Dynamic Shortest Paths and Transitive Closure: Algorithmic Techniques and Data Structures *

Camil Demetrescu †
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza", Roma, Italy

Giuseppe F. Italiano ‡
Dipartimento di Informatica, Sistemi e Produzione
Università di Roma "Tor Vergata", Roma, Italy

Abstract

In this paper, we survey fully dynamic algorithms for path problems on general directed graphs. In particular, we consider two fundamental problems: dynamic transitive closure and dynamic shortest paths. Although research on these problems spans over more than three decades, in the last couple of years many novel algorithmic techniques have been proposed. In this survey, we will make a special effort to abstract some combinatorial and algebraic properties, and some common data-structural tools that are at the base of those techniques. This will help us try to present some of the newest results in a unifying framework so that they can be better understood and deployed also by non-specialists.

*This work has been partially supported by the Sixth Framework Programme of the EU under contract number 507613 (Network of Excellence "EuroNGI: Designing and Engineering of the Next Generation Internet"), and number 001907 ("DELIS : Dynamically Evolving, Large Scale Information Systems"), and by the Italian Ministry of University and Research (Project "ALGO-NEXT: Algorithms for the Next Generation Internet and Web: Methodologies, Design and Experiments").

†Email: demetres@dis.uniroma1.it. URL: http://www.dis.uniroma1.it/~demetres.

‡Email: italiano@disp.uniroma2.it. URL: http://www.disp.uniroma2.it/users/italiano.

1 Introduction

A dynamic graph algorithm maintains a given property $\mathcal{P}$ on a graph subject to dynamic changes, such as edge insertions, edge deletions and edge weight updates. A dynamic graph algorithm should process queries on property $\mathcal{P}$ quickly, and perform update operations faster than recomputing from scratch, as carried out by the fastest static algorithm. We say that an algorithm is *fully dynamic* if it can handle both edge insertions and edge deletions. A *partially dynamic* algorithm can handle either edge insertions or edge deletions, but not both: we say that it is *incremental* if it supports insertions only, and *decremental* if it supports deletions only. In this paper, we survey fully dynamic algorithms for maintaining path problems on general directed graphs. In particular, we consider two fundamental problems.

In the *fully dynamic transitive closure problem* we wish to maintain a directed graph $G = (V, E)$ under an intermixed sequence of the following operations:

Insert(x, y):	insert an edge from x to y;
Delete(x, y):	delete the edge from x to y;
Query(x, y):	return *yes* if y is reachable from x, and return *no* otherwise.

In the *fully dynamic All Pairs Shortest Path (APSP) problem* we wish to maintain a directed graph $G = (V, E)$ with real-valued edge weights under an intermixed sequence of the following operations:

Update(x, y, w):	update the weight of edge (x, y) to the real value w; this includes as a special case both edge insertion (if the weight is set from $+\infty$ to $w < +\infty$) and edge deletion (if the weight is set to $w = +\infty$);
Distance(x, y):	output the shortest distance from x to y.
Path(x, y):	report a shortest path from x to y, if any.

Throughout the paper, we denote by m and by n the number of edges and vertices in G, respectively.

Although research on dynamic transitive closure and dynamic shortest paths problems spans over more than three decades, in the last couple of years we have witnessed a surprising resurge of interests in those two problems. The goal of this paper is to survey the newest algorithmic techniques that have been recently proposed in the literature. In particular, we will make a special effort to abstract some combinatorial and algebraic properties, and some common data-structural tools that are at the base of those techniques. This will help us try to present all the newest results in a unifying framework so that they can be better understood and deployed also by non-specialists.

1.1 History of the Problems

We first list the bounds obtainable for dynamic transitive closure with simple-minded methods. If we do nothing during each update, then we have to explore the whole graph in order to answer reachability queries: this gives $O(n^2)$ time per query and $O(1)$ time per update in the worst case. On the other extreme, we could recompute the transitive closure from scratch after each update; as this task can be accomplished via matrix multiplication [1, 39], this approach yields $O(1)$ time per query and $O(n^\omega)$ time per update in the worst case, where ω is the best known exponent for matrix multiplication (currently $\omega < 2.736$ [4]).

For the *incremental* version of transitive closure, the first algorithm was proposed by Ibaraki and Katoh [29] in 1983: its running time was $O(n^3)$ over any sequence of insertions. This bound was later improved to $O(n)$ amortized time per insertion by Italiano [30] and also by La Poutré and van Leeuwen [36]. Yellin [51] gave an $O(m^*\delta_{max})$ algorithm for m edge insertions, where m^* is the number of edges in the final transitive closure and δ_{max} is the maximum out-degree of the final graph. All these algorithms maintain explicitly the transitive closure, and so their query time is $O(1)$.

The first *decremental* algorithm was again given by Ibaraki and Katoh [29], with a running time of $O(n^2)$ per deletion. This was improved to $O(m)$ per deletion by La Poutré and van Leeuwen [36]. Italiano [31] presented an algorithm which achieves $O(n)$ amortized time per deletion on directed acyclic graphs. Yellin [51] gave an $O(m^*\delta_{max})$ algorithm for m edge deletions, where m^* is the initial number of edges in the transitive closure and δ_{max} is the maximum out-degree of the initial graph. Again, the query time of all these algorithms is $O(1)$. More recently, Henzinger and King [23] gave a randomized decremental transitive closure algorithm for general directed graphs with a query time of $O(n/\log n)$ and an amortized update time of $O(n\log^2 n)$.

Despite fully dynamic algorithms were already known for problems on undirected graphs since the earlier 80's [17], directed graphs seem to pose much bigger challenges. Indeed, the first *fully dynamic* transitive closure algorithm was devised by Henzinger and King [23] in 1995: they gave a randomized Monte Carlo algorithm with one-side error supporting a query time of $O(n/\log n)$ and an amortized update time of $O(n\widehat{m}^{0.58}\log^2 n)$, where $\widehat{m}$ is the average number of edges in the graph throughout the whole update sequence. Since $\widehat{m}$ can be as high as $O(n^2)$, their update time is $O(n^{2.16}\log^2 n)$. Khanna, Motwani and Wilson [32] proved that, when a lookahead of $\Theta(n^{0.18})$ in the updates is permitted, a deterministic update bound of $O(n^{2.18})$ can be achieved.

The situation for dynamic shortest paths has been even more dramatic. Indeed, the first papers on dynamic shortest paths date back to 1967 [37, 40, 43]. In 1985 Even and Gazit [13] and Rohnert [46] presented algorithms for maintaining shortest paths on directed graphs with arbitrary real weights. Their algorithms required $O(n^2)$ per edge insertion; however, the worst-case bounds for edge deletions were comparable to recomputing APSP from scratch. Also Ramalingam and Reps [41, 42] considered dynamic shortest path algorithms with arbitrary real weights, but in a different model. Namely, the running time of their algorithm is analyzed in terms of the output change rather than the input size (*output bounded complexity*). Frigioni *et al.* [18, 19] designed fast algorithms for graphs with bounded genus, bounded degree graphs, and bounded treewidth graphs in the same model. Again, in the worst case the running times of output-bounded dynamic algorithms are comparable to recomputing APSP from scratch.

Up to few years ago, there seemed to be few dynamic shortest path algorithms which were provably faster than recomputing APSP from scratch, and they only worked on special cases and with small integer weights. In particular, Ausiello *et al.* [2] proposed an incremental shortest path algorithm for directed graphs having positive integer weights less than C: the amortized running time of their algorithm is $O(Cn\log n)$ per edge insertion. Henzinger *et al.* [26] designed a fully dynamic algorithm for APSP on planar graphs with integer weights, with a running time of $O(n^{9/7}\log(nC))$ per operation. Fakcharoemphol and Rao in [15] designed a fully dynamic algorithm for single-source shortest paths in planar directed graphs that supports both queries and edge weight updates in $O(n^{4/5}\log^{13/5} n)$ amortized time per operation.

1.2 Novel Techniques for Dynamic Path Problems

Quite recently, many new algorithms for dynamic transitive closure and shortest path problems have been proposed.

Dynamic transitive closure. For dynamic transitive closure, King and Sagert [34] in 1999 showed how to support queries in $O(1)$ time and updates in $O(n^{2.26})$ time for general directed graphs and $O(n^2)$ time for directed acyclic graphs; their algorithm is randomized with one-side error. The bounds of King and Sagert were further improved by King [33], who exhibited a deterministic algorithm on general digraphs with $O(1)$ query time and $O(n^2 \log n)$ amortized time per update operations, where updates are insertions of a set of edges incident to the same vertex and deletions of an arbitrary subset of edges. All those algorithms are based on reductions to fast matrix multiplication and tree data structures for encoding information about dynamic paths.

Demetrescu and Italiano [10] proposed a deterministic algorithm for fully dynamic transitive closure on general digraphs that answers each query with one matrix look-up and supports updates in $O(n^2)$ amortized time. This bound can be made worst-case as shown by Sankowski in [47]. We observe that fully dynamic transitive closure algorithms with $O(1)$ query time maintain explicitly the transitive closure of the input graph, in order to answer each query with exactly one lookup (on its adjacency matrix). Since an update may change as many as $\Omega(n^2)$ entries of this matrix, $O(n^2)$ seems to be the best update bound that one could hope for this class of algorithms. This algorithm hinges upon the well known equivalence between transitive closure and matrix multiplication on a closed semiring [16, 21, 39].

In [9] the authors show how to trade off query times for updates on directed acyclic graphs: each query can be answered in time $O(n^\epsilon)$ and each update can be performed in time $O(n^{\omega(1,\epsilon,1)-\epsilon} + n^{1+\epsilon})$, for any $\epsilon \in [0,1]$, where $\omega(1,\epsilon,1)$ is the exponent of the multiplication of an $n \times n^\epsilon$ matrix by an $n^\epsilon \times n$ matrix. Balancing the two terms in the update bound yields that ϵ must satisfy the equation $\omega(1,\epsilon,1) = 1 + 2\epsilon$. The current best bounds on $\omega(1,\epsilon,1)$ [4, 28] imply that $\epsilon < 0.575$. Thus, the smallest update time is $O(n^{1.575})$, which gives a query time of $O(n^{0.575})$. This subquadratic algorithm is randomized, and has one-side error. This result has been generalized to general graphs within the same bounds by Sankowski in [47], who has also shown how to achieve an even faster update time of $O(n^{1.495})$ at the expense of a much higher $O(n^{1.495})$ query time. Roditty and Zwick presented an algorithm [44] with $O(m\sqrt{n}\,)$ update time and $O(\sqrt{n}\,)$ query time and another algorithm [45] with $O(m + n \log n)$ update time and $O(n)$ query time.

Techniques for reducing the space usage of algorithms for dynamic path problems are presented in [35]. An extensive computational study on dynamic transitive closure problems appears in [20].

