Alcom-FT Technical Report Series
ALCOMFT-TR-01-25

Fully Dynamic Transitive Closure: Breaking Through the $O(n^2)$ Barrier*

Camil Demetrescu †   Giuseppe F. Italiano ‡

Abstract

In this paper we introduce a general framework for casting fully dynamic transitive closure into the problem of reevaluating polynomials over matrices. With this technique, we improve the best known bounds for fully dynamic transitive closure. In particular, we devise a deterministic algorithm for general directed graphs that achieves $O(n^2)$ amortized time for updates, while preserving unit worst-case cost for queries. In case of deletions only, our algorithm performs updates faster in $O(n)$ amortized time.

Our matrix-based approach yields an algorithm for directed acyclic graphs which breaks through the $O(n^2)$ barrier on the single-operation complexity of fully dynamic transitive closure. We can answer queries in $O(n^\epsilon)$ time and perform updates in $O(n^{\omega(1,\epsilon,1)-\epsilon}+n^{1+\epsilon})$ time, for any $\epsilon \in [0,1]$, where $\omega(1,\epsilon,1)$ is the exponent of the multiplication of an $n\times n^\epsilon$ matrix by an $n^\epsilon \times n$ matrix. The current best bounds on $\omega(1,\epsilon,1)$ imply an $O(n^{0.575})$ query time and an $O(n^{1.575})$ update time. Our subquadratic algorithm is randomized, and has one-side error.

1 Introduction

In this paper we present fully dynamic algorithms for maintaining the transitive closure of a directed graph. A dynamic graph algorithm maintains a given property on a graph subject to dynamic changes, such as edge insertions and edge deletions. We say that an algorithm is *fully dynamic* if it can handle both edge insertions and edge deletions. A *partially dynamic* algorithm can handle either edge insertions or edge deletions, but not both: we say that it is *incremental* if it supports insertions only, and *decremental* if it supports deletions only. In the *fully dynamic transitive closure problem* we wish to maintain a directed graph $G = (V, E)$ under an intermixed sequence of the following operations:

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

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

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

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

Research on dynamic transitive closure spans over two decades. Before describing the results known, 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, 14], this approach yields $O(1)$ time per query and $O(n^\omega)$ time per update in the worst case, where ω is the best known exponent for matrix multiplication (currently $\omega < 2.736$ [2]).

Previous Work. For the *incremental* version of the problem, the first algorithm was proposed by Ibaraki and Katoh [7] in 1983: its running time was $O(n^3)$ over any sequence of insertions. This bound was later improved to $O(n)$ amortized time per insertion by Italiano [8] and also by La Poutré and van Leeuwen [13]. Yellin [15] gave an $O(m^*\delta_{max})$ algorithm for m edge insertions, where m^* is the number of edges in the final transitive closure and δ_{max} is the maximum out-degree of the final graph. All these algorithms maintain explicitly the transitive closure, and so their query time is $O(1)$.

The first *decremental* algorithm was again given by Ibaraki and Katoh [7], with a running time of $O(n^2)$ per

*This work has been partially supported by the IST Programme of the EU under contract n. IST-1999-14186 (ALCOM-FT), by the Italian Ministry of University and Scientific Research (Project "Algorithms for Large Data Sets: Science and Engineering") and by CNR, the Italian National Research Council.

†Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Via Salaria 113, 00198 Roma, Italy. Email: demetres@dis.uniroma1.it, URL: http://www.dis.uniroma1.it/~demetres/.

‡Dipartimento di Informatica, Sistemi e Produzione, Università di Roma "Tor Vergata", Via di Tor Vergata 110, 00133 Roma, Italy. Email: italiano@info.uniroma2.it, URL: http://www.info.uniroma2.it/~italiano/.

deletion. This was improved to $O(m)$ per deletion by La Poutré and van Leeuwen [13]. Italiano [9] presented an algorithm which achieves $O(n)$ amortized time per deletion on directed acyclic graphs. Yellin [15] gave an $O(m^*\delta_{max})$ algorithm for m edge deletions, where m^* is the initial number of edges in the transitive closure and δ_{max} is the maximum out-degree of the initial graph. Again, the query time of all these algorithms is $O(1)$. More recently, Henzinger and King [5] gave a randomized decremental transitive closure algorithm for general directed graphs with a query time of $O(n/\log n)$ and an amortized update time of $O(n \log^2 n)$.

