Alcom-FT Technical Report Series
ALCOMFT-TR-01-147

Università degli Studi di Roma "La Sapienza"

Dottorato di Ricerca in Ingegneria Informatica

XIII Ciclo – 2001– XIII-01-1

Fully Dynamic Algorithms for Path Problems on Directed Graphs

Camil Demetrescu

Università degli Studi di Roma "La Sapienza"

Dottorato di Ricerca in Ingegneria Informatica

XIII Ciclo - 2001– XIII-01-1

Camil Demetrescu

Fully Dynamic Algorithms for Path Problems on Directed Graphs

Thesis Committee		Reviewers
Prof. Umberto Nanni	(Advisor)	Prof. Valerie King
Prof. Giorgio Ausiello		Prof. Christos Zaroliagis
Prof. Rossella Petreschi		

Author's address:

Camil Demetrescu
Dipartimento di Informatica e Sistemistica
Università degli Studi di Roma "La Sapienza"
Via Salaria 113, I-00198 Roma, Italy
e-mail: `demetres@dis.uniroma1.it`
www: `http://www.dis.uniroma1.it/~demetres`

Abstract

In this thesis we investigate fully dynamic algorithms for path problems on directed graphs. In particular, we focus on two of the most fundamental path problems: fully dynamic transitive closure and fully dynamic single-source shortest paths.

The first part of the thesis presents a new technique which makes it possible to reduce fully dynamic transitive closure to the problem of reevaluating polynomials over matrices when updates of variables are performed. Based on this technique, we devise a new deterministic algorithm which improves the best known bounds for fully dynamic transitive closure. Our algorithm hinges upon the well-known equivalence between transitive closure and matrix multiplication on a closed semiring. We show how to maintain explicitly the transitive closure of a directed graph as a Boolean matrix in $O(n^2)$ amortized time per insertion and deletion of edges. Since an update may change as many as $\Omega(n^2)$ entries of this matrix, this seems to be the best update bound that one could hope for this class of algorithms. We note that maintaining explicitly the transitive closure allows us to answer reachability queries with just one table lookup. We also consider the case where only deletions are allowed and we show how to handle updates faster in $O(n)$ amortized time per operation while maintaining unit lookup per query; in this way we generalize to directed graphs the previous best known deletions-only result for acyclic graphs. Using the same matrix based approach, we also address the problem of maintaining implicitly the transitive closure of a directed graph and we devise the first algorithm which supports both updates and reachability queries in subquadratic time per operation. This result proves that it is actually possible to break through the $O(n^2)$ barrier on the single-operation complexity of fully dynamic transitive closure, and solves a problem that has been open for many years. Our subquadratic algorithm is randomized Monte Carlo and supports update in $O(n^{1.58})$ and query in $O(n^{0.58})$ worst-case time.

From an experimental point of view, we investigate the practical performances of fully dynamic single-source shortest paths algorithms on directed graphs with arbitrary edge weights. We also propose a variant of the best known algorithms especially designed to be simple and fast in practice while matching the same asymptotic worst-case running time. Our study provides the first experimental evidence of practical dynamic solutions for the problem that are better by several orders of magnitude than recomputing from scratch.

Acknowledgements

I wish to thank my advisor Umberto Nanni, who first sparked my interest in dynamic algorithms. His sense of humor has been of great help in overcoming many of the difficulties which arose throughout my doctoral program.

I still remember when I first knocked on Pino Italiano's office door at the University of Rome "Tor Vergata". He was friendly and kind, as usual. He wrote down a quick note on a small piece of paper about a cool open problem in dynamic transitive closure. This problem is now a main topic of this dissertation. Since then, he constantly helped me develop as a person and computer scientist. Actually, I regard meeting Pino as one of the most important events in my life.

I'm even more grateful to Giorgio Ausiello, who introduced me to Pino, and who was a tremendous source of support and encouragement throughout these years.

A special thank goes also to Alberto Marchetti-Spaccamela, for his constant friendly support and for always keeping his door open.

I thank all the people at the Department of Computer and Systems Science of the University of Rome "La Sapienza". In particular, I am grateful to Roberto Baldoni, Marco Cadoli and Marco Temperini for many useful discussions and to everyone in the Algorithm Engineering Group for providing a warm and friendly working environment.

I wish to thank my roommate Francesco Quaglia for being a true friend and an extraordinary source of advice, and for helping me realise the importance of a soun working method in doing research.

Everybody who knows Daniele Frigioni can figure out how funny and pleasant it has been to have him as a coauthor and friend. The results presented in this dissertation arose from joint works with Alberto, Daniele, Pino and Umberto [19, 20].

I feel fortunate to have had the opportunity for fruitful cooperation with the graph drawing people at the University of Roma Tre. In particular, I thank Giuseppe Di Battista, Walter Didimo, Giuseppe Liotta, Maurizio "Titto" Patrignani and Maurizio Pizzonia. I also owe a lot to Pierluigi Crescenzi and Rossella Petreschi for their valuable support and for encouraging me so many times. The results achieved with them appeared in [13, 14, 18, 22] and are not reported in this thesis.

I thank Giorgio for being Coordinator of the PhD program and for serving with Rossella and Umberto on my dissertation committee. A special mention goes to Valerie King and Christos Zaroliagis, my external reviewers, for their extremely thorough examination of my thesis and for their many constructive suggestions. In particular, I thank Valerie for her previous great work in dynamic transitive closure: without her results, I would have never come up with most of the original contributions in my thesis. I am also grateful to Mikkel Thorup for many enlightening discussions.

I am glad to mention the European IST Programme ALCOM-FT and the project "Algorithms for Large Data Sets: Science and Engineering" of the Italian Ministry for Scientific Research which allowed me to travel and to attend conferences.

I thank Radu Aldulescu and Albert Guttman for their invaluable performance of Cello Sonatas op. 5 nr. 1 in F major and op. 5 nr. 2 in G minor by Ludwig van Beethoven: their music has been a magical source of inspiration during the preparation of this dissertation.

It would be a long list to mention all the other friends I am indebted to. I gratefully thank all of them.

My parents Mihaela and Paul and my brother Emanuel deserve a warm and special acknowledgement for their love and care.

Irene is certainly the main contributor of all the best in my life, including this thesis. I thank God for the extraordinary chance of having her as wife, friend, companion, and even coauthor [15, 16, 17, 23].

Contents

Chapter 1

Introduction

In this thesis we present original contributions concerning two of the most fundamental dynamic problems on directed graphs: dynamic transitive closure and dynamic single-source shortest paths.

A *dynamic graph problem* consists of maintaining, either implicitly or explicitly, a given property in a graph subject to dynamic changes over time. Typical examples of graph properties considered in the literature include planarity, connectivity, minimum spanning tree, reachability, and shortest paths (see [40, 45, 52]). A repertoire of typical changes of the graph include insertion/deletion of nodes/edges as well as modifications of their weights.

To solve a dynamic graph problem we have to consider a *dynamic graph algorithm*, i.e., a data structure that maintains the graph under an intermixed sequence of operations of *update* of the graph and of *query* of the given property. This is opposed to classical *static* algorithms, where the graph is supposed not to change over time and the property is computed from scratch just once. The goal for an *efficient* dynamic graph algorithm is to support *both* update and query operations faster than recomputing from scratch the desired property with the best static algorithm.

We say that an algorithm is *fully dynamic* if it can handle both insertion and deletion. A dynamic graph algorithm which allows only for insertions is considered to be *incremental*, whereas a dynamic graph algorithm which allows only for deletions is called *decremental*. Incremental and decremental dynamic graph algorithms are said to be *partially dynamic*.

The design of efficient dynamic graph algorithms has attracted a lot of interest in the last ten years, both for their theoretical and practical interest. In particular, many important results have been obtained so far for undirected graphs (see, for example, [27, 30, 43, 45]). On the other hand, directed graph problems appear to be intrinsically harder, especially as far as the design of fully dynamic algorithms is concerned. In particular, the certificate complexity of reachability and of other directed graph problems has been proved to be $\Theta(n^2)$ [51], instead of $\Theta(n)$ as in the case of undirected graph problems such as connectivity and minimum spanning tree. A consequence of this fact is that a powerful technique for speeding up dynamic algorithms, sparsification [27], which yielded significant advances in undirected graph problems, is not applicable to problems on directed graphs. In spite of these difficulties, efficient solutions are known for different fundamental problems (see, for example, [5, 43, 51, 52, 53, 54]).

The first problem that we consider in this thesis is the *fully dynamic transitive closure problem*, where we wish to maintain a directed graph $G = (V, E)$ under an intermixed sequence of the following operations:

Insert(x, y): insert an edge (x, y) in G;

Delete(x, y): delete edge (x, y) from G;

Query(x, y): report *yes* if there is a path from x to y in G, and *no* otherwise.

This problem is particularly relevant to the field of databases for supporting transitivity queries on dynamic graphs of relations [71]. The problem also arises in many other areas such as compilers, interactive verification systems, garbage collection, and industrial robotics. Despite its importance, no efficient fully dynamic algorithm for maintaining the transitive closure was known before 1995.

The other problem that we address is the *fully dynamic single-source shortest paths problem*, where the goal is to maintain a weighted directed graph $G = (V, E, w)$ under an intermixed sequence of the following operations:

Increase(x, y, ϵ): increase by ϵ the weight of edge (x, y) in G;

Decrease(x, y, ϵ): decrease by ϵ the weight of edge (x, y) in G;

Query(y): report a shortest path between a fixed source node s and a node y in G, if any.

Various application scenarios for this well known problem include network optimization [2], document formatting [55], routing in communication systems, robotics, incremental compilation, traffic information systems [69], dataflow analysis. For a comprehensive review of the real world settings for the static and dynamic shortest paths problem we refer the reader to [2] and [64], respectively.

In the remainder of this chapter, we discuss previous work related to dynamic transitive closure and dynamic shortest paths problems (Section 1.1) and then we provide an overview of new results and techniques presented this thesis (Section 1.2).

1.1 Previous Work

1.1.1 Dynamic Transitive Closure

Research on dynamic transitive closure spans over two decades. We recognize a starting point in 1983 with the pioneering paper due to Ibaraki and Katoh [47]. Since then, for many years researchers have been working on partially dynamic solutions for the problem. The first fully dynamic algorithm arrived only in 1995, thanks to Henzinger and King [43].

Incremental Algorithms. All the incremental algorithms reviewed in this section have query time $O(1)$, as they explicitly maintain the transitive closure of the graph.

The first incremental solution was presented in [47] and is based on a very simple idea: when adding edge (x, y), if there exist both a path from u to x and a path from y to v, then v becomes reachable from u, if it was not already. The running time of the algorithm is $O(n^3)$ over any sequence of insertions.

This bound was later improved to $O(n)$ amortized time per insertion by Italiano [48], whose algorithm is also able to return a path between any pair of vertices, if any, in time linear in the length of the path itself. Amortized time $O(n)$ per insertion and $O(1)$ per query is also obtained by La Poutré and van Leeuwen in [56].

At last, Yellin [72] presented an algorithm with good running time on bounded degree graphs: the algorithm requires $O(m^*D)$ time for m edge insertions, where m^* is the number of edges in the final transitive closure and D is the out-degree of the final graph.

Decremental Algorithms. The first *decremental* solution was again given by Ibaraki and Katoh [47]: they proposed a depth-first based algorithm with a running time of $O(n^2)$ per deletion.

This bound was later improved to $O(m)$ per deletion by La Poutré and van Leeuwen [56]. Italiano [49] devised a decremental algorithm on directed acyclic graphs which supports deletions in $O(n)$ amortized time per operation.

Similarly to the incremental case, Yellin [72] gave also an $O(m^*D)$ algorithm for m edge deletions, where m^* is the initial number of edges in the transitive closure and D is the out-degree of the initial graph.

We remark that all the aforementioned algorithms have query time $O(1)$. Quite recently, Henzinger and King [43] gave a randomized decremental transitive closure algorithm for general directed graphs with a query time of $O(\frac{n}{\log n})$ and an amortized update time of $O(n \log^2 n)$.

Fully Dynamic Algorithms. Most of the algorithms discussed so far exploit monotonic properties which typically arise in partially dynamic problems. Such properties make it possible to amortize the cost of operations over sequences of bounded length. In addition, when only a kind of operations is permitted, it turns out that high-cost operations are followed by corresponding low-cost ones. For instance, as the number of edges in the transitive closure of a graph cannot be larger than n^2, only a constant number of insertions can let a quadratic number of new edges appear in the transitive closure; successive insertions will therefore have an intrinsically lower cost. Similar arguments cannot be exploited in a fully dynamic setting: bad intermixed sequences of insertions and deletions could force the algorithm to work hard at each step. This gives just an intuition about the reasons which make fully dynamic problems so difficult to be solved.

Before describing the results known for fully-dynamic transitive closure, we list the bounds obtainable 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, 62], 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 $\omega < 2.38$ [11]).

As already stated, the first fully dynamic transitive closure algorithm was devised in 1995 by Henzinger and King [43]: they gave a randomized Monte Carlo algorithm with one-side error supporting a query time of $O(\frac{n}{\log n})$ and an amortized update time of $O(n\hat{m}^{0.58} \log^2 n)$, where $\hat{m}$ is the average number of edges in the graph throughout the whole update sequence. Since $\hat{m}$ can be as high as $O(n^2)$, their update time is $O(n^{2.16} \log^2 n)$.

In 1996 Khanna, Motwani and Wilson [51] designed an algorithm that supports constant queries, but assumes some knowledge of the future update operations at the time of each update. Namely, they 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.

Very recently, King and Sagert [53] showed how to support queries in $O(1)$ time and updates in $O(n^{2.28})$ time for general directed graphs and $O(n^2)$ time for directed acyclic graphs. Their algorithm is randomized with one-side error: it is correct when answering yes, but has $O(\frac{1}{n^c})$ probability of error when answering no, for any constant c. This result represents a breakthrough in the area, as it is the first fully dynamic algorithm which answers reachability queries quickly without assuming knowledge of the future updates.

Finally, the bounds of King and Sagert were further improved by King [52], who exhibited a deterministic algorithm on general digraphs with $O(1)$ query time and $O(n^2 \log n)$ amortized time per update operations. The algorithm in [52] also supports generalized updates, consisting of insertions of a set of edges incident to the same vertex and deletions of an arbitrary subset of edges. We remark that this algorithm, differently from all the previous ones, does not use fast matrix multiplication as a subroutine.

Experimental Studies. In spite of the numerous and relevant theoretical results, less work has been done so far with respect to the experimental analysis of dynamic algorithms for transitive closure. A careful implementation of Italiano's incremental and decremental algorithms is presented in [60], where a comparison against several simple heuristics is carried on. Partially dynamic algorithms are also empirically analyzed in [38], where it has been shown that the algorithms due to Italiano [48] and to Cicerone *et al.* [10] (which generalizes La Poutré and van Leeuwen's algorithm) obtain the best results for the incremental problem.

Implementations of many partially dynamic algorithms are available in the LEDA Extension Package on Dynamic Graph Algorithms [3], which is an extension of the Library of Efficient Data Types and Algorithms [59]. We are aware of just two preliminary experimental studies on fully dynamic transitive closure [6, 60].

1.1.2 Dynamic Shortest Paths

Theoretical Results. The dynamic maintenance of shortest paths has a long history, and the first papers date back to 1967 [57, 63, 67]. In 1985 Even and Gazit [28] and Rohnert [68] presented algorithms for maintaining all pairs 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 shortest paths from scratch. Ausiello *et al.* [5] proposed an incremental all pairs 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.* [44] designed a fully dynamic algorithm for all pairs shortest paths on planar graphs with integer weights, with a running time of $O(n^{9/7}\log(nC))$ per operation. King [52] 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. Other results appeared in [7, 25, 32, 35].

In the case of fully dynamic single-source shortest paths, we are aware of only two dynamic solutions for general directed graphs with arbitrary (i.e., even negative) arc weights. Both these algorithms are analyzed in the *output complexity* model [36, 64, 65], where the cost of an incremental computation is not expressed as a function of the size of the current input, but is rather measured in terms of the sum of the sizes of the input changes and the output changes. We remark that in the worst case the running time of output-bounded dynamic algorithms may be comparable to recomputing from scratch with the best static algorithm.

The algorithm proposed by Ramalingam and Reps [65] requires that all the cycles in the digraph before and after any input update have positive length. It runs in $O(\hat{\delta} + \delta \log \delta)$ per update, where δ is the number of nodes affected by the input change, and $\hat{\delta}$ is the number of affected nodes plus the number of edges having at least one affected endpoint. This gives $O(m + n\log n)$ time in the worst case. In [66] the same authors also propose an output bounded solution for a generalization of the shortest path problem.

The algorithm proposed by Frigioni *et al.* [37] explicitly deals with zero-length cycles; the cost of update operations depends on the output complexity of the operation and on a structural parameter of the directed graph called k-ownership. If the graph has a k-bounded accounting function (as in the case of graphs with genus, arboricity, degree, treewidth or pagenumber limited by k) weight-decrease operations require $O(\min\{m, kn_a\} \log n)$ worst case time, while weight-increase operations require $O(\min\{m\log n, k(n_a + n_b) \log n + n\})$ worst case time. Here n_a is the number of affected nodes, and n_b is the number of nodes considered by the algorithm.

Experimental Studies. Many papers have been proposed in the algorithm engineering field concerning the practical performances of static algorithms for computing shortest paths trees (see, e.g., [8, 9, 42]), but very little is known for the experimental evaluation of dynamic shortest paths algorithms. In particular, to the best of our knowledge, there is no experimental study concerning fully dynamic single-source shortest paths in digraphs with arbitrary edge weights. In Chapter 4 of this thesis we make a step towards this direction discussing a preliminary experimental investigation which tries to fill this gap.

The case of digraphs with only positive edge weights is considered in [34], where two algorithms proposed by Ramalingam and Reps [65] and by Frigioni *et al.* [35] are compared with a static counterpart, i.e., Dijkstra's algorithm implemented with Fibonacci heaps [24, 33]. The outcome of this study shows that both on random and on real-world graphs dynamic algorithms allow to spend in updates less than 5% of the time required by the static one.

1.2 Original Contributions of the Thesis

The original contributions of this thesis are presented in Chapter 3 (fully dynamic transitive closure) and in Chapter 4 (fully dynamic single-source shortest paths). Chapter 2 introduces preliminary definitions and algebraic concepts which are the backbone for modeling path problems on directed graphs, and Chapter 5 addresses concluding remarks and open problems.

Most of the results presented in Chapter 3 and in Chapter 4 have been published in the *Proceedings of the* 41*-st Annual IEEE Symposium on Foundations of Computer Science (FOCS'00)* [20] and in the *Proceedings of the* 4*-th International Workshop on Algorithm Engineering (WAE'00)* [19], respectively.

Fully Dynamic Transitive Closure

Before stating our results, 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. It is thus quite natural to ask whether the $O(n^2)$ update bound can be actually realized for fully dynamic transitive closure on general directed graphs while maintaining one lookup per query. Another important question, if one is willing to spend more time for queries, is whether the $O(n^2)$ barrier for the single-operation time complexity of fully dynamic transitive closure can be broken. We remark that this has been an elusive goal for many years. In this thesis we affirmatively answer both questions.

Obtained Results. We devise a novel technique which allows us to reduce fully dynamic transitive closure to the problem of dynamically reevaluating polynomials over matrices when updates of variables are performed.

Using this new technique, we improve the best known bounds for fully dynamic transitive closure. In particular:

- We devise a deterministic algorithm for fully dynamic transitive closure on general digraphs which does exactly one matrix look-up per query and supports updates in $O(n^2)$ amortized time, thus improving over [52]. Our algorithm can also support within the same time bounds the generalized updates of [52], i.e., insertion of a set of edges incident to the same vertex and deletion of an arbitrary subset of edges. The space used is $O(n^2)$.

- In the special case of deletions only, our algorithm achieves $O(n)$ amortized time for deletions and $O(1)$ time for queries: this generalizes to directed graphs the bounds of [49], and improves over [43].
- We devise the first algorithm which support both updates and queries in subquadratic time per operation, proving that it is actually possible to break through the $O(n^2)$ barrier on the single-operation complexity of fully dynamic transitive closure. Our subquadratic algorithm is randomized Monte Carlo and supports update in $O(n^{1.58})$ and query in $O(n^{0.58})$ worst-case time.

Our deterministic algorithm hinges upon the equivalence between transitive closure and matrix multiplication on a closed semiring; this relation has been known for over 30 years (see e.g., the results of Munro [62], Furman [39] and Fischer and Meyer [31]) and yields the fastest known static algorithm for transitive closure. Surprisingly, no one before seems to have exploited this equivalence in the dynamic setting: some recent algorithms [43, 51, 53] make use of fast matrix multiplication, but only as a subroutine for fast updates. Differently from other approaches, the crux of our method is to use dynamic reevaluation of products of Boolean matrices as the kernel for solving dynamic transitive closure.

Fully Dynamic Single-source Shortest Paths

Recently, an important research effort has been done in the field of *algorithm engineering*, aiming at bridging the gap between theoretical results on algorithms and their implementation and practical evaluation [4, 21, 41, 50, 58, 61].

Motivated by the lack of any previous investigation about the practical performances of fully dynamic algorithms for maintaining shortest paths trees in graphs with arbitrary edge weights, we performed a preliminary experimental study of the best known algorithms for the problem [37, 65]. From the experiments, we conjectured that certain preliminary steps, performed by these algorithms during an update operation with the aim of saving time in subsequent steps, may not be worthwhile in some practical situations.

Obtained Results. These experimental observations inspired us in considering a new simplified variant of the best known algorithms for the problem. Our variant was especially designed to be simple and fast in practice while matching the same asymptotic worst-case complexity of the original algorithms.

We implemented in `C++` with the support of the `LEDA` library of efficient data types [59] the best known algorithms [37, 65], plus our new variant, and we performed a further experimental study of these algorithms on random test sets. Our investigation revealed that:

1. all the considered dynamic algorithms are faster by several orders of magnitude than recomputing from scratch with the best static algorithm: this yields clues to the fact that constant factors in practical implementations can be very small for this problem;

2. due to its simplicity, our new variant stands out as a practical solution, though the more sophisticated algorithms it is derived from may become preferable in some cases;

3. the running time of all the considered dynamic algorithms is affected by the width of the interval of edge weights. In particular, dynamic algorithms are faster if weights come from a small set of possible values.

Chapter 2

Preliminaries

2.1 Introduction

In this chapter we provide basic notation, lemmas, facts and definitions that will be useful later on for describing the original contributions of our dissertation. In particular, we provide the background for the two fundamental algorithmic problems on graphs that we study in our thesis: the transitive closure problem and the single-source shortest paths problem. Though most of the material presented in this chapter is well known from the literature, many definitions, lemmas and proofs have been reformulated to help the reader in maintaining a coherent and homogeneous view of the matter.

The chapter is organized as follows. Section 2.2 is devoted to basic yet powerful algebraic concepts that are the backbone of path problems on directed graphs. Section 2.3 formally defines the transitive closure problem, provides an algebraic framework for it based on Boolean matrices and discusses the best known algorithms, showing that transitive closure is computationally reducible to Boolean matrix multiplication. Section 2.4 formally introduces the single-source shortest paths problem, recalls the best known algorithms for it, and presents properties useful for designing efficient solutions for the dynamic versions of the problem.

2.2 Basic Algebraic Concepts

This section presents basic definitions of some algebraic concepts and related properties that are useful in dealing with path problems on directed graphs. In particular, we recall the definitions of closed semiring and of Kleene closure, showing that matrices over closed semirings form a closed semiring by themselves. We also define the concept of ring, useful for obtaining fast algorithms for Boolean matrix multiplication, and we introduce two algebraic structures that we will use for modeling path problems.

Definition 2.1 *A* CLOSED SEMIRING *is an algebraic structure* $(S, +, \cdot, 0, 1)$*, where* S *is a set of elements,* $+$ *(the* SUMMARY OPERATOR*) and* $\cdot$ *(the* EXTENSION OPERATOR*) are binary operations on* S*, and* 0 *and* 1 *are elements of* S*, satisfying the following eight properties:*

1. $(S, +, 0)$ *is a* MONOID*:*

   - S *is* CLOSED *under* $+$*:* $a + b \in S$ *for all* $a, b \in S$.
   - $+$ *is* ASSOCIATIVE*:* $a + (b + c) = (a + b) + c$ *for all* $a, b, c \in S$.
   - 0 *is an* IDENTITY *for* $+$*:* $a + 0 = 0 + a = a$ *for all* $a \in S$.

   Likewise, $(S, \cdot, 1)$ *is a monoid.*

2. 0 *is an* ANNIHILATOR*:* $a \cdot 0 = 0 \cdot a = 0$ *for all* $a \in S$.

3. $+$ *is* COMMUTATIVE*:* $a + b = b + a$ *for all* $a, b \in S$.

4. $+$ *is* IDEMPOTENT*:* $a + a = a$ *for all* $a \in S$.

5. $\cdot$ *distributes over* $+$*:* $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$ *and* $(b + c) \cdot a = (b \cdot a) + (c \cdot a)$ *for all* $a, b, c \in S$.

6. *If* $a_1, a_2, a_3, \ldots$ *is a countable sequence of elements of* S*, then*

$$a_1 + a_2 + a_3 + \cdots$$

   is well defined and in S.

7. *Associativity, commutativity, and idempotence apply to infinite as well as finite sums.*

8. $\cdot$ *distributes over countably infinite sums as well as over finite ones:*

$$a \cdot \left(\sum_j b_j \right) = \sum_j (a \cdot b_j) \text{ and } \left(\sum_j b_j \right) \cdot a = \sum_j (b_j \cdot a)$$

Based on Definition 2.1, we now introduce the closed semiring of Boolean values, which is an important system for modeling path problems on directed graphs.

Lemma 2.1 *The system of Boolean values* $S_1 = (\{0,1\}, +, \cdot, 0, 1)$ *where* $+$ *and* $\cdot$ *are defined as follows:*

+	0	1
0	0	1
1	1	1

·	0	1
0	0	0
1	0	1

is a closed semiring.

Proof. Verifying properties 1–5 is straightforward. As far as properties 6–8 are concerned, it suffices to observe that a countable sum is 0 if all terms are 0, and is 1 if at least one term is 1. □

Notice that the $+$ and $\cdot$ operators in S_1 correspond to the usual $\vee$ and $\wedge$ operators on Boolean values, respectively.

Let us now focus on the algebraic structures obtained by considering matrices over closed semirings. Later we show that such systems are closed semirings as well. Our convention for denoting matrix entries is the following: if X is a matrix, we denote by $X[i,j]$ the element in the i-th row and j-th column in X.

Definition 2.2 *Let* $(S, +, \cdot, 0, 1)$ *be a closed semiring and let* M_n *the set of* $n \times n$ *matrices over* S*. We denote by* 0_n *the* $n \times n$ *matrix of* 0*'s and by* I_n *the* $n \times n$ IDENTITY MATRIX *with* 1*'s on the main diagonal and* 0*'s elsewhere:*

$$0_n = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix} \qquad I_n = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

For $A, B \in M_n$*, we denote by* $A +_n B$ *the* $n \times n$ *matrix* C*, where:*

$$C[i,j] = A[i,j] + B[i,j],$$

and we denote by $A \cdot_n B$ *the* $n \times n$ *matrix* D*, where:*

$$D[i,j] = \sum_{k=1}^{n} A[i,k] \cdot B[k,j].$$

The fact that $(S, +, \cdot, 0, 1)$ is a closed semiring immediately leads to the following property:

Lemma 2.2 $(M_n, +_n, \cdot_n, 0_n, I_n)$ *is a closed semiring.*

From now on we will use $+$ and $\cdot$ instead of $+_n$ and $\cdot_n$ when there is no possibility of confusion with the addition and multiplication operators in the underlying closed semiring of matrix entries.

We now introduce a unary operator, the Kleene closure operator $*$, which is central to our analysis of closed semirings and, as we will see in Section 2.3, is the main tool for solving reachability problems on directed graphs.

Definition 2.3 *Let $(S, +, \cdot, 0, 1)$ be a closed semiring and $a \in S$. We define the* KLEENE CLOSURE *of a as*

$$a^* = \sum_{i=0}^{\infty} a^i,$$

where

$$a^i = \begin{cases} 1 & if\ i = 0 \\ a \cdot a^{i-1} & if\ i > 0 \end{cases}$$

is the i-th power of a. That is, a^* is the infinite sum $1 + a + a \cdot a + a \cdot a \cdot a + \cdots$.

Notice that the closure operator is derived from the base $+$ and $\cdot$ operators of closed semirings by means of infinite sums. Using the laws reported in Definition 2.1, and in particular the fact that countable infinite sums are well defined (property 6), we can prove the important feature that closed semirings are closed under $*$.

Lemma 2.3 *Let $(S, +, \cdot, 0, 1)$ be a closed semiring and $a \in S$. Then $a^* \in S$.*

Proof. The proof easily follows from the definition of Kleene closure and from properties 6–8 of closed semirings. □

A broader class of algebraic structures is the class of rings. As we will see in Section 2.3.1, using rings is necessary for obtaining efficient algorithms for matrix multiplication.

Definition 2.4 *A* RING *is an algebraic structure $(S, +, \cdot, 0, 1)$, where S is a set of elements, $+$ and $\cdot$ are binary operations on S, and 0 and 1 are elements of S, satisfying the following six properties for each $a, b, c \in S$:*

1. *$+$ and $\cdot$ are* ASSOCIATIVE*: $(a + b) + c = a + (b + c)$ and $(a \cdot b) \cdot c = a \cdot (b \cdot c)$.*
2. *$+$ is* COMMUTATIVE*: $a + b = b + a$.*
3. *$\cdot$* DISTRIBUTES *over $+$: $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$ and $(b + c) \cdot a = (b \cdot a) + (c \cdot a)$.*
4. *0 is an* IDENTITY *for $+$: $a + 0 = 0 + a = a$.*

5. 1 *is an* IDENTITY *for* $\cdot$*:* $a \cdot 1 = 1 \cdot a = a$.

6. *For each* $a \in S$ *there is an* INVERSE $-a$*:* $a + (-a) = (-a) + a = 0$.

Occasionally, we will use the operator $-$, denoting with $a - b$ the expression $a + (-b)$. The next lemma introduces the ring of integers that is another important algebraic tool used in this thesis.

Lemma 2.4 *The system* $S_2 = (\{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}, +, \cdot, 0, 1)$ *of integers where* $+$ *and* $\cdot$ *are the usual sum and product operators is a ring.*

A ring useful for obtaining efficient algorithms for Boolean matrix multiplication is the following.

Lemma 2.5 *The system* $\mathcal{Z}_{n+1} = (\{0, 1, 2, 3, \ldots, n\}, +, \cdot, 0, 1)$ *where* $+$ *and* $\cdot$ *are the usual sum and product operators performed modulo* $n+1$ *is a ring.*

2.3 Transitive Closure and Matrix Multiplication

In this section we recall basic graph-theoretic definitions, such as the notion of transitive closure, and we discuss a matrix-based framework for solving reachability problems on directed graphs. In particular, we show how the computation of the transitive closure of a directed graph is related to the problem of multiplying Boolean matrices and we describe known efficient algorithms for the problem.

Definition 2.5 *The* ADJACENCY MATRIX *of a graph* $G = (V, E)$ *with* n *nodes is a* $n \times n$ *Boolean matrix* X_G *such that* $X_G[u, v] = 1$ *if and only if* $(u, v) \in E$.

Definition 2.6 *Let* $G = (V, E)$ *be a directed graph, where* V *is the set of nodes and* $E \subseteq V \times V$ *is the set of edges. A* PATH $u \rightsquigarrow v$ *of length* k *between any two nodes* u *and* v *in* G *is a sequence of nodes* $\pi = \langle w_0, w_1, \ldots, w_k \rangle$ *such that* $w_0 = u$*,* $w_k = v$ *and* $(w_{i-1}, w_i) \in E$ *for any* $0 < i \leq k$*. If there exists a path* $u \rightsquigarrow v$*, we say that* v *is* REACHABLE *from* u *in* G*. If, for any* $i \neq j$*, it holds that* $w_i \neq w_j$*, we say that the path is* SIMPLE.

As a matrix counterpart of the previous graph-theoretic definition, we also introduce the notion of path between indices in a Boolean matrix.

Definition 2.7 *Let* X *be a Boolean matrix. A* PATH $u \rightsquigarrow v$ *of length* k *between any two indices* u *and* v *in* X *is a sequence of indices* $\pi = \langle w_0, w_2, \ldots, w_k \rangle$ *such that* $w_0 = u$*,* $w_k = v$ *and* $X[w_{i-1}, w_i] = 1$ *for any* $0 < i \leq k$.