Dynamic shortest paths. For dynamic shortest paths, King [33] presented a fully dynamic algorithm for maintaining all pairs shortest paths in directed graphs with positive integer weights less than C: the running time of her algorithm is $O(n^{2.5}\sqrt{C \log n}\,)$ per update. As in the case of dynamic transitive closure, this algorithm is based on clever tree data structures. Demetrescu and Italiano [11] proposed a fully dynamic algorithm for maintaining APSP on directed graphs with arbitrary real weights. Given a directed graph G, subject to dynamic operations, and such that each edge weight can assume at most S different *real* values, their algorithm supports each update in $O(S \cdot n^{2.5} \log^3 n)$ amortized time and each

query in optimal worst-case time. We remark that the sets of possible weights of two different edges need not be necessarily the same: namely, any edge can be associated with a different set of possible weights. The only constraint is that throughout the sequence of operations, each edge can assume at most S different real values, which seems to be the case in many applications. Differently from [33], this method uses dynamic reevaluation of products of real-valued matrices as the kernel for solving dynamic shortest paths. Finally, the same authors [7] have studied some combinatorial properties of graphs that make it possible to devise a different approach to dynamic all pairs shortest paths problems. This approach yields a fully dynamic algorithm for general directed graphs with non-negative real-valued edge weights that supports any sequence of operations in $O(n^2 \log^3 n)$ amortized time per update and unit worst-case time per distance query, where n is the number of vertices. Shortest paths can be reported in optimal worst-case time. The algorithm is deterministic, uses simple data structures, and appears to be very fast in practice. Using the same approach, Thorup [48] has shown how to achieve $O(n^2(\log n + \log^2((m+n)/n)))$ amortized time per update and $O(mn)$ space. His algorithm works with negative weights as well. In [49], Thorup has shown how to achieve worst-case bounds at the price of a higher complexity: in particular, the update bounds become $\widetilde{O}(n^{2.75})$, where $\widetilde{O}(f(n))$ denotes $O(f(n) \cdot \text{polylog } n)$.

An extensive computational study on dynamic all pairs shortest path problems appears in [8].

1.3 Organization of the paper

The remainder of this paper is organized as follows. In Section 2 we describe some combinatorial and algebraic properties of path problems on directed graphs. In Section 3 we abstract some data structures that are at the base of algorithmic techniques for dynamic path problems. Next, we focus on the newest dynamic transitive closure algorithms in Section 4 and on the newest dynamic shortest path algorithms in Section 5. In Section 6 we list some concluding remarks and open problems.

2 Combinatorial and Algebraic Properties of Path Problems

In this section we provide some background for the two algorithmic graph problems considered in this paper: transitive closure and all-pairs shortest paths. We first discuss some algebraic properties of these problems, and then we describe efficient methods for computing the Kleene closure of a matrix through reduction to matrix multiplication. In particular, we discuss two methods: the first is based on a simple doubling technique that consists of repeatedly concatenating paths to form longer paths via matrix multiplication, and the second is based on a Divide and Conquer strategy. We conclude this section by discussing a useful combinatorial property of long paths.

2.1 Path Problems and Closed Semirings

We now discuss some algebraic properties of transitive closure and shortest paths, addressing the tight relationship between these two problems and matrix sum and matrix multiplication over a closed semiring (see [5] for more details). In particular, the transitive closure of a directed graphs can be obtained from the adjacency matrix of the graph via operations on

the semiring of Boolean matrices, that we denote by $\{+, \cdot, 0, 1\}$. In this case, $+$ and $\cdot$ denote the usual sum and multiplication over Boolean matrices.

Theorem 1 *Let $G = (V, E)$ be a directed graph and let $TC(G)$ be the (reflexive) transitive closure of G. If X is the Boolean adjacency matrix of G, then the Boolean adjacency matrix of $TC(G)$ is the Kleene closure of X on the $\{+, \cdot, 0, 1\}$ Boolean semiring:*

$$X^* = \sum_{i=0}^{n-1} X^i.$$

Similarly, shortest path distances in a directed graph with real-valued edge weights can be obtained from the weight matrix of the graph via operations on the semiring of real matrices, that we denote by $\{\oplus, \odot, \mathcal{R}\}$, or more simply by $\{\min, +\}$. Here, $\mathcal{R}$ is the set of real values and $\oplus$ and $\odot$ are defined as follows. Given two real-valued matrices A and B, $C = A \odot B$ is the matrix product such that $C[x, y] = \min_{1 \leq z \leq n}\{A[x, z] + B[z, y]\}$ and $D = A \oplus B$ is the matrix sum such that $D[x, y] = \min\{A[x, y], B[x, y]\}$. We also denote by AB the product $A \odot B$ and by $AB[x, y]$ entry (x, y) of matrix AB.

Theorem 2 *Let $G = (V, E)$ be a weighted directed graph with no negative-length cycles. If X is a weight matrix such that $X[x, y]$ is the weight of edge (x, y) in G, then the distance matrix of G is the Kleene closure of X on the $\{\oplus, \odot, \mathcal{R}\}$ semiring:*

$$X^* = \bigoplus_{i=0}^{n-1} X^i.$$

2.2 Logarithmic Decomposition

We first describe a simple method for computing X^* in $O(n^\mu \cdot \log n)$ worst-case time, where $O(n^\mu)$ is the time required for computing the product of two matrices over a closed semiring. The algorithm is based on a simple path doubling argument. We focus on the $\{+, \cdot, 0, 1\}$ semiring; the case of the $\{\min, +\}$ semiring is completely analogous.

Let $\mathcal{B}_n$ be the set of $n \times n$ Boolean matrices and let $X \in \mathcal{B}_n$. We define a sequence of $\log n + 1$ polynomials $P_0, \ldots, P_{\log n}$ over Boolean matrices as:

$$P_i = \begin{cases} X & if\ i = 0 \\ P_{i-1} + P_{i-1}^2 & if\ i > 0 \end{cases}$$

It is not difficult to see that for any $1 \leq u, v \leq n$, $P_i[u, v] = 1$ if and only if there is a path $u \rightsquigarrow v$ of length at most 2^i in X. We combine paths of length ≤ 2 in X to form paths of length ≤ 4, then we concatenate all paths found so far to obtain paths of length ≤ 8 and so on. As the length of the longest detected path increases exponentially and the longest simple path is no longer than n, a logarithmic number of steps suffices to detect if any two nodes are connected by a path in the graph as stated in the following theorem.

Theorem 3 *Let X be an $n \times n$ matrix. Then $X^* = I_n + P_{\log n}$.*

2.3 Recursive Decomposition

We now show that the Kleene closure of a matrix can be computed more efficiently in time $O(n^\mu)$. The method we present is due to Munro [39] and is based on a Divide and Conquer strategy.

Let $X \in \mathcal{B}_n$. Without loss of generality, we assume that n is a power of 2. We define a mapping $\mathcal{F} : \mathcal{B}_n \to \mathcal{B}_n$ with the following equations:

$$\begin{cases} E = (A + BD^*C)^* \\ F = EBD^* \\ G = D^*CE \\ H = D^* + D^*CEBD^* \end{cases} \tag{1}$$

where A, B, C, D and E, F, G, H are obtained by partitioning X and $Y = \mathcal{F}(X)$ into submatrices of dimension $\frac{n}{2} \times \frac{n}{2}$ as follows:

$$X = \begin{array}{|c|c|}\hline \text{A} & \text{B} \\ \hline \text{C} & \text{D} \\ \hline \end{array} \qquad Y = \begin{array}{|c|c|}\hline \text{E} & \text{F} \\ \hline \text{G} & \text{H} \\ \hline \end{array}$$

Theorem 4 ([39]) *Let X be an $n \times n$ Boolean matrix. Then $\mathcal{F}(X) = X^*$.*

It is possible to think of a different definition of function $\mathcal{F}$ that provides an alternative way of computing the Kleene closure. Let $\mathcal{B}_n$ be the set of $n \times n$ Boolean matrices, let $X \in \mathcal{B}_n$, and let $\mathcal{G} : \mathcal{B}_n \to \mathcal{B}_n$ be the mapping defined by means of the following equations:

$$\begin{cases} E = A^* + A^*BHCA^* \\ F = A^*BH \\ G = HCA^* \\ H = (D + CA^*B)^* \end{cases} \tag{2}$$

where X and $Y = \mathcal{G}(X)$ are defined as:

$$X = \begin{array}{|c|c|}\hline \text{A} & \text{B} \\ \hline \text{C} & \text{D} \\ \hline \end{array} \qquad Y = \begin{array}{|c|c|}\hline \text{E} & \text{F} \\ \hline \text{G} & \text{H} \\ \hline \end{array}$$

Theorem 5 *For any $X \in \mathcal{B}_n$, $\mathcal{G}(X) = \mathcal{F}(X) = X^*$.*

Proof. If we rename syntactically submatrices A, B, C, D, and E, F, G, H in the set of equations (2) as follows:

$$\begin{array}{ll} A \longrightarrow D & E \longrightarrow H \\ B \longrightarrow C & F \longrightarrow G \\ C \longrightarrow B & G \longrightarrow F \\ D \longrightarrow A & H \longrightarrow E \end{array}$$

we obtain the set of equations (1). Thus, $\mathcal{G} = \mathcal{F}$. □

We recall that it is possible to compute E, F, G and H with two recursive calls of $\mathcal{F}$, six multiplications, and two additions of $\frac{n}{2} \times \frac{n}{2}$ matrices (see [1] for further details). Thus:

$$T(n) \leq 2T(\frac{n}{2}) + 6M(\frac{n}{2}) + 2\left(\frac{n}{2}\right)^2$$

where $M(n) = O(n^\mu)$ is the time required to multiply two $n \times n$ matrices. Solving the recurrence relation, we obtain that $T(n) = O(n^\mu)$. This yields the following theorem.

Theorem 6 *Let X be an $n \times n$ matrix and let $T(n)$ be the time required to compute recursively $\mathcal{F}(X)$. Then $T(n) = O(n^\mu)$, where $O(n^\mu)$ is the time required to multiply two matrices.*

In the case of Boolean matrix multiplication (see [4]), X^* can be computed in $o(n^{2.38})$ worst-case time. As a final note, if the size n of X and Y is not a power of 2, we can embed both X and Y in larger matrices $\widehat{X}$ and $\widehat{Y}$, respectively, having dimension that is a power of 2, of the form:

$$\widehat{X} = \begin{array}{|c|c|}\hline X & 0 \\ \hline 0 & I \\ \hline \end{array} \qquad \widehat{Y} = \begin{array}{|c|c|}\hline Y & 0 \\ \hline 0 & I \\ \hline \end{array}$$

where I is an identity matrix of the smallest possible size. Since this will at most double the size of the matrices, the asymptotic running time of computing $\widehat{Y} = \widehat{X}^*$ is not affected.

2.4 Long Paths

In this section we discuss an intuitive combinatorial property of long paths. Namely, if we pick a subset S of vertices at random from a graph G, then a sufficiently long path will intersect S with high probability. This can be very useful in finding a long path by using short searches as illustrated in Figure 1.

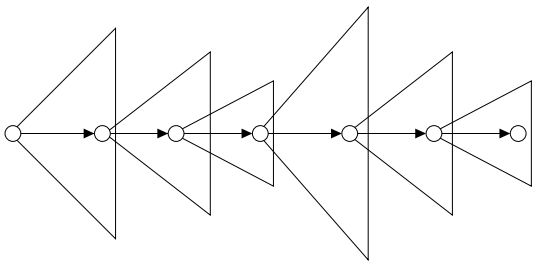

Figure 1: Finding a long path with short searches.

To the best of our knowledge, this property was first given in [22], and later on it has been used may times in designing efficient algorithms for transitive closure and shortest paths (see e.g., [11, 33, 50, 52]). The following theorem is from [50].