The first *fully dynamic* transitive closure algorithm was devised by Henzinger and King [5] in 1995: they gave a randomized Monte Carlo algorithm with one-side error supporting a query time of $O(n/\log n)$ and an amortized update time of $O(n\hat{m}^{0.58}\log^2 n)$, 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 $O(n^{2.16}\log^2 n)$. Khanna, Motwani and Wilson [10] 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 [12] showed how to support queries in $O(1)$ time and updates in $O(n^{2.26})$ time for general directed graphs and $O(n^2)$ time for directed acyclic graphs; their algorithm is randomized with one-side error. The bounds of King and Sagert were further improved by King [11], who exhibited a deterministic algorithm on general digraphs with $O(1)$ query time and $O(n^2 \log n)$ amortized time per update operations, where updates are insertions of a set of edges incident to the same vertex and deletions of an arbitrary subset of edges. We remark that all these algorithms (except [11]) use fast matrix multiplication as a subroutine.

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.

Our Results. In this paper, we affirmatively answer both questions. We first exhibit 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 [11]. Our algorithm can also support within the same time bounds the generalized updates of [11], i.e., insertion of a set of edges incident to the same vertex and deletion of an arbitrary subset of edges. 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 [9], and improves over [5].

As our second contribution, we present the first algorithm which breaks through the $O(n^2)$ barrier on the single-operation time complexity of fully dynamic transitive closure. In particular, we show how to trade off query times for updates on directed acyclic graphs: each query can be answered in time $O(n^\epsilon)$ and each update can be performed in time $O(n^{\omega(1,\epsilon,1)-\epsilon} + n^{1+\epsilon})$, for any $\epsilon \in [0, 1]$, where $\omega(1, \epsilon, 1)$ is the exponent of the multiplication of an $n \times n^\epsilon$ matrix by an $n^\epsilon \times n$ matrix. Balancing the two terms in the update bound yields that ϵ must satisfy the equation $\omega(1, \epsilon, 1) = 1 + 2\epsilon$. The current best bounds on $\omega(1, \epsilon, 1)$ [2, 6] imply that $\epsilon < 0.575$ [16]. Thus, the smallest update time is $O(n^{1.575})$, which gives a query time of $O(n^{0.575})$. Our subquadratic algorithm is randomized, and has one-side error.

All our algorithms are based on a novel technique: we introduce a general framework for maintaining polynomials defined over matrices, and we cast fully dynamic transitive closure into this framework. In particular, 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 [14], Furman [4] and Fischer and Meyer [3]) 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 [5, 10, 12] 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.

2 Two Problems on Dynamic Matrices

2.1 Dynamic Reevaluation of Polynomials over Boolean Matrices

In this section we study the problem of reevaluating polynomials over Boolean matrices and subject to updates of their variables. We define these updates so that they can be useful later on for our original problem of dynamic transitive closure. In the following, we denote by $\mathcal{M}_n$ the set of $n \times n$ Boolean matrices, by I_n the unit matrix (i.e., the $n \times n$ Boolean matrix with 1 in the diagonal and 0 elsewhere), and by $+$ and $\cdot$ sum and multiplication on Boolean matrices, respectively. We also use operation $\ominus$ defined as follows: $C = A \ominus B$, where $C[i,j] = A[i,j] \wedge (\neg B[i,j])$. Notice that this corresponds to matrix difference, with the exception that $0 \ominus 1 = 0$. Moreover, if X is a matrix, we denote by $I_{X,i}$ and $J_{X,j}$ the matrices equal to X in the i-th row and in the j-th column, respectively, and null in any other entries.

The Problem. Without loss of generality, let

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

be a polynomial defined over $\mathcal{M}_n$, where any

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

has degree exactly k and variables $X_b^a \in \mathcal{M}_n$ are distinct. 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 define a data structure for P that supports the following operations:

- $Init(X_1^1 \dots X_k^h)$: given the initial value of variables, initializes the data structure and computes the value of P from scratch.
- $SetRow(i, \Delta X, X_b^a)$: sets to 1 any entries in the i-th row of the variable X_b^a of the polynomial P as specified by matrix ΔX (i.e., $X_b^a \leftarrow X_b^a + I_{\Delta X,i}$) and updates the maintained value of P.
- $SetCol(j, \Delta X, X_b^a)$: sets to 1 any entries in the j-th column of the variable X_b^a of the polynomial P as specified by matrix ΔX (i.e., $X_b^a \leftarrow X_b^a + J_{\Delta X,j}$) and updates the maintained value of P.
- $Reset(\Delta X, X_b^a)$: resets to 0 any entries of the variable X_b^a of the polynomial P as specified by matrix ΔX (i.e., $X_b^a \leftarrow X_b^a \ominus \Delta X$) and updates the maintained value of P.
- $Query()$: returns the maintained value of the polynomial P.