The next lemma provides a direct correspondence between paths in a graph and paths in its adjacency matrix.

Lemma 2.6 *Let G be a graph and let X_G be its adjacency matrix. For any u, v, there is a path between indices u and v in X_G if and only if there is a path between nodes u and v in G.*

We also recall the concept of cycle in a directed graph.

Definition 2.8 *Let $G = (V, E)$ be a directed graph. A* Cycle *in G is a path $u \leadsto v$ such that $u = v$.*

We are now ready to formally introduce the notion of transitive closure of a directed graph.

Definition 2.9 *Let $G = (V, E)$ be a directed graph, where V is the set of nodes and $E \subseteq V \times V$ is the set of edges. The* Transitive Closure *of G is a directed graph $TC(G) = (V', E')$ such that $V = V'$ and $(u, v) \in E'$ if and only if there is a simple path $u \leadsto v$ in G. If for any $u \in V$, $(u, u) \in E'$, then we say that the transitive closure $TC(G)$ is* Reflexive.

In the following we will refer to the Transitive Closure Problem as the problem of computing the (reflexive) transitive closure of a directed graph.

Notice that in the definition of transitive closure we limit ourselves to consider simple paths. The reason of this choice is explained in the next lemma.

Lemma 2.7 *Let $G = (V, E)$ be a directed graph. For any non-simple path $\pi : u \leadsto v$ there is a corresponding simple path $\pi' : u \leadsto v$.*

Proof. Let $\pi = \langle u, \ldots, c, w_1, \ldots, w_h, c, \ldots, v\rangle$ be a non-simple path between u and v, where $\langle c, w_1, \ldots, w_h, c\rangle$ is a cycle in G. We repeatedly collapse any repetitions of the same node c in the sequence π, cutting off any intermediate nodes $w_1, \ldots, w_h$ and thus removing cycles from the path. When no repetition remains, we come up with a simple path of the form $\pi' = \langle u, \ldots, c, \ldots, v\rangle$. $\square$

The adjacency matrix representation of a graph, together with the algebraic concepts of closed semirings and of Kleene closure, are powerful tools for expressing in a compact form path relations such as transitivity. The following lemmas provide a direct correspondence between the transitive closure of a graph G and the Kleene closure of the adjacency matrix of G. Such correspondence allows us to provide efficient solutions to path problems on directed graphs in the algebraic framework of matrices.

Lemma 2.8 *Let $G = (V, E)$ be a directed graph and let X be the adjacency matrix of G. Then $X^k[u, v] = 1$ if and only if there is a path $u \leadsto v$ of length k in G.*

Proof. The proof is by induction on k. The base step ($k = 1$) is trivial. Let us now assume, by inductive hypothesis, that the claim holds for paths of length $k - 1$. We show that the claim holds for paths of length k as well. From the relation $X^k = X \cdot X^{k-1}$ and from the definition of matrix product given in Definition 2.2, we have that $X^k[u, v] = 1$ if and only if there is a node w such that both $X[u, w] = 1$ and $X^{k-1}[w, v] = 1$. Thus, using the inductive hypothesis, $X^k[u, v] = 1$ if and only if there is a path $u \leadsto v$ of length k in G obtained as concatenation of a single edge (u, w) and of a path $w \leadsto v$ of length $k - 1$. □

Lemma 2.9 *Let $G = (V, E)$ be a directed graph and let $TC(G)$ be the (reflexive) transitive closure of G. If X is the adjacency matrix of G and Y is the adjacency matrix of $TC(G)$, then $Y = X^*$.*

Proof. It suffices to prove that for any $u, v \in V$, $X^*[u, v] = 1$ if and only if there is a path of any length in G between u and v. The proof easily follows from the definition of Kleene closure and from Lemma 2.8. □

In view of Lemma 2.8 and Lemma 2.9, in the following we will consider the problem of computing the Kleene closure of a Boolean matrix as an equivalent method for computing the transitive closure of a directed graph.

Before describing some known efficient algorithms for computing the Kleene closure of a Boolean matrix, we address the problem of computing quickly $+$ and $\cdot$ operations over Boolean matrices.

2.3.1 Computing Sums and Products of Boolean Matrices

This section focuses on the computational cost of performing additions and products of matrices as introduced in Definition 2.2. In particular, we restrict our interest on products of Boolean matrices.

As a first observation, notice that the sum $C = A + B$ of two Boolean matrices A and B as introduced in Definition 2.2 can be easily carried out in $O(n^2)$ worst-case time.

Let $O(n^\omega)$ be the cost of computing the product $D = A \cdot B$ of two Boolean matrices A and B.

The simplest method for building D with $\omega = 3$ consists of performing directly the calculations specified in Definition 2.2 for any entry of D, i.e., computing $D[i, j] = \sum_{k=1}^{n} A[i, k] \cdot B[k, j]$ for any i, j.

If we assume to deal with matrices over a ring, more efficient methods are known: Strassen [70] showed that it is possible to multiply two matrices over an arbitrary ring with $\omega = \log_2 7 < 2.81$, while Coppersmith and Winograd [11] improved this bound to $\omega < 2.38$.

Though Boolean matrices do not form a ring, adapting these algorithms to compute products of $n \times n$ Boolean matrices is easy [12]: we can simply assume to perform calculations in the ring $\mathcal{Z}_{n+1}$ of integers modulo $n+1$ (see Lemma 2.5) by looking at 0 and 1 as integer numbers. The results of such computations can be converted back to Boolean values by treating any non-zero value as 1 and any zero value as 0. With this simple trick, matrix products can be performed efficiently while maintaining the results as Boolean matrices. Even if the ring of integers S_2 introduced in Lemma 2.4 would work either way, considering matrices over the ring $\mathcal{Z}_{n+1}$ of integers modulo $n+1$ is used to prevent matrix values arising in intermediate computations from getting too large.

2.3.2 Computing the Kleene Closure of a Boolean Matrix

In this section we put together concepts developed in the previous sections and we show efficient algorithms for computing the Kleene closure of a Boolean matrix through reduction to the Boolean matrix multiplication problem. In particular, we discuss two methods: the first is based on a simple efficient 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 that yields the fastest known algorithm for computing the Kleene closure of a Boolean matrix. Actually, we prove the surprising fact that computing the Kleene closure can be asymptotically as fast as multiplying two Boolean matrices.

Method 1

We first describe a simple method for computing X^* in $O(n^\omega \cdot \log n)$ worst-case time, where $O(n^\omega)$ is the time required for computing the product of two Boolean matrices. The algorithm is based on a simple doubling argument as stated in the following definition and lemmas.

Definition 2.10 *Let X be an $n \times n$ Boolean matrix. We define the sequence of $\log_2 n + 1$ polynomials $P_0, \ldots, P_{\log_2 n}$ over Boolean matrices as:*

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

Lemma 2.10 *Let X be an $n \times n$ Boolean matrix and let P_k be formed as in Definition 2.10. Then for any $1 \leq u, v \leq n$, $P_k[u,v] = 1$ if and only if there is a path $u \rightsquigarrow v$ of length at most 2^k in X.*

Proof. The proof is by induction on k. The base step for $k = 0$ is trivial. We assume by inductive hypothesis that the claim is satisfied for P_{k-1} and we prove that it is satisfied for P_k as well.

Sufficient condition: as any path of length up to 2^k between u and v in X is either of length up to 2^{k-1} or can be obtained as concatenation of two paths of length up to 2^{k-1} in X, and all these paths are correctly reported in P_{k-1} by the inductive hypothesis, it follows that $P_{k-1}[u,v] = 1$ or $P_{k-1}^2[u,v] = 1$. Thus $P_k[u,v] = P_{k-1}[u,v] + P_{k-1}^2[u,v] = 1$.

Necessary condition: if $P_k[u,v] = 1$ then at least one among $P_{k-1}[u,v]$ and $P_{k-1}^2[u,v]$ is 1. If $P_{k-1}[u,v] = 1$, then by the inductive hypothesis there is a path of length up to $2^{k-1} < 2^k$. If $P_{k-1}^2[u,v] = 1$, then there are two paths of length up to 2^{k-1} whose concatenation yields a path no longer than 2^k. □

This method hinges upon a standard doubling argument: 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 up to 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 2.1 *Let X be an $n \times n$ Boolean matrix and let P_k be formed as in Definition 2.10. Then $X^* = I_n + P_{\log_2 n}$.*

Proof. The proof easily follows from Lemma 2.10 and from Lemma 2.9 by observing that the length of the longest simple path in X is no longer than $n - 1 < 2^{\log_2 n} = n$. The I_n is required to guarantee the reflexivity of X^*. □

The next theorem discusses the time required to calculate X^* via computation of the polynomials over Boolean matrices that define $P_1, \ldots, P_{\log_2 n}$.

Theorem 2.2 *Let X be an $n \times n$ Boolean matrix and let P_k be formed as in Definition 2.10. Then it is possible to compute $X^* = I_n + P_{\log_2 n}$ in $O(n^\omega \cdot \log n)$ worst-case time, where ω is the exponent of Boolean matrix multiplication.*

Proof. The proof easily follows from the cost of computing each P_k for $k = 1$ to $\log_2 n$. □

The best known bounds for Boolean matrix multiplication (see Section 2.3.1) imply that X^* can be computed in $o(n^{2.38} \cdot \log n)$ worst-case time with this method.

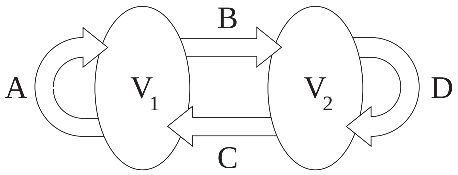


Figure 2.1: Succinct graph representation of the matrix X.

Method 2

We now show that computing the Kleene closure of a Boolean matrix can be asymptotically as fast as multiplying two Boolean matrices. The method we present is due to Munro [62] and is based on a Divide and Conquer strategy.

Definition 2.11 *Let $\mathcal{B}_n$ be the set of $n \times n$ Boolean matrices and let $X \in \mathcal{B}_n$. Without loss of generality, we assume that n is a power of 2. We define a mapping $\mathcal{F} : \mathcal{B}_n \to \mathcal{B}_n$ by means of the following equations:*

$$\begin{cases} E = (A + BD^*C)^* \\ F = EBD^* \\ G = D^*CE \\ H = D^* + D^*CEBD^* \end{cases} \tag{2.1}$$

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

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

Theorem 2.3 *Let X be an $n \times n$ Boolean matrix. Then $\mathcal{F}(X) = X^*$.*

Proof. Let $V_1 = \{1, \ldots, \frac{n}{2}\}$ and $V_2 = \{\frac{n}{2} + 1, \ldots, n\}$ a balanced partition of the indices of X into two sets of size $\frac{n}{2}$ each.

In Figure 2.1 we show a succinct representation of the graph G_X whose adjacency matrix is X. Notice that B encodes all edges that connect a node in V_1 with a node in V_2. Likewise, C connects nodes in V_2 with nodes in V_1, and A and D represent internal connections between nodes in V_1 and V_2, respectively.

We now prove that $E = (A + BD^*C)^*$ encodes explicitly all the paths in X that have both end-points in V_1. More formally, we prove that for any $i, j \in V_1$, $E[i, j] = 1$ if and only if there is a path between i and j in X. Let us consider the $\frac{n}{2} \times \frac{n}{2}$ matrix $R = A + BD^*C$. For any $u, v \in V_1$, we have that $R[u,v] = 1$ if and only if $X[u,v] = 1$ or there is a path $\langle u, w_1, \ldots, w_k, v\rangle$ in X, where $w_1, \ldots, w_k \in V_2$. In other words, R encodes explicitly all the paths in X that have both end-points u, v in V_1 and such that v is reachable from u without passing through any other node in V_1. As any path in X with both end-points in V_1 corresponds to a path in R, and all the paths in R are explicitly encoded by R^*, the claim is proved for $E = R^*$.

Regarding F, we prove that $F = EBD^*$ encodes all the paths in X that have one end-point in V_1 and the other end-point in V_2. Notice that any such path can be written in the form $\langle z_1, \ldots, z_h, w_1, \ldots, w_k\rangle$, where $z_1 \in V_1$, $z_2, \ldots, z_{h-1} \in V_1 \cup V_2$, $z_h \in V_1$, and $w_1, \ldots, w_k \in V_2$. Since $E[z_1, z_h] = 1$, $B[z_h, w_1 - \frac{n}{2}] = 1$ and $D^*[w_1 - \frac{n}{2}, w_k - \frac{n}{2}] = 1$, it holds that $F[z_1, w_k - \frac{n}{2}] = 1$. This proves the claim for F. By similar reasoning, we can prove analogous claims for G and H. $\Box$

It is possible to think of a different definition of function $\mathcal{F}$ that provides an alternative way of computing the Kleene closure. In particular, another set of equations which are equivalent to those in 2.1 can be introduced.

Lemma 2.11 *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:*

$$\begin{cases} E = A^* + A^*BHCA^* \\ F = A^*BH \\ G = HCA^* \\ H = (D + CA^*B)^* \end{cases} \tag{2.2}$$

where X and $Y = \mathcal{G}(X)$ 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 $X \in \mathcal{B}_n$, $\mathcal{G}(X) = \mathcal{F}(X) = X^*$.

Proof. If we rename syntactically submatrices A, B, C, D, and E, F, G, H in the set of equations 2.2 as follows:

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

we obtain the set of equations 2.1. If we rotate both X and Y by 180 degrees, we are exactly in the same conditions of Definition 2.11. Thus, $\mathcal{G} = \mathcal{F}$. $\Box$

In the following theorem, we discuss the time required for computing $Y = \mathcal{F}(X)$ as specified in Definition 2.11.

Theorem 2.4 *Let X be an $n \times n$ Boolean matrix and let $T(n)$ be the time required to compute recursively $\mathcal{F}(X)$. Then $T(n) = O(n^\omega)$, where $O(n^\omega)$ is the time required to multiply two Boolean matrices.*

Proof. 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. Thus:

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

where $M(n) = O(n^\omega)$ is the time required to multiply two $n \times n$ Boolean matrices. Solving the recurrence relation, we obtain that $T(n) = O(n^\omega)$. For more details, we refer the interested reader to [1]. $\square$

The best known bounds for Boolean matrix multiplication (see Section 2.3.1) imply that 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 $\hat{X}$ and $\hat{Y}$, respectively, having dimension that is a power of 2, of the form:

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

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 $\hat{Y} = \hat{X}^*$ is not affected.

2.4 Shortest Paths and Reweighting Techniques

In the previous section we focused on the reachability problem of computing whether there is a path between any pair of nodes in a directed graph. We were just interested in looking for the existence of paths, regardless of their length. In this section we consider the extended version of the problem where we look for the shortest paths between a given source node and all the other nodes in a weighted directed graph. In the following we will refer to this problem as the single-source shortest paths problem. We first provide basic definitions and properties and then we briefly survey the best known algorithms for the problem. The material presented in this section will play a central role in Chapter 4 in the design and analysis of efficient algorithms for dynamic versions of the single-source shortest paths problem.

We first provide some basic notation related to weighted directed graphs.

Definition 2.12 *We denote by $G = (V, E, w)$ a weighted directed graph where V is the set of nodes, E is the set of edges, and $w : E \to \mathcal{R}$ is a function that* *maps any edge (u, v) of the graph to a real number $w(u, v)$ called the* WEIGHT *of the edge.*

According to the previous definition, we introduce the concept of weighted length of a path as the sum of the weights of the edges in the path. This differs from Definition 2.6, where we considered just the number of traversed edges.

Definition 2.13 *Let $G = (V, E, w)$ be a weighted directed graph and let $\pi = \langle x_0, \ldots, x_k \rangle$ be a path in G. Then the* WEIGHTED LENGTH *of π is:*

$$\ell(\pi) = \sum_{i=1}^{k} w(x_{i-1}, x_i).$$

In the following, we will use the term *length* instead of *weighted length* when there is no possibility of confusion. Now we formally introduce the concept of shortest path.

Definition 2.14 *Let $G = (V, E, w)$ be a weighted directed graph. A* SHORTEST PATH *between $u, v \in V$ is a path $\pi^* : u \rightsquigarrow v$ that satisfies the equation:*

$$\ell(\pi^*) = \inf_{\pi : u \rightsquigarrow v} \ell(\pi).$$

Notice that, if there are edges of negative weight, there may be negative-length cycles $\langle c, w_1, \ldots, w_k, c \rangle$ in G. We now discuss some properties due to the presence of such cycles in the graph.

Lemma 2.12 *Let $G = (V, E, w)$ be a weighted directed graph. If there is a path of the form $\pi = \langle u, \ldots, c, w_1, \ldots, w_k, c, \ldots, v \rangle$ in G such that $\gamma = \langle c, w_1, \ldots, w_k, c \rangle$ is a negative-length cycle, then $\inf_{\pi : u \rightsquigarrow v} \ell(\pi) = -\infty$ and no shortest path exists between u and v.*

Proof. Starting from π, we obtain a shorter path π' by simply traversing γ more than once: $\pi' = \langle u, \ldots, c, w_1, \ldots, w_k, c, w_1, \ldots, w_k, c, \ldots, v \rangle$. Thus, since the more we traverse the cycle γ, the shorter we get the path, it follows that $-\infty$ is a lower bound to the length of any path $u \rightsquigarrow v$ that passes through a node lying on a negative-length cycle. As any path has finite length, no path π satisfies $\ell(\pi) = -\infty$ and no shortest path exists. □

We now introduce the natural notion of distance of a node from the source.

Definition 2.15 *Let $G = (V, E, w)$ be a weighted directed graph and let $s \in V$ be a fixed* SOURCE *node. We define a* DISTANCE *function $d_s : V \to \mathcal{R}$ such that for any $v \in V$,*

$$d_s(v) = \begin{cases} \inf_{\pi:s\rightsquigarrow v} \ell(\pi) & \textit{if } v \textit{ is reachable from } s \textit{ in } G \\ +\infty & \textit{otherwise} \end{cases}$$

Distance functions satisfy the following relevant Bellman conditions:

Lemma 2.13 *Let $G = (V, E, w)$ be a weighted directed graph and let $s \in V$ be a fixed source node. If d_s is the distance function from s, then for any $(u, v) \in E$ such that $-\infty < d_s(u) < +\infty$, the following Bellman condition is satisfied:*

$$d_s(v) \leq d_s(u) + w(u, v).$$

Moreover, if $\langle s, \ldots, u, v\rangle$ is a shortest path from s to v, then:

$$d_s(v) = d_s(u) + w(u, v).$$

Proof. Let us assume by contradiction that there is an edge $(u, v) \in E$ such that $d_s(v) > d_s(u) + w(u, v)$. Now, if $\nu = \langle s, \ldots, u\rangle$ is a shortest path to u with length $\ell(\nu) = d_s(u)$, then the path $\mu = \langle s, \ldots, u, v\rangle$ has length $\ell(\mu) = \ell(\nu) + w(u, v) = d_s(u) + w(u, v)$, which is $< d_s(v)$ by our initial assumption. This contradicts the fact that $d_s(v) = inf_{\pi:s\rightsquigarrow v}\ell(\pi)$. □

Certificates for distances of nodes from the source are provided by shortest paths trees, which represent buckets of shortest paths from the source to any other reachable node, as we show next.

Definition 2.16 *Let $G = (V, E, w)$ be a weighted directed graph and let $s \in V$ be a fixed source node. We call* SHORTEST PATHS TREE *rooted at s any tree $T(s) = (V', E')$ such that:*

1. *$V' \subseteq V$ and $E' \subseteq V' \times V' \cap E$;*
2. *for all $v \in V'$, $-\infty < d_s(v) < +\infty$;*
3. *for all $(u, v) \in E'$ the following Bellman condition is satisfied:*

$$d_s(v) = d_s(u) + w(u, v). \tag{2.3}$$

According to the previous definition, a shortest paths tree $T(s)$ contains all the nodes v that are reachable from the source s ($d_s(v) < +\infty$) without entering negative-length cycles ($d_s(v) > -\infty$). Moreover, any (simple) path in $T(s)$ is a shortest path in G.

We remark that a shortest paths tree $T(s)$ for a given source s in G may not be unique. Actually, if $(u, v), (w, v) \in E$ and both u and w lie on some shortest path from s to v, then either (u, v) or (w, v) may be in $T(s)$.

Using the previous definitions, we can now formally introduce the single-source shortest paths problem.

Definition 2.17 *Let $G = (V, E, w)$ be a weighted directed graph and let $s \in V$ a fixed source node. We define the* Single-source Shortest Paths Problem *as the problem of computing a shortest paths tree $T(s)$ of G rooted at s and the corresponding distance function d_s.*

We now discuss properties that will be useful in Chapter 4 in designing efficient dynamic algorithms for the single-source shortest paths problem. In particular, we show a reweighting technique that allows it to preprocess the weighting function of a graph so as to have no negative-weight edges while maintaining the same shortest paths in the graph, as we will see in the next two lemmas.

Definition 2.18 *Let $G = (V, E, w)$ be a weighted directed graph and let $h : V \to \mathcal{R}$ be an arbitrary function defined over the node set. We will refer to h as a* Potential *function. We define* Reweighting *function for w the function $\hat{w} : E \to \mathcal{R}$ such that for any $(u, v) \in E$, $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$.*

We remark that in the previous definition the potential function h is completely arbitrary. An important property of such reweighting is that the shortest paths in the graph stay the same after the preprocessing and it is possible to apply efficient algorithms that are designed to work only on graphs with nonnegative edge weights.

Lemma 2.14 *Let $G = (V, E, w)$ and $\hat{G} = (V, E, \hat{w})$ be two weighted directed graphs with the same sets of nodes and edges, where $\hat{w}$ is any reweighting function for w. A path π^* is a shortest path in G if and only if π^* is a shortest path in $\hat{G}$. Moreover, γ is a negative-length cycle in G if and only if γ is a negative-length cycle in $\hat{G}$.*

Proof. See [12]. □

Another property of the reweighting technique is that, provided that we have already computed a distance function d_s, if we look at d_s as a potential function, we come up with a nonnegative weighting function $\hat{w}$.

Lemma 2.15 *Let $G = (V, E, w)$ be a weighted directed graph, let $s \in V$ be a fixed source and let $d_s : V \to \mathcal{R}$ be a distance function from s. If we define $\hat{w}(u, v) = w(u, v) + d_s(u) - d_s(v)$, then $\hat{w}(u, v) \geq 0$ for any $(u, v) \in E$.*

Proof. From Lemma 2.13, we know that distance functions satisfy the Bellman condition: $d_s(v) \leq d_s(u) + w(u, v)$. Thus, $0 \leq d_s(u) + w(u, v) + d_s(u) - d_s(v) = \hat{w}(u, v)$. □

Notice that the reweighting technique with potentials equal to distances is not useful for solving the single-source shortest paths tree as it requires distances to be already computed. However, as we already stated, it is the main tool for designing efficient algorithms for the dynamic version of the problem, as we shall see in Section 4.3. Moreover, it is also useful for designing efficient algorithms for the all-pairs shortest paths problem, where we are asked to compute the shortest paths between all pairs of nodes in the graph [12].

2.4.1 Computing Shortest Paths Trees

In this section we survey the best known algorithms for the single-source shortest paths problem. In particular, we first consider the version of the problem where edge weights are constrained to be real nonnegative values and we briefly recall the well-known algorithm of Dijkstra [24]. We remark that adaptations of this algorithm are crucial to solving efficiently dynamic shortest paths problems. Then, we consider the Bellman-Ford algorithm for the unconstrained general version of the problem where edge weights are arbitrary real values. This algorithm will serve in Chapter 4 as a term of comparison in our experimental evaluation of dynamic algorithms for the single-source shortest paths problem. For more details about algorithms described in this section, we refer the interested reader to [12].

Nonnegative Edge Weights: Dijkstra's Algorithm

In Figure 2.2 we show the algorithm of Dijkstra for solving the single-source shortest paths problem. The algorithm takes as input a weighted directed graph $G = (V, E, w)$ where

$$w : E \rightarrow \mathcal{R}^+ \cup \{0\}$$

is a function that assigns each edge with a nonnegative real weight, and a fixed source node $s \in V$. The output of the algorithm is the distance function d_s as introduced in Definition 2.15, and a function

$$p_s : V \rightarrow V \cup \{\mathbf{nil}\}$$

such that $p_s(v) = u$ if and only if $(u, v) \in T(s)$. Intuitively, p_s represents the shortest paths tree $T(s)$ by assigning each node with its parent in $T(s)$. As the source has no parent in $T(s)$, we use for $p_s(s)$ the special value **nil**.

Notice that the algorithm is based on an initialization phase (lines 2–6), and a main computation loop (lines 7–13). In lines 2–4 we set the initial distances of all the nodes to $+\infty$ and all the parents to **nil**. Moreover, in line 5 we assign distance 0 to the source and in line 6 we initialize a set of nodes H. The main computation loop (line 7) progressively extracts from H all the nodes u that have minimum distance $d_s(u)$ (lines 8–9), and for each of them it looks for outgoing edges (u, v) towards nodes that are still in H (line 10). For each of such edges (u, v), in line 11 we check if the Bellman condition introduced in Lemma 2.13 is violated by (u, v). If this is the case, the distance $d_s(v)$ is improved (line 12) and the edge (u, v) is put in the shortest paths tree (line 13). We remark that the distances d_s, which are initially set to $+\infty$, are progressively reduced during the execution of the algorithm until, upon termination, they agree with the formulation given in Definition 2.15.

```
    Algorithm Dijkstra(G = (V, E, w), s) : p_s, d_s
1.  begin
2.     for each v ∈ V do
3.        d_s(v) ← +∞
4.        p_s(v) ← nil
5.     d_s(s) ← 0
6.     H ← V
7.     while H ≠ ∅ do
8.        let u ∈ H: d_s(u) = min_{w∈H} d_s(w)
9.        H ← H − {u}
10.       for each v ∈ H: (u, v) ∈ E do
11.          if d_s(v) > d_s(u) + w(u, v) then
12.             d_s(v) ← d_s(u) + w(u, v)
13.             p_s(v) ← u
14. end.
```

Figure 2.2: Dijkstra's algorithm

Lemma 2.16 *Let $G = (V, E, w)$ be a weighted directed graph with nonnegative weight function w and let $s \in V$ be a fixed source node. If $n = |V|$ and $m = |E|$, then the algorithm of Dijkstra correctly computes p_s and d_s in $O(m + n \log n)$ worst-case time.*

Proof. We concentrate on the running time of the algorithm; for the proof of correctness, we refer the interested reader to [12]. First notice that lines 2–6 require $O(n)$ time in the worst case. The **while** loop in line 7 is executed exactly n times while the **for** loop in line 10 is executed at most m times since no edge is scanned more than once during the whole execution of the algorithm. If we represent H by means of a Fibonacci Heap [12] and we assume that priorities of nodes are given by distances d_s, extractions of minima in lines 8–9 can be performed in $\log n$ worst-case time each, while each priority improvement in line 12 requires $O(1)$ time. Thus, the overall running time is $O(m + n \log n)$ in the worst-case. □

We remark that the assumption that edge weights are nonnegative is crucial for proving the correctness of the algorithm of Dijkstra. In the general case of arbitrary edge weights, the idea of keeping a priority queue and monotonically extracting nodes with minimum priority does not work anymore. As we will see in the next section, when negative-weight edges are present, the worst-case temporal complexity of the best known algorithm increases to $O(m \cdot n)$.

Arbitrary Edge Weights: Bellman-Ford's Algorithm

We now briefly describe the algorithm of Bellman-Ford for computing a shortest paths tree when arbitrary real edge weights are present in the graph. Figure 2.3 shows the pseudo-code of the algorithm.

The input and the output of the algorithm are the same of Dijkstra's algorithm, except for the function $w : E \to \mathcal{R}$ which allows negative-weight edges. The structure of the algorithm is essentially the same as the algorithm of Dijkstra, except for the fact that instead of keeping nodes in a priority queue H, we use a queue Q.

```
Algorithm Bellman-Ford(G = (V, E, w), s) : p_s, d_s
begin
   for each v ∈ V do
      d_s(v) ← +∞
      p_s(v) ← nil
   d_s(s) ← 0
   Q ← {s}
   while Q ≠ ∅ do
      u ← dequeue(Q)
      for each v ∈ H: (u, v) ∈ E do
         if d_s(v) > d_s(u) + w(u, v) then
            if v ∉ Q then enqueue(v, Q)
            d_s(v) ← d_s(u) + w(u, v)
            p_s(v) ← u
end.
```

Figure 2.3: Bellman-Ford's algorithm

The following lemma addresses the running time and the correctness of the algorithm.

Lemma 2.17 *Let $G = (V, E, w)$ be a weighted directed graph with weight function w and let $s \in V$ be a fixed source node. If $n = |V|$ and $m = |E|$, then the algorithm of Bellman-Ford correctly computes p_s and d_s in $O(m \cdot n)$ worst-case time.*

Proof. See [2]. □

Chapter 3

Fully Dynamic Transitive Closure

3.1 Introduction

In this chapter we present original results concerning the fully dynamic transitive closure problem. As a main contribution, we devise a new technique which allows us to reduce fully dynamic transitive closure to the problem of dynamically reevaluating polynomials over matrices when updates of variables are performed.

Based on this new technique, we devise a fully dynamic version of the static algorithm for computing the transitive closure of a directed graph presented in Section 2.3.2 and referred to as Method 1. In particular, we show how to achieve $O(n^2 \log n)$ amortized time per update and unit cost per query. Our algorithm revisits the best known algorithm for fully dynamic transitive closure presented in [52] in terms of completely different data structures, and features better initialization time.

Using the same matrix-based approach, we then devise a new deterministic algorithm which improves the best known bounds for fully dynamic transitive closure achieved in [52]. Our algorithm hinges upon the well-known equivalence between transitive closure and matrix multiplication on a closed semiring (see Section 2.3.2, Method 2). Updates are supported in $O(n^2)$ amortized time and reachability queries are answered with just one matrix lookup. We also address the case where only deletions are allowed and we show how to handle updates in $O(n)$ amortized time per operation while maintaining constant time per query; in this way we generalize to directed graphs the bounds of [49] and we improve over [43].

At last, we present the first algorithms which support both updates and queries in subquadratic time per operation, proving that it is actually possible

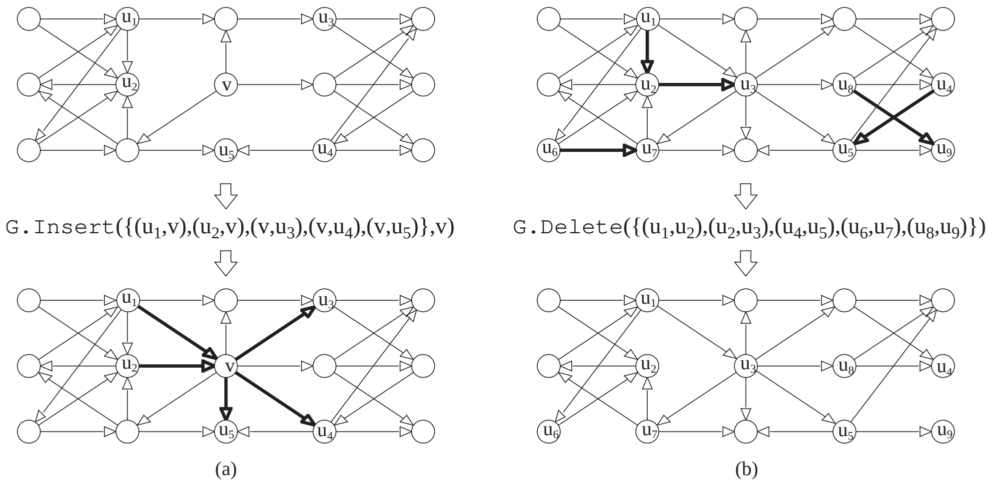


Figure 3.1: (a) `Insert` operation; (b) `Delete` operation as in Definition 3.1.

to break through the $O(n^2)$ barrier on the single-operation complexity of fully dynamic transitive closure.

In the remainder of this chapter we first formally define the fully dynamic transitive closure problem and we give preliminary definitions (Section 3.2). Sections 3.4 to 3.7 are devoted to our technical contributions: we introduce two problems on dynamic matrices (Section 3.4) and we design fast algorithms for fully dynamic transitive closure based on efficient solutions to these problems (Sections 3.5 to 3.7). Section 3.8 summarizes the most interesting techniques used throughout the chapter and the achieved results.

3.2 Statement of the Problem

In this section we give a formal definition of the fully dynamic transitive closure problem. We assume the reader to be familiar with the preliminary concepts discussed in Section 2.2 and in Section 2.3.