Theorem 7 *Let $S \subseteq V$ be a set of vertices chosen uniformly at random. Then the probability that a given simple path has a sequence of more than $\frac{cn \log n}{|S|}$ vertices, none of which are from S, for any $c > 0$, is, for sufficiently large n, bounded by $2^{-\alpha c}$ for some positive α.*

As shown in [52], it is possible to choose set S deterministically by a reduction to a hitting set problem [3, 38]. A similar technique has also been used in [33].

2.5 Locality

Recently, Demetrescu and Italiano [7] proposed a new approach to dynamic path problems based on maintaining classes of paths characterized by local properties, i.e., properties that hold for all proper subpaths, even if they may not hold for the entire paths. They showed that this approach can play a crucial role in the dynamic maintenance of shortest paths. For instance, they considered a class of paths defined as follows:

Definition 1 *A path π in a graph is locally shortest if and only if every proper subpath of π is a shortest path.*

This definition is inspired by the optimal-substructure property of shortest paths: all subpaths of a shortest path are shortest. However, a locally shortest path may not be shortest.

The fact that locally shortest paths include shortest paths as a special case makes them an useful tool for computing and maintaining distances in a graph. Indeed, paths defined locally have interesting combinatorial properties in dynamically changing graphs. For example, it is not difficult to prove that the number of locally shortest paths that may change due to an edge weight update is $O(n^2)$ if updates are partially dynamic, i.e., increase-only or decrease-only:

Theorem 8 *Let G be a graph subject to a sequence of increase-only or decrease-only edge weight updates. Then the amortized number of paths that start or stop being locally shortest at each update is $O(n^2)$.*

Unfortunately, Theorem 8 may not hold if updates are fully dynamic, i.e., increases and decreaes of edge weights are intermixed. To cope with pathological sequences, a possible solution is to retain information about the history of a dynamic graph, considering the following class of paths:

Definition 2 *A historical shortest path (in short, historical path) is a path that has been shortest at least once since it was last updated.*

Here, we assume that a path is updated when the weight of one of its edges is changed. Applying the locality technique to historical paths, we derive locally historical paths:

Definition 3 *A path π in a graph is locally historical if and only if every proper subpath of π is historical.*

Like locally shortest paths, also locally historical paths include shortest paths, and this makes them another useful tool for maintaining distances in a graph:

Lemma 1 *If we denote by SP, LSP, and LHP respectively the sets of shortest paths, locally shortest paths, and locally historical paths in a graph, then at any time the following inclusions hold: $SP \subseteq LSP \subseteq LHP$.*

Differently from locally shortest paths, locally historical paths exhibit interesting combinatorial properties in graphs subject to fully dynamic updates. In particular, it is possible to prove that the number of paths that become locally historical in a graph at each edge weight update depends on the number of historical paths in the graph.

Theorem 9 *(Demetrescu and Italiano [7]) Let G be a graph subject to a sequence of update operations. If at any time throughout the sequence of updates there are at most $O(h)$ historical paths in the graph, then the amortized number of paths that become locally historical at each update is $O(h)$.*

To keep changes in locally historical paths small, it is then desirable to have as few historical paths as possible. Indeed, it is possible to transform every update sequence into a slightly longer equivalent sequence that generates only a few historical paths. In particular, there exists a simple *smoothing* strategy that, given any update sequence Σ of length k, produces an operationally equivalent sequence $F(\Sigma)$ of length $O(k \log k)$ that yields only $O(\log k)$ historical shortest paths between each pair of vertices in the graph. We refer the interested reader to [7] for a detailed description of this smoothing strategy. According to Theorem 9, this technique implies that only $O(n^2 \log k)$ locally historical paths change at each edge weight update in the smoothed sequence $F(\Sigma)$.

As elaborated in [7], locally historical paths can be maintained very efficiently. Since by Lemma 1 locally historical paths include shortest paths, this yields the fastest known algorithm for fully dynamic all pairs shortest paths.

3 Algorithmic Techniques

In this section we describe some algorithmic techniques which are the kernel of the best known algorithms for maintaining transitive closure and shortest paths.

We start with some observations which are common to all the techniques considered here. First of all, we note that the algebraic structure of path problems allows one to support insertions in a natural fashion. Indeed, insertions correspond to the $\oplus$ operation on closed semirings. This perhaps can explain the wealth of fast incremental algorithms for dynamic path problems on directed graphs [29, 30, 36, 51]. However, there seems to be no natural setting for deletions in this algebraic framework. Thus, in designing fully dynamic algorithms it seems quite natural to focus on special techniques and data structures for supporting deletions.

The second remark is that most fully dynamic algorithms are surprisingly based on the same decompositions used for the best static algorithms (see Section 2). The main difference is that they maintain dynamic data structures at each level of the decomposition. As we will see, the definition of a suitable interface between data structures at different levels plays an important role in the design of efficient dynamic algorithms.

In the remainder of this section we describe data structures that are able to support dynamic operations at each level. In Sections 4 and 5, the data structures surveyed here will be combined with the decompositions shown in Section 2 to obtain efficient algorithms for dynamic path problems on directed graphs.

3.1 Tools for Trees

In this section we describe a tree data structure for keeping information about dynamic path problems. The first appearance of this tool dates back to 1981, when Even and Shiloach [14] showed how to maintain a breadth-first tree of an undirected graph under any sequence of edge deletions; they used this as a kernel for decremental connectivity on undirected graphs. Later on, Henzinger and King [23] showed how to adapt this data structure to fully dynamic transitive closure in directed graphs. Recently, King [33] designed an extension of this tree data structure to weighted directed graphs for solving fully dynamic all pairs shortest paths.

The Problem. The goal is to maintain information about breadth-first search (BFS) on an undirected graph G undergoing deletions of edges. In particular, in the context of dynamic

path problems, we are interested in maintaining BFS trees of depth up to d, with $d \leq n$. For the sake of simplicity, we describe only the case where deletions do not disconnect the underlying graph. The general case can be easily handled by means of "phony" edges (i.e., when deleting an edge that disconnects the graph, we just replace it by a phony edge).

It is well known that BFS partitions vertices into levels, so that there can be edges only between adjacent levels. More formally, let r be the vertex where we start BFS, and let level ℓ_i contains vertices encountered at distance i from r ($\ell_0 = \{r\}$): edges incident to a vertex at level ℓ_i can have their other endpoints either at level ℓ_{i-1}, ℓ_i, or ℓ_{i+1}, and no edge can connect vertices at levels ℓ_i and ℓ_j for $|j - i| > 1$. Let $\ell(v)$ be the level of vertex v.

Given an undirected graph $G = (V, E)$ and a vertex $r \in V$, we would like to support any intermixed sequence of the following operations:

- `Delete`(x, y): delete edge (x, y) from G.
- `Level`(u): return the level $\ell(u)$ of vertex u in the BFS tree rooted at r (return $+\infty$ if u is not reachable from r).

In the remainder of this paper, to indicate that an operation `Y`() is performed on a data structure `X`, we use the notation `X.Y`().

Data Structure. We maintain information about BFS throughout the sequence of edge deletions by simply keeping explicitly those levels. In particular, for each vertex v at level ℓ_i in T, we maintain the following data structures: UP(v), SAME(v) and DOWN(v) containing the edges connecting v to level ℓ_{i-1}, ℓ_i, and ℓ_{i+1}, respectively. Note that for all $v \neq r$, UP(v) must contain at least one edge (i.e., the edge from v to its parent in the BFS tree). In other words, a non-empty UP(v) witnesses the fact that v is actually entitled to belong to that level. This property will be important during edge deletions: whenever UP(v) gets emptied because of deletions, v looses its right to be at that level and must be demoted at least one level down.

Implementation of Operations. When edge (x, y) is being deleted, we proceed as follows. If $\ell(x) = \ell(y)$, simply delete (x, y) from SAME(y) and from SAME(x). The levels encoded in UP, SAME and DOWN still capture the BFS structure of G. Otherwise, without loss of generality let $\ell(x) = \ell_{i-1}$ and $\ell(y) = \ell_i$. Update the sets UP, SAME and DOWN by deleting x from UP(y) and y from DOWN(x). If UP(y) $\neq \emptyset$, then there is still at least one edge connecting y to level ℓ_{i-1}, and the levels will still reflect the BFS structure of G after the deletion.

The main difficulty is when UP(y) $= \emptyset$ after the deletion of (x, y). In this case, deleting (x, y) causes y to loose its connection to level ℓ_{i-1}. Thus, y has to drop down at least one level. Furthermore, its drop may cause a deeper landslide in the levels below. This case can be handled as follows.

We use a FIFO queue Q, initialized with vertex y. We will insert a vertex v in the queue Q whenever we discover that UP(v) $= \emptyset$, i.e., vertex v has to be demoted at least one level down. We will repeat the following demotion step until Q is empty:

Demotion Step :

1. Remove the first vertex in Q, say v.

2. Delete v from its level $\ell(v) = \ell_i$ and tentatively try to place v one level down, i.e., in ℓ_{i+1}.
3. Update the sets UP, SAME and DOWN consequently:
   (a) For each edge (u, v) in SAME(v), delete (u, v) from SAME(u) and insert (u, v) in DOWN(u) and UP(v) (as UP(v) was empty, this implies that UP(v) will be initialized with the old set SAME(v)).
   (b) For each edge (v, z) in DOWN(v), move edge (v, z) from UP(z) to SAME(z) and from DOWN(v) to SAME(v); if the new UP(z) is empty, insert z in the queue Q. Note that this will empty DOWN(v).
   (c) If UP(v) is still empty, insert v again into Q.

Analysis. It is not difficult to see that applying the Demotion Step until the queue is empty will maintain correctly the BFS levels. Level queries can be answered in constant time. To bound the total time required to process any sequence of edge deletions, it suffices to observe that each time an edge (u, v) is examined during a demotion step, either u or v will be dropped one level down. Thus, edge (u, v) can be examined at most $2d$ times in any BFS levels up to depth d throughout any sequence of edge deletions. This implies the following theorem.

Theorem 10 *Maintaining BFS levels up to depth d requires $O(md)$ time in the worst case throughout any sequence of edge deletions in an undirected graph with m initial edges.*

This means that maintaining BFS levels requires d times the time needed for constructing them. Since $d \leq n$, we obtain a total bound of $O(mn)$ if there are no limits on the depth of the BFS levels.

◁ ◇ ▷

As it was shown in [23, 33], it is possible to extend the BFS data structure presented in this section to deal with weighted directed graphs. In this case, a shortest path tree is maintained in place of BFS levels: after each edge deletion or edge weight increase, the tree is reconnected by essentially mimicking Dijkstra's algorithm rather than BFS. Details can be found in [33].

3.2 Tools for Dynamic Matrices

As it was mentioned earlier, one can look at path problems on graphs from the matrix viewpoint. In this section, we will describe matrix data structures for keeping information about paths in dynamic directed graphs. In particular, we consider the problem of maintaining products of matrices subject to updates of entries. For the sake of simplicity, we will show how to deal with products of two matrices only: details about the extension to polynomials over matrices of arbitrary degree are spelled out in [6, 10]. We state the problem so that it can be useful later on for implementing dynamic transitive closure and dynamic shortest paths.

In the following, we denote by $\mathcal{M}_n$ the set of $n \times n$ Boolean matrices, by I_n the unit matrix (i.e., the $n \times n$ Boolean matrix with 1 in the diagonal and 0 elsewhere), and by $+$ and $\cdot$ sum and multiplication on Boolean matrices, respectively. We also use operation $\ominus$ defined as follows: $C = A \ominus B$, where $C[i, j] = A[i, j] \wedge (\neg B[i, j])$. Notice that this corresponds to matrix difference, with the exception that $0 \ominus 1 = 0$. Moreover, if X is a matrix, we denote

by $I_{X,i}$ and $J_{X,j}$ the matrices equal to X in the i-th row and in the j-th column, respectively, and null in any other entries.