Note that for $k = 2$ this problem is simply related to matrix multiplication. Thus, it becomes interesting when $k > 2$.

We add to the previous five operations a further update operation especially designed for dynamic transitive closure:

- $LazySet(\Delta X, X_b^a)$: sets to 1 any entries of the variable X_b^a of the polynomial P as specified by matrix ΔX (i.e., $X_b^a \leftarrow X_b^a + \Delta X$), but does *not* update the maintained value of P.

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

Before stating our bounds, we observe that it is easy to support $Init$ in $O(h \cdot k \cdot n^\omega + h \cdot n^2)$, $SetRow$ and $SetCol$ in $O(k \cdot n^\omega)$, $Reset$ in $O(k \cdot n^\omega + h \cdot n^2)$, and $LazySet$ and $Query$ in $O(n^2)$. We now show how to improve the time bounds for $SetRow$, $SetCol$ and $Reset$ at the price of increasing the time complexity of $Init$. In particular, we support $SetRow$, $SetCol$ and $Reset$ in $O(k \cdot n^2)$ amortized time, $Query$ and $LazySet$ in $O(n^2)$, and $Init$ in $O(h \cdot k^2 \cdot n^3)$ when $k > 1$ and $O(h \cdot n^2)$ when $k = 1$. In the special case where no $SetRow$, $SetCol$ and $LazySet$ operations are performed, our data structure is able to reset any number of entries of any input variable in $O(k \cdot n^3)$ worst case time, i.e., $O(k \cdot n)$ amortized time per reset entry. We remark that in our application (dynamic transitive closure), k and h will be constant. We mention that we can also maintain dynamically witnesses for P: i.e., for any x, y s.t. $P[x,y] = 1$, we can exhibit in $O(k)$ worst-case time a sequence of indices $v_1, \dots, v_{k-1}$ and a term index a such that $X_1^a[x, v_1] \cdot X_2^a[v_1, v_2] \cdots X_k^a[v_{k-1}, y] = 1$.

Our Data Structure. The data structure we use for representing a polynomial is described below:

- We maintain each term $T = X_1 \cdot X_2 \cdots X_k$ of the polynomial as the sum of sub-terms

  $$T = \sum_{b=0}^{k} M_{b,b-1}^L \cdot M_{b,k-b-1}^R$$

  where $M_{b,j}^L$ and $M_{b,j}^R$ are auxiliary variables defined as follows:

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

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

- For each product of Boolean matrices $L \cdot R$ computed in the data structure, we maintain three matrices of witness sets initialized as follows. For any triple of indices x, y, z:

– $Left_{LR}[y,z] = \{x \mid L[x,y] = 1 \wedge R[y,z] = 1\}$

– $Prod_{LR}[x,z] = \{y \mid L[x,y] = 1 \wedge R[y,z] = 1\}$

– $Right_{LR}[x,y] = \{z \mid L[x,y] = 1 \wedge R[y,z] = 1\}$

Clearly, for any x, y, $(L \cdot R)[x,y] = 1$ if and only if $|Prod_{LR}[x,y]| > 0$.

- For each extended sum of Boolean matrices $M_1 + M_2 + \ldots + M_q$ computed in the data structure, we maintain a matrix of witness sets $Sum_{M_1 \cdots M_q}$ such that $Sum_{M_1 \cdots M_q}[x,y] = \{j \mid M_j[x,y] = 1\}$. Again, for any x, y, $(M_1 + M_2 + \ldots + M_q)[x,y] = 1$ if and only if $|Sum_{M_1 \cdots M_q}[x,y]| > 0$.

Implementation of Operations. Since maintaining matrices of sets $Prod_{LR}$ upon changes of L and R is the backbone of our data structure, we define operations $SetLRow$, $SetRCol$, $SetCrossLR$, $LazySetL$, $LazySetR$, $ResetL$ and $ResetR$ for doing so. Notice that $Left_{LR}$ and $Right_{LR}$ are only used to achieve the desired time bounds.

- $SetLRow(i, L, R, \Delta L)$: perform $L \leftarrow L + I_{\Delta L, i}$ and for each u, v s.t. $L[i,u] = 1$ and $R[u,v] = 1$ add i, u, and v to $Left_{LR}[u,v]$, $Prod_{LR}[i,v]$, and $Right_{LR}[i,u]$, respectively.

- $SetRCol(j, L, R, \Delta R)$: perform $R \leftarrow R + J_{\Delta R, j}$ and for each u, v s.t. $L[u,v] = 1$ and $R[v,j] = 1$ add u, v, and j to $Left_{LR}[v,j]$, $Prod_{LR}[u,j]$, and $Right_{LR}[u,v]$, respectively.