Definition 3.1 *Let $G = (V, E)$ be a directed graph and let $TC(G) = (V, E')$ be its transitive closure. The* Fully Dynamic Transitive Closure Problem *consists of maintaining a data structure* `G` *for graph G under an intermixed sequence $\sigma = \langle \texttt{G.Op}_1, \ldots, \texttt{G.Op}_k \rangle$ of* Initialization, Update, *and* Query *operations. Each operation $\texttt{G.Op}_j$ on data structure* `G` *can be either one of the following:*

- `G.Init`(A)*: perform the initialization operation $E \leftarrow A$, where $A \subseteq V \times V$.*

- `G.Insert`(v, I): *perform the update* $E \leftarrow E \cup \{(u, v) \mid u \in V \wedge (u, v) \in I\} \cup \{(v, u) \mid u \in V \wedge (v, u) \in I\}$, *where* $I \subseteq E$ *and* $v \in V$. *We call this kind of update a* v-CENTERED *insertion in* G.

- `G.Delete`(D): *perform the update* $E \leftarrow E - D$, *where* $D \subseteq E$.

- `G.Query`(x, y): *perform a query operation on* $TC(G)$ *and return* 1 *if* $(x, y) \in E'$ *and* 0 *otherwise.*

Few remarks are in order at this point. First, the generalized `Insert` and `Delete` updates that we consider in our operational statement of the problem have been first introduced by King in [52]. With just one operation, they are able to change the graph by adding or removing a whole set of edges, rather than a single edge (see Figure 3.1). Notice that we provide an operational (and not algorithmic) definition of operations, giving no detail about what the actual implementation should do for supporting them. Second, we consider explicitly operations of initialization of the graph G and, more generally than in the traditional definitions of dynamic problems, we allow them to appear everywhere in sequence σ.

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.

◁ ◇ ▷

In Lemma 2.9 we proved that, if $G = (V, E)$ is a directed graph and X_G is its adjacency matrix, computing the Kleene closure X_G^* of X_G is equivalent to computing the (reflexive) transitive closure $TC(G)$ of G. For this reason, in this chapter, instead of attacking directly the problem introduced in Definition 3.1, we study an equivalent problem on matrices. Before defining it formally, we provide some preliminary notation.

Definition 3.2 *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* j*-th column, respectively, and null in any other entries:*

$$I_{X,i}[x, y] = \begin{cases} X[x, y] & if\ x = i \\ 0 & otherwise \end{cases}$$

$$J_{X,i}[x, y] = \begin{cases} X[x, y] & if\ y = i \\ 0 & otherwise \end{cases}$$

Definition 3.3 *Let* X *and* Y *be* $n \times n$ *Boolean matrices. Then* $X \subseteq Y$ *if and only if* $X[x, y] = 1 \ \Rightarrow\ Y[x, y] = 1$ *for any* $x, y \in \{1, \ldots, n\}$.

We are now ready to define a dynamic version of the problem of computing the Kleene closure of a Boolean matrix.

Definition 3.4 *Let X be an $n \times n$ Boolean matrix and let X^* be its Kleene closure. We define the* Fully Dynamic Boolean Matrix Closure Problem *as the problem of maintaining a data structure* `X` *for matrix X under an intermixed sequence $\sigma = \langle \texttt{X.Op}_1, \ldots, \texttt{X.Op}_k \rangle$ of initialization, update, and query operations. Each operation $\texttt{X.Op}_j$ on data structure* `X` *can be either one of the following:*

- $\texttt{X.Init}^*(Y)$*: perform the initialization operation $X \leftarrow Y$, where Y is an $n \times n$ Boolean matrix.*
- $\texttt{X.Set}^*(i, \Delta X)$*: perform the update $X \leftarrow X + I_{\Delta X,i} + J_{\Delta X,i}$, where ΔX is an $n \times n$ Boolean matrix and $i \in \{1, \ldots, n\}$. We call this kind of update an i-*Centered *set operation on X and we call ΔX* Update Matrix*.*
- $\texttt{X.Reset}^*(\Delta X)$*: perform the update $X \leftarrow X - \Delta X$, where $\Delta X \subseteq X$ is an $n \times n$ Boolean update matrix.*
- $\texttt{X.Lookup}^*(x, y)$*: return the value of $X^*[x, y]$, where $x, y \in \{1, \ldots, n\}$.*

Algebraic operations $+$ and $-$ are performed by casting Boolean matrices into the ring of $n \times n$ matrices over the ring S_2 of integers introduced in Lemma 2.4. Integer matrices are converted back to Boolean matrices by looking at any zero value as 0 and any nonzero value as 1.

Notice that $\texttt{Set}^*$ is allowed to modify only the i-th row and the i-th column of X, while $\texttt{Reset}^*$ and $\texttt{Init}^*$ can modify any entries of X.

It is also interesting to observe that in our dynamic setting, in order to define `Reset` updates which use the $-$ operator to flip matrix entries from 1 to 0, we require operations on Boolean matrices to be performed in the broader algebraic structure of rings. For the static version of the problem discussed in Section 2.3 rings instead of closed semirings were used as well, but only for obtaining efficient algorithms and not for defining the problem itself.

We stress the strong correlation between Definition 3.4 and Definition 3.1: if G is a graph and X is its adjacency matrix, operations $\texttt{X.Init}^*$, $\texttt{X.Set}^*$, $\texttt{X.Reset}^*$, and $\texttt{X.Lookup}^*$ are equivalent to operations `G.Init`, `G.Insert`, `G.Delete`, and `G.Query`, respectively.

3.3 Overview of Our Approach

In this section we give an overview of the new ideas presented in this paper, discussing the most significant aspects of our techniques.

The technical contributions of this chapter span over four sections: Section 3.4 presents two problems on dynamic matrices that will be central to designing three efficient fully dynamic algorithms for transitive closure in Section 3.5, Section 3.6 and Section 3.7, respectively.

Our approach consists of reducing fully dynamic transitive closure to the problem of maintaining efficiently polynomials over matrices subject to updates of their variables. In particular, we focus on the equivalent problem of fully dynamic Kleene closure and we show that efficient data structures for it can be realized using efficient data structures for maintaining polynomials over matrices.

Suppose that we have a polynomial over Boolean matrices, e.g., $P(X,Y,Z,W) = X + YZ^2W$, where matrices X, Y, Z and W are its variables. The value $P(X,Y,Z,W)$ of the polynomial can be computed via sum and multiplication of matrices X, Y, Z and W in $O(n^{2.38})$ (see Section 2.3.1). Now, what kind of modifications can we perform on a variable, e.g., variable Z, so as to have the chance of updating the value of $P(X,Y,Z,W)$ in less than $O(n^{2.38})$ time?

In Section 3.4.1 we show a data structure that allows us to reevaluate correctly $P(X,Y,Z,W)$ in just $O(n^2)$ amortized time after flipping to 1 any entries of Z that were 0, provided they lie on a row or on a column (`SetRow` or `SetCol` operation), of after flipping to 0 *any* entries of Z that were 1 (`Reset` operation). This seems a step forward, but are this kind of updates of variables powerful enough to be useful our original problem of fully dynamic transitive closure? Unfortunately, the answer is no. Actually, we also require the more general `Set` operation of flipping to 1 *any* entries of Z that were 0. Now, if we want to have our polynomial always up to date after each variable change of this kind, it seems that there is no way of doing any better than recomputing everything from scratch.

So let us lower our expectations on our data structure for maintaining P, and tolerate errors. In exchange, our data structure must support efficiently the general `Set` operation. The term "errors" here means that we maintain a "relaxed" version of the correct value of the polynomial, where some 0's may be incorrect. The only important property that we require is that any 1's that appear in the correct value of the polynomial after performing a `SetRow` or `SetCol` operation must also appear in the relaxed value that we maintain. This allows us to support any `Set` operation efficiently in a lazy fashion (so in the following we call it `LazySet`) and is powerful enough for our original problem of fully dynamic transitive closure.

Actually, doing things lazily while maintaining the desired properties in our data structure for polynomials is the major technical difficulty in Section 3.4.1. Sections 3.5 and 3.6 then show two methods to solve the fully dynamic Boolean matrix closure problem by using polynomials of Boolean matrices as if they were building blocks. The second method yields the fastest known algorithm for fully dynamic transitive closure with constant query time. If we give up maintaining polynomials of degree > 1, using a surprisingly simple lazy technique we can even support certain kinds of variable updates in subquadratic worst-case time per operation (see Section 3.4.2). This turns out to be once again applicable to fully dynamic transitive closure, yielding the first subquadratic algorithms known so far for the problem (see Section 3.7).

3.4 Dynamic Matrices

In this section we consider two problems on dynamic matrices and we devise fast algorithms for solving them. As we already stated, these problems will be central to designing efficient algorithms for the fully dynamic Boolean matrix closure problem introduced in Definition 3.4. In more detail, in Section 3.4.1 we address the problem of reevaluating polynomials over Boolean matrices under modifications of their variables. We propose a data structure for maintaining efficiently the special class of polynomials of degree 2 consisting of single products of Boolean matrices. We show then how to use this data structure for solving the more general problem on arbitrary polynomials. In Section 3.4.2 we study the problem of finding an implicit representation for integer matrices that makes it possible to update as many as $\Omega(n^2)$ entries per operation in $o(n^2)$ worst-case time at the price of increasing the lookup time required to read a single entry.

3.4.1 Dynamic Polynomials over Boolean Matrices

We now study the problem of maintaining the value of polynomials over Boolean matrices under updates of their variables. We define these updates so that they can be useful later on for our original problem of dynamic Boolean matrix closure. We first need some preliminary definitions.

Definition 3.5 *Let X be a data structure. We denote by X_i the value of X at* Time *i, i.e., the value of X after the i-th operation in a sequence of operations that modify X. By convention, we assume that at time 0 any numerical value in X is zero. In particular, if X is a Boolean matrix, $X_0 = 0_n$.*

In the following definition we formally introduce our first problem on dynamic matrices.

Definition 3.6 *Let $\mathcal{B}_n$ be the set of $n \times n$ Boolean matrices and let*

$$P = \sum_{a=1}^{h} T_a$$

be a polynomial[1] *with h terms defined over $\mathcal{B}_n$, where each*

$$T_a = \prod_{b=1}^{k} X_b^a$$

has degree exactly k and variables $X_b^a \in \mathcal{B}_n$ are distinct. We consider the problem of maintaining a data structure `P` *for the polynomial P under an intermixed sequence $\sigma = \langle \texttt{P.Op}_1, \ldots, \texttt{P.Op}_l \rangle$ of initialization, update, and query operations. Each operation $\texttt{P.Op}_j$ on the data structure* `P` *can be either one of the following:*

- `P.Init`$(Z_1^1, \ldots, Z_k^h)$: *perform the initialization $X_b^a \leftarrow Z_b^a$ of the variables of polynomial P, where each Z_b^a is an $n \times n$ Boolean matrix.*

- `P.SetRow`$(i, \Delta X, X_b^a)$: *perform the row update operation $X_b^a \leftarrow X_b^a + I_{\Delta X, i}$, where ΔX is an $n \times n$ Boolean update matrix. The operation sets to 1 the entries in the i-th row of variable X_b^a of polynomial P as specified by matrix ΔX.*

- `P.SetCol`$(i, \Delta X, X_b^a)$: *perform the column update operation $X_b^a \leftarrow X_b^a + J_{\Delta X, i}$, where ΔX is an $n \times n$ Boolean update matrix. The operation sets to 1 the entries in the i-th column of variable X_b^a of polynomial P as specified by matrix ΔX.*

- `P.LazySet`$(\Delta X, X_b^a)$: *perform the update operation $X_b^a \leftarrow X_b^a + \Delta X$, where ΔX is an $n \times n$ Boolean update matrix. The operation sets to 1 the entries of variable X_b^a of polynomial P as specified by matrix ΔX.*

- `P.Reset`$(\Delta X, X_b^a)$: *perform the update operation $X_b^a \leftarrow X_b^a - \Delta X$, where ΔX is an $n \times n$ Boolean update matrix such that $\Delta X \subseteq X_b^a$. The operation resets to 0 the entries of variable X_b^a of polynomial P as specified by matrix ΔX.*

[1] In the following, we omit specifying explicitly the dependence of a polynomial on its variables, and we denote by P both the function $P(X_1, \ldots, X_k)$ and the value of this function for fixed values of $X_1, \ldots, X_k$, assuming that the correct interpretation is clear from the context.

- `P.Lookup()`: *answer a query about the value of P by returning an $n \times n$ Boolean matrix Y_j, such that $M_j \subseteq Y_j \subseteq P_j$, where M is an $n \times n$ Boolean matrix whose value at time j is defined as follows:*

$$M_j = \sum_{\substack{1 \le i \le j :\\ \mathtt{Op}_i \neq \mathtt{LazySet}}} (P_i - P_{i-1}).$$

and P_i is the value of polynomial P at time i. According to this definition, we allow the answer about the value of P to be affected by one-sided error.

Algebraic operations $+$ and $-$ are performed by casting Boolean matrices into the ring of $n \times n$ matrices over the ring S_2 of integers introduced in Lemma 2.4. Integer matrices are converted back to Boolean matrices by looking at any zero value as 0 *and any nonzero value as* 1.

`SetRow` and `SetCol` are allowed to modify only the i-th row and the i-th column of variable X_b^a, respectively, while `LazySet`, `Reset` and `Init` can modify any entries of X_b^a. It is crucial to observe that in the operational definition of `Lookup` we allow one-sided errors in answering queries on the value of P. In particular, in the answer there have to be no incorrect 1's and the error must be bounded: `Lookup` has to return a matrix Y that contains *at least* the 1's in M, and *no more* than the 1's in P. As we will see later on, this operational definition simplifies the task of designing efficient implementations of the operations and is still powerful enough to be useful for our original problem of dynamic Boolean matrix closure.

The following lemma shows that the presence of errors is related to the presence of `LazySet` operations in sequence σ. In particular, it shows that, if no `LazySet` operation is performed, then `Lookup` makes no errors and returns the correct value of polynomial P.

Lemma 3.1 *Let P be a polynomial and let $\sigma = \langle \mathtt{P.Op}_1, \ldots, \mathtt{P.Op}_k \rangle$ be a sequence of operations on P. If $\mathtt{Op}_i \neq \mathtt{LazySet}$ for all $1 \le i \le j \le k$, then $M_j = P_j$.*

Proof. The proof easily follows by telescoping the sum that defines M_j: $M_j = P_j - P_{j-1} + P_{j-1} - P_{j-2} + \cdots + P_2 - P_1 + P_1 - P_0 = P_j - P_0 = P_j$. □

Errors in the answers given by `Lookup` may appear as soon as `LazySet` operations are performed in sequence σ. To explain how M is defined mathematically, notice that $M_0 = 0_n$ by Definition 3.5 and M sums up all the changes that the value of P has undergone up to the j-th operation, except for the changes due to `LazySet` operations, which are ignored. This means that, if any entry $P[x,y]$ flips from 0 to 1 or vice-versa due to an operation $\texttt{Op}_j$ different from `LazySet`, so does $M[x,y]$ and thus $Y[x,y]$.

As a side note, we remark that it is straightforward to extend the results of this section to the general class of polynomials with terms of different degrees and multiple occurrences of the same variable.

◁ ◇ ▷

We now focus on the problem of implementing the operations introduced in Definition 3.6. A simple-minded implementation of the operations on `P` is the following:

- Maintain variables X_b^a, terms T_a, and a matrix Y that contains the value of the polynomial.
- Recompute from scratch T_a and the value of $Y = P = T_1 + \cdots + T_h$ after each `Init`, `SetRow`, `SetCol` and `Reset` that change X_b^a.
- Do *nothing* after a `LazySet` operation, except for updating X_b^a. This means that Y may no longer be equal to P after the operation.
- Let `Lookup` return the maintained value of Y.

It is easy to verify that at any time j, i.e., after the j-th operation, $\texttt{Op}_j \neq$`LazySet` implies $Y = P$ and $\texttt{Op}_j =$`LazySet` implies $Y = M$. In other words, the value Y returned by `Lookup` oscillates between the exact value P of the polynomial and the value M obtained without considering `LazySet` operations.

With the simple-minded implementation above, we can support `Init` in $O(h \cdot k \cdot n^\omega + h \cdot n^2)$ time, `SetRow` and `SetCol` in $O(k \cdot n^\omega)$ time, `Reset` in $O(k \cdot n^\omega + h \cdot n^2)$ time, and `Lookup` and `LazySet` in $O(n^2)$ time.

The remainder of this section provides more efficient solutions for the problem. In particular, we present a data structure that supports `Lookup` and `LazySet` operations in $O(n^2)$ worst-case time, `SetRow`, `SetCol` and `Reset` operations in $O(k \cdot n^2)$ amortized time, and `Init` operations in $O(h \cdot k \cdot n^\omega + h \cdot n^2)$ worst-case time. The space used is $O(h \cdot k^2 \cdot n^2)$. Before considering the general case where polynomials have arbitrary degree k, we focus on the special class of polynomials where $k = 2$.

Data Structure for Polynomials of Degree $k = 2$

We define a data structure for P that allows us to maintain explicitly the value Y_j of the matrix Y at any time j during a sequence $\langle \texttt{P.Op}_1, \ldots, \texttt{P.Op}_l \rangle$ of operations. This makes it possible to perform `Lookup` operations in optimal quadratic time. We avoid recomputing from scratch the value of Y after each update as in the simple-minded method, and we propose efficient techniques for propagating to Y the effects of changes of variables X_b^a due to `SetRow`, `SetCol` and `Reset` operations. In case of `LazySet`, we only need to update the affected variables, leaving the other elements in the data structure unaffected. This, of course, implies that after a `LazySet` at time j, the maintained value Y_j will be clearly not synchronized with the correct value P_j of the polynomial. Most technical difficulties of this section come just from this lazy maintenance of Y_j.

Our data structure for representing a polynomial of degree 2 of the form $P = X_1^1 \cdot X_2^1 + \ldots + X_1^h \cdot X_2^h$ is presented below.

Data Structure 3.1 *We maintain the following elementary data structures with $O(h \cdot n^2)$ space:*

1. *$2h$ matrices X_1^a and X_2^a for $1 \le a \le h$;*
2. *h integer matrices $Prod_1, \ldots, Prod_h$ such that $Prod_a$ maintains a "lazy" count of the number of witnesses of the product $T_a = X_1^a \cdot X_2^a$.*
3. *an integer matrix S such that $S[x,y] = |\{a : Prod_a[x,y] > 0\}|$. We assume that $Y_j[x,y] = 1 \Leftrightarrow S[x,y] > 0$.*
4. *$2h$ integer matrices $LastFlip_X$, one for each matrix $X = X_b^a$. For any entry $X[x,y] = 1$, $LastFlip_X[x,y]$ is the time of the most recent operation that caused $X[x,y]$ to flip from 0 to 1. More formally:*

$$LastFlip_{X_j}[x,y] = \max_{1 \le t \le j} \{t \mid X_t[x,y] - X_{t-1}[x,y] = 1\}$$

   if $X_j[x,y] = 1$, and is undefined otherwise;

5. *$2h$ integer vectors $LastRow_X$, one for each matrix $X = X_b^a$. $LastRow_X[i]$ is the time of the last `Init` or `SetRow` operation on the i-th row of X, and zero if no such operation was ever performed. More formally:*

$$LastRow_{X_j}[i] = \max_{1 \le t \le j} \{0,\ t \mid \texttt{Op}_t = \texttt{Init}(\ldots) \vee \texttt{Op}_t = \texttt{SetRow}(i, \Delta X, X)\}$$

   We also maintain similar vectors $LastCol_X$;

6. *a counter $Time$ of the number of performed operations;*

Before getting into the full details of our implementation of operations, we give an overview of the main ideas. We consider how the various operations should affect the data structure. In particular, we suppose that an operation changes any entries of variable X_1^a in a term $T_a = X_1^a \cdot X_2^a$, and we define what our implementation should do on matrix $Prod_a$:

`SetRow/SetCol`: if some entry $X_1^a[x, y]$ is flipping to 1, then y becomes a witness in the product $X_1^a \cdot X_2^a$ for any pair x, z such that $X_2^a[y, z] = 1$. Then we should put y in the count $Prod_a[x, z]$, if it is not already counted. Moreover, if some entry $X_1^a[x, y]$ was already 1, but for some pair x, z the index y is not counted in $Prod_a[x, z]$, then we should put y in the count $Prod_a[x, z]$.

`LazySet`: if some entry $X_1^a[x, y]$ is flipping to 1, then y becomes a witness for any pair x, z such that $X_2^a[y, z] = 1$. Then we should put y in the count $Prod_a[x, z]$, if it is not already counted, *but we do not do this.*

`Reset`: if some entry $X_1^a[x, y]$ is flipping to 0, then y is no longer a witness for all pairs x, z such that $X_2^a[y, z] = 1$. Then we should remove y from the count $Prod_a[x, z]$, if it is currently counted.

Note that after performing `LazySet` there may be triples (x, y, z) such that both $X_1^a[x, y] = 1$ and $X_2^a[y, z] = 1$, but y is not counted in $Prod_a[x, z]$. Now the problem is: is there any property that we can exploit to tell if a given y is counted or not in $Prod_a[x, z]$ whenever both $X_1^a[x, y] = 1$ and $X_2^a[y, z] = 1$?

We introduce a predicate $\mathcal{P}_a(x, y, z)$, $1 \le x, y, z \le n$, such that $\mathcal{P}_a(x, y, z)$ is true if and only if the last time any of the two entries $X_1^a[x, y]$ and $X_2^a[y, z]$ flipped from 0 to 1 is before the time of the last update operation on the x-th row or the y-th column of X_1^a and the time of the last update operation on the y-th row or the z-th column of X_2^a. In short:

$$\mathcal{P}_a(x, y, z) \;:=\; \max\{LastFlip_{X_1^a}[x, y], LastFlip_{X_2^a}[y, z]\} \le \\ \max\{LastRow_{X_1^a}[x], LastCol_{X_1^a}[y], LastRow_{X_2^a}[y], LastCol_{X_2^a}[z]\}$$

The property $\mathcal{P}_a$ answers our previous question and allows it to define the following invariant that we maintain in our data structure. We remark that we do not need to maintain $\mathcal{P}_a$ explicitly in our data structure as it can be computed on demand in constant time by accessing $LastFlip$ and $LastRow$.

Invariant 3.1 *For any term $T_a = X_1^a \cdot X_2^a$ in polynomial P, at any time during a sequence of operations σ, the following invariant holds for any pair of indices x, z:*

$$Prod_a[x, z] = |\{y \;:\; X_1^a[x, y] = 1 \;\wedge\; X_2^a[y, z] = 1 \;\wedge\; \mathcal{P}_a(x, y, z)\}|$$

According to Invariant 3.1, it is clear that the value of each entry $Prod_a[x, z]$ is a "lazy" count of the number of witnesses of the Boolean matrix product $T_a[x, z] = (X_1^a \cdot X_2^a)[x, z]$. Notice that, since $T_a[x, z] = 1 \Leftrightarrow \exists y \;:\; X_1^a[x, y] = 1 \;\wedge\; X_2^a[y, z] = 1$, we have that $Prod_a[x, z] > 0 \;\Rightarrow\; T_a[x, z] = 1$. Thus, we may think of $\mathcal{P}_a$ as a "relaxation" property.

We implement the operations introduced in Definition 3.6 as described next, assuming that the operation $Time \leftarrow Time + 1$ is performed just before each operation:

`Init`

procedure `Init`$(Z_1^1, Z_2^1, \ldots, Z_1^h, Z_2^h)$
begin
    for each a **do** $X_1^a \leftarrow Z_1^a$; $X_2^a \leftarrow Z_2^a$
    { initialize members 2–5 of Data Structure 3.1 }
end

`Init` assigns the value of variables X_1^a and X_2^a and initializes elements 2–5 of Data Structure 3.1. In particular, $LastFlip_X[x, y]$ is set to $Time$ for any $X[x, y] = 1$ and the same is done for $LastRow[i]$ and $LastCol[i]$ for any i. $Prod_a$ is initialized by computing the product $X_1^a \cdot X_2^a$ in the ring of integers, i.e., looking at X_b^a as integer matrices.

`Lookup`

function `Lookup`()
begin
    return Y s.t. $Y[x, y] = 1 \;\Leftrightarrow\; S[x, y] > 0$
end

`Lookup` simply returns a binarized version Y of matrix S defined in Data Structure 3.1.

`SetRow`

procedure `SetRow`$(i, \Delta X, X_b^a)$
1. **begin**
2. $X_b^a \leftarrow X_b^a + I_{\Delta X, i}$
3.   {update $LastFlip_{X_b^a}$}
4.   **if** $b = 1$ **then**
5.   **for each** $x : X_1^a[i, x] = 1$ **do**
6.   **for each** $y : X_2^a[x, y] = 1$ **do**
7.   **if not** $\mathcal{P}_a(i, x, y)$ **then**
8. $Prod_a[i, y] \leftarrow Prod_a[i, y] + 1$
9.   **if** $Prod_a[i, y] = 1$ **then** $S[i, y] \leftarrow S[i, y] + 1$
10.   **else**{$b = 2$: similar to `P.SetCol`$(i, \Delta X, X_1^a)$}
11. $LastRow_{X_b^a}[i] \leftarrow Time$
12. **end**

After performing an i-centered insertion in X_b^a on line 2 and after updating $LastFlip_{X_b^a}$ on line 3, `SetRow` checks on lines 5–7 for any triple (i, x, y) such that the property $\mathcal{P}_a(i, x, y)$ is still not satisfied, but will be satisfied thanks to line 11, and increases $Prod_a$ and S accordingly (lines 8–9).

`SetCol`

procedure `SetCol`$(i, \Delta X, X_b^a)$
begin
  $X_b^a \leftarrow X_b^a + J_{\Delta X, i}$
  {update $\mathit{LastFlip}_{X_b^a}$}
  if $b = 1$ **then**
    for each $x : X_1^a[x,i] = 1$ **do**
      for each $y : X_2^a[i,y] = 1$ **do**
        if not $\mathcal{P}_a(x,i,y)$ **then**
          $\mathit{Prod}_a[x,y] \leftarrow \mathit{Prod}_a[x,y] + 1$
          if $\mathit{Prod}_a[x,y] = 1$ **then** $S[x,y] \leftarrow S[x,y] + 1$
  else {$b = 2$: similar to `P.SetRow`$(i, \Delta X, X_1^a)$}
  $\mathit{LastCol}_{X_1^a}[i] \leftarrow \mathit{Time}$
end

Similar to `SetRow`.

`LazySet`

procedure `LazySet`$(\Delta X, X_b^a)$
begin
  $X_b^a \leftarrow X_b^a + \Delta X$
  {update $\mathit{LastFlip}_{X_b^a}$}
end

`LazySet` simply sets to 1 any entries in X_b^a and updates $\mathit{LastFlip}_{X_b^a}$. We remark that no other object in the data structure is changed.

`Reset`

procedure `Reset`$(\Delta X, X_b^a)$
1. **begin**
2.   **if** $b = 1$ **then**
3.   **for each** $x, y : \Delta X[x,y] = 1$ **do**
4.   **if** $\max\{\mathit{LastRow}_{X_1^a}[x], \mathit{LastCol}_{X_1^a}[y]\} \geq \mathit{LastFlip}_{X_1^a}[x,y]$ **then**
5.   **for each** $z : X_1^a[y,z] = 1$ **do**
6.   **if** $\mathcal{P}_a(x,y,z)$ **then**
7. $\mathit{Prod}_a[x,z] \leftarrow \mathit{Prod}_a[x,z] - 1$
8.   **if** $\mathit{Prod}_a[x,z] = 0$ **then** $S[x,z] \leftarrow S[x,z] - 1$
9.   **else** { here $\max\{\mathit{LastRow}_{X_1^a}[x], \mathit{LastCol}_{X_1^a}[y]\} < \mathit{LastFlip}_{X_1^a}[x,y]$ }
10.   **for each** $z : X_1^a[y,z] = 1 \wedge \mathit{LastCol}_{X_2^a}[z] > \mathit{LastFlip}_{X_1^a}[x,y]$ **do**
11.   **if** $\mathcal{P}_a(x,y,z)$ **then**
12. $\mathit{Prod}_a[x,z] \leftarrow \mathit{Prod}_a[x,z] - 1$
13.   **if** $\mathit{Prod}_a[x,z] = 0$ **then** $S[x,z] \leftarrow S[x,z] - 1$
14.   **else** {$b = 2$ similar to $b = 1$}
15. $X_b^a \leftarrow X_b^a - \Delta X$
16. **end**

In lines 2-14, using $\mathit{LastRow}_{X_b^a}$, $\mathit{LastCol}_{X_b^a}$, and $\mathit{LastFlip}_{X_b^a}$, `Reset` updates Prod_a and S so as to maintain Invariant 3.1. Namely, for each reset entry

(x, y) specified by ΔX (line 3), it looks for triples (x, y, z) such that $\mathcal{P}(x, y, z)$ is going to be no more satisfied due to the reset of $X_b^a[x, y]$ to be performed (lines 5–6 and lines 10–11); $Prod_a$ and S are adjusted accordingly (lines 7–8 and lines 12-13).

The distinction between the two cases $\max\{LastRow_{X_1^a}[x], LastCol_{X_1^a}[y]\} \geq LastFlip_{X_1^a[x,y]}$ and $\max\{LastRow_{X_1^a}[x], LastCol_{X_1^a}[y]\} < LastFlip_{X_1^a[x,y]}$ in line 4 and in line 9, respectively, is important to achieve fast running times as it will be discussed in the proof of Theorem 3.2. Here we only point out that if the test in line 4 succeeds, then we can scan any z s.t. $X_1^a[y, z] = 1$ without affecting the running time. If this is not the case, then we need to process *only* indices z such that the test $LastCol_{X_2^a}[y, z] > LastFlip_{X_1^a}[x, y]$ is satisfied, and avoid scanning other indices. For this reason line 10 must be implemented very carefully by maintaining indices z in a list and by using a move-to-front strategy that brings index z to the front of the list as any operation `Init`$(\ldots)$, `SetRow`$(z, \ldots)$ or `SetCol`$(z, \ldots)$ is performed on z. In this way indices are sorted according to the dates of operations on them.

As last step, `Reset` resets the entries of X_b^a as specified by ΔX (line 15).

◁ ◇ ▷

The correctness of our implementation of operations `Init`, `SetRow`, `SetCol`, `LazySet`, `Reset` and `Lookup` is discussed in the following theorem.

Theorem 3.1 *At any time j,* `Lookup` *returns a matrix Y_j that satisfies the relation $M_j \subseteq Y_j \subseteq P_j$ as in Definition 3.6.*

Proof. We first remind that Y is the binarized version of S as follows from the implementation of `Lookup`.

To prove that $Y \subseteq P$, observe that `SetRow` increases $Prod_a[i, y]$ (line 8), and possibly S (line 9), only if both $X_1^a[i, x] = 1$ and $X_2^a[x, y] = 1$: this implies that $T_a[i, y] = 1$ and $P[i, y] = 1$.

To prove that $M \subseteq Y$, notice that at time j *after* performing an operation `Op`$_j$=`SetRow`$(i, \Delta X, X_b^a)$ on the i-th row of X_1^a, $\mathcal{P}(i, x, y)$ is satisfied for any triple (i, x, y) such that $X_1^a[i, x] = 1$ and $X_2^a[x, y] = 1$ thanks to the operation $LastRow_{X_b^a}[i] \leftarrow Time$ (line 11). For X_2^a the proof is analogous. Now, all such triples (i, x, y) are enumerated by `SetRow` (lines 5–6): for each of them such that $\mathcal{P}_a(i, x, y)$ was false at time $j - 1$, $Prod_a[i, y]$ is increased and possibly $S[i, y]$ is increased as well (lines 7-9). If $P[i, y]$ flips from 0 to 1, then necessarily $X_1^a[i, x]$ flips from 0 to 1 for some x, and then, as stated above w.r.t. $\mathcal{P}_a$, $Prod_a[i, y]$ gets increased. Thus, recalling that Y is the binarized version of S, we have for any y:

$$P_j[i, y] - P_{j-1}[i, y] = 1 \;\Rightarrow\; Y_j[i, y] - Y_{j-1}[i, y] = 1.$$

From the definition of M in Definition 3.6 we have that:

$$M_j[i,y] - M_{j-1}[i,y] = 1 \;\Leftrightarrow\; P_j[i,y] - P_{j-1}[i,y] = 1.$$

This proves the relation $M \subseteq Y$. A similar argument is valid also for `SetCol`, while `LazySet` does not affect S at all.

To complete the proof we remark that $Y = P$ just after any `Init` operation and that `Reset` leaves the data structure as if reset entries were never set to 1. Indeed, `Reset` can be viewed as a sort of "undo" procedure that cancels the effects of previous `SetRow`, `SetCol` or `Init` operations. □

We now analyze the complexity of our implementation of the operations on polynomials.