The Problem. Given two Boolean matrices $X_1, X_2 \in \mathcal{M}_n$, we consider the problem of maintaining a data structure for product $P = X_1 \cdot X_2$, under an intermixed sequence of update and query operations of the following kind:

- `SetRow`$(i, \Delta X, X_b)$: perform the row update operation $X_b \leftarrow X_b + I_{\Delta X,i}$, where ΔX is an $n \times n$ Boolean update matrix, and $b = 1, 2$. The operation sets to 1 the entries in the i-th row of variable X_b of product P as specified by matrix ΔX.
- `SetCol`$(i, \Delta X, X_b)$: perform the column update operation $X_b \leftarrow X_b + J_{\Delta X,i}$, where ΔX is an $n \times n$ Boolean update matrix, and $b = 1, 2$. The operation sets to 1 the entries in the i-th column of variable X_b of product P as specified by matrix ΔX.
- `Reset`$(\Delta X, X_b)$: perform the update operation $X_b \leftarrow X_b \ominus \Delta X$, where ΔX is an $n \times n$ Boolean update matrix such that $\Delta X \subseteq X_b$, and $b = 1, 2$. The operation resets to 0 the entries of variable X_b of product P as specified by matrix ΔX.
- `Lookup`(): return the maintained value of P.

We add to the previous five operations a further update operation especially designed for lazy evaluation in dynamic path problems:

- `LazySet`$(\Delta X, X_b)$: perform the update operation $X_b \leftarrow X_b + \Delta X$, where ΔX is an $n \times n$ Boolean update matrix, and $b = 1, 2$. The operation sets to 1 the entries of variable X_b of product P as specified by matrix ΔX. However, the maintained value of P is not updated by this operation.

Let C_P be the correct value of P that we would have by recomputing it from scratch after each update, and let M_P be the actual value that we maintain. If no `LazySet` operation is ever performed, then always $M_P = C_P$. Otherwise, M_P is not necessarily equal to C_P, and we guarantee the following weaker property on M_P: if $C_P[u, v]$ flips from 0 to 1 due to a `SetRow`/`SetCol` operation on a variable X_b, then $M_P[u, v]$ flips from 0 to 1 as well. This means that `SetRow` and `SetCol` always correctly reveal new 1's in the maintained value of P, possibly taking into account the 1's inserted through previous `LazySet` operations. This property will be crucial for dynamic path problems.

We observe that `SetRow` and `SetCol` can be easily implemented in $O(n^2)$ worst-case time by recomputing just the affected entries of the product matrix P. However, the main difficulty lies behind the interaction between `Reset` and `LazySet` operations: in particular, the presence of `LazySet` makes it difficult to tell right from wrong in the maintained data structure for P. We next show how to support all operations in $O(n^2)$ amortized time.

Data Structure. To maintain the product $P = X_1 \cdot X_2$, we keep three matrices of witness sets initialized as follows. For any triple of indices x, y, z:

- $Left[y, z] = \{x \mid X_1[x, y] = 1 \wedge X_2[y, z] = 1\}$
- $Prod[x, z] = \{y \mid X_1[x, y] = 1 \wedge X_2[y, z] = 1\}$
- $Right[x, y] = \{z \mid X_1[x, y] = 1 \wedge X_2[y, z] = 1\}$

Clearly, for any x, y, $(X_1 \cdot X_2)[x, y] = 1$ if and only if $|Prod[x, y]| > 0$.

Implementation of Operations. Operations can be realized as follows:

- `SetRow`$(i, \Delta X, X_1)$: perform $X_1 \leftarrow X_1 + I_{\Delta X,i}$ and for each u, v s.t. $X_1[i,u] = 1$ and $X_2[u,v] = 1$ add i, u, and v to $Left[u,v]$, $Prod[i,v]$, and $Right[i,u]$, respectively. `SetCol`$(i, \Delta X, X_2)$ is similar.
- `SetRow`$(i, \Delta X, X_2)$: perform $X_2 \leftarrow X_2 + I_{\Delta X,i}$ and for each u, v s.t. $X_1[u,i] = 1$ and $X_2[i,v] = 1$ add u, i, and v to $Left[i,v]$, $Prod[u,v]$, and $Right[u,i]$, respectively. `SetCol`$(i, \Delta X, X_1)$ is similar.
- `LazySet`$(\Delta X, X_b)$: perform $X_b \leftarrow X_b + \Delta X$ without updating $Left$, $Prod$, and $Right$.
- `Reset`$(\Delta X, X_1)$: do $X_1 \leftarrow X_1 \ominus \Delta X$. Then for each x, y s.t. $\Delta X[x,y] = 1$ and for each $z \in Right[x,y]$ remove x, y, z from $Left[y,z]$, $Prod[x,z]$, and $Right[x,y]$, respectively. $Reset(\Delta X, X_2)$ is similar.

Theorem 11 *Any* `SetRow`, `SetCol`, `LazySet`, *operation on a product $X_1 \cdot X_2$ can be supported in $O(n^2)$ worst-case time. The cost of any* `Reset` *can be charged to previous set operations.*

Proof. It is easy to see that any set operation can be realized in $O(n^2)$ worst-case time. Let

$$\Phi = \sum_{u,v} |Prod[u,v]|$$

be a potential function associated to $X_1 \cdot X_2$, $0 \leq \Phi \leq n^3$. The rest of the proof follows from the fact that any set operation increases Φ by at most n^2 units and that any reset operation decreases Φ by at most n units for each reset entry. Note that `LazySet` does not affect Φ. □

◁ ◇ ▷

The techniques presented in this section can be extended to deal with polynomials of arbitrary degree, e.g., products of more than two terms, and that the space usage can be reduced from $O(n^3)$ to $O(n^2)$. Details can be found in [6, 10]. Moreover, similar data structures can be given for settings different from the semiring of Boolean matrices. In particular, in [11] the problem of maintaining polynomials of matrices over the {min,+} semiring is addressed. The running time of operations for maintaining polynomials in this semiring is given below.

Theorem 12 *Let P be a polynomial with constant degree of matrices over the $\{\min, +\}$ semiring. Any* `SetRow`, `SetCol`, `LazySet`, *and* `Reset` *operation on variables of P can be supported in $O(D \cdot n^2)$ amortized time, where D is the maximum number of different values assumed by entries of variables during the sequence of operations.* `Query` *operations on the value of P are supported in constant time.*

4 Dynamic Transitive Closure

In this section we survey the best known algorithms for fully dynamic transitive closure. We will first start in Section 4.1 with the algorithm of King [33], whose main ingredients are the logarithmic decomposition of Section 2.2 and the tools for trees described in Section 3.1. This methods yields $O(n^2 \log n)$ amortized time per update and $O(1)$ per query. We will next

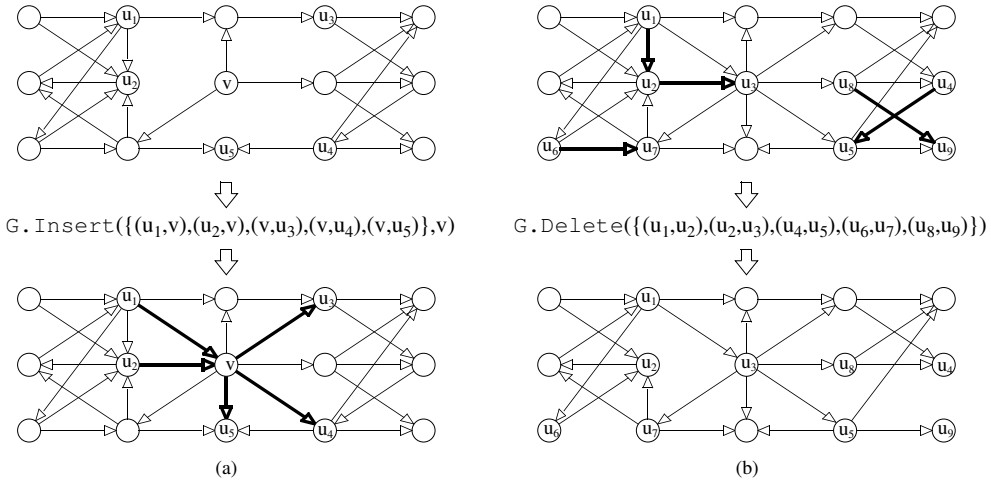

Figure 2: (a) `Insert` operation; (b) `Delete` operation.

show in Section 4.2 how to obtain the same bounds with the techniques for dynamic matrices described in Section 3.2 in place of the tools for trees. We will then conclude in Section 4.3 with the algorithm by Demetrescu and Italiano [10], which uses the same matrix-based approach but fully exploits the power of the recursive decomposition presented in Section 2.3. Using this method, updates are supported in $O(n^2)$ amortized time and reachability queries are answered with one matrix lookup. We start with a formal definition of the fully dynamic transitive closure problem.

The Problem. Let $G = (V, E)$ be a directed graph and let $TC(G) = (V, E')$ be its transitive closure. We consider the problem of maintaining a data structure for graph G under an intermixed sequence of update and query operations of the following kinds:

- `Insert`(v, I): perform the update $E \leftarrow E \cup I$, where $I \subseteq E$ and $v \in V$. This operation assumes that all edges in I are incident to v. We call this kind of update a v-Centered insertion in G.
- `Delete`(D): perform the update $E \leftarrow E - D$, where $D \subseteq E$.
- `Query`(x, y): perform a query operation on $TC(G)$ and return 1 if $(x, y) \in E'$ and 0 otherwise.

We note that these generalized `Insert` and `Delete` updates are able to change, with just one operation, the graph by adding or removing a whole set of edges, rather than a single edge (see Figure 2). Differently from other variants of the problem, we do not address the issue of returning actual paths between nodes, and we just consider the problem of answering reachability queries.

4.1 Tree-based Dynamic Transitive Closure with Logarithmic Decomposition

In this section we describe how to maintain the Transitive Closure of a directed graph in $O(n^2 \log n)$ amortized time per update operation. The algorithm that we describe has been

designed by King [33] and is based on the tree data structure presented in Section 3.1 and on the logarithmic decomposition described in Section 2.2. To support queries efficiently, the algorithm uses also a *counting* technique that consists of keeping a count for each pair of vertices x, y of the number of insertion operations that yielded new paths between them. Counters are maintained so that there is a path from x to y if and only if the counter for that pair is non-zero. The counting technique has been first introduced in [34]: in that case, counters keep track of the number of different distinct paths between pairs of vertices in an acyclic directed graph. We show the data structure used for maintaining the Transitive Closure and how operations `Insert`, `Delete` and `Query` are implemented.