- $SetCrossLR(i, L, R, \Delta L, \Delta R)$: perform $L \leftarrow L + J_{\Delta L, i}$ and $R \leftarrow R + I_{\Delta R, i}$ and for each u, v s.t. $L[u,i] = 1$ and $R[i,v] = 1$ add u, i, and v to $Left_{LR}[i,v]$, $Prod_{LR}[u,v]$, and $Right_{LR}[u,i]$, respectively.

- $LazySetL(L, R, \Delta L)$: perform $L \leftarrow L + \Delta L$ without updating $Left_{LR}$, $Prod_{LR}$, and $Right_{LR}$. $LazySetR(L, R, \Delta R)$ works analogously on R.

- $ResetL(L, R, \Delta L)$: do $L \leftarrow L \ominus \Delta L$. Then for each x, y s.t. $\Delta L[x,y] = 1$ and for each $z \in Right_{LR}[x,y]$ remove x, y, z from $Left_{LR}[y,z]$, $Prod_{LR}[x,z]$, and $Right_{LR}[x,y]$, respectively. $ResetR(L, R, \Delta R)$ works analogously on R.

Theorem 1 *Any $SetLRow$, $SetRCol$, $SetCrossLR$, $LazySetL$, $LazySetR$, $ResetL$ or $ResetR$ operation on a product $L \cdot R$ can be supported in $O(n^2)$ amortized time. Moreover, any intermixed sequence of $ResetL$ and $ResetR$ operations requires $O(n^3)$ overall time in the worst case.*

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

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

be a potential function associated to $L \cdot R$, $0 \le \Phi \le n^3$. The rest of the proof follows from the fact that any set operation increases Φ by at most n^2 units and that any reset operation decreases Φ by at most n units for each reset entry. Note that neither $LazySetL$ nor $LazySetR$ affect Φ. To achieve the $O(n^3)$ bound for sequences of reset only updates, we require matrices to be compactly represented so that any matrix operation $A + B$ or $A \ominus B$ costs $O(s)$, where s is the number of 1's in B. □

We now define our update operations on polynomials in terms of the above operations.

- $SetRow(i, \Delta X, X_b^a)$: update the term $T = T_a$ as follows. For each $j \leftarrow 1$ to $b-2$ do $SetRCol$ on the product $M_{b-1,j}^L = X_{b-1-j} \cdot M_{b-1,j-1}^L$ to propagate any i-th column changes. For each $j \leftarrow 1$ to $k-b$ do $SetLRow$ on the product $M_{b-1,j}^R = M_{b-1,j-1}^R \cdot X_{b+j}$ to propagate any i-th row changes. Finally, do a $SetCrossLR$ operation on the product $M_{b-1,b-2}^L \cdot M_{b-1,k-b}^R$ and then update the sums that define T_a and P. Also perform $LazySetL$ or $LazySetR$ on each remaining product that involves X_b^a maintained in the data structure.

- $SetCol(i, \Delta X, X_b^a)$: analogous to $SetRow$. It updates $M_{b,j}^L$ and $M_{b,j}^R$ for any j.

- $LazySet(\Delta X, X_b^a)$: do $LazySetL$ or $LazySetR$ on each product that involves X_b^a maintained in the data structure.

- $Reset(\Delta X, X_b^a)$: do $ResetL$ or $ResetR$ on each product that involves X_b^a maintained in the data structure and propagate changes to any affected products and sums.

Analysis. To prove the correctness of operations, we first observe that for any reset entry $X[u,v]$ of a variable X, $Reset$ leaves the data structure as if $X[u,v]$ was never set to 1: this implies that we can concentrate on set operations only. Again, let C_P be the correct value of P that we would have by recomputing it from scratch after each update, and let M_P be the actual value that we maintain. Let us consider a $SetCol$ operation on the i-th column of X_b^a or a $SetRow$ operation on the i-th row of X_{b+1}^a. For any entry $C_P[x,y]$ that flips from 0 to 1 due to this operation there must be a sequence of indices $w_1, \ldots, u, i, v, \ldots, w_{k-1}$ such that either $\Delta X_b^a[u,i] = 1$ or $\Delta X_{b+1}^a[i,v] = 1$ and for which the product $X_1^a[x,w_1] \cdots X_b^a[u,i] \cdot X_{b+1}^a[i,v] \cdots X_k^a[w_{k-1},y]$

flips from 0 to 1. As this product is always considered and updated by $SetRow$ or $SetCol$ during the propagation steps, it follows that $M_P[x,y]$ flips from 0 to 1 as well. If no $LazySet$ is ever performed, then any 1 introduced in C_P also appears in M_P: as for any x, y, $C_P[x,y] = 0 \Rightarrow M_P[x,y] = 0$ then $M_P = C_P$. The running times of $SetRow$, $SetCol$, $LazySet$, and $Reset$ follow from Theorem 1: we remark that we can charge the cost of resetting any entry of any input or auxiliary variable maintained in the data structure to previous $Init$, $SetRow$, $SetCol$, or $LazySet$ operations.