Theorem 3.2 *Any* `Lookup`*,* `SetRow`*,* `SetCol` *and* `LazySet` *operation requires $O(n^2)$ time in the worst case. Any* `Init` *requires $O(h \cdot n^\omega + h \cdot n^2)$ worst-case time, where ω is the exponent of matrix multiplication. The cost of any* `Reset` *operation can be charged to previous* `SetRow`*,* `SetCol` *and* `Init` *operations. The maximum cost charged to each* `Init` *is $O(h \cdot n^3)$. The space required is $O(h \cdot n^2)$.*

Proof. It is straightforward to see from the pseudocode of the operations that any `SetRow`, `SetCol` and `LazySet` operation requires $O(n^2)$ time in the worst case.

`Init` takes $O(h \cdot n^\omega + h \cdot n^2)$ in the worst case: in more detail, each $Prod_a$ can be directly computed via matrix multiplication and any other initialization step requires no more than $O(n^2)$ worst-case time.

To prove that the cost of any `Reset` operation can be charged to previous `SetRow`, `SetCol` and `Init` operations, we use a potential function

$$\Phi_a = \sum_{x,y} Prod_a[x,y]$$

associated to each term T_a of the polynomial. From the relation:

$$Prod_a[x,z] = |\{y \;:\; X_1^a[x,y] = 1 \;\wedge\; X_2^a[y,z] = 1 \;\wedge\; \mathcal{P}_a(x,y,z)\}|$$

given in Invariant 3.1, it follows that $0 \le Prod_a[x,z] \le n$ for all x, z. Thus, $0 \le \Phi_a \le n^3$.

Now, observe that `SetRow` increases Φ_a by at most n^2 units per operation, while `Init` increases Φ_a by at most n^3 units per operation. Note that *LazySet* does not affect Φ_a. We can finally address the case of `Reset` operations. Consider the distinction between the two cases

$$\max\{LastRow_{X_1^a}[x], LastCol_{X_1^a}[y]\} \ge LastFlip_{X_1^a[x,y]}$$

in line 4 and

$$\max\{LastRow_{X_1^a}[x], LastCol_{X_1^a}[y]\} < LastFlip_{X_1^a[x,y]}$$

in line 9. In the first case, we can charge the cost of processing any triple (x, y, z) to some previous operation on the x-th row of X_1^a or to some previous operation on the y-th column of X_1^a; in the second case, we consider only those (x, y, z) for which some operation on the z-th column of $X_2^a[y, z]$ was performed *after* both $X_1^a[x, y]$ and $X_2^a[y, z]$ were set to 1. In both cases, any `Reset` operation decreases Φ_a by at most n units for each reset entry of X_b^a, and this can be charged to previous operations which increased Φ_a. □

The complex statement of the charging mechanism encompasses the dynamics of our data structure. In particular, we allow `Reset` operations to charge up to a $O(n^3)$ cost to a single `Init` operation. Thus, in an arbitrary mixed sequence with any number of `Init`, `Reset` takes $O(n^3)$ amortized time per update. If, however, we allow `Init` operations to appear in σ only every $\Omega(n)$ `Reset` operations, the bound for `Reset` drops down to $O(n^2)$ amortized time per operation.

As a consequence of Theorem 3.2, we have the following corollaries that refine the analysis of the running time of `Reset` operations.

Corollary 3.1 *If we perform just one* `Init` *operation in a sequence σ of length $\Omega(n)$, or more generally one* `Init` *operation every $\Omega(n)$* `Reset` *operations, then the amortized cost of* `Reset` *is $O(n^2)$ per operation.*

Corollary 3.2 *If we perform just one* `Init` *operation in a sequence σ of length $\Omega(n^2)$, or more generally one* `Init` *operation every $\Omega(n^2)$* `Reset` *operations, and no operations* `SetRow` *and* `SetCol`*, then the amortized cost of* `Reset` *is $O(n)$ per operation.*

In the following, we show how to extend the previous techniques in order to deal with the general case of polynomials of degree $k > 2$.

Data Structure for Polynomials of Degree $k > 2$

To support terms of degree $k > 2$ in P, we consider an equivalent representation $\widehat{P}$ of P such that the degree of each term is 2. This allows us to maintain a data structure for $\widehat{P}$ with the operations defined in the previous paragraph.

Lemma 3.2 *Consider a polynomial*

$$P = \sum_{a=1}^{h} T_a = \sum_{a=1}^{h} X_1^a \cdots X_k^a$$

with h terms where each term T_a has degree exactly k and variables X_b^a are Boolean matrices. Let $\widehat{P}$ be the polynomial over Boolean matrices of degree 2 defined as

$$\widehat{P} = \sum_{a=1}^{h} \sum_{b=0}^{k} L_{b,b-1}^a \cdot R_{b,k-b-1}^a$$

where $L_{b,j}^a$ and $R_{b,j}^a$ are polynomials over Boolean matrices of degree ≤ 2 defined as

$$L_{b,j}^a = \begin{cases} X_{b-j}^a \cdot L_{b,j-1}^a & if\ j \in [0, b-1] \\ I_n & if\ j = -1 \end{cases}$$

$$R_{b,j}^a = \begin{cases} R_{b,j-1}^a \cdot X_{b+1+j}^a & if\ j \in [0, k-b-1] \\ I_n & if\ j = -1 \end{cases}$$

Then $P = \widehat{P}$.

Proof. To prove the claim, it suffices to check that

$$T_a = \sum_{b=0}^{k} L_{b,b-1}^a \cdot R_{b,k-b-1}^a$$

Unrolling the recursion for $L_{b,b-1}^a$, we obtain:

$$L_{b,b-1}^a = X_1^a \cdot L_{b,b-2}^a = X_1^a \cdot X_2^a \cdot L_{b,b-3}^a = \cdots = X_1^a \cdot X_2^a \cdots X_b^a \cdot I_n$$

Likewise, $R_{b,k-b-1}^a = I_n \cdot X_{b+1}^a \cdots X_k^a$ holds. Thus, by idempotence of the closed semiring of Boolean matrices, we finally have:

$$\sum_{b=0}^{k} L_{b,b-1}^a \cdot R_{b,k-b-1}^a = \sum_{b=0}^{k} X_1^a \cdots X_b^a \cdot X_{b+1}^a \cdots X_k^a = X_1^a \cdots X_k^a = T_a.$$

$\square$

Since $\widehat{P}$, $L_{b,j}^a$ and $R_{b,j}^a$ are all polynomials of degree ≤ 2, they can be represented and maintained efficiently by means of instances of Data Structure 3.1. Our data structure for maintaining polynomials of degree > 2 is presented below:

Data Structure 3.2 *We maintain explicitly the k^2 polynomials $L_{b,j}^a$ and $R_{b,j}^a$ with instances of Data Structure 3.1. We also maintain polynomial $\widehat{P}$ with an instance Y of Data Structure 3.1.*

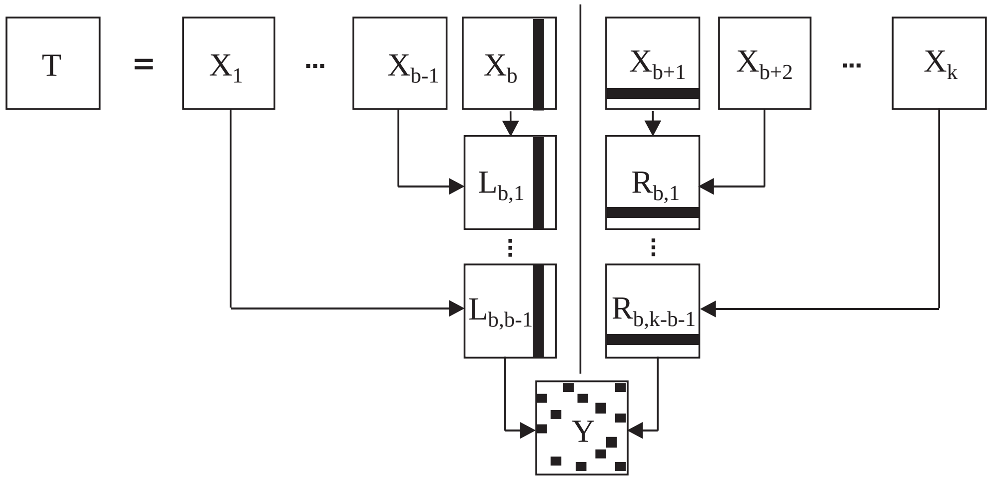


Figure 3.2: Revealing new 1's in Y while updating a term T of degree k by means of a `SetCol` operation on variable X_b or by means of a `SetRow` operation on variable X_{b+1}.

We now consider how to support `SetRow`, `SetCol`, `LazySet`, `Reset`, `Init` and `Lookup` in the case of arbitrary degree. We denote by $\texttt{SetRow}_{k=2}$ and $\texttt{SetCol}_{k=2}$ the versions of `SetRow` and `SetCol` implemented for $k = 2$.

`SetCol, SetRow`

procedure `SetCol`$(i, \Delta X, X_b^a)$
1. **begin**
2. $X_b^a \leftarrow X_b^a + J_{\Delta X,i}$
3. **for** $j \leftarrow 1$ **to** $b-1$ **do**
4. $\texttt{L}^{\texttt{a}}_{\texttt{b,j}}.\texttt{SetCol}_{k=2}(i, \Delta L^a_{b,j-1}, L^a_{b,j-1})$ { it holds $L^a_{b,j} = X^a_{b-j} \cdot L^a_{b,j-1}$ }
5. **for** $j \leftarrow 1$ **to** $k-b-1$ **do**
6. $\texttt{R}^{\texttt{a}}_{\texttt{b,j}}.\texttt{SetRow}_{k=2}(i, \Delta R^a_{b,j-1}, R^a_{b,j-1})$ { it holds $R^a_{b,j} = R^a_{b,j-1} \cdot X^a_{b+1+j}$ }
7. $\texttt{Y.SetCol}_{k=2}(i, \Delta L^a_{b,b-1}, L^a_{b,b-1})$
8. $\texttt{Y.SetRow}_{k=2}(i, \Delta R^a_{b,k-b-1}, R^a_{b,k-b-1})$
9. **for** $j \leftarrow 1$ **to** $k-b$ **do** $\texttt{L}^{\texttt{a}}_{\texttt{b+j,j}}.\texttt{LazySet}(\Delta X^a_b, X^a_b)$
10. **for** $j \leftarrow 1$ **to** $b-2$ **do** $\texttt{R}^{\texttt{a}}_{\texttt{b-j-1,j}}.\texttt{LazySet}(\Delta X^a_b, X^a_b)$
11. **end**

The main idea behind `SetCol` is to exploit associativity of Boolean matrix multiplication in order to propagate changes of intermediate polynomials that are always limited to a row or a column and thus can be efficiently handled by means of operations like $\texttt{SetRow}_{k=2}$ and $\texttt{SetCol}_{k=2}$.

In lines 3–4 `SetCol` propagates via $\texttt{SetCol}_{k=2}$ the changes of the i-th column of X_b^a to $L^a_{b,1}$, then the changes of the i-th column of $L^a_{b,1}$ to $L^a_{b,2}$, and so on through the recursive decomposition:

$$\begin{array}{lllll}
L^a_{b,0} & = & X^a_b \cdot I_n & = & X^a_b \\
L^a_{b,1} & = & X^a_{b-1} \cdot L^a_{b,0} & = & X^a_{b-1} \cdot X^a_b \\
L^a_{b,2} & = & X^a_{b-2} \cdot L^a_{b,1} & = & X^a_{b-2} \cdot X^a_{b-1} \cdot X^a_b \\
\vdots & & \vdots & & \vdots \\
L^a_{b,b-1} & = & X^a_1 \cdot L^a_{b,b-2} & = & X^a_1 \cdots X^a_{b-2} \cdot X^a_{b-1} \cdot X^a_b
\end{array}$$

Likewise, in lines 5–6 it propagates via $\texttt{SetRow}_{k=2}$ a null matrix of changes of the i-th row of X^a_{b+1} to $R^a_{b,1}$, then the changes (possibly none) of the i-th row of $R^a_{b,1}$ (due to the late effects of some previous `LazySet`) to $R^a_{b,2}$, and so on through the recursive decomposition:

$$\begin{array}{lllll}
R^a_{b,0} & = & I_n \cdot X^a_b & = & X^a_{b+1} \\
R^a_{b,1} & = & R^a_{b,0} \cdot X^a_{b+1} & = & X^a_{b+1} \cdot X^a_{b+2} \\
R^a_{b,2} & = & R^a_{b,1} \cdot X^a_{b+2} & = & X^a_{b+1} \cdot X^a_{b+2} \cdot X^a_{b+3} \\
\vdots & & \vdots & & \vdots \\
R^a_{b,k-b-1} & = & R^a_{b,k-b-2} \cdot X^a_k & = & X^a_{b+1} \cdot X^a_{b+2} \cdot X^a_{b+3} \cdots X^a_k
\end{array}$$

We remark that both loops in lines 3–4 and in lines 5–6 reveal, gather and propagate any 1's that appear in the intermediate polynomials due to the late effects of some previous `LazySet`. In particular, even if the presence of lines 5–6 may seem strange because $\Delta X^a_{b+1} = 0_n$, these lines are executed just for this reason.

Finally, in lines 7–8 changes of $L^a_{b,b-1}$ and $R^a_{b,k-b-1}$ are propagated to Y, which represents the maintained value of $\widehat{P}$, and in lines 9–10 new 1's are lazily inserted in any other polynomials that feature X^a_b as a variable.

We omit the pseudocode for `SetRow` because it is similar to `SetCol`.

`Reset, LazySet, Init, Lookup`

$\texttt{Reset}(\Delta X, X^a_b)$ can be supported by propagating via $\texttt{Reset}_{k=2}$ any changes of X^a_b to any intermediate polynomial $L^a_{u,v}$ and $R^a_{u,v}$ that contains it, then changes of such polynomials to any polynomials which depend on them and so on up to Y.

$\texttt{LazySet}(\Delta X, X^a_b)$ can be supported by performing $\texttt{LazySet}_{k=2}$ operations on each polynomial $L^a_{u,v}$ and $R^a_{u,v}$ that contains X^a_b.

$\texttt{Init}(Z^1_1, \ldots, Z^h_k)$ can be supported by invoking $\texttt{Init}_{k=2}$ on each polynomial $L^w_{u,v}$, $R^w_{u,v}$ and by propagating the intermediate results up to Y.

$\texttt{Lookup}()$ can be realized by returning the maintained value Y of $\widehat{P}$.

To conclude this section, we discuss the correctness and the complexity of our operations in the case of polynomials of arbitrary degree.

Theorem 3.3 *At any time j,* `Lookup` *returns a matrix Y_j that satisfies the relation $M_j \subseteq Y_j \subseteq P_j$ as in Definition 3.6.*

Proof. Since $\widehat{P} = P$ by Lemma 3.2, we prove that:

$$\widehat{P}_j \supseteq Y_j \supseteq M_j = \sum_{\substack{1 \le i \le j :\\ \texttt{Op}_i \neq \texttt{LazySet}}} (\widehat{P}_i - \widehat{P}_{i-1}).$$

To this aim, it is sufficient to prove that any 1 that appears (or disappears) in the correct value of $\widehat{P}$ due to an operation different from `LazySet` appears (or disappears) in Y as well, and that any entry of Y equal to 1 is also equal to 1 in $\widehat{P}$.

- `SetCol`/`SetRow`: assume a `SetCol` operation is performed on the i-th column of variable X_b^a (see Figure 3.2). By induction, we assume that all new 1's are correctly revealed in the i-th column of our data structure for $L_{b,j}^a$ after the j-th iteration of $\texttt{SetCol}_{k=2}$ in line 4. Notice that $\Delta L_{b,j}^a = J_{\Delta L_{b,j}^a, i}$, that is changes of $L_{b,j}^a$ are limited to the i-th column: this implies that these changes can be correctly propagated by means of a `SetCol` operation to any polynomial that features $L_{b,j}^a$ as a variable. As a consequence, by Theorem 3.1, the $j+1$-th iteration of $\texttt{SetCol}_{k=2}$ in line 4 correctly reveals all new 1's in our data structure for $L_{b,j+1}^a$, and again these new 1's all lie on its i-th column. Thus, at the end of the loop in lines 3–4, all new 1's appear correctly in the i-th column of $L_{b,b-1}^a$. Similar considerations apply also for $R_{b,k-b-1}^a$. To prove that lines 7–8 insert correctly in Y all new 1's that appear in $\widehat{P}$ and that $Y \subseteq \widehat{P}$ we use again Theorem 3.1 and the fact that any 1 that appears in $\widehat{P}$ also appears in $L_{b,b-1}^a \cdot R_{b,k-b-1}^a$. Indeed, for any entry $\widehat{P}[x,y]$ that flips from 0 to 1 due to a change of the i-th column of X_b^a or the i-th row of X_{b+1}^a there is a sequence of indices $x = u_0, u_1, \ldots, u_{b-1}, u_b = i, u_{b+1}, \ldots, u_{k-1}, u_k = y$ such that $X_j^a[u_{j-1}, u_j] = 1$, $1 \le j \le k$, and either one of $X_b^a[u_{b-1}, i]$ or $X_{b+1}^a[i, u_{b+1}]$ just flipped from 0 to 1 due to the `SetRow`/`SetCol` operation. The proof for `SetRow` is completely analogous.

- `Reset`: assume a `Reset` operation is performed on variable X_b^a. As $\texttt{Reset}_{k=2}$ can reset any subset of entries of variables, and not only those lying on a row or a column as in the case of $\texttt{SetRow}_{k=2}$ and $\texttt{SetCol}_{k=2}$, the correctness of propagating any changes of X_b^a to the polynomials that depend on it easily follows from Theorem 3.1.

- `Init`: each `Init` operation recomputes from scratch all polynomials in Data Structure 3.2. Thus $Y = \widehat{P}$ after each `Init` operation. □

Theorem 3.4 *Any* `Lookup` *and* `LazySet` *operation requires* $O(n^2)$ *time in the worst case. Any* `SetRow` *and* `SetCol` *operation requires* $O(k \cdot n^2)$ *amortized time, and any* `Init` *operation takes* $O(h \cdot k \cdot n^{\omega} + h \cdot n^2)$ *worst-case time. The cost of any* `Reset` *operation can be charged to previous* `SetRow`, `SetCol` *and* `Init` *operations. The maximum cost charged to each* `Init` *is* $O(h \cdot k \cdot n^3)$*. The space required is* $O(h \cdot k^2 \cdot n^2)$*.*

Proof. The proof easily follows from Theorem 3.2. □

As in the previous paragraph, we have the following corollaries.

Corollary 3.3 *If we perform just one* `Init` *operation in a sequence* σ *of length* $\Omega(n)$*, or more generally one* `Init` *operation every* $\Omega(n)$ `Reset` *operations, then the amortized cost of* `Reset` *is* $O(k \cdot n^2)$ *per operation.*

Corollary 3.4 *If we perform just one* `Init` *operation in a sequence* σ *of length* $\Omega(n^2)$*, or more generally one* `Init` *operation every* $\Omega(n^2)$ `Reset` *operations, and we perform no operations* `SetRow` *and* `SetCol`*, then the amortized cost of* `Reset` *is* $O(k \cdot n)$ *per operation.*

3.4.2 Dynamic Matrices over Integers

In this section we study the problem of finding an implicit representation for a matrix of integers that makes it possible to support simultaneous updates of multiple entries of the matrix very efficiently at the price of increasing the lookup time required to read a single entry. This problem on dynamic matrices will be central to designing the first subquadratic algorithm for fully dynamic transitive closure that will be described in Section 3.7. We formally define the problem as follows:

Definition 3.7 *Let* M *be an* $n \times n$ *integer matrix. We consider the problem of performing an intermixed sequence* $\sigma = \langle \texttt{M.Op}_1, \ldots, \texttt{M.Op}_l \rangle$ *of operations on* M*, where each operation* $\texttt{M.Op}_j$ *can be either one of the following:*

- `M.Init`(X)*: perform the initialization* $M \leftarrow X$*, where* X *is an* $n \times n$ *integer matrix.*
- `M.Update`(J, I)*: perform the update operation* $M \leftarrow M + J \cdot I$*, where* J *is an* $n \times 1$ *column integer vector, and* I *is a* $1 \times n$ *row integer vector. The product* $J \cdot I$ *is an* $n \times n$ *matrix defined for any* $1 \le x, y \le n$ *as:*

$$(J \cdot I)[x, y] = J[x] \cdot I[y]$$

- `M.Lookup`(x, y)*: return the integer value* $M[x, y]$*.*

It is straightforward to observe that `Lookup` can be supported in unit time and operations `Init` and `Update` in $O(n^2)$ worst-case time by explicitly performing the algebraic operations specified in the previous definition.

In the following we show that, if one is willing to give up unit time for `Lookup` operations, it is possible to support `Update` in $O(n^{\omega(1,\epsilon,1)-\epsilon})$ worst-case time for each update operation, for any ϵ, $0 \leq \epsilon \leq 1$, 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. Queries on individual entries of M are answered in $O(n^\epsilon)$ worst-case time via `Lookup` operations and `Init` still takes $O(n^2)$ worst-case time.

We now sketch the main ideas behind the algorithm. We follow a simple lazy approach: we log at most n^ϵ update operations without explicitly computing them and we perform a global reconstruction of the matrix every n^ϵ updates. The reconstruction is done through fast rectangular matrix multiplication. This yields an implicit representation for M which requires us to run through logged updates in order to answer queries about entries of M.

Data Structure

We maintain the following elementary data structures with $O(n^2)$ space:

- an $n \times n$ integer matrix $Lazy$ that maintains a lazy representation of M;
- an $n \times n^\epsilon$ integer matrix Buf_J for buffering update column vectors J;
- an $n^\epsilon \times n$ integer matrix Buf_I for buffering update row vectors I;
- a counter t of the number of `Update` operations performed since the last `Init`, modulo n^ϵ.

Before proposing our implementation of the operations introduced in Definition 3.7, we discuss a simple invariant property that we maintain in our data structure and that guarantees the correctness of the implementation of the operations that we are going to present. We use the following notation:

Definition 3.8 *We denote by $Buf_J\langle j\rangle$ the $n \times j$ matrix obtained by considering only the first j columns of Buf_J. Similarly, we denote by $Buf_I\langle i\rangle$ the $i \times n$ matrix obtained by considering only the first i rows of Buf_I.*

Invariant 3.2 *At any time t in the sequence of operations σ, the following invariant is maintained:*

$$M = Lazy + Buf_J\langle t\rangle \cdot Buf_I\langle t\rangle.$$

`Update`

procedure `Update`(J, I)
1. **begin**
2. $\quad t \leftarrow t + 1$
3. $\quad$ **if** $t \le n^\epsilon$ **then**
4. $\quad\quad Buf_J[\cdot, t] \leftarrow J$
5. $\quad\quad Buf_I[t, \cdot] \leftarrow I$
6. $\quad$ **else**
7. $\quad\quad t \leftarrow 0$
8. $\quad\quad Lazy \leftarrow Lazy + Buf_J \cdot Buf_I$
9. **end**

`Update` first increases t and, if $t \le n^\epsilon$, it copies column vector J onto the t-th column of Buf_J (line 4) and row vector I onto the t-th row of Buf_I (line 5). If $t > n^\epsilon$, there is no more room in Buf_J and Buf_I for buffering updates. Then the counter t is reset in line 7 and the reconstruction operation in line 8 synchronizes $Lazy$ with M via rectangular matrix multiplication of the $n \times n^\epsilon$ matrix Buf_J by the $n^\epsilon \times n$ matrix Buf_I.

`Lookup`

procedure `Lookup`(x, y)
begin
$\quad$ **return** $Lazy[x, y] + \sum_{j=1}^{t} Buf_J[x, j] \cdot Buf_I[j, y]$
end

`Lookup` runs through the first t columns and rows of buffers Buf_J and Buf_I, respectively, and returns the value of $Lazy$ corrected with the inner product of the x-th row of $Buf_J\langle t\rangle$ by the y-th column of $Buf_I\langle t\rangle$.

`Init`

procedure `Init`(X)
begin
$\quad Lazy \leftarrow X$
$\quad t \leftarrow 0$
end

`Init` simply sets the value of $Lazy$ and empties the buffers by resetting t.

The following theorem discusses the time and space requirements of operations `Update`, `Lookup`, and `Init`. As already stated, the correctness easily follows from the fact that Invariant 3.2 is maintained throughout any sequence of operations.

Theorem 3.5 *Each* `Update` *operation can be supported in* $O(n^{\omega(1,\epsilon,1)-\epsilon})$ *worst-case time and each* `Lookup` *in* $O(n^\epsilon)$ *worst-case time, where* $0 \le \epsilon \le 1$ *and*

$\omega(1, \epsilon, 1)$ *is the exponent for rectangular matrix multiplication.* `Init` *requires* $O(n^2)$ *time in the worst case. The space required is* $O(n^2)$.

Proof. An amortized update bound follows trivially from amortizing the cost of the rectangular matrix multiplication $Buf_J \cdot Buf_I$ against n^ϵ update operations. This bound can be made worst-case by standard techniques, i.e., by keeping two copies of the data structures: one is used for queries and the other is updated by performing matrix multiplication in the background.

As fas as `Lookup` is concerned, it answers queries on the value of $M[x, y]$ in $\Theta(t)$ worst-case time, where $t \leq n^\epsilon$. □

Corollary 3.5 *If* $O(n^\omega)$ *is the time required for multiplying two* $n \times n$ *matrices, then we can support* `Update` *in* $O(n^{2-(3-\omega)\epsilon})$ *worst-case time and* `Lookup` *in* $O(n^\epsilon)$ *worst-case time. Choosing* $\epsilon = 1$, *the best known bound for matrix multiplication* ($\omega < 2.38$) *implies an* $O(n^{1.38})$ `Update` *time and an* $O(n)$ `Lookup` *time.*

Proof. A rectangular matrix multiplication between a $n \times n^\epsilon$ matrix by a $n^\epsilon \times n$ matrix can be performed by computing $O((n^{1-\epsilon})^2)$ multiplications between $n^\epsilon \times n^\epsilon$ matrices. This is done in $O\left((n^{1-\epsilon})^2 \cdot (n^\epsilon)^\omega\right)$. The amortized time of the reconstruction operation $Lazy \leftarrow Lazy + Buf_J \cdot Buf_I$ is thus $O\left(\frac{(n^{1-\epsilon})^2 \cdot (n^\epsilon)^\omega + n^2}{n^\epsilon}\right) = O(n^{2-(3-\omega)\epsilon})$. The rest of the claim follows from Theorem 3.5. □

3.5 Transitive Closure Updates in $O(n^2 \log n)$ Time

In this section we show a first method for casting fully dynamic transitive closure into the problem of reevaluating polynomials over Boolean matrices presented in Section 3.4.1.

Based on the technique developed in Section 3.4.1, we revisit the dynamic graph algorithm given in [52] in terms of dynamic matrices and we present a matrix-based variant of it which features better initialization time while maintaining the same bounds on the running time of update and query operations, i.e., $O(n^2 \cdot \log n)$ time per update and $O(1)$ time per query. The space requirement of our algorithm is $M(n) \cdot \log n$, where $M(n)$ is the space used for representing a polynomial over Boolean matrices. As stated in Theorem 3.4, $M(n)$ is $O(n^2)$ if h and k are constant.

In the remainder of this section we first describe our data structure and then we show how to support efficiently operations introduced in Definition 3.4 for the equivalent problem of fully dynamic Boolean matrix closure.

3.5.1 Data Structure

In Section 2.3.2 (Method 1) we showed that the Kleene closure of a Boolean matrix X can be computed from scratch via matrix multiplication by computing $\log_2 n$ polynomials $P_k = P_{k-1} + P_{k-1}^2$, $1 \leq k \leq \log_2 n$, introduced in Definition 2.10. In the static case where X^* has to be computed only once, intermediate results can be thrown away as only the final value $X^* = P_{\log_2 n}$ 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 $P_1, \ldots, P_{\log_2 n}$ with the property that each of them has degree ≤ 3:

Definition 3.9 *Let X be an $n \times n$ Boolean matrix. We define the sequence of $\log_2 n + 1$ polynomials over Boolean matrices $P_0, \ldots, P_{\log_2 n}$ as:*

$$P_k = \begin{cases} X & if\ k = 0 \\ P_{k-1} + P_{k-1}^2 + P_{k-1}^3 & if\ k > 0 \end{cases}$$

Before describing our data structure for maintaining the Kleene closure of X, we discuss some useful properties. In particular, we give claims similar to Lemma 2.10 and Theorem 2.1.

Lemma 3.3 *Let X be an $n \times n$ Boolean matrix and let P_k be formed as in Definition 3.9. Then for any $1 \leq u, v \leq n$, $P_k[u, v] = 1$ if and only if there is a path $u \leadsto v$ of length at most 3^k in X.*

Proof. Similar to the proof of Lemma 2.10, except for the fact that we also consider paths of length up to 3^k obtained as concatenation of 3 paths of length 3^{k-1}. □

Lemma 3.4 *Let X be an $n \times n$ Boolean matrix and let P_k be formed as in Definition 3.9. Then $X^* = I_n + P_{\log_2 n}$*[2]*.*

Proof. The proof is similar to the proof of Theorem 2.10 and follows by observing that the length of the longest simple path in X is no longer than $n - 1 < 3^{\log_3 n} \leq 3^{\log_2 n}$. I_n is required to guarantee the reflexivity of X^*. □

Our data structure for maintaining X^* is the following:

Data Structure 3.3 *We maintain an $n \times n$ Boolean matrix X and we maintain the $\log_2 n$ polynomials $P_1 \ldots P_{\log_2 n}$ of degree 3 given in Definition 3.9 with instances of Data Structure 3.2 presented in Section 3.4.1.*

[2] In general, if X is a Boolean matrix, $P_0 = X$ and $P_k = P_{k-1} + P_{k-1}^2 + \cdots + P_{k-1}^d$, it is possible to prove that $X^* = I_n + P_q$ for any $q >= \log_d n$.

As we will see in Section 3.5.2, the reason for considering the extra term P^3_{k-1} in our data structure is that polynomials need to be maintained using not only `SetRow`/`SetCol`, but also `LazySet`. As stated in Definition 3.6, using `LazySet` yields a weaker representation of polynomials, and this forces us to increase the degree if complete information about X^* has to be maintained. This aspect will be discussed in more depth in the proof of Theorem 3.6.

3.5.2 Implementation of Operations

In this section we show that operations `Init`*, `Set`*, `Reset`* and `Lookup`* introduced in Definition 3.4 can all be implemented in terms of operations `Init`, `LazySet`, `SetRow`, and `SetCol` (described in Section 3.4.1) on polynomials $P_1 \ldots P_{\log_2 n}$.

`Init`*

```
procedure Init*(X)
begin
    Y ← X
    for k = 1 to log₂ n do
        P_k.Init(Y)
        Y ← P_k.Lookup()
end
```

`Init`* performs $\mathtt{P}_k$`.Init` operations on each P_k by propagating intermediate results from X to P_1, then from P_1 to P_2, and so on up to $P_{\log_2 n}$.

`Lookup`*

```
procedure Lookup*(x, y)
begin
    Y ← P_{log₂ n}.Lookup()
    return I_n + Y[x, y]
end
```

`Lookup`* returns the value of $P_{\log_2 n}[x, y]$.

`Set`*

```
procedure Set*(i, ΔX)
begin
    ΔY ← ΔX
    for k = 1 to log₂ n do
        P_k.LazySet(ΔY, P_{k-1})
        P_k.SetRow(i, ΔY, P_{k-1})
        P_k.SetCol(i, ΔY, P_{k-1})
        ΔY ← P_k.Lookup()
end
```

`Set*` propagates changes of P_{k-1} to P_k for any $k = 1$ to $\log_2 n$. Notice that any new 1's that appear in P_{k-1} are inserted in the object P_k via `LazySet`, but only the i-th row and the i-th row column of P_{k-1} are taken into account by `SetRow` and `SetCol` in order to determine changes of P_k. As re-inserting 1's already present in a variable is allowed by our operations on polynomials, for the sake of simplicity in line 7 we assign the update matrix ΔY with P_k and not with the variation of P_k.

`Reset*`

```
   procedure Reset*(ΔX)
1. begin
2.    ΔY ← ΔX
3.    for k = 1 to log₂ n do
4.       Y ←P_k.Lookup()
5.       P_k.Reset(ΔY, P_{k-1})
6.       ΔY ← Y−P_k.Lookup()
7. end
```

`Reset*` performs P_k.`Reset` operations on each P_k by propagating changes specified by ΔX to P_1, then changes of P_1 to P_2, and so on up to $P_{\log_2 n}$. Notice that we use an auxiliary matrix Y to compute the difference between the value of P_k before and after the update and that the computation of ΔY in line 6 always yields a Boolean matrix.