Data Structure. Given a directed graph $G = (V, E)$, we maintain $\log n + 1$ levels. On each level i, $0 \leq i \leq \log n$ we maintain the following data structures:

- a graph $G_i = (V, E_i)$ such that $(x, y) \in E_i$ if there is a path from x to y in G of length $\leq 2^i$. Note that the converse is not necessarily true: i.e., $(x, y) \in E_i$ may not necessarily imply, however, that there is a path from x to y in G of length $\leq 2^i$. We maintain G_0 and $G_{\log n}$ such that $G_0 = G$ and $G_{\log n} = TC(G)$.
- for each $v \in V$, a BFS tree $Out_{i,v}$ of depth 2 of G_i rooted at v and a BFS tree $In_{i,v}$ of depth 2 of $\widehat{G_i}$ rooted at v, where $\widehat{G_i}$ is equal to G_i, except for the orientation of edges, which is reversed. We maintain the BFS trees with instances of the data structure presented in Section 3.1.
- a matrix $Count_i[x, y] = |\{\ v\ :\ x \in In_{i,v}\ \wedge\ y \in Out_{i,v}\ \}|$. Note that $Count_i[x, y] > 0$, if there is a path from x to y in G of length $\leq 2^{i+1}$.

Implementation of Operations. Operations can be realized as follows:

- `Insert`(v, I): for each $i = 0$ to $\log n$, do the following: add I to E_i, rebuild $Out_{i,v}$ and $In_{i,v}$, updating $Count_i$ accordingly, and add to I any (x, y) such that $Count_i[x, y]$ flips from 0 to 1.
- `Delete`(D): for each $i = 0$ to $\log n$, do the following: remove D from E_i, for each $(x, y) \in D$ do $Out_{i,v}$`.Delete`(x, y) and $In_{i,v}$`.Delete`(x, y), updating $Count_i$ accordingly, and add to D all (x, y) such that $Count_i[x, y]$ flips from positive to zero.
- `Query`(x, y): return 1 if $(x, y) \in E_{\log n}$ and 0 otherwise.

We note that an `Insert` operation simply rebuilds the BFS trees rooted at v on each level of the decomposition. It is important to observe that the trees rooted at other vertices on any level i might not be valid BFS trees of the current graph G_i, but are valid BFS trees of some older version of G_i that did not contain the newly inserted edges. A `Delete` operation, instead, maintains dynamically the tree data structures on each level, removing the deleted edges as described in Section 3.1 and propagating changes up to the decomposition.

Analysis. To prove the correctness of the algorithm, we need to prove that, if there is a path from x to y in G of length $\leq 2^i$, then $(x, y) \in E_i$. Conversely, it is easy to see that $(x, y) \in E_i$ only if there is a path from x to y in G of length $\leq 2^i$. We first consider `Insert` operations. It is important to observe that, by the problem's definition, the set I contains only edges incident to v for $i = 0$, but this might not be the case for $i > 0$, since entries $Count_i[x, y]$ with $x \neq v$ and $y \neq v$ might flip from 0 to 1 during a v-centered insertion. However, we follow a lazy approach and we only rebuild the BFS trees on each level rooted at v. The correctness of this follows from the simple observation that any new paths that appear due to a v-centered insertion pass through v, so rebuilding the trees rooted at v is enough to keep track of these new paths. To prove this, we proceed by induction. We assume that for any new paths $x \rightsquigarrow v$ and $v \rightsquigarrow y$ of length $\leq 2^i$, $(x, v), (v, y) \in E_i$ at the beginning of loop iteration i. Since $x \in In_{i,v}$ and $y \in Out_{i,v}$ after rebuilding $In_{i,v}$ and $Out_{i,v}$, $Count_i[x, y]$ is increased by one if no path $x \rightsquigarrow v \rightsquigarrow y$ of length $\leq 2^{i+1}$ existed before the insertion. Thus $(x, y) \in E_{i+1}$ at the beginning of loop iteration $i + 1$. To complete our discussion of correctness, we note that deletions act as "undo" operations that leave the data structures as if deleted edges were never inserted. The running time is established as follows.

Theorem 13 *Any* `Insert` *operation requires* $O(n^2 \log n)$ *worst-case time, the cost of* `Delete` *operations can be charged to previous insertions, and* `Query` *operations are answered in constant time.*

Proof. The bound for `Insert` operations derives from the observation that reconstructing BFS trees on each of the $\log n+1$ levels requires $O(n^2)$ time in the worst case. By Theorem 10, any sequence of `Delete` operations can be supported in $O(d)$ times the cost of building a BFS tree of depth up to d. Since $d = 2$ in the data structure, this implies that the cost of deletions can be charged to previous insertion operations. The bound for `Query` operations is straightforward. □

The previous theorem implies that updates are supported in $O(n^2 \log n)$ amortized time per operation.

4.2 Matrix-based Dynamic Transitive Closure with Logarithmic Decomposition

In this section we show how to rephrase the fully dynamic transitive closure algorithm by King [33] presented in Section 4.1 in terms of the matrix-based tools described in Section 3.2. Using matrices instead of trees will pave the road for the faster dynamic algorithm that will be illustrated in Section 4.3, which is based on the recursive decomposition of Section 2.3 instead of the logarithmic decomposition of Section 2.2.

Data Structure. In Section 2.2 we have shown that the Kleene closure of a Boolean matrix X can be computed from scratch via matrix multiplication by computing $\log n$ polynomials $P_i = P_{i-1} + P_{i-1}^2$, $1 \leq i \leq \log n$. In the static case where X^* has to be computed only once, intermediate results can be thrown away as only the final value $X^* = P_{\log n}$ is required. In the dynamic case, instead, intermediate results provide useful information for updating efficiently X^* whenever X gets modified.

In this section we consider a slightly different definition of polynomials $P_1, \ldots, P_{\log n}$ with the property that each of them has degree ≤ 3. Let X be an $n \times n$ Boolean matrix. We define a sequence of $\log n + 1$ polynomials over Boolean matrices $Q_0, \ldots, Q_{\log n}$ as:

$$Q_i = \begin{cases} X & if\ i = 0 \\ Q_{i-1} + Q_{i-1}^2 + Q_{i-1}^3 & if\ i > 0 \end{cases}$$

Before describing the data structure for maintaining the Kleene closure of X, we note that for any $1 \leq u, v \leq n$, $Q_i[u, v] = 1$ if and only if there is a path $u \rightsquigarrow v$ of length at most 3^i in X. This yields the following theorem, which corresponds to Theorem 3, given in Section 2.2 for polynomials P_i of degree 2:

Theorem 14 *Let X be an $n \times n$ Boolean matrix. Then $X^* = I_n + Q_{\log n}$.*

The data structure for maintaining X^* is as follows. We maintain an $n \times n$ Boolean matrix X and we maintain the $\log n$ polynomials $Q_1 \ldots Q_{\log n}$ of degree 3 with instances of the data structure for Boolean matrices presented in Section 3.2. The reason for considering degree 3 instead of degree 2 in the data structure will be addressed in the next section.

Implementation of Operations. Operations can be realized as follows:

- `Insert`(v, I): first let $\Delta X[x, y]$ be 1 for each $(x, y) \in I$, and 0 otherwise. Then for each $i = 1$ to $\log n$, do the following: perform Q_i.`LazySet`$(\Delta X, Q_{i-1})$, Q_i.`SetRow`$(v, \Delta X, Q_{i-1})$, Q_i.`SetCol`$(v, \Delta X, Q_{i-1})$, and $\Delta X \leftarrow Q_i$.`Lookup`().
- `Delete`(D): first let $\Delta X[x, y]$ be 1 for each $(x, y) \in D$, and 0 otherwise. Then for each $i = 0$ to $\log n$, do the following: perform Q_i.`Reset`$(\Delta X, Q_{i-1})$ and $\Delta X \leftarrow Q_i^{old}$.`Lookup`$() - Q_i^{new}$.`Lookup`(), where Q_i^{old} and Q_i^{new} denote polynomial Q_i *before* and *after* performing the `Reset` operation, respectively.
- `Query`(x, y): return $Q_{\log n}$.`Lookup`$()[x, y]$.

We note that both `Insert` and `Delete` operations propagate changes of Q_{i-1} to Q_i for any $i = 1$ to $\log n$. In case of `Insert`, any new 1's that appear in Q_{i-1} are inserted in the data structure for polynomial Q_i via `LazySet`, but only the changes of the v-th row and the v-th row column of Q_{i-1} are taken into account by `SetRow` and `SetCol` in order to determine changes of Q_i. This is conceptually similar to rebuilding only the BFS trees rooted at v in the algorithm presented in Section 4.1.

Analysis. To prove the correctness of the algorithm, we need to prove that, if there is a path from x to y in G of length $\leq 2^i$, then $Q_i[x, y] = 1$. Conversely, it is easy to see that $Q_i[x, y] = 1$ only if there is a path from x to y in G of length $\leq 2^i$. We first consider `Insert` operations. It is important to observe that, by the problem's definition, ΔX is zero outside the v-th row and the v-th column for $i = 0$, but this might not be the case for $i > 0$. However, as we said before we follow a lazy approach, and only the changes of the v-th row and on the v-th column of Q_{i-1} are taken into account by `SetRow` and `SetCol` in order to determine changes of Q_i. Other 1's are logged into the data structure via `LazySet`. Similarly to the algorithm presented in Section 4.1, the correctness of this follows from the simple observation that any new paths that appear due to a v-centered insertion pass through v. We proceed by induction. We assume that for any new paths $x \rightsquigarrow v$ and $v \rightsquigarrow y$ of length $\leq 2^{i-1}$, $Q_{i-1}[x, v] = 1$ and $Q_{i-1}[v, y] = 1$ at the beginning of loop iteration i. If both the portions

$x \rightsquigarrow i$ and $i \rightsquigarrow y$ of π have length up to 2^{i-1}, then π gets recorded in Q^2_{i-1}, and therefore in Q_i, thanks to one of $\texttt{Q}_i\texttt{.SetRow}(v, \Delta X, Q_{i-1})$ or $\texttt{Q}_i\texttt{.SetCol}(v, \Delta X, Q_{i-1})$. On the other hand, if v is close to (but does not coincide with) one endpoint of π, the appearance of π may be recorded in Q^3_{i-1}, but not in Q^2_{i-1}. This is the reason why degree 2 does not suffice for Q_i in this dynamic setting. To complete our discussion of correctness, we note that deletions act as "undo" operations that leave the data structures as if deleted edges were never inserted.

The running time is the same as the algorithm presented in Section 4.1. Observing that polynomials have constant degree, the proof follows easily from Theorem 11.

Theorem 15 *Any* `Insert` *operation requires* $O(n^2 \log n)$ *worst-case time, the cost of* `Delete` *operations can be charged to previous insertions, and* `Query` *operations are answered in constant time.*

The previous theorem implies that updates are supported in $O(n^2 \log n)$ amortized time per operation.

4.3 Matrix-based Dynamic Transitive Closure with Recursive Decomposition

In this section we discuss a second and more powerful method for casting fully dynamic transitive closure into the problem of reevaluating polynomials over Boolean matrices presented in Section 3.2.

This method hinges upon the well-known equivalence between transitive closure and matrix multiplication on a closed semiring discussed in Section 2.3 and yields a new deterministic algorithm that improves the best known bounds for fully dynamic transitive closure. The algorithm supports any update operation in $O(n^2)$ amortized time and answers any reachability query with just one matrix lookup. The space used is $O(n^2)$.

Data Structure. Let X be a Boolean matrix and let X^* be its Kleene closure. In Section 2.3 we have defined two equivalent functions $\mathcal{F}$ and $\mathcal{G}$ with the property that:

$$\mathcal{F}(X) = \mathcal{G}(X) = X^*$$

We now define another function $\mathcal{H}$ such that $\mathcal{H}(X) = X^*$, obtained by combining the definitions of $\mathcal{F}$ and $\mathcal{G}$, and designed to be well-suited for efficient reevaluation in a fully dynamic setting.