The space usage of our data structure is $O(n^3)$ and allows it to report witnesses in constant time. If one is willing to give up the dynamic maintenance of witnesses, the space can be reduced to $O(n^2)$. Details are deferred to the full paper.

2.2 Maintaining Dynamic Matrices over Integers

We present a dynamic algorithm for maintaining the special class of polynomials over matrices consisting of a single $n \times n$ matrix of integers M when updates of submatrices and queries on single entries are performed. Updates are of the form $M \leftarrow M + J \cdot I$, where J and I are any column and row vectors, respectively. Notice that it is easy to support such updates in $O(n^2)$ worst-case time and queries with one matrix lookup. Our algorithm achieves $O(n^{\omega(1,\epsilon,1)-\epsilon})$ worst-case time for each update operation, for any ϵ, $0 \le \epsilon \le 1$, 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. Queries on individual entries of M are answered in $O(n^\epsilon)$ worst-case time.

We now sketch the main ideas behind the algorithm. We follow a 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.

Our Data Structure. Let $C_{n,n}$ be the $n \times n$ correct matrix, i.e., the matrix that we would have obtained by explicitly performing the update operations. Let $M_{n,n}$ be the actual matrix maintained by our lazy algorithm. Initially, and just after any reconstruction operation, $M_{n,n} = C_{n,n}$. We also maintain a $n \times n^\epsilon$ matrix J_{n,n^ϵ} and a $n^\epsilon \times n$ matrix $I_{n^\epsilon,n}$. We denote by $Update(J_k, I_k)$ the k-th update operation since the last reconstruction: this operation specifies the matrix $J_k \cdot I_k$ to be added to $C_{n,n}$. To do $Update(J_k, I_k)$, we copy the input vector J_k onto the k-th column of J_{n,n^ϵ} and the input vector I_k onto the k-th row of $I_{n^\epsilon,n}$. If $k = n^\epsilon$, we synchronize $M_{n,n}$ with $C_{n,n}$ by performing the reconstruction operation $M_{n,n} \leftarrow M_{n,n} + J_{n,n^\epsilon} \cdot I_{n^\epsilon,n}$.

Theorem 2 *Let t be the number of updates performed at any time since the last reconstruction and let $J_{n,t}$ and $I_{t,n}$ be the matrices obtained by considering only the first t columns and rows of J_{n,n^ϵ} and $I_{n^\epsilon,n}$, respectively. Then the following invariant is maintained: $C_{n,n} = M_{n,n} + J_{n,t} \cdot I_{t,n}$.*

Theorem 3 *Each update operation can be supported in $O(n^{\omega(1,\epsilon,1)-\epsilon})$ worst-case time and each query 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.*

Sketch of Proof: An amortized update bound follows trivially from amortizing the cost of the rectangular matrix multiplication $J_{n,n^\epsilon} \cdot I_{n^\epsilon,n}$ 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.

By Theorem 2, if t is the number of updates performed since the last reconstruction, a query on the value of $C_{n,n}[i,j]$ can be answered in $\Theta(t)$ time by simply computing

$$M_{n,n}[i,j] + \sum_{k=1}^{t} J_{n,t}[i,k] \cdot I_{t,n}[k,j].$$

As $t = O(n^\epsilon)$, this takes $O(n^\epsilon)$ worst-case time. □

3 A Fully-Dynamic Algorithm for Transitive Closure

In this section we show how to reduce dynamic transitive closure to the problem of dynamically reevaluating polynomials over Boolean matrices considered in Section 2.1. This allows us to design a fully dynamic algorithm for transitive closure with $O(n^2)$ update time and unit cost per query.

Let $G = (V, E)$ be a directed graph and let A_G be its adjacency matrix (i.e., $A_G[u,v] = 1$ if and only if $(u,v) \in E$). Note that the Kleene closure of A_G, defined as

$$A_G^* = \sum_{i=0}^{\infty} (A_G)^i,$$

is the adjacency matrix of the transitive closure of G, as $A_G^*[u,v] = 1$ if and only if there is a path from u to v in G. In the following, we focus on the problem of dynamically maintaining the Kleene closure A_G^* upon modifications of A_G over time: this is equivalent to maintaining the transitive closure of G.