3.5.3 Analysis

In what follows we discuss the correctness and the complexity of our implementation of operations `Init*`, `Set*`, `Reset*`, and `Lookup*` presented in Section 3.5.2. We recall that X is an $n \times n$ Boolean matrix and P_k, $0 \leq k \leq \log_2 n$, are the polynomials introduced in Definition 3.9.

Theorem 3.6 *If at any time during a sequence σ of operations there is a path of length up to 2^k between x and y in X, then $P_k[x, y] = 1$.*

Proof. By induction. The base is trivial. We assume that the claim holds inductively for P_{k-1}, and we show that, after any operation, the claim holds also for P_k.

- `Init*`: since any `Init*` operation rebuilds from scratch P_k, the claim holds from Lemma 3.3.
- `Set*`: let us assume that a `Set*` operation is performed on the i-th row and column of X and a new path π of length up to 2^k, say $\pi =$

$\langle x, \ldots, i, \ldots, y \rangle$, appears in X due to this operation. We prove that $P_k[x, y] = 1$ after the operation.

Observe that $\mathtt{P}_k$`.LazySet`$(\Delta P_{k-1}, P_{k-1})$ puts in place any new 1's in any occurrence of the variable P_{k-1} in data structure $\mathtt{P}_k$. We remark that, although the maintained value of P_k in data structure $\mathtt{P}_k$ is not updated by `LazySet` and therefore the correctness of the current operation is not affected, this step is very important: indeed, new 1's corresponding to new paths of length up to 2^{k-1} that appear in X will be useful in future `Set`* operations for detecting the appearance of new paths of length up to 2^k.

If both the portions $x \rightsquigarrow i$ and $i \rightsquigarrow y$ of π have length up to 2^{k-1}, then π gets recorded in P_{k-1}^2, and therefore in P_k, thanks to one of $\mathtt{P}_k$`.SetRow`$(i, \Delta P_{k-1}, P_{k-1})$ or $\mathtt{P}_k$`.SetCol`$(i, \Delta P_{k-1}, P_{k-1})$. On the other hand, if i is close to (but does not coincide with) one endpoint of π, the appearance of π may be recorded in P_{k-1}^3, but not in P_{k-1}^2. This is the reason why degree 2 does not suffice for P_k in this dynamic setting.

- `Reset`*: by inductive hypothesis, we assume that $P_{k-1}[x, y]$ flips to zero after a `Reset`* operation only if no path of length up to 2^{k-1} remains in X between x and y. Since any $\mathtt{P}_k$`.Reset` operation on P_k leaves it as if cleared 1's in P_{k-1} were never set to 1, $P_k[x, y]$ flips to zero only if no path of length up to 2^k remains in X.

□

We remark that the condition stated in Theorem 3.6 is only sufficient because P_k may keep track of paths having length strictly more than 2^k, though no longer than 3^k. However, for $k = \log_2 n$ the condition is also necessary as no shortest path can be longer than $n = 2^k$. Thus, it is straightforward to see that a path of any length between x and y exists at any time in X if and only if $P_{\log_2 n}[x, y] = 1$.

The following theorem establishes the running time and the space requirements of operations `Init`*, `Set`* and `Reset`*.

Theorem 3.7 *Any* `Init`* *operation can be performed in $O(n^\omega \cdot \log n)$ worst-case time, where ω is the exponent of matrix multiplication; any* `Set`* *takes $O(n^2 \cdot \log n)$ amortized time. The cost of* `Reset`* *operations can be charged to previous* `Init`* *and* `Set`* *operations. The maximum cost charged to each* `Init` *is $O(n^3 \cdot \log n)$. The space required is $O(n^2 \cdot \log n)$.*

Proof. The proof follows from Theorem 3.4 by considering the time bounds of operations on polynomials described in Section 3.4.1. As each maintained polynomial has constant degree $k = 3$, it follows that the space used is $O(n^2 \cdot \log n)$. □

Corollary 3.6 *If we perform just one* `Init*` *operation in a sequence* σ *of length* $\Omega(n)$*, or more generally one* `Init` *operation every* $\Omega(n)$ `Reset` *operations, then the amortized cost of* `Reset` *is* $O(n^2 \cdot \log n)$ *per operation.*

Corollary 3.7 *If we perform just one* `Init*` *operation in a sequence* σ *of length* $\Omega(n^2)$*, or more generally one* `Init` *operation every* $\Omega(n^2)$ `Reset` *operations, and we perform no operations* `SetRow` *and* `SetCol`*, then the amortized cost of* `Reset` *is* $O(n \cdot \log n)$ *per operation.*

In the traditional case where $\texttt{Op}_1$ =`Init*` and $\texttt{Op}_i \neq$`Init*` for any $i > 1$, i.e., `Init*` is just performed once at the beginning of the sequence of operations, previous corollaries state that both `Set*` and `Reset*` are supported in $O(n^2 \cdot \log n)$ amortized time. In the decremental case where only `Reset*` operations are performed, the amortized time is $O(n \cdot \log n)$ per update.

◁ ◇ ▷

The algorithm that we presented in this section can be viewed as a variant which features very different data structures of the fully dynamic transitive closure algorithm presented by King in [52].

King's algorithm is based on a data structure for a graph $G = (V, E)$ that maintains a logarithmic number of edge subsets $E_0, \ldots, E_{\log_2 n}$ with the property that $E_0 = E$ and $(x, y) \in E_i$ if there is a path $x \leadsto y$ of length up to 2^i in G. Moreover, if y is not reachable from x in G, then $(x, y) \notin E_i$ for all $0 \leq i \leq \log_2 n$.

The maintained values of our polynomials $P_0, \ldots, P_{log_2 n}$ here correspond to the sets $E_0, \ldots, E_{\log_2 n}$.

The algorithm by King also maintains $\log_2 n$ forests $F_0, \ldots, F_{\log_2 n-1}$ such that F_i uses edges in E_i and includes $2n$ trees $Out_i(v)$ and $In_i(v)$, two for each node $v \in V$, such that $Out_i(v)$ contains all nodes reachable from v using at most 2 edges in E_i, and $In_i(v)$ contains all nodes that reach v using at most 2 edges in E_i. For each pair of nodes, also a table $Count_i$ is maintained, where $Count_i[x, y]$ is the number of nodes v such that $x \in In_i(v)$ and $y \in Out_i(v)$. Now, E_i is maintained so as to contain edges (x, y) such that $Count_{i-1}[x, y] > 0$. Trees $In_i(v)$ and $Out_i(v)$ are maintained for any node v by means of deletions-only data structures [29] which are rebuilt from scratch after each v-centered insertion of edges.

Our data structures for polynomials over Boolean matrices P_i play the same role as King's forests F_i of In_i and Out_i trees and of counters $Count_i$.

While King's data structures require $O(n^3 \cdot \log n)$ worst-case initialization time on dense graphs, the strong algebraic properties of Boolean matrices allow us to exploit fast matrix multiplication subroutines for initializing more efficiently our data structures in $O(n^\omega \cdot \log n)$ time in the worst case, where $\omega = 2.38$.

3.6 Transitive Closure Updates in $O(n^2)$ Time

In this section we show our second and more powerful method for casting fully dynamic transitive closure into the problem of reevaluating polynomials over Boolean matrices presented in Section 3.4.1.

This method hinges upon the well-known equivalence between transitive closure and matrix multiplication on a closed semiring discussed in Section 2.3.2 and yields a new deterministic algorithm that improves the best known bounds for fully dynamic transitive closure. Our 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 $O(n^2)$.

3.6.1 Data Structure

Let X be a Boolean matrix and let X^* be its Kleene closure. Before discussing the dynamic case, we recall the main ideas of method 2 presented in Section 2.3.2 for computing statically X^*. In Definition 2.11, in Theorem 2.3 and in Lemma 2.11 we showed that, if we decompose X and X^* into submatrices A, B, C, D and E, F, G, H of size $\frac{n}{2} \times \frac{n}{2}$ as follows:

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

then A, B, C, D and E, F, G, H satisfy both Equation 2.1:

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

and Equation 2.2:

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

We also defined two equivalent functions $\mathcal{F}$ and $\mathcal{G}$ based on the previous sets of equations with the property that:

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

and we addressed a method for computing either one of them in $O(n^\omega)$ worst-case time (see Theorem 2.4), where ω is the exponent of Boolean matrix multiplication. This recalled the surprising result, known since the early 70's, that computing the Kleene closure of an $n \times n$ Boolean matrix X can be asymptotically as fast as multiplying two $n \times n$ Boolean matrices [62].

We now define another function $\mathcal{H}$ such that $\mathcal{H}(X) = X^*$, based on a new set of equations obtained by combining Equation 2.1 and Equation 2.2. Our goal is to define $\mathcal{H}$ is such a way that it is well-suited for efficient reevaluation in a fully dynamic setting.

Lemma 3.5 *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:*

$$\left\{ \begin{array}{lll} P = D^* & & \\ E_1 = (A + BP^2C)^* & E_2 = E_1BH_2^2CE_1 & E = E_1 + E_2 \\ F_1 = E_1^2BP & F_2 = E_1BH_2^2 & F = F_1 + F_2 \\ G_1 = PCE_1^2 & G_2 = H_2^2CE_1 & G = G_1 + G_2 \\ H_1 = PCE_1^2BP & H_2 = (D + CE_1^2B)^* & H = H_1 + H_2 \end{array} \right. \tag{3.1}$$

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

Proof. We prove that E_1+E_2, F_1+F_2, G_1+G_2 and H_1+H_2 are sub-matrices of X^*:

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

We first observe that, by definition of Kleene closure (see Definition 2.3), $X = X^* \Rightarrow X = X^2$. Thus, since $E_1 = (A + BP^2C)^*$, $H_2 = (D + CE_1^2B)^*$ and $P = D^*$ are all closures, then we can replace E_1^2 with E_1, H_2^2 with H_2 and P^2 with P. This implies that $E_1 = (A + BPC)^* = (A + BD^*C)^*$ and then $E_1 = E$ by Equation 2.1. Now, since by Theorem 2.3 E is a sub-matrix of X^* and encodes explicitly all paths in X with both end-points in $V_1 = \{1, \ldots, \frac{n}{2}\}$, and since $E_2 = EB(D+CEB)^*CE$, then $E_2 \subseteq E$. It follows that $E_1 + E_2 = E + E_2 = E$. By similar reasoning, we can prove that $F_1 + F_2$, $G_1 + G_2$ and $H_1 + H_2$ are sub-matrices of X^*. In particular, for $H = H_1 + H_2$ we also need to observe that $D^* \subseteq H_2$. □

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 3.1, which is apparently useless, will be clear in Lemma 3.7 after presenting a fully dynamic version of the algorithm which defines $\mathcal{H}$.

In the next lemma 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.

Theorem 3.8 *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.*

Proof. It is possible to compute E, F, G and H with three recursive calls of $\mathcal{H}$, a constant number c_m of multiplications, and a constant number c_s of additions of $\frac{n}{2} \times \frac{n}{2}$ matrices. Thus:

$$T(n) \leq 3\,T\left(\frac{n}{2}\right) + c_m M\left(\frac{n}{2}\right) + c_s\left(\frac{n}{2}\right)^2$$

where $M(n) = O(n^\omega)$ is the time required to multiply two $n \times n$ Boolean matrices. Solving the recurrence relation, since $\log_2 3 < \max\{\omega, 2\} = \omega$, we obtain that $T(n) = O(n^\omega)$ (see e.g., the Master Theorem in [12]). $\square$

The previous theorem showed that, even if $\mathcal{H}$ needs to compute one more closure than $\mathcal{F}$ and $\mathcal{G}$, asymptotically the running time does not get worse.

◁ ◇ ▷

In the following, we study how to reevaluate efficiently $\mathcal{H}(X) = X^*$ under changes of X. Our data structure for maintaining the Kleene closure X^* is the following:

Data Structure 3.4 *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 12 polynomials over $n \times n$ Boolean matrices with the data structure presented in Section 3.4.1:

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

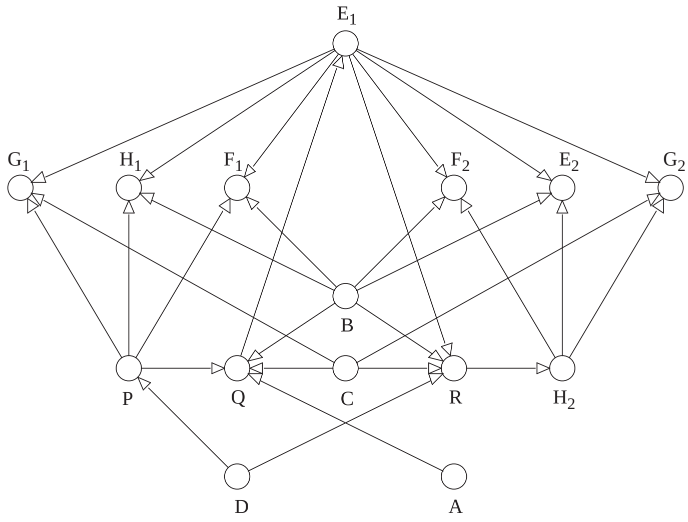


Figure 3.3: Data dependencies between polynomials and closures.

and we recursively maintain 3 *Kleene closures* P*,* E_1 *and* H_2*:*

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

with instances of size $\frac{n}{2} \times \frac{n}{2}$ *of Data Structure 3.4.*

It is worth to note that Data Structure 3.4 is recursively defined: 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. In Figure 3.3 we show the acyclic graph of dependencies between objects in our data structure: there is an arc from node u to node v if the polynomial associated to u is a variable of the polynomial associated to v. For readability, we do not report nodes for the final polynomials E, F, G, H. A topological sort of this graph, e.g., $\tau = \langle P, Q, E_1, R, H_2, F_1, G_1, H_1, E_2, F_2, G_2, E, F, G, H\rangle$, yields a correct evaluation order for the objects in the data structure and thus gives a method for computing $\mathcal{H}(X)$.

We remark that our data structure has memory of all the intermediate values produced when computing $\mathcal{H}(X)$ from scratch and maintains such values upon updates of X. As it was already observed in Section 3.5, maintaining intermediate results of some static algorithm for computing X^* is a fundamental idea for updating efficiently X^* whenever X gets modified.

Since our data structure reflects the way $\mathcal{H}(X)$ 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

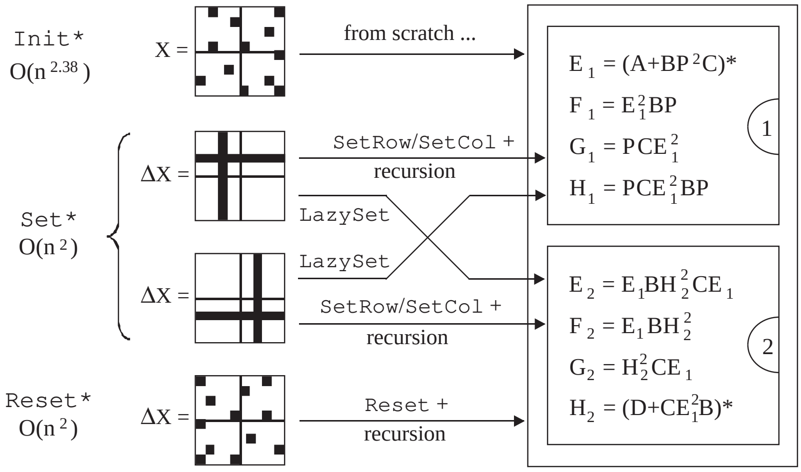


Figure 3.4: Overview of operations `Init`*, `Set`* and `Reset`*.

E_2, F_2, G_2, H_2:

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

In the next section we show how to implement operations `Init`*, `Set`*, `Reset`* and `Lookup`* introduced in Definition 3.4 in terms of operations `Init`, `LazySet`, `SetRow` and `SetCol` (see Section 3.4.1) on the polynomials of Data Structure 3.4.

3.6.2 Implementation of Operations

From a high-level point of view, our approach is the following. We maintain X_1^* and X_2^* *in tandem* (see Figure 3.4): whenever a `Set`* operation is performed on X, we update X^* by computing how either X_1^* or X_2^* are affected by this change. Such 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, `Reset`* operations update both X_1^* and X_2^* and leave the data structure as if any reset entry were never set to 1.

We now describe in detail our implementation. To keep pseudocodes shorter and more readable, we assume that implicit `Lookup` and `Lookup`* operations are performed in order to retrieve the current value of objects so as to use them in subsequent steps. Furthermore, we do not deal explicitly with base recursion steps.

`Set`*

Before describing our implementation of `Set`*, we first define a useful shortcut for performing simultaneous `SetRow` and `SetCol` operations with the same i on more than one variable in a polynomial P:

procedure `P.Set`$(i, \Delta X_1, \ldots, \Delta X_q)$
begin
    `P.SetRow`$(i, \Delta X_1, X_1)$
    `P.SetCol`$(i, \Delta X_1, X_1)$
    $\vdots$
    `P.SetRow`$(i, \Delta X_q, X_q)$
    `P.SetCol`$(i, \Delta X_q, X_q)$
end

Similarly, we give a shortcut[3] for performing simultaneous `LazySet` operations on more than one variable in a polynomial P:

procedure `P.LazySet`$(\Delta X_1, \ldots, \Delta X_q)$
begin
    `P.LazySet`$(\Delta X_1, X_1)$
    $\vdots$
    `P.LazySet`$(\Delta X_q, X_q)$
end

We also define an auxiliary operation `LazySet`* on closures which performs `LazySet` operations for variables A, B, C and D on the polynomials Q, R, F_1, G_1, H_1, E_2, F_2, and G_2 and recurses on the closure P which depend directly on them. We assume that, if M is a variable of a polynomial maintained in our data structure, $\Delta M = M_{curr} - M_{old}$ is the difference between the current value M_{curr} of M and the old value M_{old} of M.

procedure `LazySet`*(ΔX)
begin
    $X \leftarrow X + \Delta X$
    `Q.LazySet`$(\Delta A, \Delta B, \Delta C)$
    `R.LazySet`$(\Delta B, \Delta C, \Delta D)$
    { similarly for F_1, G_1, H_1, E_2, F_2, and G_2 }
    `P.LazySet`*(ΔD)
end

Using the shortcuts `Set` and `LazySet` and the new operation `LazySet`*, we are now ready to define `Set`*.

[3] For the sake of simplicity, we use the same identifier `LazySet` for both the shortcut and the native operation on polynomials, assuming to use the shortcut in defining `Set`*.

procedure `Set*`$(i, \Delta X)$
1. **begin**
2. $X \leftarrow X + I_{\Delta X,i} + J_{\Delta X,i}$
3. **if** $1 \le i \le \frac{n}{2}$ **then**
4. `Q.Set`$(i, \Delta A, \Delta B, \Delta C)$
5. `E`$_1$`.Set*`$(i, \Delta Q)$
6. `F`$_1$`.Set`$(i, \Delta E_1, \Delta B)$
7. `G`$_1$`.Set`$(i, \Delta C, \Delta E_1)$
8. `H`$_1$`.Set`$(i, \Delta C, \Delta E_1, \Delta B)$
9. `R.Set`$(i, \Delta C, \Delta E_1, \Delta B)$
10. `H`$_2$`.LazySet*`(ΔR)
11. `G`$_2$`.LazySet`$(\Delta C, \Delta E_1)$
12. `F`$_2$`.LazySet`$(\Delta E_1, \Delta B)$
13. `E`$_2$`.LazySet`$(\Delta E_1, \Delta B, \Delta C)$
14. **else** $\{ \frac{n}{2} + 1 \le i \le n \}$
15. $i \leftarrow i - \frac{n}{2}$
16. `P.Set*`$(i, \Delta D)$
17. `R.Set`$(i, \Delta B, \Delta C, \Delta D)$
18. `H`$_2$`.Set*`$(i, \Delta R)$
19. `G`$_2$`.Set`$(i, \Delta H_2, \Delta C)$
20. `F`$_2$`.Set`$(i, \Delta B, \Delta H_2)$
21. `E`$_2$`.Set`$(i, \Delta B, \Delta H_2, \Delta C)$
22. `Q.Set`$(i, \Delta B, \Delta P, \Delta C)$
23. `E`$_1$`.LazySet*`(ΔQ)
24. `F`$_1$`.LazySet`$(\Delta B, \Delta P)$
25. `G`$_1$`.LazySet`$(\Delta P, \Delta C)$
26. `H`$_1$`.LazySet`$(\Delta B, \Delta P, \Delta C)$
27. `E.Init`(E_1, E_2)
28. `F.Init`(F_1, F_2)
29. `G.Init`(G_1, G_2)
30. `H.Init`(H_1, H_2)
31. **end**

`Set`* performs an i-centered update in X and runs through the closures and the polynomials of Data Structure 3.4 to propagate any changes of A, B, C, D to E, F, G, H. The propagation order is $\langle Q$, E_1, F_1, G_1, H_1, R, H_2, G_2, F_2, E_2, E, F, G, $H\rangle$ if $1 \le i \le \frac{n}{2}$ and $\langle P$, R, H_2, G_2, F_2, E_2, Q, E_1, F_1, G_1, $H_1\rangle$ if $\frac{n}{2}+1 \le i \le n$ and is defined according to a topological sort of the graph of dependencies between objects in Data Structure 3.4 shown in Figure 3.3.

Roughly speaking, `Set`* updates the objects in the data structure according to the value of i as follows:

1. If $1 \le i \le \frac{n}{2}$, fully updates Q, R, E_1, F_1, G_1, H_1 (lines 4–9) and lazily updates E_2, F_2, G_2, H_2 (lines 10–13). See Figure 3.5 (a).

2. If $\frac{n}{2} + 1 \le i \le n$, fully updates P, Q, R, E_2, F_2, G_2, H_2 (lines 16–22) and lazily updates E_1, F_1, G_1, H_1 (lines 23–26). See Figure 3.5 (b).

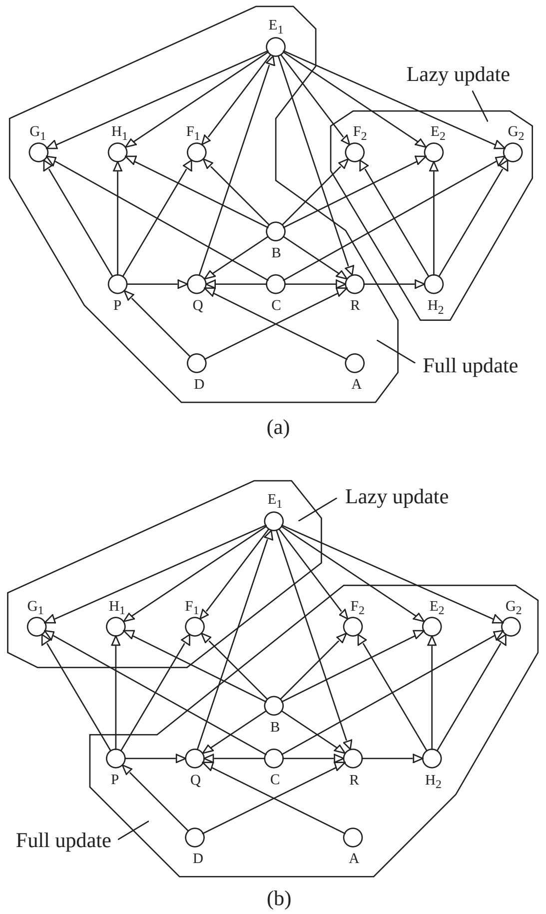


Figure 3.5: Portions of Data Structure 3.4 affected during a `Set*` operation when: (a) $1 \leq i \leq \frac{n}{2}$; (b) $\frac{n}{2} + 1 \leq i \leq n$.

We highlight that it is not always possible to perform efficiently full updates of all the objects of Data Structure 3.4. Actually, some objects may change everywhere, and not only in a row and column. Such unstructured changes imply that we can only perform lazy updates on such objects, as they cannot be efficiently manipulated by means of i-centered `SetRow` and `SetCol` operations.

We now explain in detail the operations performed by `Set*` according to the two cases $1 \leq i \leq \frac{n}{2}$ and $\frac{n}{2} + 1 \leq i \leq n$.

Case 1: $1 \leq i \leq \frac{n}{2}$**.**

In this case an i-centered update of X may affect the i-th row and the i-th column of A, the i-th row of B and the i-th column of C, while D is not affected at all by this kind of update (see Figure 3.4). The operations performed by `Set*` when $1 \le i \le \frac{n}{2}$ are therefore the following:

Line 2**:** an i-centered set operation is performed on X.

Line 4**:** $Q = A + BP^2C$ is updated by performing `SetRow` and `SetCol` operations for any variables A, B and C being changed. $P = D^*$ does not change since, as already observed, D is not affected by the change. Notice that new 1's may appear in Q only in the i-th row and column due to this operation.

Line 5**:** `Set*` is recursively called to propagate the changes of Q to E_1. We remark that E_1 may change also outside the i-th row and column due to this operation. Nevertheless, as we will see in Lemma 3.6, the fact that E_1 is a closure implies that new 1's appear in a very structured way. This will make it possible to propagate changes efficiently to any polynomial that, in turn, depends on E_1.

Lines 6–9**:** polynomials F_1, G_1, H_1 and R are updated by performing `SetRow` and `SetCol` operations for any variables E_1, B and C being changed. We recall that such operations take into account only the entries of ΔE_1 lying in the i-th row and in the i-th column, albeit other entries may be non-zero. Again, Lemma 3.6 and Lemma 3.7 will show that this is sufficient.

Lines 10–13**:** $H_2 = R^*$ is not updated, but new 1's that appear in R are lazily inserted in the data structure of H_2 by calling `LazySet*`. Then `LazySet` operations are carried out on polynomials G_2, F_2, E_2 to insert in the data structures that maintain them any new 1's that appear in C, E_1 and B.

Lines 27–30**.** Recompute polynomials E, F, G and H from scratch. This is required as F_1, G_1 and H_2 may change everywhere and not only in a row and a column. Differently from the case of E_1, whose change is structured as it is a closure, we cannot exploit any particular structure of ΔF_1, ΔG_1 and ΔH_2 for reducing ourselves to use `SetRow` and `SetCol` and we are forced to use `Init`. Note that, since E, F, G and H have all degree 1, this is not a bottleneck in terms of running time.

Case 2: $\frac{n}{2} + 1 \le i \le n$.

In this case an i-centered update of X may affect only the i-th row and the i-th column of D, the i-th row of C and the i-th column of B, while A is not affected at all by this kind of update (see Figure 3.4).

Operations performed by `Set`* are completely analogous to the case $1 \leq i \leq \frac{n}{2}$, except for the fact that we need to rescale the index i in line 15 and we have also to perform a recursive call to update P in line 16.

`Reset`*

Before describing our implementation of `Reset`*, we define a useful shortcut[4] for performing simultaneous `Reset` operations on more than one variable in a polynomial P.

procedure `P.Reset`$(\Delta X_1, \ldots, \Delta X_q)$
begin
  `P.Reset`$(\Delta X_1, X_1)$
  $\vdots$
  `P.Reset`$(\Delta X_q, X_q)$
end

Using this shortcut, we are now ready to define `Reset`*. We assume that, if M is a variable of a polynomial maintained in our data structure, $\Delta M = M_{old} - M_{curr}$ is the difference between the value M_{old} of M just before calling `Reset`* and the current value M_{curr} of M.

procedure `Reset`$^*(\Delta X)$
begin
  $X \leftarrow X - \Delta X$
  `P.Reset`$^*(\Delta D)$
  `Q.Reset`$(\Delta A, \Delta B, \Delta P, \Delta C)$
  $\mathtt{E}_1$`.Reset`$^*(\Delta Q)$
  `R.Reset`$(\Delta D, \Delta C, \Delta E_1, \Delta B)$
  $\mathtt{H}_2$`.Reset`$^*(\Delta R)$
  $\mathtt{F}_1$`.Reset`$(\Delta E_1, \Delta B, \Delta P)$
  { similarly for G_1, H_1, E_2, F_2, G_2, and then for E, F, G, H }
end

`Reset`* resets any entries of X as specified by ΔX and runs through the closures and the polynomials in the data structure to propagate any changes of A, B, C, D to E, F, G, H. The propagation is done according to a topological order τ of the graph of dependencies shown in Figure 3.3 and is the same order followed by `Init`*, which has a similar structure. Actually, we could think of `Reset`* as a function that "undoes" any previous work performed by `Init`* and `Set`* on the data structure, leaving it as if the reset entries of X were never set to 1.

[4] For the sake of simplicity, we use the same identifier `Reset` for both the shortcut and the native operation on polynomials, assuming to use the shortcut in defining `Reset`*.

`Init*`

```
    procedure Init*(Z)
1.  begin
2.      X ← Z
3.      P.Init*(D)
4.      Q.Init(A, B, P, C)
5.      E1.Init*(Q)
6.      R.Init(D, C, E1, B)
7.      H2.Init*(R)
8.      F1.Init(E1, B, P)
9.      { similarly for G1, H1, E2, F2, G2, and then for E, F, G, H }
10. end
```

`Init*` sets the initial value of X (line 2) and initializes the objects in Data Structure 3.4 according to the topological order τ of the graph of dependencies as explained in the previous subsection (lines 3–9).

`Lookup*`

```
procedure Lookup*(x, y)
begin
    return Y[x, y]
end
```

`Lookup*` simply returns the maintained value of $Y[x, y]$.

3.6.3 Analysis

Now we discuss the correctness and the complexity of our implementation. Before providing the main claims, we give some preliminary definitions and lemmas which are useful for capturing algebraic properties of the changes that polynomials in our data structure undergo during a `Set*` operation.

The next definition recalls a property of Boolean update matrices that is related to the operational concept of i-centered update.

Definition 3.10 *We say that a Boolean update matrix ΔX is $i-centered$ if $\Delta X = I_{\Delta X,i} + J_{\Delta X,i}$, i.e., all entries lying outside the i-th row and the i-th column are zero.*

If the variation ΔX of some matrix X during an update operation is i-centered and X is a variable of a polynomial P that has to be efficiently reevaluated, then we can use `P.SetRow` and `P.SetCol` operations which are especially designed for doing so. But what happens if X changes by a ΔX that is not i-centered? Can we still update efficiently the polynomial P without recomputing it from scratch via `Init`? This is the case of E_1 and ΔE_1 while performing a `Set`* update with $1 \le i \le \frac{n}{2}$. In the following we show that, under certain hypotheses on X and ΔX (which are satisfied by E_1 and ΔE_1), we can still solve the problem efficiently.

While the property of being i-centered is related to an update matrix by itself, the following two definitions are concerned with properties of an update matrix ΔX with respect to the matrix X to which it is applied:

Definition 3.11 *If X is a Boolean matrix and ΔX is a Boolean update matrix, we say that ΔX is $i-transitive$ with respect to X if $I_{\Delta X,i} = I_{\Delta X,i} \cdot X$ and $J_{\Delta X,i} = X \cdot J_{\Delta X,i}$.*

Definition 3.12 *If X is a Boolean matrix and ΔX is a Boolean update matrix, we say that ΔX is $i-complete$ with respect to X if $\Delta X = J_{\Delta X,i} \cdot I_{\Delta X,i} + X \cdot I_{\Delta X,i} + J_{\Delta X,i} \cdot X$.*

Using the previous definitions we can show that the variation of X^* due to an i-centered update of X is i-transitive and i-complete.

Lemma 3.6 *Let X be a Boolean matrix and let ΔX be an i-centered update matrix. If we denote by ΔX^* the matrix $(X + \Delta X)^* - X^*$, then ΔX^* is i-transitive and i-complete with respect to X^*.*

Proof. The following equalities prove the first condition of i-transitivity:

$$I_{\Delta X^*,i} \cdot X^* = I_{(X+\Delta X)^* - X^*,i} \cdot X^* = I_{(X+\Delta X)^* \cdot X^* - X^* \cdot X^*,i} =$$

$$I_{(X+\Delta X)^* - X^*,i} = I_{\Delta X^*,i}.$$

The other conditions can be proved analogously. The hypothesis that ΔX is i-centered is necessary for the i-completeness. □

The following lemma shows under what conditions for ΔX and X it is possible to perform operations of the kind $X \leftarrow X + \Delta X$ on a variable X of a polynomial by reducing such operations to i-centered updates even if ΔX is not i-centered.