Lemma 2 *Let $\mathcal{B}_n$ be the set of $n \times n$ Boolean matrices, let $X \in \mathcal{B}_n$ and let $\mathcal{H} : \mathcal{B}_n \to \mathcal{B}_n$ be the mapping defined by means of the following equations:*

$$\begin{cases} P = D^* & & \\ E_1 = (A + BP^2C)^* & E_2 = E_1BH_2^2CE_1 & E = E_1 + E_2 \\ F_1 = E_1^2BP & F_2 = E_1BH_2^2 & F = F_1 + F_2 \\ G_1 = PCE_1^2 & G_2 = H_2^2CE_1 & G = G_1 + G_2 \\ H_1 = PCE_1^2BP & H_2 = (D + CE_1^2B)^* & H = H_1 + H_2 \end{cases} \tag{3}$$

where X *and* $Y = \mathcal{H}(X)$ *are defined as:*

$$X = \begin{array}{|c|c|}\hline A & B \\ \hline C & D \\ \hline \end{array} \qquad Y = \begin{array}{|c|c|}\hline E & F \\ \hline G & H \\ \hline \end{array}$$

Then, for any $X \in \mathcal{B}_n$, $\mathcal{H}(X) = X^*$.

Observe that $\mathcal{H}$ provides a method for computing the Kleene closure of an $n \times n$ Boolean matrix, provided that we are able to compute Kleene closures of Boolean matrices of size $\frac{n}{2} \times \frac{n}{2}$. The reason of using E_1^2, H_2^2 and P^2 instead of E_1, H_2 and P in Equation 3, will be clarified in the following. The aim of this section is to study how to reevaluate efficiently $\mathcal{H}(X) = X^*$ under changes of X. Before describing a data structure for maintaining X^* based on $\mathcal{H}$, we show that a Divide et Conquer algorithm which recursively uses $\mathcal{H}$ to solve sub-problems of smaller size requires asymptotically the same time of computing the product of two Boolean matrices (see [6, 10]).

Theorem 16 *Let* X *be an* $n \times n$ *Boolean matrix and let* $T(n)$ *be the time required to compute recursively* $\mathcal{H}(X)$. *Then* $T(n) = O(n^\omega)$, *where* $O(n^\omega)$ *is the time required to multiply two Boolean matrices.*

The data structure for maintaining the Kleene closure X^* is the following: we maintain two $n \times n$ Boolean matrices X and Y decomposed in sub-matrices A, B, C, D, and E, F, G, H:

$$X = \begin{array}{|c|c|}\hline A & B \\ \hline C & D \\ \hline \end{array} \qquad Y = \begin{array}{|c|c|}\hline E & F \\ \hline G & H \\ \hline \end{array}$$

We also maintain the following polynomials over $n \times n$ Boolean matrices with the data structure presented in Section 3.2:

$$\begin{array}{lll} Q = A + BP^2C & E_2 = E_1BH_2^2CE_1 & E = E_1 + E_2 \\ F_1 = E_1^2BP & F_2 = E_1BH_2^2 & F = F_1 + F_2 \\ G_1 = PCE_1^2 & G_2 = H_2^2CE_1 & G = G_1 + G_2 \\ H_1 = PCE_1^2BP & R = D + CE_1^2B & H = H_1 + H_2 \end{array}$$

and we recursively maintain Kleene closures P, E_1 and H_2:

$$P = D^* \quad E_1 = Q^* \quad H_2 = R^*$$

with smaller instances of size $\frac{n}{2} \times \frac{n}{2}$ of the data structure.

Notice the recursive definition: P, E_1 and H_2 are Kleene closures of $\frac{n}{2} \times \frac{n}{2}$ matrices. Also observe that the polynomials Q, F_1, G_1, H_1, E_2, F_2, G_2, R, E, F, G and H that we maintain have all constant degree ≤ 6. It is easy to see that the sequence $\tau = \langle P,\ Q,\ E_1,\ R,\ H_2,\ F_1,\ G_1,\ H_1,\ E_2,\ F_2,\ G_2,\ E,\ F,\ G,\ H\rangle$ yields a correct evaluation order for any intermediate values that arise in computing $Y = \mathcal{H}(X)$. We remark that the data structure stores all these intermediate values and maintains such values upon updates of X.

Since the data structure reflects the way $Y = \mathcal{H}(X)$ is computed, it basically represents X^* as the sum of two Boolean matrices: the first, say X_1^*, is defined by submatrices E_1, F_1, G_1, H_1, and the second, say X_2^*, by submatrices E_2, F_2, G_2, H_2:

$$X_1^* = \begin{array}{|c|c|}\hline E_1 & F_1 \\ \hline G_1 & H_1 \\ \hline \end{array} \qquad X_2^* = \begin{array}{|c|c|}\hline E_2 & F_2 \\ \hline G_2 & H_2 \\ \hline \end{array}$$

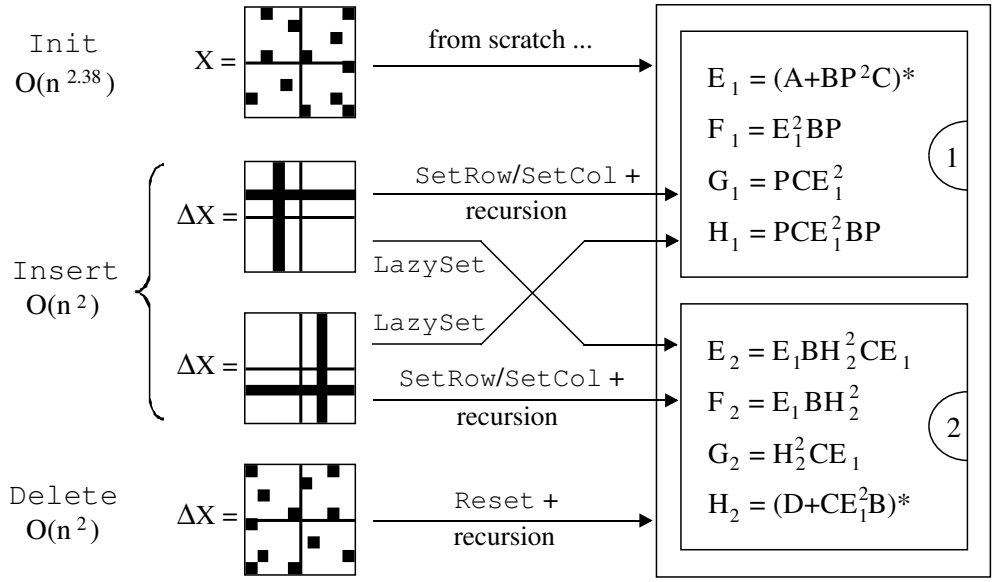

Figure 3: Overview of operations `Insert` and `Delete`.

Implementation of Operations. From a high-level point of view, the approach is the following. Maintain X_1^* and X_2^* *in tandem* (see Figure 3): whenever an `Insert` operation is performed on X, update X^* by computing how either X_1^* or X_2^* are affected by this change. Updates are lazily performed so that neither X_1^* nor X_2^* encode complete information about X^*, but their sum does. On the other side, `Delete` operations update both X_1^* and X_2^* and leave the data structure as if any reset entry were never set to 1. As in the algorithm presented in Section 4.2, update operations are realized via `SetRow`, `SetCol`, `LazySet`, and `Reset` on the polynomials in the data structure. We now briefly sketch how the operations are realized.

- `Insert`(v, I): set $X[x, y]$ to 1 for each $(x, y) \in I$ incident to v, and then consider the two cases:

  1. $1 \leq v < \frac{n}{2}$: the v-centered update of X may affect the v-th row and the v-th column of A, the v-th row of B and the v-th column of C, while D is not affected at all by this kind of update (see Figure 3). Do the next steps in the order specified by sequence τ.

     Fully update X_1^* by calling `SetRow`/`SetCol` on Q, F_1, G_1, H_1, R, and recurring on E_1. Lazily update X_2^* by calling `LazySet` on G_2, F_2, E_2, and recurring on H_2.
  2. $1 \leq v < n$: the v-centered update of X may affect only the v-th row and the v-th column of D, the v-th row of C and the v-th column of B, while A is not affected at all by this kind of update. Do the next steps in the order specified by sequence τ.

     Fully update X_2^* by calling `SetRow`/`SetCol` on R, G_2, F_2, E_2, and recurring on P, H_2. Lazily update X_1^* by calling `LazySet` on F_1, G_1, H_1, and recurring on E_1.

  In both cases, recompute polynomials E, F, G, H to update Y.

- `Delete`(D): clear entries $X[x, y]$ for each $(x, y) \in D$, and propagate changes to $Y = X^*$ by performing `Reset` operations on each polynomial in the data structure in the order specified by sequence τ. Data structures P, E_1 and H_2 of are treated recursively.

- `Query`(x, y): return $Y[x, y]$.

Analysis. For more implementation details and for a detailed analysis of the algorithm we refer the interested reader to [6, 10]. Below we report the running time of operations.

Theorem 17 *Any* `Insert` *operation requires* $O(n^2)$ *worst-case time, the cost of* `Delete` *operations can be charged to previous insertions, and* `Query` *operations are answered in constant time.*

The previous theorem implies that updates are supported in $O(n^2)$ amortized time per operation. Based on dynamic determinant computation, Sankowski [47] has shown how to make the amortized bounds worst-case.

5 Dynamic Shortest Paths

In this section we survey the best known algorithms for fully dynamic shortest paths. Those algorithms can be seen as a natural evolution of the techniques described so far for dynamic transitive closure. They are not a trivial extension of transitive closure algorithms, however, as dynamic shortest path problems look more complicated in nature. As an example, consider the deletion of an edge (u, v). In the case of transitive closure, reachability between a pair of vertices x and y can be re-established by *any* replacement path avoiding edge (u, v). In case of shortest paths, after deleting (u, v), we have to look for the *best* replacement path avoiding edge (u, v).

We start with a formal definition of the fully dynamic all pairs shortest paths problem. Next, we survey the algorithm by King [33], whose main ingredients are the long paths property of Section 2.4 and the tools for trees described in Section 3.1. This methods yields $O(n^{2.5}\sqrt{C \cdot \log n})$ amortized time per update and $O(1)$ per query in graphs with positive integer weights less than C. Finally, we describe the algorithm by Demetrescu and Italiano [11], which is based on the long paths property of Section 2.4 and the tools for matrices described in Section 3.2. Using this method, updates are supported in $O(S \cdot n^{2.5} \log^3 n)$ amortized time and reachability queries are answered in $O(1)$ in graphs where each edge can assume at most S different real values.

The Problem. Let $G = (V, E)$ be a weighted directed graph. We consider the problem of maintaining a data structure for G under an intermixed sequence of update and query operations of the following kinds:

- `Decrease`(v, w): decrease the weight of edges incident to v in G as specified by a new weight function w. We call this kind of update a v-CENTERED decrease in G.

- `Increase`(w): increase the weight of edges in G as specified by a new weight function w.

- `Query`(x, y): return the distance between x and y in G.

As in fully dynamic transitive closure, we consider generalized update operations where we modify a whole set of edges, rather than a single edge (see Figure 2). Again, we do not address the issue of returning actual paths between vertices, and we just consider the problem of answering distance queries.