The Problem. Let X be a $n \times n$ Boolean matrix. Without loss of generality we assume that n is a power of 2. We present a data structure for maintaining both X and its Kleene closure X^* under the following four operations:

- $Init^*(X)$: initializes the data structure and computes X^* from scratch.
- $Set^*(i, \Delta X, X^*)$: sets to 1 any entries in the i-th row and in the i-th column of X as specified by matrix ΔX (i.e., $X \leftarrow X + I_{\Delta X,i} + J_{\Delta X,i}$) and propagates the updates to X^*.
- $Reset^*(\Delta X, X^*)$: resets to 0 any entries of X as specified by matrix ΔX (i.e., $X \leftarrow X \ominus \Delta X$) and propagates the updates to X^*.
- $Lookup^*(x, y)$: returns the current value of $X^*[x, y]$.

Note that Set^* is allowed to modify only one row and one column of X, while $Reset^*$ can modify any entries of X. Our data structure supports each Set^* and $Reset^*$ operation in $O(n^2)$ amortized time, each $Lookup^*$ in unit worst-case time (as the closure is updated after each modification), and each $Init^*$ in $O(n^3)$ worst-case time. Using Set^*, $Reset^*$ and $Query^*$ we can implement insertion of a set of edges incident to the same vertex, deletion of an arbitrary subset of edges and transitive closure query in the original graph problem. We mention that we can also maintain dynamically witnesses for X^*: this allows us to report a path from vertex x to vertex y in the graph in $O(\ell)$ time, where ℓ is the path length.

Our Data Structure. We split X and X^* into four $\frac{n}{2} \times \frac{n}{2}$ sub-matrices A, B, C, D and E, F, G, H, respectively:

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

It is well known (see e.g., [1]) that E, F, G, H are related to A, B, C, D by either one of the following equivalent groups of relations:

$$\begin{array}{ll} E = (A + BD^*C)^* & E = A^* + A^*BHCA^* \\ F = EBD^* & F = A^*BH \\ G = D^*CE & G = HCA^* \\ H = D^* + D^*CEBD^* & H = (D + CA^*B)^* \end{array} \tag{1}$$

Our data structure for X^* is based on another set of relations:

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

It is easy to prove that our relations are equivalent to those in (1) since $E_1^2 = E_1$, $P^2 = P$, $H_2^2 = H_2$, and for any x, y, $A^*[x, y] = 1 \Rightarrow E_1[x, y] = 1$ and $D^*[x, y] = 1 \Rightarrow H_2[x, y] = 1$.

Our data structure maintains explicitly a decomposition of X^* based on equations in (2) where each involved polynomial is dynamically maintained as described in Section 2.1. Notice that P, E_1 and H_2 are themselves closures of polynomials, and thus can be recursively represented through dynamic polynomials on Boolean matrices of smaller size.

Decomposition (2) basically represents X^* as the sum of two Boolean matrices: the first, say X_1^*, is defined by submatrices E_1, F_1, G_1, H_1, and the second, say X_2^* by submatrices E_2, F_2, G_2, H_2. We maintain X_1^* and X_2^* in tandem: 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. Updates are performed so that neither X_1^* nor X_2^* encode complete information about X^*, but their sum does. On the other hand, $Reset^*$ operations update both X_1^* and X_2^* and leave the data structure as if the reset entries were never set to 1.

Implementation of Operations. If a matrix X gets modified, we denote by X' the matrix after the update.

- $Init^*(X)$: Recursively perform $Init$ operations over all polynomials in Decomposition (2). Intermediate results are properly propagated through the decomposition.
- $Set^*(i, \Delta X, X^*)$: Let V_1 and V_2 be two sets of indices such that $V_1 = \{1, \ldots, \frac{n}{2}\}$ and $V_2 = \{\frac{n}{2} + 1, \ldots, n\}$. We distinguish two cases:
  - $i \in V_1$: the update may affect a row and/or a column of A, a row of B and a column of C. Perform the following steps for updating X_1^*:

    A1. Update $Q = A + BP^2C$ by performing $SetRow$ / $SetCol$ operations for any variables A, B and C being changed. Let $\Delta Q = Q' \ominus Q$: note that all entries of ΔQ lying outside the i-th row and column are zero.

    A2. Recursively call $Set^*(i, \Delta Q, Q^*)$ to update E_1. Let $\Delta E_1 = E_1' \ominus E_1$: note that some entries of ΔE_1 lying outside the i-th row and column may be non-zero.

    A3. Update polynomials F_1, G_1 and H_1 by performing $SetRow$ / $SetCol$ operations for any variables E_1, B and C being changed. Note that we 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. We will show later that this is sufficient.