Lemma 3.7 *If X is a Boolean matrix such that $X = X^*$ and ΔX is an i-transitive and i-complete update matrix with respect to X, then $X + \Delta X = (X + I_{\Delta X,i} + J_{\Delta X,i})^2$.*

Proof. Since $X = X^*$ it holds that $X = X^2$ and $X = X + I_{\Delta X,i} \cdot J_{\Delta X,i}$. The proof follows from Definition 3.11 and Definition 3.12 and from the facts: $I^2_{\Delta X,i} \subseteq I_{\Delta X,i}$, $J^2_{\Delta X,i} \subseteq J_{\Delta X,i}$ and $\Delta X = \Delta X + I_{\Delta X,i} + J_{\Delta X,i}$. □

It follows that, under the hypotheses of Lemma 3.7, if we replace any occurrence of X in P with X^2 and we perform both `P.SetRow`$(i, I_{\Delta X,i}, X)$

and $\texttt{P.SetCol}(i, J_{\Delta X,i}, X)$, then new 1's in P correctly appear. This is the reason why in Data Structure 3.4 we used E_1^2, H_2^2, and P^2 instead of E_1, H_2, and P, respectively.

Before stating the main theorem of this section which establishes the correctness of operations on our data structure, we discuss a general property of polynomials and closures over Boolean matrices that will be useful in proving the theorem.

Lemma 3.8 *Let P and Q be polynomials or closures over Boolean matrices and let $\widehat{P}$ and $\widehat{Q}$ be relaxed functions such that $\widehat{P}(X) \subseteq P(X)$ and $\widehat{Q}(Y) \subseteq Q(Y)$ for any values of variables X and Y. Then, for any X:*

$$\widehat{Q}(\widehat{P}(X)) \subseteq Q(P(X))$$

Proof. Let $\widehat{Y} = \widehat{P}(X)$ and $Y = P(X)$. By definition, we have: $\widehat{Y} \subseteq Y$ and $\widehat{Q}(\widehat{Y}) \subseteq Q(\widehat{Y})$. By exploiting a monotonic behavior of polynomials and closures over Boolean matrices, we have: $\widehat{Y} \subseteq Y \Rightarrow Q(\widehat{Y}) \subseteq Q(Y)$. Thus: $\widehat{Q}(\widehat{Y}) \subseteq Q(\widehat{Y}) \subseteq Q(Y) \Rightarrow \widehat{Q}(\widehat{Y}) \subseteq Q(Y) \Rightarrow \widehat{Q}(\widehat{P}(X)) \subseteq Q(P(X))$. □

Theorem 3.9 *Let $\mathcal{H}$ be the function defined in Lemma 3.5, let X and Y be the matrices maintained in Data Structure 3.4, and let M be a Boolean matrix whose value at any time j is defined as:*

$$M_j = \sum_{\substack{1 \le i \le j:\\ \texttt{Op}_i \neq \texttt{LazySet}^*}} \mathcal{H}(X_i) - \mathcal{H}(X_{i-1}).$$

If we denote by X_j and Y_j the values of X and Y after the j-th operation, respectively, then the relation $M_j \subseteq Y_j \subseteq \mathcal{H}(X_j)$ is satisfied.

Proof. The proof is by induction on the size n of matrices in Data Structure 3.4. The base is trivial. We assume that the claim holds for instances of size $\frac{n}{2}$ and we prove that it holds also for instances of size n.

- $\texttt{Op}_j\texttt{=Init}^*$: since $\texttt{Init}^*$ performs $\texttt{Init}$ operations on each object, then $Y_j = \mathcal{H}(X_j)$.
- $\texttt{Op}_j\texttt{=Set}^*$: we first prove that $Y_j \subseteq \mathcal{H}(X_j)$. Observe that Y is obtained as a result of a composition of functions that relax the correct intermediate values of polynomials and closures of Boolean matrices in our data structure allowing them to contain less 1's. Indeed, by the properties of $\texttt{Lookup}$ described in Section 3.4.1, we know that, if P is the correct value of a polynomial at any time, then $\texttt{P.Lookup}() \subseteq P$. Similarly, by inductive hypothesis, if K is a Kleene closure of an $\frac{n}{2} \times \frac{n}{2}$ Boolean

matrix, then at any time `K.Lookup`$^*(x, y) = 1 \Rightarrow K[x, y] = 1$. The claim then follows by Lemma 3.8, which states that the composition of relaxed functions computes values containing at most the 1's contained in the values computed by the correct functions.

To prove that $M_j \subseteq Y_j$, based on the definition of M, it suffices to verify that $\Delta\mathcal{H}(X) \subseteq \Delta Y$, where $\Delta\mathcal{H}(X) = \mathcal{H}(X_j) - \mathcal{H}(X_{j-1})$ and $\Delta Y = Y_j - Y_{j-1}$. In particular, we prove that if $\mathcal{H}[x, y]$ flips from 0 to 1 due to operation `Set`*, then either $X_1^*[x, y]$ flips from 0 to 1 (due to lines 4–8 when $1 \leq i \leq \frac{n}{2}$), or $X_2^*[x, y]$ flips from 0 to 1 (due to lines 17–21 when $\frac{n}{2} + 1 \leq i \leq n$).

Without loss of generality, assume that the `Set`* operation is performed with $1 \leq i \leq \frac{n}{2}$ (the proof is completely analogous if $\frac{n}{2} + 1 \leq i \leq n$).

As shown in Figure 3.4, sub-matrices A, B and C may undergo i-centered updates due to this operation and so their variation can be correctly propagated through `SetRow` and `SetCol` operations to polynomial Q (line 4) and to polynomials F_1, G_1 and H_1 (lines 6–8). As ΔQ is also i-centered due to line 4, any variation of Q, that is assumed to be elsewhere correct from previous operations, can be propagated to closure E_1 through a recursive call of `Set`* in line 5. By the inductive hypothesis, this propagation correctly reveals any new 1's in E_1. We remark that E_1 may contain less 1's than E due to any previous `LazySet` operations done in line 23.

Observe now that E_1 occurs in polynomials F_1, G_1 and H_1 and that ΔE_1 is not necessarily i-centered. This would imply that we cannot propagate directly changes of E_1 to these polynomials, as no efficient operation for doing so was defined in Section 3.4.1. However, by Lemma 3.6, ΔE_1 is i-transitive and i-complete with respect to E_1. Since $E_1 = E_1^*$, by Lemma 3.7 performing both `SetRow`$(i, I_{\Delta E_1, i}, E_1)$ and `SetCol`$(i, J_{\Delta E_1, i}, E_1)$ operations on data structures $\mathtt{F}_1$, $\mathtt{G}_1$ and $\mathtt{H}_1$ in lines 6–8 is sufficient to correctly reveal new 1's in F_1, G_1 and H_1.

Again, note that F_1, G_1 and H_1 may contain less 1's than F, G and H, respectively, due to any previous `LazySet` operations done in lines 23–26. We have then proved that lines 4–8 correctly propagate any i-centered update of X to X_1^*.

To conclude the proof, we observe that E_1 also occurs in polynomials E_2, F_2, G_2, R and indirectly affects H_2. Unfortunately, we cannot update H_2 efficiently as ΔR is neither i-centered, nor i-transitive/i-complete with respect to R. So in lines 9–13 we limit ourselves to update explicitly R and to log any changes of E_1 by performing `LazySet` operations on polynomials G_2, F_2, and E_2 and a `LazySet`* operation on H_2. This is sufficient to guarantee the correctness of subsequent `Set`* operations for $\frac{n}{2}+1 \le i \le n$.

- `Op`$_j$=`Reset`*: this operation runs in judicious order through the objects in the data structure and undoes the effects of previous `Set`* and `Init`* operations. Thus, any property satisfied by Y still holds after performing a `Reset`* operation.

□

Corollary 3.8 *Let* `X` *be an instance of Data Structure 3.4 and let* $\sigma = \langle$`X.Op`$_1$, $\ldots$, `X.Op`$_k\rangle$ *be a sequence of operations on* `X`*. If* `Op`$_i \neq$ `LazySet`* *for all* $1 \le i \le j \le k$*, then* $M_j = \mathcal{H}(X_j)$*.*

Proof. Since $\mathcal{H}(0_n) = 0_n^* = 0_n$, the proof easily follows by telescoping the sum that defines M_j: $M_j = \mathcal{H}(X_j) - \mathcal{H}(X_{j-1}) + \mathcal{H}(X_{j-1}) - \mathcal{H}(X_{j-2}) + \cdots + \mathcal{H}(X_2) - \mathcal{H}(X_1) + \mathcal{H}(X_1) - \mathcal{H}(X_0) = \mathcal{H}(X_j) - \mathcal{H}(X_0) = \mathcal{H}(X_j)$. □

To conclude this section, we address the running time of operations and the space required to maintain an instance of our data structure.

Theorem 3.10 *Any* `Init`* *operation can be performed in* $O(n^\omega)$ *worst-case time, where* ω *is the exponent of matrix multiplication; any* `Set`* *takes* $O(n^2)$ *amortized time. The cost of* `Reset`* *operations can be charged to previous* `Init`* *and* `Set`* *operations. The maximum cost charged to each* `Init`* *is* $O(n^3)$*. The space required is* $O(n^2)$*.*

Proof. Since all the polynomials in Data Structure 3.4 are of constant degree and involve a constant number of terms, the amortized cost of any `SetRow`, `SetCol`, `LazySet`, and `Reset` operation on them is quadratic in $\frac{n}{2}$ (see Theorem 3.4). Let $T(n)$ be the time complexity of any `Set`*, `LazySet`* and `Reset`* operation. Then:

$$T(n) \le 3\,T(\frac{n}{2}) + \frac{c\,n^2}{4}$$

for some suitably chosen constant $c > 0$. As $\log_2 3 < 2$, this implies that $T(n) = O(n^2)$.

`Init`* recomputes recursively $\mathcal{H}$ from scratch using `Init` operations on polynomials, which require $O(n^\omega)$ worst-case time each. We can then prove that the running time of `Init`* is $O(n^\omega)$ exactly as in Theorem 3.8.

To conclude the proof, observe that if $K(n)$ is the space used to maintain all the objects in Data Structure 3.4, and $M(n)$ is the space required to maintain a polynomial with the data structure of Section 3.4.1, then:

$$K(n) \le 3\,K(\frac{n}{2}) + 12\,M(n).$$

Since $M(n) = O(n^2)$ by Theorem 3.4, then $K(n) = O(n^2)$. $\square$

Corollary 3.9 *If we perform just one* `Init*` *operation in a sequence* σ *of length* $\Omega(n)$*, or more generally one* `Init*` *operation every* $\Omega(n)$ `Reset*` *operations, then the amortized cost of* `Reset*` *is* $O(n^2)$ *per operation.*

Corollary 3.10 *If we perform just one* `Init*` *operation in a sequence* σ *of length* $\Omega(n^2)$*, or more generally one* `Init*` *operation every* $\Omega(n^2)$ `Reset*` *operations, and we perform no* `Set*` *operations, then the amortized cost of* `Reset*` *is* $O(n)$ *per operation.*

In the traditional case where $\mathtt{Op}_1 = $`Init*` and $\mathtt{Op}_i \neq$`Init*` for any $i > 1$, i.e., `Init*` is performed just once at the beginning of the sequence of operations, previous corollaries state that both `Set*` and `Reset*` are supported in $O(n^2)$ amortized time. In the decremental case where only `Reset*` operations are performed, the amortized time is $O(n)$ per update.

3.7 Breaking Through the $O(n^2)$ Barrier

In this section we present the first algorithm that supports both updates and queries in subquadratic time per operation, showing that it is actually possible to break through the $O(n^2)$ barrier on the single-operation complexity of fully dynamic transitive closure. This result is obtained by means of a new technique that consists of casting fully dynamic transitive closure into the problem of dynamically maintaining matrices over integers presented in Section 3.4.2. As already shown in Section 3.5 and in Section 3.6, dynamic matrices, thanks to their strong algebraic properties, play a crucial role in designing efficient algorithms for the fully dynamic transitive closure problem.

The remainder of this section is organized as follows. In Section 3.7.1 we present a subquadratic algorithm for directed acyclic graphs based on dynamic matrices that answers queries in $O(n^\epsilon)$ time and performs updates in $O(n^{\omega(1,\epsilon,1)-\epsilon} + n^{1+\epsilon})$ time, for any $0 \leq \epsilon \leq 1$, 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. According to the current best bounds on $\omega(1,\epsilon,1)$, we obtain an $O(n^{0.58})$ query time and an $O(n^{1.58})$ update time. The algorithm we propose is randomized, and has one-sided error. In Section 3.7.2 we also devise a second simple method for obtaining subquadratic updates under certain hypotheses on the sequence of operations. Like our first algorithm, this method is built on top of dynamic matrices as well.

3.7.1 Counting Paths in Acyclic Directed Graphs

In this section we study a variant of the fully dynamic transitive closure problem presented in Definition 3.1 and we devise the first algorithm that supports both update and query in subquadratic time per operation. In the variant that we consider, the graph that we maintain is constrained to be acyclic; furthermore, `Insert` and `Delete` operations work on single edges rather than on set of edges. We shall discuss later how to extend our algorithm to deal with more than one edge at a time.

Definition 3.13 *Let $G = (V, E)$ be a directed acyclic graph and let $TC(G) = (V, E')$ be its transitive closure. We consider the problem of maintaining a data structure* `G` *for the graph G under an intermixed sequence $\sigma = \langle \texttt{G.Op}_1, \ldots, \texttt{G.Op}_k \rangle$ of update and query operations. Each operation $\texttt{G.Op}_j$ on the data structure* `G` *can be either one of the following:*

- `G.Insert`(x, y)*: perform the update $E \leftarrow E \cup \{(x, y)\}$, such that the graph obtained after the update is still acyclic.*
- `G.Delete`(x, y)*: perform the update $E \leftarrow E - \{(x, y)\}$, where $(x, y) \in E$.*
- `G.Query`(x, y)*: perform a query operation on $TC(G)$ by returning 1 if $(x, y) \in E'$ and 0 otherwise.*

In this version of the problem, we do not deal explicitly with initialization operations.

Data Structure

In [53] King and Sagert showed that keeping a count of the number of distinct paths between any pair of vertices in a directed acyclic graph G allows it to maintain the transitive closure of G upon both insertions and deletions of edges. Unfortunately, these counters may be as large as 2^n: to perform $O(1)$ time arithmetic operations on counters, an $O(n)$ wordsize is required. As shown in [53], the wordsize can be reduced to $2c \lg n$ for any $c \geq 5$ based on the use of arithmetic operations performed modulo a random prime number. This yields a fully dynamic randomized Monte Carlo algorithm for transitive closure with the property that "yes" answers on reachability queries are always correct, while "no" answers are wrong with probability $O(\frac{1}{n^c})$. We recall that this algorithm performs reachability queries in $O(1)$ and updates in $O(n^2)$ worst-case time on directed acyclic graphs.

We now present an algorithm that combines the path counting approach of King and Sagert with our technique of implicit matrix representation. Both techniques are very simple, but surprisingly their combination solves a problem that has been open for many years.

Data Structure 3.5 *We keep a count of the number of distinct paths between any pair of vertices in graph G by means of an instance M of the dynamic matrix data structure described in Section 3.4.2. We assume that $M[x, y]$ is the number of distinct paths between node x and node y in graph G. Since G is acyclic, this number is well-defined.*

Implementation of Operations

We now show how to implement operations `Insert`, `Delete` and `Query` in terms of operations `Update` and `Lookup` on our data structure as described in Section 3.4.2. We assume all arithmetic operations are performed in constant time.

`Insert`

procedure `Insert`(x, y)
1. **begin**
2. $E \leftarrow E \cup \{(x, y)\}$
3. **for** $z = 1$ **to** n **do**
4. $J[z] \leftarrow$ `M.Lookup`(z, x)
5. $I[z] \leftarrow$ `M.Lookup`(y, z)
6. `M.Update`(J, I)
7. **end**

`Insert` first puts edge (x, y) in the graph and then, after querying matrix `M`, computes two vectors J and I such that $J[z]$ is the number of distinct paths $z \leadsto x$ in G and $I[z]$ is the number of distinct paths $y \leadsto z$ in G (lines 3–5). Finally, it updates `M` in line 6. The operation performed on M is $M \leftarrow M + J \cdot I$: this means that the number $M[u, v]$ of distinct paths between any two nodes (u, v) is increased by the number $J[u]$ of distinct paths $u \leadsto x$ times the number $I[v]$ of distinct paths $y \leadsto v$, i.e., $M[u, v] \leftarrow M[u, v] + J[u] \cdot I[v]$.

`Delete`

procedure `Delete`(x, y)
1. **begin**
2. $E \leftarrow E - \{(x, y)\}$
3. **for** $z = 1$ **to** n **do**
4. $J[z] \leftarrow$ `M.Lookup`(z, x)
5. $I[z] \leftarrow$ `M.Lookup`(y, z)
6. `M.Update`$(-J, I)$
7. **end**

`Delete` is identical to `Insert`, except for the fact that it removes the edge (x, y) from the graph and performs the update of `M` in line 6 with $-J$ instead of J. The operation performed on M is $M \leftarrow M - J \cdot I$: this means that the number $M[u, v]$ of distinct paths between any two nodes (u, v) is decreased by the number $J[u]$ of distinct paths $u \rightsquigarrow x$ times the number $I[v]$ of distinct paths $y \rightsquigarrow v$, i.e., $M[u,v] \leftarrow M[u,v] - J[u] \cdot I[v]$.

`Query`

```
    procedure Query(x, y)
    begin
        if M.Lookup(x, y) > 0 then return 1
        else return 0
    end
```

`Query` simply looks up the value of $M[x,y]$ and returns 1 if the current number of distinct paths between x and y is positive, and zero otherwise.

◁ ◇ ▷

We are now ready to discuss the running time of our implementation of operations `Insert`, `Delete`, and `Query`.

Theorem 3.11 *Any* `Insert` *and any* `Delete` *operation can be performed in* $O(n^{\omega(1,\epsilon,1)-\epsilon} + n^{1+\epsilon})$ *worst-case time, for any* $0 \le \epsilon \le 1$*, 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. Any* `Query` *takes* $O(n^\epsilon)$ *in the worst case. The space required is* $O(n^2)$*.*

Proof. We recall that, by Theorem 3.5, each entry of M can be queried in $O(n^\epsilon)$ worst-case time, and each `Update` operation can be performed in $O(n^{\omega(1,\epsilon,1)-\epsilon})$ worst-case time. Since I and J can be computed in $O(n^{1+\epsilon})$ worst-case time by means of n queries on M, we can support both insertions and deletions in $O(n^{\omega(1,\epsilon,1)-\epsilon} + n^{1+\epsilon})$ worst-case time, while a reachability query for any pair of vertices (x,y) can be answered in $O(n^\epsilon)$ worst-case time by simply querying the value of $M[x,y]$. □

Corollary 3.11 *Any* `Insert` *and any* `Delete` *operation requires* $O(n^{1.58})$ *worst-case time, and any* `Query` *requires* $O(n^{0.58})$ *worst-case time.*

Proof. Balancing the two terms in the update bound $O(n^{\omega(1,\epsilon,1)-\epsilon} + n^{1+\epsilon})$ yields that ϵ must satisfy the equation $\omega(1,\epsilon,1) = 1 + 2\epsilon$. The current best bounds on $\omega(1,\epsilon,1)$ [11, 46] imply that $\epsilon < 0.58$ [73]. Thus, the smallest update time is $O(n^{1.58})$, which gives a query time of $O(n^{0.58})$. □

The algorithm we presented is deterministic. However, as the numbers involved may be as large as 2^n, performing arithmetic operations in constant time requires wordsize $O(n)$. To reduce wordsize to $O(\log n)$ while maintaining the same subquadratic bounds ($O(n^{1.58})$ per update and $O(n^{0.58})$ per query) we perform all arithmetic operations modulo some random prime number as explained in [53]. Again, this produces a randomized Monte Carlo algorithm, where "yes" answers on reachability queries are always correct, while "no" answers are wrong with probability $O(\frac{1}{n^c})$ for any constant $c \geq 5$.

It is also not difficult to extend our subquadratic algorithm to deal with insertions/deletions of more than one edge at a time. In particular, we can support any insertion/deletion of up to $O(n^{1-\eta})$ edges incident to a common vertex in $O(n^{\omega(1,\epsilon,1)-\epsilon} + n^{2-(\eta-\epsilon)})$ worst-case time. We emphasize that this is still $o(n^2)$ for any $1 > \eta > \epsilon > 0$. Indeed, rectangular matrix multiplication can be trivially implemented via matrix multiplication: this implies that $\omega(1,\epsilon,1) < 2 - (2-\omega)\epsilon$, where $\omega = \omega(1,1,1) < 2.38$ is the current best exponent for matrix multiplication [11].

3.7.2 A Deterministic Algorithm

In this section we briefly report another idea for reducing fully dynamic transitive closure to the problem of maintaining dynamic integer matrices. We show a simple deterministic algorithm which works on general directed graphs and supports reachability queries in $O(n^{0.616})$ time, insertion of an edge in $O(n^{1.616})$ time, and deletion of an edge among the $O(n^{\eta})$ most recently inserted edges in $O(n^{1.616+\eta})$ time. The algorithm is subquadratic if $\eta < 0.384$ and all bounds are worst-case.

We remark that constraining deletions to remove only "recent" edges represents a rather strong limitation; nevertheless, we report the method for its technical interest. The main idea that the algorithm is based on consists of stacking edge insertions and undoing them when a recently inserted edge has to be deleted.

Data Structure and Implementation of Operations

We maintain a matrix M such that $M[x,y]$ is the number of previous edge insertions which caused at least a new simple path from x to y to appear in the graph. Clearly, there is a path in the graph from x to y if and only if $M[x,y] > 0$, and this allows us to answer reachability queries.

To insert an edge (x,y), we compute a row vector Out_y such that $Out_y[v] = 1$ if and only if there is a path from y to v in the graph and zero otherwise. We also compute a column vector In_x that keeps track of nodes which reach x. Then we perform the operation $M \leftarrow M + In_x \cdot Out_y$ and we log the pair (In_x, Out_y) in a stack.

To delete an edge (x,y) which is among the k most recently inserted edges, we pop the topmost k pairs of vectors (In_x, Out_y) from the stack and perform $M \leftarrow M - In_x \cdot Out_y$ for each pair in backward order to "undo" the last k edge insertions. Finally, we re-insert in any order all removed edges, but (x,y).

Notice that a single deletion costs $2k$ times more than a single insertion and leaves M as if (x, y) was never inserted.

If we maintain M with the data structure presented in Section 3.4.2, we can support any operation of the kind $M \leftarrow M \pm In_x \cdot Out_y$ in $O(n^{2-0.624\epsilon})$ time and any query about an entry of M in $O(n^{\epsilon})$ time for any $0 \leq \epsilon \leq 1$ (see Corollary 3.5). Since computing In_x and Out_y requires $O(n^{1+\epsilon})$ time due to the non-constant access to M, we have that any insertion requires $O(n^{2-0.624\epsilon} + n^{1+\epsilon})$ worst-case time. If we balance the two terms solving the equation $2 - 0.624\epsilon = 1 + \epsilon$, we obtain $\epsilon = \frac{1}{1.624} = 0.616$ which gives an $O(n^{1.616})$ insertion time and $O(n^{0.616})$ query time. Choosing $k = n^{\eta}$ yields $O(n^{1.616+\eta})$ time per deletion, which is subquadratic for any $\eta < 0.384$.

3.8 Conclusions

In this chapter we have presented new time and space efficient algorithms for maintaining the transitive closure of a directed graph under edge insertions and edge deletions.

As a main contribution, we have introduced a general framework for casting fully dynamic transitive closure into the problem of dynamically reevaluating polynomials over matrices when updates of variables are performed. Such technique has turned out to be very flexible and powerful, leading both to revisit the best known algorithm for fully dynamic transitive closure [52] in terms of completely different data structures, and to design a new and faster algorithm for the problem.

In particular, efficient data structures for maintaining polynomials over Boolean matrices allowed us to devise the rather complex deterministic algorithm described in Section 3.6, which supports updates in quadratic amortized time and queries with just one matrix lookup. Our algorithm improves the best bounds for fully dynamic transitive closure achieved in [52] and is the fastest algorithm with constant query time known in literature for this problem.

In addition, a surprisingly simple technique for efficiently maintaining dynamic matrices of integers under simultaneous updates of multiple entries, combined with a previous idea of counting paths in acyclic digraphs [53], yielded the randomized algorithm presented in Section 3.7.1: this algorithm, for the first time in the study of fully dynamic transitive closure, breaks through the $O(n^2)$ barrier on the single-operation complexity of the problem.

Chapter 4

Fully Dynamic Shortest Paths

4.1 Introduction

This chapter features a preliminary experimental study of the best known fully dynamic algorithms for the single-source shortest paths problem and presents a new variant inspired by the experiments. Our variant is especially designed to be simple and fast in practice while matching the worst-case asymptotic complexity of the best algorithms.

The chapter is dominated by the concept of reweighting, originally designed by Edmonds and Karp for network flows problems [26] and discussed in Section 2.4 of this dissertation. We shall see that in the dynamic setting this technique allows it to reduce the general problem of updating shortest paths in case of arbitrary edge weights to a problem with nonnegative edge weights. This opens the possibility of using dynamic adaptations of the classical algorithm of Dijkstra instead of recomputing from scratch by means of Bellman-Ford's algorithm (see Section 2.4.1), and allows it to reduce dramatically the running time of performing updates from $O(mn)$ to $O(m + n \log n)$. Actually, the reweighting technique is the kernel for solving efficiently the fully dynamic single-source shortest paths problem and is the common skeleton of all the best known algorithms for the problem, including our new variant.

Our experimental study was aimed at evaluating and comparing the effectiveness of different algorithms constructed around the skeleton provided by the reweighting method, and our main goal was that of identifying with experimental evidence the more convenient algorithm to use in practice in a fully dynamic setting.

The results of our preliminary experiments on random test sets showed that:

1. all the considered dynamic algorithms based on the reweighting technique are faster by several orders of magnitude than recomputing from

scratch with the best static algorithm: this yields clues to the fact that constant factors in practical implementations can be very small for this problem;

2. the running time of all the considered dynamic algorithms is affected by the width of the interval of edge weights. In particular, dynamic algorithms are faster if weights come from a small set of possible values;

3. due to its simplicity, our new variant stands out as a practical solution, though more complex algorithms may become preferable in some specific cases.

The chapter is organized as follows. Section 4.2 formally defines the fully dynamic single-source shortest paths problem and gives preliminary definitions. In Section 4.3 we revisit the reweighting technique in a dynamic context and we discuss different implementations of update operations based on this technique. We briefly describe implementations known from the literature plus a new simple and practical algorithm which we suggest. We finally focus on our experimental framework in Section 4.4 and report the results of our experimentation in Section 4.5.

4.2 Statement of the Problem

In this section we give a formal definition of the fully dynamic single-source shortest paths problem that we study in this chapter. We assume that the reader is familiar with the preliminary concepts discussed in Section 2.4.

Definition 4.1 *Let $G = (V, E, w)$ be an edge weighted directed graph, let $s \in V$ be a fixed source node, let $d_s : V \to \mathcal{R}$ be the distance function w.r.t. s, and let $T(s)$ be a shortest path tree rooted at s. We define the* FULLY DYNAMIC SINGLE-SOURCE SHORTEST PATHS PROBLEM *as the problem of maintaining a data structure* `G` *for the graph G under an intermixed sequence $\sigma = \langle \texttt{G.Op}_1, \ldots, \texttt{G.Op}_k \rangle$ of* INITIALIZATION, UPDATE, *and* QUERY *operations. Each operation $\texttt{G.Op}_j$ on data structure* `G` *can be one of the following:*

- `G.Init`(w'): *perform the initialization operation $w(x, y) \leftarrow w'(x, y)$ for any $(x, y) \in E$, where $w' : E \to \mathcal{R}$. The operation sets the weight of each edge in the graph.*

- `G.Increase`(x, y, ϵ): *perform the update $w(x, y) \leftarrow w(x, y) + \epsilon$, where $(x, y) \in E$ and $\epsilon \in \mathcal{R}^+$. The operation increases the weight of edge (x, y) by the amount ϵ.*

- `G.Decrease`(x, y, ϵ): *perform the update* $w(x, y) \leftarrow w(x, y) - \epsilon$, *where* $(x, y) \in E$ *and* $\epsilon \in \mathcal{R}^+$. *The operation decreases the weight of edge* (x, y) *by the amount* ϵ.

- `G.Query`(x): *perform a query operation on* d_s *and* $T(s)$ *and return a pair* $\langle d, \pi \rangle$ *such that* $d = d_s(x)$ *and* π *is the path* $s \leadsto x$ *in the shortest paths tree* $T(s)$.

This definition is a dynamic counterpart of Definition 2.17 given in Section 2.4. Notice that we focus on the fixed-topology case where a single update is allowed to change the weight of an edge, but not to remove and insert nodes and edges of the graph.

In next section we address the issue of implementing efficiently operations `Increase` and `Decrease` introduced in Definition 4.1. In particular, we show that the reweighting technique introduced in Section 2.4 plays a crucial role in designing solutions which are asymptotically faster than recomputing shortest paths from scratch after each update with the best known static algorithm.

4.3 Fully Dynamic Algorithms by Reweighting

In Section 2.4 we showed that, given a weighted directed graph $G = (V, E, w)$ with n nodes and m edges, if s is the source and d_s is the distance function, then we can obtain a new graph $\hat{G} = (V, E, \hat{w})$ where: $\hat{w}(x, y) = w(x, y) + d_s(x) - d_s(y)$. We recalled the surprising property that shortest paths stay the same in both G and $\hat{G}$ (Lemma 2.14). We also learned from Lemma 2.15 that the weight of any edge $\hat{w}(x, y)$ is nonnegative for any $(x, y) \in E$.

Previous properties let us immediately think of the possibility of using the more efficient algorithm of Dijkstra instead of Bellman-Ford's one for computing distances and shortest paths. Unfortunately, in the static case, this does not work for our single-source problem as using such potentials requires distances to be already computed. Thus, the best static solution for the single-source shortest paths problem remains Bellman-Ford's algorithm, which runs in $O(mn)$ worst-case time (see Section 2.4.1).

In a dynamic setting, instead, the reweighting technique is a powerful tool. Roughly speaking, the main idea consists of maintaining explicitly during a sequence of operations the current distances and exploiting them for defining a nonnegative reweighting function. To perform an update, we can then run a Dijkstra's computation on the reweighted graph and the distances which we obtain can be viewed as *variations* of distances in the original graph since the update.

The skeleton of a dynamic algorithm based on the reweighting technique is shown in Figure 4.1. As the most expensive operation is the Dijkstra com-

1.	*Change the weight of an edge*
	↓
2.	*Reweight the updated graph*
	↓
3.	*Run Dijkstra on the reweighted graph*
	↓
4.	*Use obtained distances to update distances and shortest paths tree in the original graph*

Figure 4.1: Skeleton of an update operation based on the reweighting technique.

putation, the time required for performing steps 1–4 is $O(m + n \log n)$. We remark that all the best known solutions for the fully dynamic single-source shortest paths problem are designed around this skeleton. Before describing algorithms for the problem, we define the following data structure:

Data Structure 4.1 *We maintain a weighted directed graph $G = (V, E, w)$, the distances $d_s(v)$ of any node v from the source s in G, and a shortest paths tree $T(s)$ rooted at s represented as a vector of parents p_s such that (x, y) is in $T(s)$ if and only if $p_s(y) = x$.*

According to this definition, explicitly maintaining both shortest paths and distances makes it possible to perform `Query` operations in optimal time, so we shall not discuss further this operation. We remark that this choice is common to all known algorithms for the problem introduced in Definition 4.1.

In the remainder of the section we focus on the problem of supporting efficiently operations `Increase` and `Decrease` using Data Structure 4.1. We consider separately the case of `Increase` and `Decrease` operations and we discuss different implementations known from the literature plus a variant which we propose.

4.3.1 Supporting `Increase` Operations

Observe that an operation `Increase`(x, y, ϵ) which increases the weight of an edge (x, y) that is not in $T(s)$ has nothing to do: actually, no node changes its distance due to this update. On the other hand, if (x, y) is an edge in $T(s)$, then only nodes in the subtree $T(y)$ rooted at y may change their distance d_s as a consequence of the update. This suggests that a clever dynamic algorithm should avoid working on nodes which do not belong to $T(y)$.