5.1 Tree-based Dynamic Shortest Paths with Stitching

In this section we describe how to maintain all pairs shortest paths in a directed graph with non-negative integer edge weights less than C in $O(n^{2.5}\sqrt{C \log n})$ amortized time per update operation. The algorithm that we describe has been designed by King [33] and it builds on the tree data structure presented in Section 3.1 and on the long paths property described in Section 2.4.

Data Structure. Given a weighted directed graph G, we maintain for each vertex v:

- a shortest paths tree Out_v of G of depth $d \leq \sqrt{nC \log n}$ rooted at v;
- a shortest paths tree In_v of $\widehat{G}$ of depth $d \leq \sqrt{nC \log n}$ rooted at v, where $\widehat{G}$ is equal to G, except for the orientation of edges, which is reversed;
- a set S containing $\sqrt{nC \log n}$ vertices of G chosen uniformly at random, referred to as "blockers";
- a complete weighted directed graph G_S with vertex set S such that, with very high probability, the weight of (x, y) in G_S is equal to the distance between x and y in G. For the long paths property given in Section 2.4, these weights are no greater than d.
- an integer distance matrix $dist$.

We maintain the trees with instances of the data structure presented in Section 3.1, adapted to deal with weighted directed graphs and to include only short paths, i.e., vertices of distance up to d from the root. Information about longer paths will be obtained by stitching together these short paths.

Implementation of Operations. The main idea of the algorithm is to exploit the long paths property of Section 2.4, maintaining dynamically only shortest paths of (weighted) length up to $\sqrt{C \log n}$ with the data structure presented in Section 3.1, and stitching together these paths to update longer paths using any static $O(n^3)$ all pairs shortest paths algorithm on a contraction with $O(\sqrt{nC \log n})$ vertices of the original graph. Operations are realized as follows:

- `Decrease`(v, w): rebuild In_v and Out_v to update G_S, i.e., paths of length up to d. Apply the algorithm `Stitch` below to update longer paths.
- `Increase`(w): update edges with increased weight in any In_v and Out_v that contain them, and then update G_S, i.e., paths of length up to d. Apply the algorithm `Stitch` below to update longer paths.
- `Query`(x, y): return $dist(x, y)$.

Details about the stitching algorithm are given below:

- `Stitch`():

  1. Let $dist^{(d)}(x, y)$ be the distance from x to y of length up to d, obtained from all the trees In_v and Out_v.
  2. Compute the distances $dist()$ between all vertices in S using any static $O(n^3)$ APSP algorithm on G_S.
  3. Compute the distances from vertices in V to vertices in S. This can be done for a pair $x \in V$ and $s \in S$ by computing
  $$dist(x, s) \leftarrow \min\{dist^{(d)}(x, s), \min_{s' \in S}\{dist^{(d)}(x, s') + dist(s', s)\}.$$
  4. Compute the distances between vertices in V. This can be done for a pair $x, y \in V$ by computing
  $$dist(x, y) \leftarrow \min\{dist^{(d)}(x, y), \min_{s \in S}\{dist(x, s) + dist^{(d)}(s, y)\}.$$

Analysis. The stitching algorithm is dominated by the last step, which takes time $O(n^2|S|) = O(n^2(n \log n/d))$. Shortest path trees of length up to d can be maintained in $O(n^2 d)$ amortized time with the data structure of Section 3.1. Choosing $d = \Theta(\sqrt{nC \log n})$ yields an amortized update bound of $O(n^{2.5}\sqrt{C \log n})$.

Theorem 18 *Any* `Decrease` *and* `Increase` *operation requires* $O(n^{2.5}\sqrt{C \log n})$ *amortized time, and* `Query` *can be answered in constant time.*

The set S can be computed deterministically, as illustrated in [33]. This makes the whole algorithm deterministic with the same bounds.

5.2 Matrix-based Dynamic Shortest Paths with Logarithmic Decomposition

In this section we describe how to maintain all pairs shortest paths in a directed graph where each edge can assume at most S different real weights in $O(Sn^{2.5} \log^3 n)$ amortized time per update operation. The algorithm that we describe has been designed by Demetrescu and Italiano [11] and is based on the matrix data structure presented in Section 3.2 and on the long paths property described in Section 2.4.

Data Structure. We denote by $Q_i[x, y]$ the length of a shortest path $x \rightsquigarrow y$ that uses at most 2^i edges, i.e., $Q_i = \bigoplus_{0 \le j \le 2^i} X^j$, where X is the weight matrix of G. Thus, $X^* = Q_{\log n}$ if there are no negative-length cycles. We maintain $\log n$ levels: level i keeps track of paths with at most 2^i edges. For each level we maintain a polynomial P_i defined on three variables L, R, C, which are updated via `LazySet`, `SetRow` and `SetCol`, respectively. The value of polynomial P_{i-1} is used to update variables of polynomial P_i. More in detail, we maintain:

- $\log n$ polynomials $P_1, \ldots, P_{\log n}$ defined as $P_i(L, R, C) = L \oplus CRL \oplus LCR$ represented with $\log n$ instances of the data structure presented in Section 3.2, adapted to work on the $\{\min, +\}$ semiring; we maintain each P_i so that, with very high probability, $\mathtt{P}_i$.`Lookup`()$[x, y]$ is less or equal than the length of a shortest path $x \rightsquigarrow y$ that uses at most 2^i edges, i.e., $\mathtt{P}_i$.`Lookup`() $\le Q_i$;

- a counter $Time$ of the number of performed `Decrease` operations;
- an array $SetTime$ such that $SetTime[u] = t$ if vertex u has been last considered in a `Decrease` operation at time t.

Implementation of Operations. We say that a vertex u is *out of fashion* at level i if $Time - SetTime[u] > (cn \log n)/2^{i-1}$, and we say that it is *in fashion* otherwise. When a vertex v goes out of fashion, we cast v off by raising to $+\infty$ the corresponding column of C and the corresponding row of R. Actually, only rows and columns of R and C corresponding to vertices that are in fashion carry information about shortest paths, whereas rows and columns corresponding to vertices that went out of fashion do not participate in this task. The correctness of this approach hinges on the long path property of Section 2.4, i.e., given any two vertices x and y, we can find with very high probability a vertex z that is in fashion on some level such that the shortest distance from x to y is given either by $C[x, z] + RL[z, y]$ or by $LC[x, z] + R[z, y]$ on that level. The reason for the terms CRL and LCR of degree 3 in polynomials P_i is because z might be more than 2^{i-1} edges far away from either endpoint of the shortest path $x \rightsquigarrow y$. We note that vertices that are in fashion play exactly the same role as the set S of blockers in Section 5.1. Operations are as follows.

- `Decrease`(v, w): first pick at random a vertex u and let $\Delta[x, y] = w(x, y)$. Then, for each $i = 1$ to $\log n$, do the following: perform P_i.`LazySet`(Δ, L), P_i.`SetRow`(u, Δ, R), P_i.`SetRow`(v, Δ, R), P_i.`SetCol`(u, Δ, C), P_i.`SetCol`(v, Δ, C), and $\Delta \leftarrow P_i$.`Lookup`$()$. At each level, cast off vertices which are no longer in fashion via `Reset` operations on variables R and C.
- `Increase`(w): we let $\Delta[x, y] = w(x, y)$ $i = 1$ to $\log n$, do the following: perform P_i.`Reset`(D, P_{i-1}) and $D \leftarrow P_i$.`Lookup`$()$.
- `Query`: return $P_{\log n}$.`Lookup`$()[x, y]$.

Analysis. `Decrease`(v, w) brings into fashion at most two vertices u and v. The first vertex is v, the center of the update: it is brought into fashion because it may contain the latest information about shortest paths that go through v. The second vertex u is chosen at random, so as to ensure that vertices that are in fashion are well distributed throughout the graph, as shown in the following lemma.

Lemma 3 *At any time during a sequence of updates, on any path that uses $\geq 2^{i-1}$ edges there is at least one vertex that is in fashion at level i with probability $p \geq 1 - \frac{1}{n^c}$, for any constant $c > 0$.*

We prove that, with very high probability, $\mathtt{P}_i$.`Lookup`$()[x, y]$ is less than or equal to the length of a shortest path $x \rightsquigarrow y$ that uses at most 2^i edges.

Theorem 19 *At any time t during a sequence of updates, $X^* \leq \mathtt{P}_i$.`Lookup`$() \leq Q_i$ with probability $p \geq 1 - \frac{1}{n^c}$, for any constant $c > 0$.*

Proof. We proceed by induction on the number of levels. The base is trivial. Assume by induction that the claim holds for $\mathtt{P}_{i-1}$. Let $\pi_{x,y}$ be any path $x \rightsquigarrow y$ of length $\ell(\pi_{x,y})$ that uses q edges, $2^{i-1} < q \leq 2^i$. By Lemma 3, with probability $p \geq 1 - \frac{1}{n^c}$ there is at least one vertex on

$\pi_{x,y}$ that is in fashion at level i. Let z be the most recent among them, with $t_z = SetTime[z]$. Then, at time t_z, operations $\mathtt{P}_i.\mathtt{SetRow}(z, Y, R)$, $\mathtt{P}_i.\mathtt{SetCol}(z, Y, C)$ and $\mathtt{P}_i.\mathtt{LazySet}(Y, L)$ synchronized L, column z of C, and row z of R with Y, where $Y = \mathtt{P}_{k-1}.\mathtt{Lookup}()$ at that time. By the inductive hypothesis, this implies that, at time t_z, $C[x, z] \leq Q_{i-1}[x, z]$, $LC[x, z] \leq Q^2_{i-1}[x, z]$, $R[z, y] \leq Q_{i-1}[z, y]$, and $RL[z, y] \leq Q^2_{i-1}[z, y]$. By the definition of `Lookup` given in Section 3.2 and by noting that $\ell(\pi_{x,y}) \geq \min\{Q_{i-1}[x, z] + Q^2_{k-1}[z, y],\ Q^2_{i-1}[x, z] + Q_{i-1}[z, y]\}$, we have that at time t_z

$$\mathtt{P}_i.\mathtt{Lookup}()[x, y] \leq \min\{C[x, z] + RL[z, y],\ LC[x, z] + R[z, y]\} \leq$$
$$\min\{Q_{i-1}[x, z] + Q^2_{i-1}[z, y],\ Q^2_{i-1}[x, z] + Q_{i-1}[z, y]\} \leq \ell(\pi_{x,y}).$$

Since a vertex z is brought into fashion not only if it is chosen randomly, but also if a z-centered operation that decreases the weights of edges incident to z is performed, and since z is the most recent vertex on $\pi_{x,y}$, $\ell(\pi_{x,y})$ did not decrease since time t_z. As edge increases are handled via $\mathtt{P}_i.\mathtt{Reset}$ operations on P_i which "undo" the effects of previous set operations, this implies that the above inequalities were valid at time t_z and are still valid at time t. Moreover, since $\mathtt{P}_i.\mathtt{LazySet}(Y, L)$ always updates P_i by considering the contribution of the term L of degree 1, it follows that $\mathtt{P}_i.\mathtt{Lookup}() \leq \mathtt{P}_{i-1}.\mathtt{Lookup}() \leq Q_{i-1}$. Thus, if $\pi^*_{x,y}$ is a path that uses at most 2^i edges and $\ell(\pi^*_{x,y}) = Q_i[x, y]$ at time t, it holds that $\mathtt{P}_i.\mathtt{Lookup}()[x, y] \leq \ell(\pi^*_{x,y}) = Q_i[x, y]$. □

As $Q_{\log n} = X^*$, Theorem 19 implies that $\mathtt{P}_{\log n}.\mathtt{Lookup}() = X^*$ with very high probability, which proves the correctness of the algorithm. To conclude this section, we discuss the running time of the algorithm.