A4. Update the polynomial $R = D + CE_1^2B$ without updating H_2 and do $LazySet$ operations on F_2, G_2, E_2 and on any inner polynomial in the decomposition of R^* for any variables E_1, B, C and for any sub-variable of R being changed.

A5. Recompute polynomials E, F, G and H.

– $i \in V_2$: the update may affect a row and/or a column of D, a column of B and a row of C. Perform the following steps for updating X_2^*:

B1. Update R, H_2, G_2, F_2, E_2 similarly to Q, E_1, F_1, G_1, and H_1 in steps A1–A3.

B2. Recursively call $Set^*(i, \Delta D, D^*)$ to update P.

B3. Process Q, Q^*, E_1, F_1, G_1, and H_1 similarly to R, R^*, H_2, G_2, F_2, E_2 in step A4.

B4. Recompute polynomials E, F, G and H.

- $Reset^*(\Delta X, X^*)$:
  1. Recursively call $Reset^*(\Delta D, D^*)$ to update P, where ΔD is the sub-matrix of ΔX corresponding to D.
  2. Update $Q = A + BP^2C$ through $Reset$ operations for any variables A, B, P, and C being changed, and recursively call $Reset^*(\Delta Q, Q^*)$ for updating E_1.
  3. Update $R = D + CE_1^2B$ through $Reset$ operations for any variables D, C, E_1, and B being changed, and recursively call $Reset^*(\Delta R, R^*)$ for updating H_2.
  4. Update F_1, F_2, G_1, G_2, H_1, and E_2 by means of $Reset$ operations on them for any variables P, E_1, H_2, B and C being changed.
  5. Update polynomials E, F, G and H by means of $Reset$ operations for any variables E_1, E_2, F_1, F_2, G_1, G_2, H_1 and H_2 being changed.

Analysis. Now we discuss the correctness and the complexity of operations on our data structure.

Definition 1 *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.*

Definition 2 *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 *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 $\Delta X = J_{\Delta X,i} \cdot I_{\Delta X,i} + X \cdot I_{\Delta X,i} + J_{\Delta X,i} \cdot X$.*

Lemma 1 *If X is a Boolean matrix and ΔX is an i-centered update matrix, then $\Delta X^* = (X + \Delta X)^* - X^*$ is i-transitive and i-complete with respect to X^*.*

Lemma 2 *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 $X + \Delta X = (X + I_{\Delta X,i} + J_{\Delta X,i})^2$.*

Sketch of 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 2 and Definition 3 and from the facts: $I_{\Delta X,i}^2 = I_{\Delta X,i}$, $J_{\Delta X,i}^2 = J_{\Delta X,i}$ and $\Delta X = \Delta X + I_{\Delta X,i} + J_{\Delta X,i}$. □

Theorem 4 *Let C_{X^*} be the correct value of X^* that we would have by recomputing it from scratch after each update, and let $M_{X^*} = X_1^* + X_2^*$ be the actual value that we maintain. For any pair of indices x, y, $1 \le x, y \le n$, the following is true. If $C_{X^*}[x, y]$ flips from 0 to 1 due to a $Set^*(i, \Delta X, X^*)$ operation, then $M_{X^*}[x, y]$ flips from 0 to 1 as well. If new 1's are only added in X through Set^* operations, then $M_{X^*} = C_{X^*}$.*

Sketch of Proof: We prove that if $C_{X^*}[x, y]$ flips from 0 to 1 due to a $Set^*(i, \Delta X, X^*)$ operation, then either $X_1^*[x, y]$ flips from 0 to 1 (due to steps A1–A3 when $i \in V_1$), or $X_2^*[x, y]$ flips from 0 to 1 (due to step B1 when $i \in V_2$). The proof proceeds by induction on the size of matrices in Decomposition (2).

Without loss of generality, assume that the Set^* operation is performed with $i \in V_1$ (if $i \in V_2$ the proof is completely analogous). Sub-matrices A, B and C may undergo i-centered updates due to this operation and so their variation can be correctly propagated through $SetRow/SetCol$ operations to polynomial Q in step A1, and to polynomials F_1, G_1 and H_1 in step A3. As ΔQ is also i-centered due to step A1, 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 step A2. 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 step B3.

