A New Combinatorial Approach For Sparse Graph Problems

Guy E. Blelloch, Virginia Vassilevska, and Ryan Williams

Computer Science Department, Carnegie Mellon University, Pittsburgh, PA
{guyb,virgi,ryanw}@cs.cmu.edu

Abstract. We give a new combinatorial data structure for representing arbitrary Boolean matrices. After a short preprocessing phase, the data structure can perform fast vector multiplications with a given matrix, where the runtime depends on the sparsity of the input vector. The data structure can also return *minimum witnesses* for the matrix-vector product. Our approach is simple and implementable: the data structure works by precomputing small problems and recombining them in a novel way. It can be easily plugged into existing algorithms, achieving an asymptotic speedup over previous results. As a consequence, we achieve new running time bounds for computing the transitive closure of a graph, all pairs shortest paths on unweighted undirected graphs, and finding a maximum node-weighted triangle. Furthermore, any asymptotic improvement on our algorithms would imply a $o(n^3/\log^2 n)$ combinatorial algorithm for Boolean matrix multiplication, a longstanding open problem in the area. We also use the data structure to give the first asymptotic improvement over $O(mn)$ for all pairs least common ancestors on directed acyclic graphs.

1 Introduction

A large collection of graph problems in the literature admit essentially two solutions: an *algebraic* approach and a *combinatorial* approach. Algebraic algorithms rely on the theoretical efficacy of fast matrix multiplication over a ring, and reduce the problem to a small number of matrix multiplications. These algorithms achieve unbelievably good theoretical guarantees, but can be impractical to implement given the large overhead of fast matrix multiplication. Combinatorial algorithms rely on the efficient preprocessing of small subproblems. Their theoretical guarantees are typically worse, but they usually lead to more practical improvements. Combinatorial approaches are also interesting in that they have the capability to tackle problems that seem to be currently out of the reach of fast matrix multiplication. For example, many sparse graph problems are not known be solvable quickly with fast matrix multiplication, but a combinatorial approach can give asymptotic improvements. (Examples are below.)

In this paper, we present a new combinatorial method for preprocessing an $n \times n$ dense Boolean matrix A in $O(n^{2+\varepsilon})$ time (for any $\varepsilon > 0$) so that sparse vector multiplications with A can be done faster, while matrix updates are not too expensive to handle. In particular,

– for a vector v with t nonzeros, $A \cdot v$ can be computed in $O(\frac{n}{\log n}(t/\kappa + n/\ell))$ time, where ℓ and κ are parameters satisfying $\binom{\ell}{\kappa} \leq n^{\varepsilon}$, and
– row and/or column updates to the matrix can be performed in $O(n^{1+\varepsilon})$ time.

The matrix-vector multiplication can actually return a vector w of *minimum witnesses*; that is, $w[i] = k$ iff k is the minimum index satisfying $A[i,k] \cdot v[k] \neq 0$. The data structure is simple, does not use devious "word tricks" or hide any large constants, and can be implemented on a pointer machine.[1] We apply our data structure to four fundamental graph problems: transitive closure, all pairs shortest paths, minimum weight triangle, and all pairs least common ancestors. All four are known to be solvable in $n^{3-\delta}$ time for some $\delta > 0$, but the algorithms are algebraic and do not exploit the potential sparsity of graphs. With the right settings of the parameters ℓ and κ, our data structure can be applied to all the above four problems, giving the best runtime bounds for *sparse problems* to date.

Transitive Closure: We have a directed graph on n nodes and m edges, and wish to find all pairs of nodes (u, v) whether there is a path from u to v in the graph. Transitive closure has myriad applications and a long history of ideas. The best known theoretical algorithms use $O(M(n))$ time [10, 13] where $M(n)$ is the complexity of $n \times n$ Boolean matrix product, and $O(mn/\log n + n^2)$ time [5, 3]. Algebraic matrix multiplication implies an $O(n^{\omega})$ algorithm, where $\omega < 2.376$ [6], and combinatorial matrix multiplication gives an $O(n^3/\log^2 n)$ runtime [2, 14, 18]. Our data structure can be used to implement transitive closure in $O(mn(\log(\frac{n^2}{m})/\log^2 n)+n^2)$ time. This constitutes the first combinatorial improvement on the bounds of $O(n^3/\log^2 n)$ and $O(mn/\log n + n^2)$ that follow from Four Russians preprocessing, and it establishes the best known running time for general sparse graphs.