Lemma 4 *At any time during a sequence of updates, there are at most $\frac{cn \log n}{2^{i-1}}$ vertices that are in fashion at level i for any constant $c > 0$.*

Proof. The proof easily follows from observing that a vertex u goes out of fashion after $\frac{cn \log n}{2^{i-1}}$ `Decrease` operations that do not update $SetTime[u]$. □

If we assume that edge weights can assume at most S different real values, then by a simple scaling argument variables of polynomial P_i at level i can assume at most $S \cdot 2^i$ different real values. By Theorem 12, maintaining polynomial P_i at level i would require $O(S \cdot 2^i \cdot n^2)$ amortized time per operation. However, by Lemma 4 at any time all but at most $\frac{cn \log n}{2^{i-1}}$ rows of R and columns of C are set to $+\infty$. As shown in [11], this yields the following bounds.

Theorem 20 *In any intermixed sequence of operations* `Decrease` *and* `Increase` *on a graph whose edges can assume at most S different real values, each update costs $O(S \cdot n^{2.5} \log^3 n)$ amortized time.*

As in the case of the algorithm described in Section 5.1, the bounds of Theorem 20 can be made deterministic.

5.3 Locality-based Dynamic Shortest Paths

In this section we address the algorithm by Demetrescu and Italiano [7], who devised the first deterministic near-quadratic update algorithm for fully dynamic all-pairs shortest paths. This algorithm is also the first solution to the problem in its generality. The algorithm is

based on the notions of historical paths and locally historical paths in a graph subject to a sequence of updates, as discussed in Section 2.5.

The main idea is to maintain dynamically the locally historical paths of the graph in a data structure. Since by Lemma 1 shortest paths are locally historical, this guarantees that information about shortest paths is maintained as well.

To support an edge weight update operation, the algorithm implements the smoothing strategy mentioned in Section 2.5 and works in two phases. It first removes from the data structure all maintained paths that contain the updated edge: this is correct since historical shortest paths, in view of their definition, are immediately invalidated as soon as they are touched by an update. This means that also potentially uniform paths that contain them are invalidated and have to be removed from the data structure. As a second phase, the algorithm runs an all-pairs modification of Dijkstra's algorithm [12], where at each step a shortest path with minimum weight is extracted from a priority queue and it is combined with existing historical shortest paths to form new potentially uniform paths. At the end of this phase, paths that become potentially uniform after the update are correctly inserted in the data structure.

The update algorithm spends constant time for each of the $O(zn^2)$ new potentially uniform path (see Theorem 9). Since the smoothing strategy lets $z = O(\log n)$ and increases the length of the sequence of updates by an additional $O(\log n)$ factor, this yields $O(n^2 \log^3 n)$ amortized time per update. The interested reader can find further details about the algorithm in [7].

Using the same approach, but with a different smoothing strategy, Thorup [48] has shown how to achieve $O(n^2(\log n + \log^2(m/n)))$ amortized time per update and $O(mn)$ space. His algorithm works with negative weights as well.

6 Conclusions and Open Problems

In this paper we have surveyed the newest developments in the area of fully dynamic algorithms for directed graphs. In particular, we have focused on dynamic transitive closure and dynamic shortest paths. Throughout the paper, we have attempted to present all the algorithmic techniques within a unifying framework by abstracting the algebraic and combinatorial properties and the data-structural tools that lie at their foundations.

This bulk of recent work has raised some new and perhaps intriguing questions. First, can we reduce the space usage for dynamic shortest paths to $O(n^2)$? Second, and perhaps more importantly, can we solve efficiently fully dynamic *single-source* reachability and shortest paths on general graphs? Finally, are there any general techniques for making increase-only algorithms fully dynamic? Similar techniques have been widely exploited in the case of fully dynamic algorithms on undirected graphs [24, 25, 27].

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison Wesley, 1974.

[2] G. Ausiello, G.F. Italiano, A. Marchetti-Spaccamela, and U. Nanni. Incremental algorithms for minimal length paths. *Journal of Algorithms*, 12(4):615–38, 1991.

[3] V. Chvátal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235, 1979.

[4] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9:251–280, 1990.

[5] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[6] C. Demetrescu. *Fully Dynamic Algorithms for Path Problems on Directed Graphs*. PhD thesis, Department of Computer and Systems Science, University of Rome "La Sapienza", February 2001.

[7] C. Demetrescu and G. F. Italiano. A new approach to dynamic all pairs shortest paths. *Journal of the Association for Computing Machinery (JACM)*, 51(6):968–992, 2004.

[8] C. Demetrescu and G. F. Italiano. Experimental analysis of dynamic all pairs shortest path algorithms. *To appear in ACM Transactions on Algorithms*, 2005. Special issue devoted to the best papers selected from the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'04).

[9] C. Demetrescu and G. F. Italiano. Trade-offs for fully dynamic reachability on dags: Breaking through the $O(n^2)$ barrier. *Journal of the Association for Computing Machinery (JACM)*, 52(2):147–156, 2005.

[10] C. Demetrescu and G.F. Italiano. Fully dynamic transitive closure: Breaking through the $O(n^2)$ barrier. In *Proc. of the 41st IEEE Annual Symposium on Foundations of Computer Science (FOCS'00)*, pages 381–389, 2000. Full paper available at the URL: `http://arXiv.org/abs/cs.DS/0104001`.

[11] C. Demetrescu and G.F. Italiano. Fully dynamic all pairs shortest paths with real edge weights. *To appear in Journal of Computer and System Sciences*, 2005.

[12] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[13] S. Even and H. Gazit. Updating distances in dynamic graphs. *Methods of Operations Research*, 49:371–387, 1985.

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

[15] J. Fakcharoemphol and S. Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. In *Proc. of the 42nd IEEE Annual Symposium on Foundations of Computer Science (FOCS'01), Las Vegas, Nevada*, pages 232–241, 2001.

[16] M. J. Fischer and A. R. Meyer. Boolean matrix multiplication and transitive closure. In *Conference Record 1971 Twelfth Annual Symposium on Switching and Automata Theory*, pages 129–131, East Lansing, Michigan, 13–15 October 1971. IEEE.

[17] G.N. Frederickson. Data structures for on-line updating of minimum spanning trees. *SIAM J. Comput.*, 14:781–798, 1985.

[18] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Semi-dynamic algorithms for maintaining single source shortest paths trees. *Algorithmica*, 22(3):250–274, 1998.

[19] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic algorithms for maintaining shortest paths trees. *Journal of Algorithms*, 34:351–381, 2000.

[20] D. Frigioni, T. Miller, U. Nanni, and C. D. Zaroliagis. An experimental study of dynamic algorithms for transitive closure. *ACM Journal of Experimental Algorithms*, 6(9), 2001.

[21] M.E. Furman. Application of a method of fast multiplication of matrices in the problem of finding the transitive closure of a graph. *Soviet Math. Dokl.*, 11(5), 1970. English translation.

[22] D. H. Greene and D.E. Knuth. *Mathematics for the analysis of algorithms.* Birkhäuser, 1982.

[23] M. Henzinger and V. King. Fully dynamic biconnectivity and transitive closure. In *Proc. 36th IEEE Symposium on Foundations of Computer Science (FOCS'95)*, pages 664–672, 1995.

[24] M. Henzinger and V. King. Maintaining minimum spanning forests in dynamic graphs. *SIAM Journal on Computing*, 31(2):364–374, 2001.

[25] M.R. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM*, 46(4):502–516, 1999.

[26] M.R. Henzinger, P. Klein, S. Rao, and S. Subramanian. Faster shortest-path algorithms for planar graphs. *Journal of Computer and System Sciences*, 55(1):3–23, August 1997.

[27] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM*, 48:723–760, 2001.

[28] X. Huang and V.Y. Pan. Fast rectangular matrix multiplication and applications. *Journal of Complexity*, 14(2):257–299, June 1998.

[29] T Ibaraki and N. Katoh. On-line computation of transitive closure for graphs. *Information Processing Letters*, 16:95–97, 1983.

[30] G.F. Italiano. Amortized efficiency of a path retrieval data structure. *Theoretical Computer Science*, 48(2–3):273–281, 1986.

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

[32] S. Khanna, R. Motwani, and R. H. Wilson. On certificates and lookahead in dynamic graph problems. *Algorithmica*, 21(4):377–394, 1998.

[33] V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Proc. 40th IEEE Symposium on Foundations of Computer Science (FOCS'99)*, pages 81–99, 1999.

[34] 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.

[35] 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 Computing and Combinatorics Conference (COCOON), LNCS 2108*, pages 268–277, 2001.

[36] J. A. La Poutré and J. van Leeuwen. Maintenance of transitive closure and transitive reduction of graphs. In *Proc. Workshop on Graph-Theoretic Concepts in Computer Science*, pages 106–120. Lecture Notes in Computer Science 314, Springer-Verlag, Berlin, 1988.

[37] P. Loubal. A network evaluation procedure. *Highway Research Record 205*, pages 96–109, 1967.

[38] L. Lovász. On the ratio of optimal integral and fractional covers. *Discrete Mathematics*, 13:383–390, 1975.

[39] I. Munro. Efficient determination of the transitive closure of a directed graph. *Information Processing Letters*, 1(2):56–58, 1971.

[40] J. Murchland. The effect of increasing or decreasing the length of a single arc on all shortest distances in a graph. Technical report, LBS-TNT-26, London Business School, Transport Network Theory Unit, London, UK, 1967.

[41] G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest path problem. *Journal of Algorithms*, 21:267–305, 1996.

[42] G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *Theoretical Computer Science*, 158:233–277, 1996.

[43] V. Rodionov. The parametric problem of shortest distances. *U.S.S.R. Computational Math. and Math. Phys.*, 8(5):336–343, 1968.

[44] L. Roditty and U. Zwick. Improved dynamic reachability algorithms for directed graphs. In *Proceedings of 43th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 679–688, 2002.

[45] L. Roditty and U. Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC)*, pages 184–191, 2004.

[46] H. Rohnert. A dynamization of the all-pairs least cost problem. In *Proc. 2nd Annual Symposium on Theoretical Aspects of Computer Science, (STACS'85), LNCS 182*, pages 279–286, 1985.

[47] P. Sankowski. Dynamic transitive closure via dynamic matrix inverse. In *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science (FOCS'04), Rome, Italy*, 2004.

[48] M. Thorup. Fully-dynamic all-pairs shortest paths: Faster and allowing negative cycles. In *Proceedings of the 9th Scandinavian Workshop on Algorithm Theory (SWAT'04)*, pages 384–396, 2004.

[49] M. Thorup. Worst-case update times for fully-dynamic all-pairs shortest paths. In *Proceedings of the 37th ACM Symposium on Theory of Computing (STOC 2005)*, 2005.

[50] J.D. Ullman and M. Yannakakis. High-probability parallel transitive-closure algorithms. *SIAM Journal on Computing*, 20(1):100–125, 1991.

[51] D. M. Yellin. Speeding up dynamic transitive closure for bounded degree graphs. *Acta Informatica*, 30:369–384, 1993.

[52] U. Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM*, 49(3):289–317, 2002.