Notice 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 such operation was defined in Section 2.1. However, by Lemma 1 ΔE_1 is i-transitive and i-complete with respect to E_1. Since $E_1 = E_1^*$, by Lemma 2 performing both $SetRow(i, I_{\Delta E_1,i}, E_1)$ and $SetCol(i, J_{\Delta E_1,i}, E_1)$ operations on polynomials F_1, G_1 and H_1 in step A3 is sufficient to correctly reveal new 1's in them. 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 step B3. We have then proved that steps A1–A3 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 step A4 we limit ourselves to update explicitly R and to log any changes of E_1 by performing $LazySet$ operations on polynomials E_2, F_2, G_2 and on any smaller polynomials in the decomposition of R^*. This is sufficient to guarantee the correctness of subsequent Set^* operations for $i \in V_2$. □

Theorem 5 *Any Set^* or $Reset^*$ operation requires $O(n^2)$ amortized time.*

Sketch of Proof: Since all the polynomials in Decomposition (2) are of constant degree and involve a constant number of terms, the amortized cost of any $Query$, $SetRow$, $SetCol$, $LazySet$ and $Reset$ operation on them is quadratic in $\frac{n}{2}$. Let $T(n)$ be the time complexity of both Set^* and $Reset^*$ operations. Then

$$T(n) \leq 3T(\frac{n}{2}) + \frac{cn^2}{4}$$

for some suitably chosen constant $c > 0$. □

The space usage is dominated by the space usage of the data structure for polynomials described in Section 2.1. We can thus maintain dynamically witnesses for X^*, which allows us to report paths between vertices in optimal time, at the expense of using $O(n^3)$ space. As already pointed out in Section 2.1, if one is willing to give up the dynamic maintenance of witnesses, the space can be reduced to $O(n^2)$.

A Deletions-Only Algorithm. In the special case where $Reset^*$ only operations are allowed, we can directly represent X^* by either one group of relations in (1). Again, we use the data structure described in Section 2.1 for maintaining each polynomial and we properly propagate any changes through the decomposition by means of $Reset$ operations on polynomials and recursive calls to $Reset^*$. As any number of $Reset$ operations on polynomials of constant degree can be supported in $O(n^3)$ worst-case time, it is not difficult to prove that any sequence of $Reset^*$ operations on X^* also requires $O(n^3)$ time in the worst case, i.e., $O(n)$ amortized time per reset entry of X. The $Lookup$ time remains $O(1)$ as the closure is always explicitly updated.

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

We now show how the dynamic data structure described in Section 2.2 can be used for maintaining the transitive closure of a directed acyclic graph $G = (V, E)$ in subquadratic time per query/update operation. In [12] King and Sagert show 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 [12], 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.

Our Data Structure. We follow the approach of [12], but maintain a matrix M such that $M[x, y]$ stores the number of distinct paths in G between vertices x and y, for any $x, y \in V$, with the data structure of Section 2.2. Whenever a new edge (x, y) is inserted in G, we compute two vectors J and I such that $J[u] = M[u, x]$ and $I[u] = M[y, u]$ for any $u \in V$ and then we perform $Update(J, I)$ on M. In case of deletion of an edge (x, y), we compute I and J as above and then we perform $Update(-J, I)$ on M. It is easy to prove that, if G is acyclic, this method correctly maintains the number of distinct paths between any pair of vertices. We recall that, by Theorem 3, each entry of M can be queried in $O(n^\epsilon)$ worst-case time, and each $Update$ 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 $M[x, y]$. Balancing the two terms in the update bound yields that ϵ must satisfy the equation $\omega(1, \epsilon, 1) = 1 + 2\epsilon$. The current best bounds on $\omega(1, \epsilon, 1)$ [2, 6] imply that $\epsilon < 0.575$ [16]. Thus, the smallest update time is $O(n^{1.575})$, which gives a query time of $O(n^{0.575})$.

The algorithm is deterministic but uses wordsize $O(n)$. To reduce wordsize to $O(\log n)$ while maintaining the same subquadratic bounds ($O(n^{1.575})$ per update and $O(n^{0.575})$ per query) we use the technique of [12]: 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 $\omega = \omega(1,1,1) < 2.736$ is the current best exponent for matrix multiplication [2].

Acknowledgments

We are indebted to Valerie King, Garry Sagert and Mikkel Thorup for many useful discussions.

References

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

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

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

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

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

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

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

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

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

[10] S. Khanna, R. Motwani, and R. H. Wilson. On certificates and lookahead on dynamic graph problems. In *Proc. 7th ACM-SIAM Symp. Discrete Algorithms*, pages 222–231, 1996.

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

[12] V. King and G. Sagert. A fully dynamic algorithm for maintaining the transitive closure. In *Proc. 31st ACM Symposium on Theory of Computing (STOC'99)*, pages 492–498, 1999.

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

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

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

[16] U. Zwick. 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)*, pages 310–319, Los Alamitos, CA, November 8–11 1998.