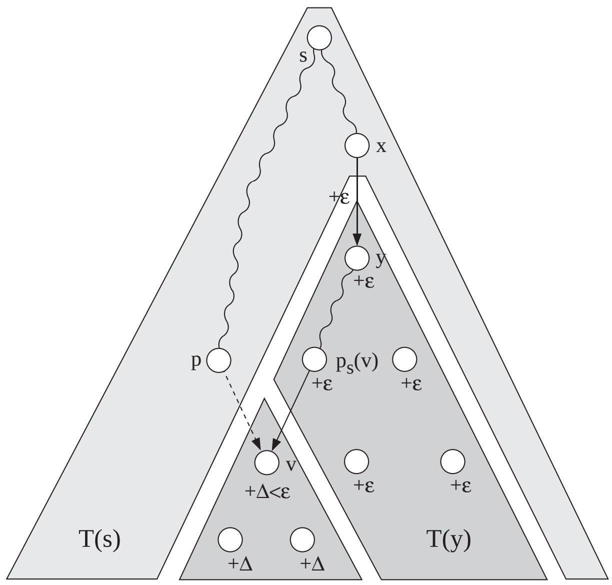


Figure 4.2: Shortest paths tree after an `Increase`(x, y, ϵ) on edge (x, y). Node v chooses as new parent node p and increases its distance from s by $\Delta < \epsilon$. Subtree $T(v)$ detaches from $T(y)$.

Figure 4.2 shows a shortest paths tree $T(s)$ after performing an `Increase` operation on edge (x, y). In the figure, node v chooses as new parent in $T(s)$ node p, which guarantees that its distance is increased by less than ϵ. As a consequence, subtree $T(y)$ is detached from $T(y)$, and $T(y)$ contracts.

Sometimes, it may happen that the new path $s \rightsquigarrow p \rightarrow v$ has the same length as the old shortest path $s \rightsquigarrow v$ before the update: thus, the distance $d_s(v)$ does not increase after the update, and the same holds for all nodes which are in the subtree $T(v)$ rooted at v. A subtle dynamic algorithm should avoid performing high-cost operations on such nodes. The problem here is that, while nodes in $T(y)$ are easily identifiable, nodes in $T(y)$ which are not going to change their distance after the update are less easy to detect if we still have not computed their distance variation.

In general, if we denote by $d'_s(v)$ the distance of node v after the update, then the distance variation $\delta_s(v) = d'_s(v) - d_s(v)$ satisfies the inequality: $0 \leq \delta_s(v) \leq \epsilon$. In the following we say that nodes v such that $\delta_s(v) > 0$ are AFFECTED.

We now discuss five possible implementations of the `Increase` operation, which we denote by `BFM`, `REW`, `DF`, `DFMN`, and `RR`. `BFM` is a simple-minded algorithm based on the static algorithm of Bellman Ford; `REW` is a raw implementation of the reweighting method, and `DF` is a heuristic variant of `REW` which we suggest; finally, `DFMN` [37, 19] and `RR` [64, 65] are the best known fully dynamic algorithms from the literature. We remark that `DFMN` and `RR` are more sophisticated variants of `REW` as well. The running time of all these algorithms is $O(m + n \log n)$ worst-case time, except for `BFM`, which runs in $O(mn)$ worst-case time.

`BFM`

A simple-minded implementation of both operations `Increase` and `Decrease` consists of recomputing d_s and $T(s)$ from scratch via Bellman-Ford's algorithm (see Section 2.4.1) after each update:

procedure `Increase`(x, y, ϵ)
1. **begin**
2. $\quad w(x, y) \leftarrow w(x, y) + \epsilon$
3. $\quad$ **if** $(x, y) \notin T(s)$ **then return**
4. $\quad \langle p_s, d_s \rangle \leftarrow$ `Bellman-Ford`$(\hat{G},\ s)$
5. **end**

The running time is dominated by the computation of Bellman-Ford's algorithm in line 4 and is clearly $O(mn)$.

`REW`

We now present a raw implementation of the reweighting technique summarized in Figure 4.1 which yields an algorithm for operation `Increase` that is better than recomputing from scratch through the best static algorithm. The implementation is as follows:

procedure `Increase`(x, y, ϵ)
1. **begin**
2. $\quad w(x, y) \leftarrow w(x, y) + \epsilon$
3. $\quad$ **if** $(x, y) \notin T(s)$ **then return**
4. $\quad$ **for each** $(u, v) \in E$ **do**
5. $\qquad \hat{w}(u, v) \leftarrow w(u, v) + d_s(u) - d_s(v)$
6. $\quad \langle \hat{p}_s, \hat{d}_s \rangle \leftarrow$ `Dijkstra`$(\hat{G},\ s)$
7. $\quad$ **for each** $v \in V$ **do**
8. $\qquad d_s(v) \leftarrow d_s(v) + \hat{d}_s(v)$
9. $\qquad p_s(v) \leftarrow \hat{p}_s(v)$
10. **end**

We first increase the weight of (x, y) so as to obtain an updated graph G and compute the reweighting function $\hat{w}$ from w (lines 4–5); then we apply Dijkstra's algorithm on $\hat{G}$ in order to compute distances $\hat{d}_s$ in $\hat{G}$ (line 6). We finally get back updated distances d_s in G from distances $\hat{d}_s$ in $\hat{G}$ (lines 7–8).

The correctness of this method is easily proved by observing that, after increasing the weight of (x, y), the updated $\hat{G}$ is the reweighted version of the updated G and does not contain negative-length cycles. Thus, Lemma 2.14

also holds for $\hat{G}$ and G. This allows us to get back correctly distances in G from distances in $\hat{G}$ (lines 7–8). The correctness of line 9 follows from Lemma 2.14.

The running time is dominated by the computation of Dijkstra's algorithm in line 6 and is clearly $O(m + n \log n)$ (see Section 2.4.1).

Notice that `REW` is very similar to `BFM`: also `REW` recomputes from scratch shortest paths, but exploits the fact that $\hat{G}$ has nonnegative weights and runs `Dijkstra` on $\hat{G}$ instead of calling `Bellman-Ford` on G.

In practical implementations we can avoid computing explicitly $\hat{G}$ in lines 4–5 and retrieving back distances d_s from $\hat{d}_s$ and shortest paths tree p_s from $\hat{p}_s$ in lines 7–9, by using the relations $\hat{w}(x,y) = w(x,y) + d_s(x) - d_s(y)$ and $\hat{d}_s(v) = d'_s(v) - d_s(v)$ on the fly.

`DF`

We now describe our new variant. The observation that no node outside $T(y)$ can change its distance as a consequence of the weight increase led us to consider a heuristic improvement of `REW` which in many practical cases yields dramatic speedup. The main idea consists of starting Dijkstra's computation with a priority queue H already loaded with only the nodes in $T(y)$. The priority $\delta_s(v)$ of each node v in H is initially given as:

$$\delta_s(v) = \min_{(p,v)\in E \,\wedge\, p \notin T(y)} \{\hat{d}_s(p) + \hat{w}(p,v)\}.$$

It is easy to see that, by our definition of $\hat{w}$, $\hat{d}_s(p) = 0$ for any $p \notin T(y)$. The implementation of `Increase` that we propose comes next.

    procedure `Increase`(x, y, ϵ)
1. **begin**
2. $w(x,y) \leftarrow w(x,y) + \epsilon$
3.   **if** $(x,y) \notin T(s)$ **then return**
4.   **for each** $v \in T(y)$ **do**
5. $\delta_s(v) \leftarrow \min_{(p,v)\in E \,\wedge\, p \notin T(y)} \{\ \hat{w}(p,v)\ \}$
6. $H \leftarrow \{v \in V\ :\ v \in T(y)\}$
7.   **while** $H \neq \emptyset$ **do**
8.   **let** $u \in H$: $\delta_s(u) = \min_{w \in H} \delta_s(w)$
9. $d_s(u) \leftarrow d_s(u) + \delta_s(u)$
10. $H \leftarrow H - \{u\}$
11.   **for each** $v \in H$: $(u,v) \in E$ **do**
12.   **if** $\delta_s(v) > \delta_s(u) + \hat{w}(u,v)$ **then**
13. $\delta_s(v) \leftarrow \delta_s(u) + \hat{w}(u,v)$
14. $p_s(v) \leftarrow u$
15. **end**

Observe that our dynamic algorithm is an adaptation of Dijkstra's algorithm shown in Section 2.4.1. Indeed, lines 7–14 are equal to the main **while** loop of Dijkstra's algorithm, except that we use δ as distances and

$\hat{w}$ as weights, like we were using a reweighted graph. Moreover, the set H is no more initialized with just the source s, but with all the nodes in $T(y)$ (line 6). Initial priorities of nodes are no more $+\infty$, and 0 for the source, but are computed in lines 4–5 as explained above.

To make a direct comparison with `REW`, the main **while** loop is started in `DF` with a configuration of the data structures which is the same that would have been reached by performing several previous steps in Dijkstra's call of `REW`. This may save in practice a large amount of work, especially in the case where the size of the subtree $T(y)$ is much smaller than the size of the whole shortest paths tree $T(s)$. Since in the worst case $T(y)$ may be as large as $T(s)$, the asymptotic running time of `DF` is the same of `REW`, i.e., $O(m + n \log n)$.

◁ ◇ ▷

We now briefly recall two fully dynamic algorithms known from the literature: `RR` [64, 65] and `DFMN` [37, 19]. They feature the same structure of `DF`, but instead of inserting in H all the nodes in $T(y)$, only put in H affected nodes, i.e., nodes which are actually going to increase their distances after the update. This means that they avoid performing useless high-cost operations such as extractions of zero-valued minima from H in Dijkstra's main loop, remembering that a distance zero in $\hat{G}$ corresponds to a null variation of distance in G. While `RR` requires graphs not to have zero-length cycles for doing so, `DFMN` has no restrictions. The common structure of both `RR` and `DFMN` is the following:

procedure `Increase`(x, y, ϵ)
1. **begin**
2. $\quad w(x,y) \leftarrow w(x,y) + \epsilon$
3. $\quad$ **if** $(x,y) \notin T(s)$ **then return**
4. $\quad$ { identify affected nodes }
5. $\quad$ **for each** $v \in T(y)$ **do**
6. $\qquad \delta_s(v) \leftarrow \min_{(p,v)\in E \wedge p \text{ } is \text{ } not \text{ } affected}\{\ \hat{w}(p,v)\ \}$
7. $\quad H \leftarrow \{\ affected\ nodes\ \}$
8. $\quad$ **while** $H \neq \emptyset$ **do**
9. $\qquad$ **let** $u \in H$: $\delta_s(u) = \min_{w \in H} \delta_s(w)$
10. $\qquad H \leftarrow H - \{u\}$
11. $\qquad d_s(u) \leftarrow d_s(u) + \delta_s(u)$
12. $\qquad$ **for each** $v \in H$: $(u,v) \in E$ **do**
13. $\qquad\quad$ **if** $\delta_s(v) > \delta_s(u) + \hat{w}(u,v)$ **then**
14. $\qquad\qquad \delta_s(v) \leftarrow \delta_s(u) + \hat{w}(u,v)$
15. $\qquad\qquad p_s(v) \leftarrow u$
16. **end**

Note that, differently from `DF`, the algorithm performs an additional phase of identifying affected nodes in line 4. Affected nodes are then used in place of nodes in $T(y)$ in the rest of the pseudocode.

We provide only an informal description of the two algorithms, referring the interested reader to [64, 65, 37, 19] for more details.

`RR`

In addition to the objects in Data Structure 4.1 `RR` maintains a subset SP of the edges of G, containing the edges of G that belong to at least one shortest path from s to the other nodes of G, i.e., $SP = \{\ (u,v) \in E\ :\ d_s(v) = d_s(u) + w(u,v)\ \}$.

If G has only positive-length cycles, then the directed graph $SP(G) = (V, SP)$ with node set V and edge set SP is acyclic and is used for finding affected nodes as follows.

We assume that the weight of edge (x, y) has been increased. `RR` maintains a work set containing nodes that have been identified as affected, but have not yet been processed. Initially, y is inserted in that set only if there are no further edges in SP entering y after the operation. Nodes in the work set are processed one by one, and when node u is processed, all edges (u, v) leaving u are deleted from SP, and v is inserted in the work set. All nodes that are identified as affected during this phase are inserted in the work set.

`RR` runs in $O(m_a + n_a + n_a \log n_a)$ per update, where n_a is the number of nodes affected by the update and m_a is the number of edges having at least one affected endpoint. This gives $O(m + n \log n)$ time in the worst case.

`DFMN`

`DFMN` is able to detect affected nodes even in presence of zero-length cycles in the graph.

The procedure colors the nodes in $T(y)$ as follows: $z \in T(y)$ is white if it changes neither the parent in $T(y)$ nor the distance; z is pink if it changes the parent; z is red if it changes the distance. In addition, a new parent in the shortest paths tree is given to each pink node. This is done by inserting the nodes in a heap, extracting them in non-decreasing order of their old distance from the source, and searching, for each of these nodes, an alternative shortest path from the source. To achieve this goal a quite complicated procedure is given that deals with zero-length cycles. At the end of the procedure, we retrieve affected nodes which are red.

`DFMN` has a worst-case complexity per update which depends on a structural parameter of the graph called *k-ownership*. If the graph has a k-bounded accounting function (as in the case of graphs with genus, arboricity, degree, treewidth or pagenumber limited by k) `Decrease` operations require $O(\min\{m, kn_a\}\ \log n)$ worst-case time, while `Increase` operations require $O(\min\{m \log n, k(n_a + n_b) \log n + n\})$ worst-case time. Here n_a is the number of nodes affected by the update and n_b is the number of nodes considered by

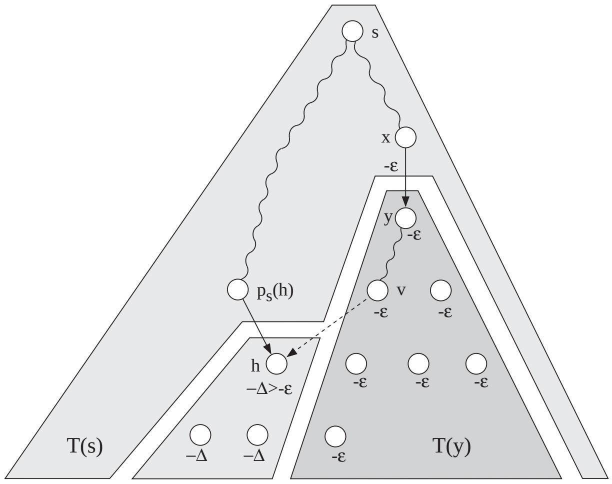


Figure 4.3: Shortest paths tree after a `Decrease`(x, y, ϵ) on edge (x, y). Node h chooses as new parent node v and decreases distance from s by $\Delta < \epsilon$. Subtree $T(h)$ is attached to $T(y)$.

the algorithm. In terms of n and m, this gives $O(m + n \log n)$ time in the worst case.

4.3.2 Supporting `Decrease` Operations

Assume that an operation `Decrease`(x, y, ϵ) reduces the weight of edge (x, y) by a positive quantity ϵ. If $d_s(x) + w_{x,y} - \epsilon \geq d_s(y)$, then distances of nodes do not change due to this update and `Decrease` has nothing else to do.

On the other hand, if $d_s(x) + w_{x,y} - \epsilon < d_s(y)$, then the nodes in $T(y)$ decrease their distance by ϵ, and the updated subtree $T'(y)$ may include nodes not contained in $T(y)$. All these nodes are AFFECTED by the input change. If y is affected, then the new shortest paths from s to the affected nodes contain (x, y). If y is not affected then no negative cycle is added to G, and no affected nodes exist. Node x is affected if and only if the update operation introduces a negative-length cycle, and this allows it to detect such cycles dynamically during each `Decrease` operation.

Figure 4.3 shows a shortest paths tree $T(s)$ after performing a `Decrease` operation on edge (x, y). In the figure node h chooses as new parent in $T(s)$ node v which guarantees that its distance is decreased by less than ϵ. As a consequence, subtree $T(h)$ is attached to $T(y)$, which gets expanded.

Differently from the case of `Increase` operations where refinements were possible by exploiting the combinatorial property that affected nodes are a subset of $T(y)$, we only report one implementation, which is common to all the algorithms `DF`, `RR`, and `DFMN` based on the reweighting technique:

procedure `Decrease`(x, y, ϵ)
begin
    $w(x, y) \leftarrow w(x, y) - \epsilon$
    if $d_s(x) + w_{x,y} - \epsilon \geq d_s(y)$ **then return**
    $\delta_s(y) \leftarrow -\epsilon$
    $H \leftarrow \{y\}$
    while $H \neq \emptyset$ **do**
        let $u \in H$: $\delta_s(u) = \min_{w \in H} \delta_s(w)$
        $H \leftarrow H - \{u\}$
        $d_s(u) \leftarrow d_s(u) + \delta_s(u)$
        for each $v \in V$: $(u, v) \in E$ **do**
            if $\delta_s(v) > \delta_s(u) + \hat{w}(u, v)$ **then**
                if $v = x$ **then** { a negative-length cycle has been detected }
                $H \leftarrow H \cup \{v\}$
                $\delta_s(v) \leftarrow \delta_s(u) + \hat{w}(u, v)$
                $p_s(v) \leftarrow u$
end

Notice that this is basically a Dijkstra computation on the reweighted graph, except for the following differences:

1. the computation starts from node y instead of the source s;

2. the priorities of nodes in H are strictly negative;

3. the initial priority $\delta_s(y)$ of y is $-\epsilon$;

4. if a negative-length cycle is introduced, then it includes edge (x, y), and thus it can reported whenever node x gets negative priority.

The running time of this implementation of the `Decrease` operation is clearly $O(m + n \log n)$: this is the best known update bound for the fully dynamic single-source shortest paths problem.

$\triangleleft \diamond \triangleright$

We remark that the variant `DF` which we proposed in this section was inspired by the results of a first suite of experiments on random test sets aimed at comparing previous algorithms for the single-source shortest paths problem in a dynamic setting. In those experiments, we measured the number of affected nodes per update and we discovered that, when edge weights are chosen from a large set of possible values, this number is very close to the number of nodes in $T(y)$, where (x, y) is the updated edge. The obvious logical explanation was the following: the probability that a node v has two incoming edges (u_1, v) and (u_2, v) such that $d_s(u_1) + w(u_1, v) = d_s(u_2) + w(u_2, v) = d_s(v)$

(i.e., the probability of having two equivalent shortest paths which lead to v) gets smaller as the range of possible values of edge weights increases.

We then had the idea of removing from algorithms RR and DFMN the step of identifying affected nodes and we implemented our version DF which works directly on $T(y)$.

After implementing DF, we performed a second suite of experiments that we present in the next two sections, aimed at confirming the considerations that inspired the design of DF. As we will see in Section 4.5, our considerations were fully confirmed by the experiments, which showed the clear superiority of DF when edge weights are chosen from a large set of possible values.

4.4 Experimental Setup

In this section we describe our experimental framework, presenting the problem instance generators, the performance indicators we consider, and some relevant implementation details. All codes being compared have been implemented by the author as C++ classes using advanced data types of LEDA [59] (version 3.6.1).

The software package, including algorithm implementations and test sets generators used in the preparation of this chapter, is available over the Internet at the URL:

`ftp://www.dis.uniroma1.it/pub/demetres/experim/dsplib-1.1/`

Our experiments were performed on a SUN Workstation Sparc Ultra 10 with a single 300 MHz processor and 128 MB of main memory running UNIX Solaris 5.7. All C++ programs were compiled by the GNU g++ compiler version 1.1.2 with optimization level O4. Each experiment consisted of maintaining both the distance of nodes from the source and the shortest paths tree in a random directed graph by means of different algorithms upon a random mixed sequence of Increase and Decrease operations.

4.4.1 Algorithms Under Evaluation

The algorithms that we consider in our experimental study are: BFM, DFMN, RR and DF. We did not include the REW implementation of the reweighting technique as it proved itself to be very slow if compared to DFMN, RR and DF.

We put effort to producing C++ codes for algorithms DFMN, RR and DF in such a way that their running times can be compared as fairly as possible. In particular, we avoided creating "out of the paper" implementations of algorithms DFMN and RR. For example, in RR we do not explicitly maintain the shortest paths dag SP with a separate data structure so as to avoid additional maintenance overhead that may penalize RR when compared against the other algorithms: actually, we maintain this information implicitly using the distance labels of nodes. In general, we tried to keep in mind the high-level algorithmic ideas while devising fast codes.

For these reasons, we used just one code for performing `Decrease` and we focused on hacking and tweaking codes for `Increase`. We believe that the effect of using LEDA data structures does not affect the ranking of different algorithms. More details about our codes can be directly found in our experimental package distributed over the Internet. In the remainder of this paper, we refer to `ALL-DECR` as the `Decrease` code and to `DFMN`, `RR` and `DF` as the `Increase` codes.

4.4.2 Graph and Sequence Generators

We used four random generators for synthesizing the graphs and the sequences of updates:

- `gen_graph(n,m,s,min,max)`: builds a random directed graph with n nodes, m edges and integer edge weights w s.t. `min`$\leq w \leq$`max`, forming no negative or zero length cycle and with all nodes reachable from the source node `s`. Reachability from the source is obtained by first generating a connecting path through the nodes as suggested in [9]; remaining edges are then added by uniformly and independently selecting pairs of nodes in the graph. To avoid introducing negative and zero length cycles we use the method of potentials described in [41]. The idea consists of first generating a random potential function $\phi : V \to \mathcal{R}$ and a random positive weight function $\nu : E \to \mathcal{R}^+$: the weight of an edge (u,v) is then generated as $w(u,v) \leftarrow \phi(u) - \phi(v) + \nu(u,v)$. By telescoping sums over cycles, we can easily prove that all cycles have nonnegative length.

- `gen_graph_z(n,m,s,min,max)`: similar to `gen_graph`, but all cycles in the generated graphs have exactly length zero. This is obtained by considering edge weights which are differences of potentials: $w(u,v) \leftarrow \phi(u) - \phi(v)$.

- `gen_seq(G,q,min,max)`: issues a mixed sequence of `q` `Increase` and `Decrease` operations on edges chosen at random in graph `G` without introducing negative and zero length cycles. Weights of edges are updated so that they always fit in the range [`min`,`max`]. Negative and zero length cycles are avoided by using the same potentials ϕ used in the generation of weights of edges of `G` and generating a different positive function ν. Optionally, the following additional constraints are supported:

  - *Modifying Sequence*: each `Increase` or `Decrease` operation is chosen among the operations that actually modify some shortest path from the source.

  – *Alternated Sequence*: the sequence has the form `Increase-Decrease-Increase-Decrease`..., where each pair of consecutive `Increase-Decrease` updates is performed on the same edge.

- `gen_seq_z(G,q,min,max)`: similar to `gen_seq`, but the update operations in the generated sequences force cycles in the graph `G` to have length zero. Again, this is obtained by considering edge weights as just difference of potentials ϕ, without using ν.

All our generators are based on the LEDA pseudo-random source of numbers. We initialized the random source with a different odd seed for each graph and sequence we generated.

4.4.3 Performance Indicators

We considered several performance indicators for evaluating and comparing the different codes. In particular, for each experiment and for each code we measured: (a) the average running time per update operation during the whole sequence of updates; (b) the average number of nodes processed in the distance-update phase of algorithms. Again, this is per update operation during the whole sequence of updates.

Indicator (b) is very important for comparing the dynamic algorithms as it measures the actual portion of the shortest paths tree for which they perform high-cost operations such as extractions of minima from a priority queue. It is interesting to observe that, if an `Increase` operation is performed on an edge (x, y), the value of the indicator (b) measured for both `RR` and `DFMN` reports the number of affected nodes that change their distance from the source since that update, while the value of (b) measured for `DF` reports the number of nodes in the shortest paths tree $T(y)$ rooted at y before the update.

Other measured indicators were: (c) the maximum running time per update operation; (d) the average number of scanned edges per update operation during the whole sequence of updates; (e) the total time required for initializing the data structures.

The running times were measured by the UNIX system call `getrusage()` and are reported in milliseconds. Indicators (b) and (d) were measured by annotating the codes with probes. The values of all the indicators are obtained by averaging over 15 trials. Each trial consists of a graph and a sequence randomly generated through `gen_graph` or `gen_graph_z` and `gen_seq` or `gen_seq_z`, respectively, and is obtained by initializing the pseudo-random generator with a different seed.

4.5 Experimental Results

The purpose of this section is to identify with experimental evidence the more convenient algorithm to use in practice in a fully dynamic environment. We performed several preliminary experiments on random graphs and random update sequences for different parameters defining the test sets aiming at comparing and separating the performances of the different algorithms. In particular, our main goals were the following:

1. Estimating the practical advantages of using fully dynamic algorithms based on the reweighting technique instead of dynamic adaptations of the static algorithm of Bellman-Ford.

2. Evaluating the impact of constants hidden in the asymptotic bounds. This is particularly important as all the considered implementations of `Increase` and `Decrease` operations feature the same $O(m + n \log n)$ bound on the running time.

3. Identifying under which conditions it is worth to identify affected nodes before updating the distances, as done by `RR` and `DFMN`, and when the simpler strategy of working on the larger subtree $T(y)$ adopted by `DF` is to be preferred.

Our first experiment showed that the time required by an `Increase` may significantly depend upon the width of the interval of the edge weights in the graph:

- *Increasing edge weight interval*: we ran our `DFMN`, `RR` and `DF` codes on mixed sequences of 2000 modifying update operations performed on graphs with 300 nodes and $m = 0.5n^2 = 45000$ edges and with edge weights in the range $[-2^k, 2^k]$ for values of k increasing from 3 to 10. The results of this test for `Increase` operations are shown in Figure 4.4. It is interesting to note that the smaller is the width of the edge weight interval, the larger is the gap between the number of affected nodes considered by `RR` during any `Increase` operation on an edge (x, y), and the number of nodes in $T(y)$ scanned by `DF`. In particular, `RR` is faster than `DF` for weight intervals up to $[-32, 32]$, while `DF` improves upon `RR` for larger intervals. This experimental result agrees with the fact that `RR` is theoretically efficient in output bounded sense, but spends more time than `DF` for identifying affected nodes. The capacity of identifying affected nodes even in the presence of zero cycles penalizes `DFMN` that is always slower than `RR` and `DF` on these problem instances with no zero cycles.

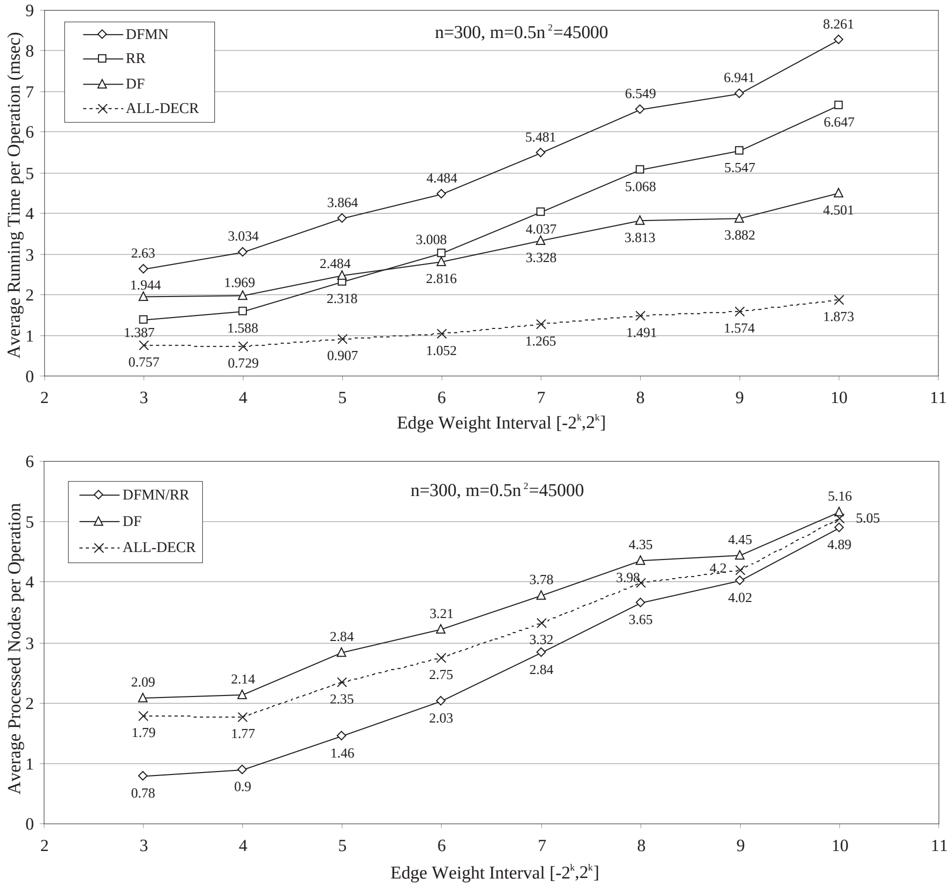
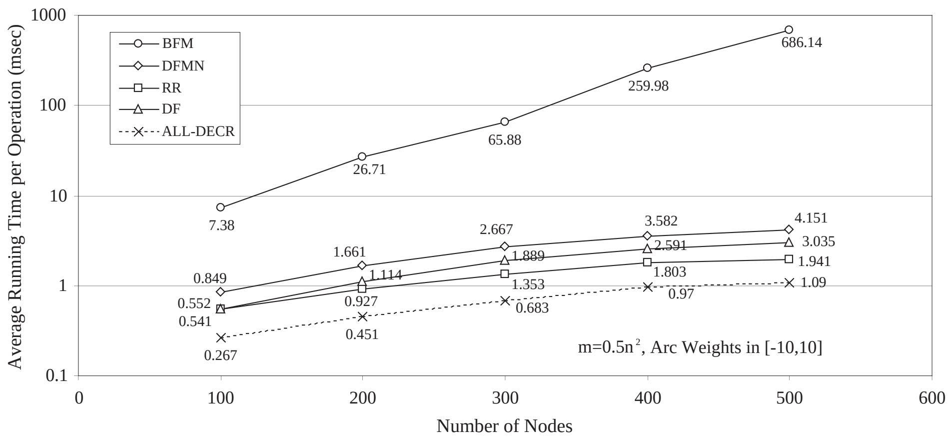


Figure 4.4: Experiments performed with $n = 300$ and $m = 0.5n^2 = 45000$ for edge weight intervals increasing from $[-8, 8]$ to $[-1024, 1024]$.

In our second suite of experiments, we ran BFM, DFMN, RR and DF codes on random sequences of 2000 modifying updates performed both on dense and sparse graphs with no negative and zero cycles and for two different ranges of the edges weights. In particular, we performed two suites of tests:

- *Increasing number of nodes*: we measured the running times on dense graphs with $100 \leq n \leq 500$ and $m = 0.5n^2$ for edge weights in $[-10, 10]$ and $[-1000, 1000]$. We repeated the experiment on larger sparse graphs with $1000 \leq n \leq 3000$ and $m = 30n$ for the same edge weights intervals and we found that the performance indicators follow the same trend of those of dense graphs that are shown in Figure 4.5. This experiment agrees with the first one and confirms that, on graphs with edge density 50%, DF beats RR for large weight intervals and RR beats DF for small



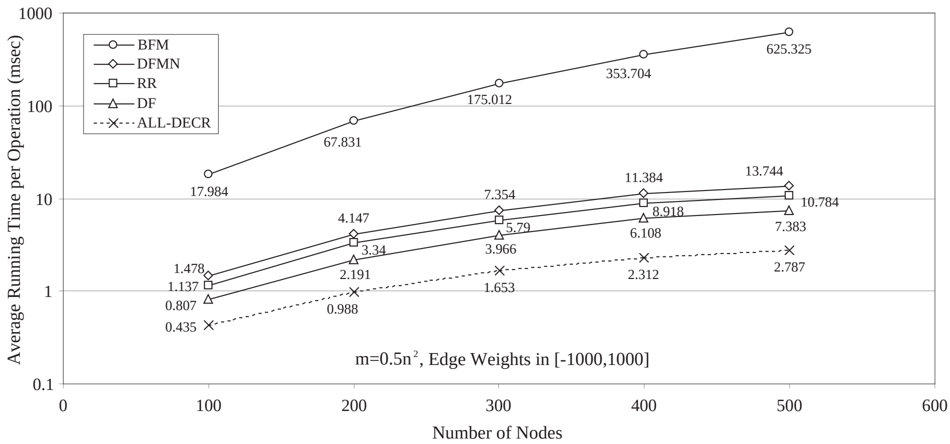


Figure 4.5: Experiments performed with $100 \leq n \leq 500$ and $m = 0.5n^2$ for edge weights in the range $[-10, 10]$ and $[-1000, 1000]$.

weight intervals. Notice that the dynamic codes we considered are better by several orders of magnitude than recomputing from scratch through the LEDA `BFM` code.

- *Increasing number of edges*: we retained both the running times and the number of nodes processed in the distance-update phase of algorithms on dense graphs with $n = 300$ and $0.05n^2 \leq m \leq 0.9n^2$ for edge weights in $[-10, 10]$ and $[-1000, 1000]$. We repeated the experiment on larger sparse graphs with $n = 2000$ and $10n \leq m \leq 50n$ for the same edge weights intervals and again we found similar results. Performance indicators for this experiment on dense graphs are shown in Figure 4.6 and Figure 4.7 and agree with the ones measured in the first test for what concerns the edge weight interval width. However, it is interesting to note that even

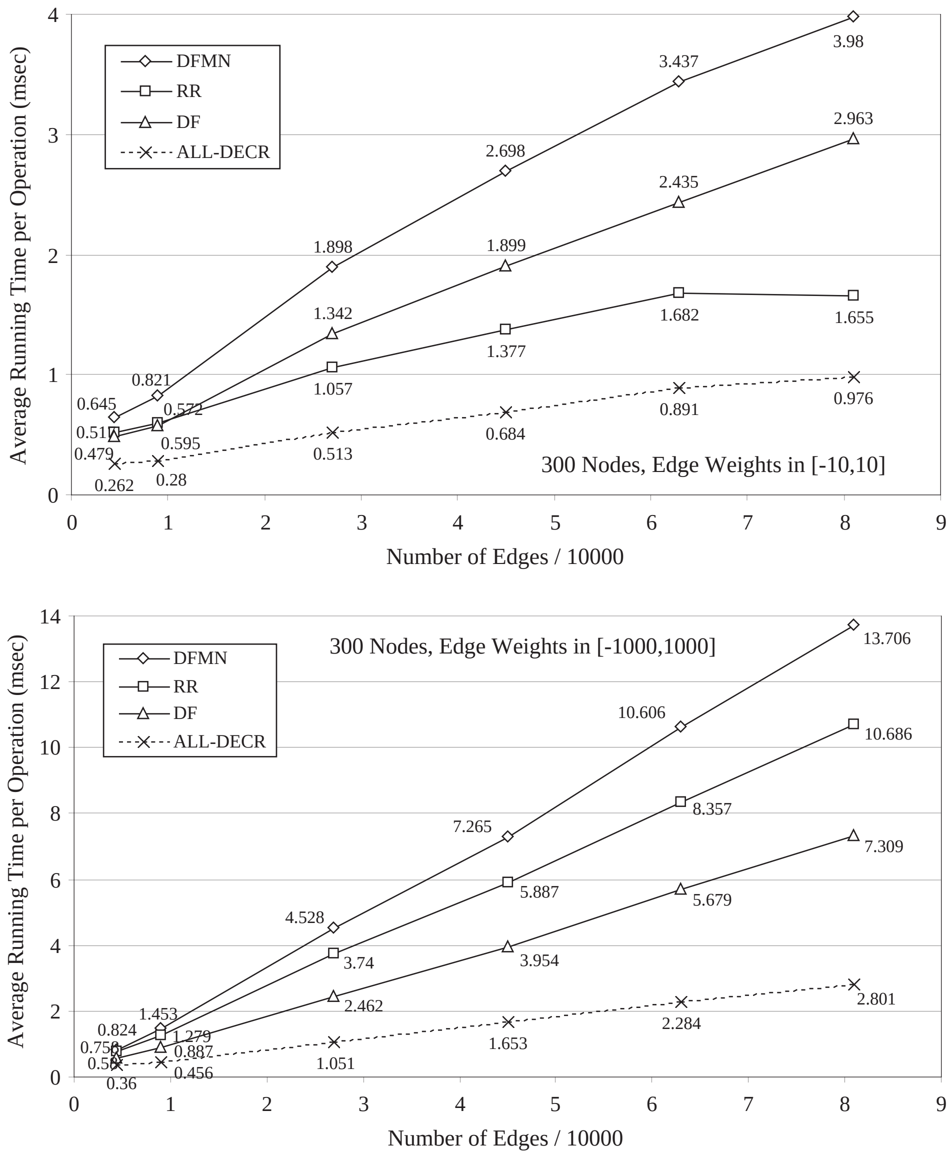
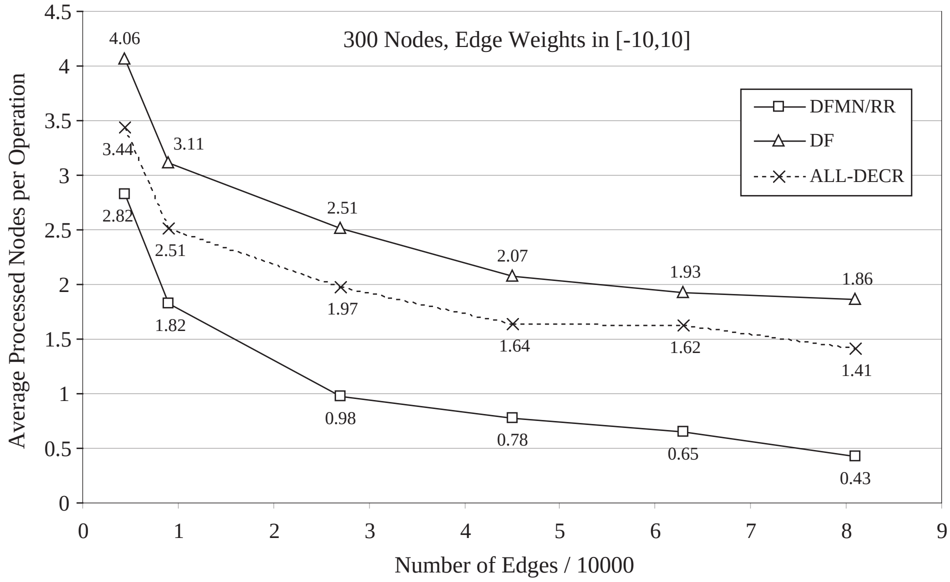


Figure 4.6: Running Time: experiments performed with $n = 300$, $0.05n^2 \leq m \leq 0.9n^2$ for edge weights in the range $[-10, 10]$ and $[-1000, 1000]$.



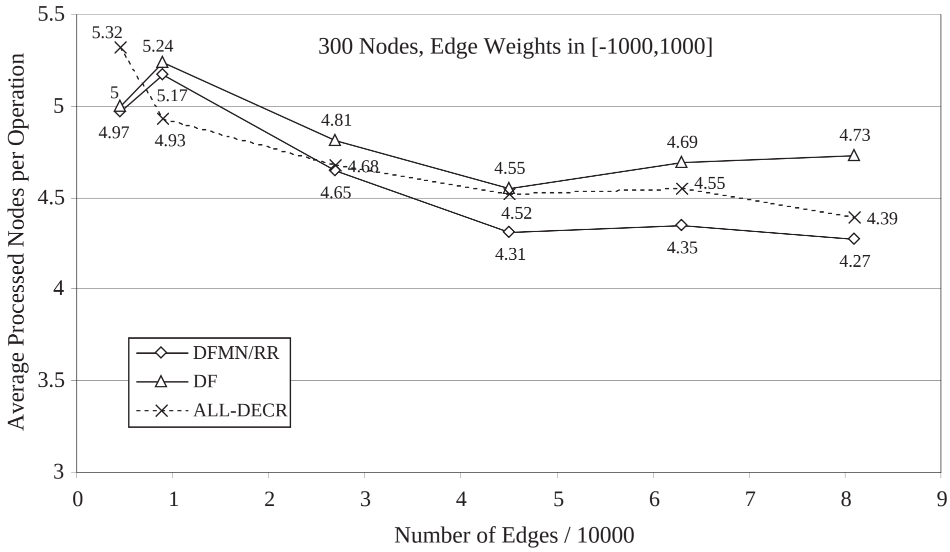


Figure 4.7: Average number of processed nodes: experiments performed with $n = 300$, $0.05n^2 \leq m \leq 0.9n^2$ for edge weights in the range $[-10, 10]$ and $[-1000, 1000]$.

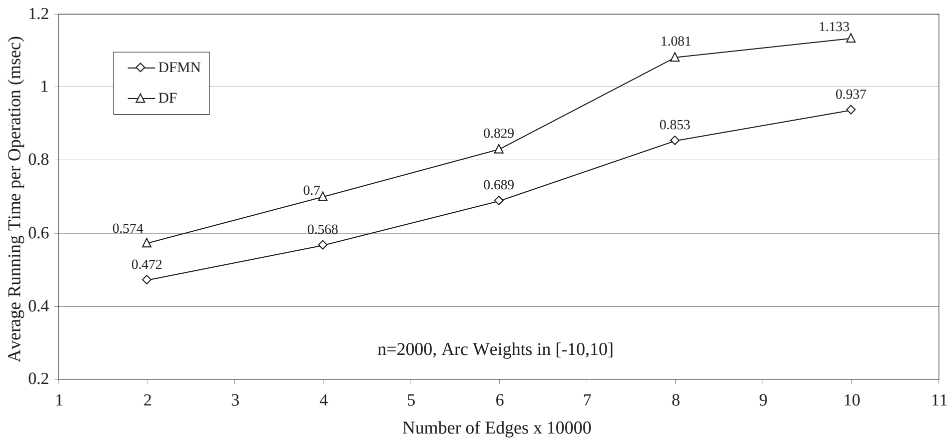


Figure 4.8: Experiments performed with $n = 2000$, $10n \leq m \leq 50n$ for edge weights in $[-10, 10]$. All cycles have zero length during updates.

for small weight ranges, if the edge density is less than 10%, the running time of `DF` slips beneath that of `RR`.

As from the previous tests our `DFMN` code is always slower than `RR` and `DF`, our third experiment aims at investigating if families of problem instances exist for which `DFMN` is a good choice for a practical dynamic algorithm. As it is able to identify affected nodes even in presence of zero cycles, we were not surprised to see that `DFMN` beats in practice `DF` in a dynamic setting where graphs have many zero cycles. We remark that `RR` is not applicable in this context.

- *Increasing number of edges and zero cycles*: we ran `DFMN` and `DF` codes on random graphs with 2000 nodes, $10n \leq m \leq 50n$, weights in $[-10, 10]$, all zero cycles, and subject to 2000 random alternated and modifying updates per sequence. We used generators `gen_graph_z` and `gen_seq_z` to build the input samples. Figure 4.8 shows the measured running times of `Increase` operations for this experiment, proving the superiority of `DFMN` w.r.t. `DF` on such instances.

Performance indicators (c), (d) and (e) provided no interesting additional hint on the behavior of the algorithms and therefore we omit them from our discussion: the interested reader can find in the experimental package the detailed results tables of our tests.

4.6 Conclusions

In this chapter we have considered a preliminary experimental study of the fully dynamic single-source shortest paths problem. We have also suggested a new variant `DF` of the best known algorithms for the problem.

Our variant was especially designed to be simple and fast in practice while matching the worst-case asymptotic complexity of the best algorithms. It was inspired by the results of a first suite of experiments on random test sets aimed at comparing previous algorithms for the single-source shortest paths problem in a dynamic setting. In those experiments, we measured the average number of affected nodes per edge weight update and we discovered that, when edge weights in the graph are chosen from a large set of possible values, this number is very close to the number of nodes in the subtree rooted at the head of the modified edge. In this case, we conjectured that algorithms `RR` and `DFMN`, which spend time in identifying affected nodes with the hope of speeding up the final phase of updating distances, may not be the fastest choice in practice. This led us to consider a simplified version `DF` which avoids looking for affected nodes, and we implemented it.

We performed a second suite of experiments, presented in this chapter, which confirmed our conjecture and showed the clear superiority of `DF` when edge weights are chosen from a large set of possible values.

Some results provided by our experiments were:

- all the considered dynamic algorithms based on the reweighting technique are faster by several orders of magnitude than recomputing from scratch with the best static algorithm `BFM`: this yields clues to the fact that constant factors in practical implementations can be very small for this problem;

- the running time of all the considered dynamic algorithms is affected by the width of the interval of edge weights. In particular, dynamic algorithms are faster if weights come from a small set of possible values;

- `RR` is preferable when edge weights are small integers, `DFMN` is preferable when there are many zero-length cycles in the graph, and `DF` is the best choice in any other case;

We have also showed that the reweighting technique of Edmonds and Karp, originally designed for network flows problems, is the key to solving efficiently the fully dynamic single-source shortest paths problem. Indeed, this method is the skeleton of all the best known algorithms for the problem, including our variant. All these algorithms perform updates in $O(m + n \log n)$ worst-case time per operation, instead of the $O(mn)$ worst-case time of recomputing from scratch.

Chapter 5

Conclusions and Further Directions

In this dissertation we have investigated two of the most fundamental dynamic problems on directed graphs: the fully dynamic transitive closure (Chapter 3) and the fully dynamic single-source shortest paths problem (Chapter 4).

In Chapter 3 we have studied the fully dynamic transitive closure problem theoretically and we have devised new algorithms which improve the best known bounds for the problem. These results have been obtained by means of a novel technique which consists of casting fully dynamic transitive closure into the problem of maintaining polynomials over matrices. In particular, we have proposed efficient data structures for maintaining polynomials over Boolean matrices subject to updates of their variables (Section 3.4.1) and for maintaining implicitly matrices over integers subject to simultaneous updates of multiple entries (Section 3.4.2).

The first data structure led us to devise a new deterministic fully dynamic transitive closure algorithm which supports updates in $O(n^2)$ amortized time and answers reachability queries with just one table lookup (Section 3.6). This algorithm hinges upon the equivalence between transitive closure and matrix multiplication on a closed semiring and is the fastest known fully dynamic transitive closure algorithm with constant query time.

A surprisingly simple method for supporting in subquadratic time operations on dynamic matrices (Section 3.4.2), combined with a previous idea of counting paths in acyclic digraphs [53], yielded the randomized algorithm for acyclic digraphs presented in Section 3.7.1: this algorithm, for the first time in the study of fully dynamic transitive closure, breaks through the $O(n^2)$ barrier on the single-operation complexity of the problem, supporting updates in $O(n^{1.58})$ and queries in $O(n^{0.58})$ worst-case time.

In Chapter 4 we have addressed the fully dynamic single-source shortest paths problem, reporting a preliminary experimental study of the best known algorithms for it. We have also discussed how experiments helped us devise a new variant of those algorithms, especially designed to be as fast as possible in practice.

Experimental results showed that recomputing a shortest paths tree from scratch with the best static algorithm may be several orders of magnitude slower than updating it by means of an efficient fully dynamic algorithm: this confirmed the theoretical analysis and suggested that constants hidden in the asymptotic bounds known for this problem are rather small in practice. Experiments also showed that our variant is always faster than the other considered algorithms, except for the case where edge weights of the graph are chosen from a small set of possible values.

◁ ◇ ▷

To conclude, we address some open problems and directions for future research which we consider a natural continuation of the work of this thesis. We mention a few of them:

- Extending the matrix-based framework presented in Chapter 3 to the more general case of fully dynamic all-pairs shortest paths; we recall that, similarly to transitive closure, this problem can described in terms of algebraic operations on closed semirings as well [1].

- Devising *deterministic* subquadratic algorithms for fully dynamic transitive closure, as well as considering the design of subquadratic solutions for *general* digraphs.

- Investigating lower bounds for fully dynamic transitive closure.

- Evaluating experimentally the performance of the deterministic algorithms presented in Section 3.5 and in Section 3.6. In particular, since our algorithms use matrix-based data structures, it would be interesting to investigate cache effects in a dynamic setting.

- Study better trade-offs between query and update operations for fully dynamic transitive closure.

- Considering query-update trade-offs for fully dynamic single source shortest paths.

Bibliography

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

[2] Ahuja, R., Magnanti, T., and Orlin, J. *Network Flows: Theory, Algorithms and Applications.* Prentice Hall, Englewood Cliffs, NJ, 1993.

[3] Alberts, D., Cattaneo, G., Italiano, G., Nanni, U., and Zaroliagis, C. A software library of dynamic graph algorithms. In *Proc. Workshop on Algorithms and Experiments (ALEX'98)* (1998), pp. 129–136.

[4] Anderson, R. The role of experiment in the theory of algorithms. In *Proceedings of the 5th DIMACS Challenge Workshop* (1996). Available over the Internet at the URL:
`http://www.cs.amherst.edu/ dsj/methday.html`.

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

[6] Brown, G., and King, V. Space-efficent methods for maintaining shortest paths and transitive closure: Theory and practice. Personal communication, July 2000.

[7] Chaudhuri, S., and Zaroliagis, C. Shortest paths in digraphs of small treewidth. part I: Sequential algorithms. *Algorithmica 27*, 3 (2000), 212–226. Special Issue on Treewidth.

[8] Cherkassky, B., and Goldberg, A. Negative-cycle detection algorithms. In *Proc. European Symposium on Algorithms (ESA'96), LNCS 1136* (1996), pp. 349–363.

[9] Cherkassky, B., Goldberg, A., and Radzik, T. Shortest paths algorithms: Theory and experimental evaluation. *Mathematical Programming 73* (1996), 129–174.

[10] Cicerone, D., Frigioni, D., Nanni, U., and Pugliese, F. Counting edges in a digraph. In *Proc. Workshop on Graph-Theoretic Concepts in Computer Science (WG'96), LNCS 1197* (1996), pp. 85–100.

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

[12] Cormen, T., Leiserson, C., and Rivest, R. *Introduction to Algorithms*. The MIT Press, 1990.

[13] Crescenzi, P., Demetrescu, C., Finocchi, I., and Petreschi, R. Reversible Execution and Visualization of Programs with LEONARDO. *Journal of Visual Languages and Computing 11*, 2 (2000). An extended abstract appears in *Proceedings of the 1-st Workshop on Algorithm Engineering* (WAE'97), Venice, Italy, September 1997, 146-155. Leonardo is available at the URL: `http://www.dis.uniroma1.it/~demetres/Leonardo/`.

[14] Demetrescu, C., Di Giacomo, E., Finocchi, I., and Liotta, G. Visualizing geometric algorithms with wave: System demonstration. In *Proc. of the 10th Annual Fall Workshop on Computational Geometry (CG'00), University at Stony Brook* (2000).

[15] Demetrescu, C., and Finocchi, I. A General-Purpose Logic-Based Visualization Framework. In *Proceedings of the 7th International Conference in Central Europe on Computer Graphics, Visualization and Interactive Digital Media (WSCG'99)* (1999), pp. 55–62.

[16] Demetrescu, C., and Finocchi, I. Smooth Animation of Algorithms in a Declarative Framework. In *Proceedings of the 15th IEEE Symposium on Visual Languages (VL'99)* (1999), pp. 280–287.

[17] Demetrescu, C., and Finocchi, I. A Technique for Generating Graphical Abstractions of Program Data Structures. In *Proceedings of the 3rd International Conference on Visual Information Systems (VISUAL'99)* (1999), LNCS 1614, pp. 785–792.

[18] Demetrescu, C., Finocchi, I., and Liotta, G. Visualizing algorithms over the web with the publication-driven approach. In *Proc. of the 4-th Workshop on Algorithm Engineering (WAE'00), Saarbrücken, Germany. September 5-8* (2000).

[19] Demetrescu, C., Frigioni, D., Marchetti-Spaccamela, A., and Nanni, U. Maintaining shortest paths in digraphs with arbitrary arc weights: An experimental study. In *Proceedings of the 4-st Workshop*

on Algorithm Engineering (WAE'00), Saarbrücken, Germany, September 5-8 (2000).

[20] DEMETRESCU, C., AND ITALIANO, G. 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)* (2000), pp. 381–389.

[21] DEMETRESCU, C., AND ITALIANO, G. What do we learn from experimental algorithmics? In *Proceedings of the 25th International Symposium on Mathematical Foundations of Computer Science (MFCS'00), Bratislava, Slovak Republic, August 28 - September 1* (2000).

[22] DEMETRESCU, C. AND DI BATTISTA, G. AND FINOCCHI, I. AND LIOTTA, G. AND PATRIGNANI, M. AND PIZZONIA, M. Infinite Trees and the Future. In *Proceedings of the 7-th International Symposium on Graph Drawing (GD '99), LNCS* (1999).

[23] DEMETRESCU, C. AND FINOCCHI, I. Break the "right" cycles and get the "best" drawing. In *Proc. of the 2nd International Workshop on Algorithm Engineering and Experiments (ALENEX'00)* (2000), B. Moret and A. Goldberg, Eds., pp. 171–182.

[24] DIJKSTRA, E. A note on two problems in connection with graphs. *Numerische Mathematik 1* (1959), 269–271.

[25] DJIDJEV, H., PANTZIOU, G., AND ZAROLIAGIS, C. Improved algorithms for dynamic shortest paths. *Algorithmica 28* (2000), 367–389.

[26] EDMONDS, J., AND KARP, R. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM 19* (1972), 248–264.

[27] EPPSTEIN, D., GALIL, Z., ITALIANO, G., AND NISSENZWEIG, A. Sparsification - a technique for speeding up dynamic graph algorithms. *Journal of the ACM 44* (1997).

[28] EVEN, S., AND GAZIT, H. Updating distances in dynamic graphs. *Methods of Operations Research 49* (1985), 371–387.

[29] EVEN, S., AND SHILOACH, Y. An on-line edge-deletion problem. *Journal of the ACM 28* (1981), 1–4.

[30] FEIGENBAUM, J., AND KANNAH, S. Dynamic graph algorithms. In *Handbook of Discrete and Combinatorial Mathematics.* pp. 583–591.

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

[32] Franciosa, P., Frigioni, D., and Giaccio, R. Semi dynamic shortest paths and breadth-first search on digraphs. In *Proc. 14th Annual Symposium on Theoretical Aspects of Computer Science, (STACS'97), LNCS 1200* (1997), pp. 26–40.

[33] Fredman, M., and Tarjan, R. Fibonacci heaps and their use in improved network optimization algorithms. *Journal of the ACM 34* (1987), 596–615.

[34] Frigioni, D., Ioffreda, M., Nanni, U., and Pasqualone, G. Experimental analysis of dynamic algorithms for the single source shortest path problem. *ACM Journal on Experimental Algorithmics 3* (1998).

[35] Frigioni, D., Marchetti-Spaccamela, A., and Nanni, U. Fully dynamic output bounded single source shortest path problem. In *Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'96)* (1996), pp. 212–221.

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

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

[38] Frigioni, D., Miller, T., Nanni, U., Pasqualone, G., Shaefer, G., and Zaroliagis, C. An experimental study of dynamic algorithms for directed graphs. In *Proc. European Symposium on Algorithms (ESA'98), LNCS 1461* (1998), pp. 320–331.

[39] Furman, M. 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.

[40] Galil, Z., Italiano, G., and Sarnak, N. Fully dynamic planarity testing. In *Proc. ACM Symp. on Theory of Computing (STOC'92)* (1992), pp. 495–506.

[41] Goldberg, A. Selecting problems for algorithm evaluation. In *Proc. 3-rd Workshop on Algorithm Engineering (WAE'99), LNCS 1668* (1999), pp. 1–11.

[42] Goldberg, A., and Radzik, T. A heuristic improvement of the bellman-ford algorithm. *Applied Math. Letters 6* (1993), 3–6.

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

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

[45] Holm, J., de Lichtenberg, K., and Thorup, M. Poly-logarithmic deterministic fully dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. In *Proc. 30th Symp. on Theory of Computing (STOC'98)* (1998), pp. 79–89.

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

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

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

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

[50] Johnson, D. A theoretician's guide to the experimental analysis of algorithms. In *Proceedings of the 5th DIMACS Challenge Workshop* (1996). Available over the Internet at the URL:
`http://www.cs.amherst.edu/ dsj/methday.html`.

[51] Khanna, S., Motwani, R., and Wilson, R. H. On certificates and lookahead on dynamic graph problems. In *Proc. 7th ACM-SIAM Symp. Discrete Algorithms* (1996), pp. 222–231.

[52] King, V. 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)* (1999), pp. 81–99.

[53] King, V., and Sagert, G. A fully dynamic algorithm for maintaining the transitive closure. In *Proc. 31st ACM Symposium on Theory of Computing (STOC'99)* (1999), pp. 492–498.

[54] Klein, P., and Subramanian, S. Fully dynamic approximation schemes for shortest paths problems in planar graphs. In *Proc. International Workshop on Algorithms and Data Structures, LNCS 709, pp. 443–451* (1993).

[55] Knuth, D., and Plass, M. Breaking paragraphs into lines. *Software-practice and experience 11* (1981), 1119–1184.

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

[57] Loubal, P. A network evaluation procedure. *Highway Research Record 205* (1967), 96–109.

[58] McGeoch, C. A bibliography of algorithm experimentation. In *Proceedings of the 5th DIMACS Challenge Workshop* (1996). Available over the Internet at the URL:
`http://www.cs.amherst.edu/ dsj/methday.html`.

[59] Mehlhorn, K., and Naher, S. leda, a platform for combinatorial and geometric computing. *Communications of the ACM 38* (1995), 96–102.

[60] Miller, T., and Zaroliagis, C. A first experimental study of a dynamic transitive closure algorithm. In *Proc. 1st Workshop on Algorithm Engineering (WAE'97)* (1997), pp. 64–73.

[61] Moret, B. Towards a discipline of experimental algorithmics. In *Proceedings of the 5th DIMACS Challenge Workshop* (1996). Available over the Internet at the URL:
`http://www.cs.amherst.edu/~dsj/methday.html`.

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

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

[64] RAMALINGAM, G. Bounded incremental computation. In *Lecture Notes in Computer Science 1089* (1996).

[65] RAMALINGAM, G., AND REPS, T. An incremental algorithm for a generalization of the shortest path problem. *Journal of Algorithms 21* (1996), 267–305.

[66] RAMALINGAM, G., AND REPS, T. On the computational complexity of dynamic graph problems. *Theoretical Computer Science 158* (1996), 233–277.

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

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

[69] SCHULZ, F., WAGNER, D., AND WEIHE, K. Dijkstra's algorithm online: an empirical case study from public railroad transport. In *Proc. 3rd Workshop on Algorithm Engineering (WAE'99)* (1999), pp. 110–123.

[70] STRASSEN, V. Gaussian elimination is not optimal. *Numerische Mathematik 14* (1969), 354–356.

[71] YANNAKAKIS, M. Graph-theoretic methods in database theory. In *Proc. 9-th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (1990), pp. 230–242.

[72] YELLIN, D. M. Speeding up dynamic transitive closure for bounded degree graphs. *Acta Informatica 30* (1993), 369–384.

[73] ZWICK, U. All pairs shortest paths in weighted directed graphs - exact and almost exact algorithms. In *Proc. of the 39th IEEE Annual Symposium on Foundations of Computer Science (FOCS'98)* (Los Alamitos, CA, November 8–11 1998), pp. 310–319.

Università *La Sapienza*
Dottorato di Ricerca in Ingegneria Informatica

Collection of Theses

V-93-1 Marco Cadoli. *Two Methods for Tractable Reasoning in Artificial Intelligence: Language Restriction and Theory Approximation.* June 1993.

V-93-2 Fabrizio d'Amore. *Algorithms and Data Structures for Partitioning and Management of Sets of Hyperrectangles.* June 1993.

V-93-3 Miriam Di Ianni. *On the complexity of flow control problems in Store-and-Forward networks.* June 1993.

V-93-4 Carla Limongelli. *The Integration of Symbolic and Numeric Computation by p-adic Construction Methods.* June 1993.

V-93-5 Annalisa Massini. *High efficiency self-routing interconnection networks.* June 1993.

V-93-6 Paola Vocca. *Space-time trade-offs in directed graphs reachability problem.* June 1993.

VI-94-1 Roberto Baldoni. *Mutual Exclusion in Distributed Systems.* June 1994.

VI-94-2 Andrea Clementi. *On the Complexity of Cellular Automata.* June 1994.

VI-94-3 Paolo Giulio Franciosa. *Adaptive Spatial Data Handling.* June 1994.

VI-94-4 Andrea Schaerf. *Query Answering in Concept-Based Knowledge Representation Systems: Algorithms, Complexity, and Semantic Issues.* June 1994.

VI-94-5 Andrea Sterbini. *2-Thresholdness and its Implications: from the Synchronization with PVchunk to the Ibaraki-Peled Conjecture.* June 1994.

VII-95-1 Piera Barcaccia. *On the Complexity of Some Time Slot Assignment Problems in Switching Systems.* June 1995.

VII-95-2 Michele Boreale. *Process Algebraic Theories for Mobile Systems.* June 1995.

VII-95-3 Antonella Cresti. *Unconditionally Secure Key Distribution Protocols.* June 1995.

VII-95-4 Vincenzo Ferrucci. *Dimension-Independent Solid Modeling.* June 1995.

VII-95-5 Esteban Feuerstein. *On-line Paging of Structured Data and Multi-threaded Paging.* June 1995.

VII-95-6 Michele Flammini. *Compact Routing Models: Some Complexity Results and Extensions.* June 1995.

VII-95-7 Giuseppe Liotta. *Computing Proximity Drawings of Graphs.* June 1995.

VIII-96-1 Luca Cabibbo. *Querying and Updating Complex-Object Databases.* May 1996.

VIII-96-2 Diego Calvanese. *Unrestricted and Finite Model Reasoning in Class-Based Representation Formalisms.* May 1996.

VIII-96-3 Marco Cesati. *Structural Aspects of Parameterized Complexity.* May 1996.

VIII-96-4 Flavio Corradini. *Space, Time and Nondeterminism in Process Algebras.* May 1996.

VIII-96-5 Stefano Leonardi. *On-line Resource Management with Application to Routing and Scheduling.* May 1996.

VIII-96-6 Rosario Pugliese. *Semantic Theories for Asynchronous Languages.* May 1996.

IX-97-1 Paola Alimonti. *Local search and approximability of MAX SNP problems.* May 1997.

IX-97-2 Tiziana Calamoneri. *Does Cubicity Help to Solve Problems?.* May 1997.

IX-97-3 Paolo Di Blasio. *A Calculus for Concurrent Objects: Design and Control Flow Analysis.* May 1997.

IX-97-4 Bruno Errico. *Intelligent Agents and User Modelling.* May 1997.

IX-97-5 Roberta Mancini. *Modelling Interactive Computing by exploiting the Undo.* May 1997.

IX-97-6 Riccardo Rosati. *Autoepistemic Description Logics.* May 1997.

IX-97-7 Luca Trevisan. *Reductions and (Non-)Approximability.* May 1997.

X-98-1 Gianluca Battaglini. *Analysis of Manufacturing Yield Evaluation of VLSI/WSI Systems: Methods and Methodologies.* April 1998.

X-98-2 Piergiorgio Bertoli. *Using OMRS in practice: a case study with Acl-2.* April 1998.

X-98-3 Chiara Ghidini. *A semantics for contextual reasoning: theory and two relevant applications.* April 1998.

X-98-4 Roberto Giaccio. *Visiting complex structures.* April 1998.

X-98-5 Giampaolo Greco. *Dimension and structure in Combinatorics.* April 1998.

X-98-6 Paolo Liberatore. *Compilation of intractable problems and its application to artificial intelligence.* April 1998.

X-98-7 Fabio Massacci. *Efficient approximate tableaux and an application to computer security.* April 1998.

X-98-8 Chiara Petrioli. *Energy-Conserving Protocols for Wireless Communications.* April 1998.

X-98-9 Giulio Balestreri. *An Algebraic Semantics for the Shared Spaces Coordination Languages.* April 1999.

XI-99-1 Luca Becchetti. *Efficient Resource Management in High Bandwidth Networks.* April 1999.

XI-99-2 Nicola Cancedda. *Text Generation from Message Understanding Conference Templates.* April 1999.

XI-99-3 Luca Iocchi. *Design and Development of Cognitive Robots.* April 1999.

XI-99-4 Francesco Quaglia. *Consistent checkpointing in distributed computations: theoretical results and protocols.* April 1999.

XI-99-5 Milton Romero. *Disparity/Motion Estimation For Stereoscopic Video Processing.* April 1999.

XI-99-6 Massimiliano Parlione. *Remote Class Inheritance.* April 2000.

XII-00-1 Marco Daniele. *Advances in Planning as Model Checking.* April 2000.

XII-00-2 Walter Didimo. *Flow Techniques and Optimal Drawings of Graphs.* April 2000.

XII-00-3 Leonardo Tininini. *Querying Aggregate Data.* April 2000.

XII-00-4 Enver Sangineto. *Classificazione Automatica d'Immagini Tramite Astrazione Geometrica".* April 2001.

XIII-01-1 Camil Demetrescu. *Fully Dynamic Algorithms for Path Problems on Directed Graphs.* April 2001.

XIII-01-2 Giovanna Melideo. *Tracking Causality in Distributed Computations.* April 2001.

XIII-01-3 Maurizio Patrignani. *Visualization of Large Graphs.* April 2001.

XIII-01-4 Maurizio Pizzonia. *Engineering of Graph Drawing Algorithms for Applications.* April 2001.