All Pairs Shortest Paths (APSP): We want to construct a representation of a given graph, so that for any pair of nodes, a shortest path between the pair can be efficiently obtained from the representation. The work on APSP is deep and vast; here we focus on the simplest case where the graph is *undirected* and *unweighted*. For this case, Galil and Margalit [11] and Seidel [15] gave $O(n^{\omega})$ time algorithms. These algorithms do not improve on the simple $O(mn + n^2)$ algorithm (using BFS) when $m = o(n^{\omega-1})$. The first improvement over $O(mn)$ was given by Feder and Motwani [9] who gave an $O(mn \log(\frac{n^2}{m})/\log n)$ time algorithm. Recently, Chan presented new algorithms that take $\hat{O}(mn/\log n)$ time.[2] We show that APSP on undirected unweighted graphs can be computed in $O(mn \log(\frac{n^2}{m})/\log^2 n)$ time. Our algorithm modifies Chan's $O(mn/\log n + n^2 \log n)$ time solution, implementing its most time-consuming procedure efficiently using our data structure.

[1] When implemented on the w-word RAM, the multiplication operation runs in $O(\frac{n}{w}(t/k+n/\ell))$. In fact, all of the combinatorial algorithms mentioned in this paper can be implemented on a w-word RAM in $O(T(n)(\log n)/w)$ time, where $T(n)$ is the runtime stated.

[2] The $\hat{O}$ notation suppresses poly$(\log\log n)$ factors.

All Pairs Weighted Triangles: Here we have a directed graph with an arbitrary weight function w on the nodes. We wish to compute, for all pairs of nodes v_1 and v_2, a node v_3 that such that (v_1, v_3, v_2, v_1) is a cycle and $\sum_i w(v_i)$ is minimized or maximized. The problem has applications in data mining and pattern matching. Recent research has uncovered interesting algebraic algorithms for this problem [17], the current best being $O(n^{2.575})$, but again it is somewhat impractical, relying on fast rectangular matrix multiplication. (We note that the problem of finding a *single* minimum or maximum weight triangle has been shown to be solvable in $n^{\omega+o(1)}$ time [8].) The proof of the result in [17] also implies an $O(mn/\log n)$ algorithm. Our data structure lets us solve the problem in $O(mn\log(\frac{n^2}{m})/\log^2 n)$ time.

Least Common Ancestors on DAGs: Given a directed acyclic graph G on n nodes and m edges, fix a topological order on the nodes. For all pairs of nodes s and t we want to compute the highest node in topological order that still has a path to both s and t. Such a node is called a least common ancestor (LCA) of s and t. The all pairs LCA problem is to determine an LCA for every pair of vertices in a DAG. In terms of n, the best algebraic algorithm for finding all pairs LCAs uses the minimum witness product and runs in $O(n^{2.575})$ [12, 7]. Czumaj, Kowaluk, and Lingas [12, 7] gave an algorithm for finding all pairs LCAs in a *sparse* DAG in $O(mn)$ time. We improve this runtime to $O(mn\log(\frac{n^2}{m})/\log n)$.

1.1 On the optimality of our algorithms

We have claimed that all the above problems (with the exception of the last one) can be solved in $O(mn\log(n^2/m)/\log^2 n)$ time. How does this new runtime expression compare to previous work? It is easier to see the impact when we let $m = n^2/s$ for a parameter s. Then

$$\frac{mn\log(n^2/m)}{\log^2 n} = \frac{n^3}{\log^2 n} \cdot \frac{\log s}{s} \tag{1}$$

Therefore, our algorithms yield asymptotic improvements on "medium density" graphs, where the number of edges is in the range $n^{2-o(1)} \leq m \leq o(n^2)$.

At first glance, such an improvement may appear small. We stress that our algorithms have essentially reached the best that one can do for these problems, without resorting to fast matrix multiplication or achieving a major breakthrough in combinatorial matrix algorithms. As all of the above problems can be used to simulate Boolean matrix multiplication, (1) implies that an $O\left(mn\frac{\log(n^2/m)}{f(n)\log^2 n}\right)$ algorithm for any of the above problems and any unbounded function $f(n)$ would entail an asymptotic improvement on combinatorial Boolean matrix multiplication, a longstanding open problem. Note that an $O\left(\frac{mn}{f(n)\log n}\right)$ combinatorial algorithm does not imply such a breakthrough: let $m = n^2/s$ and $f(n) = o(\log n)$ and observe $mn/(f(n)\log n) = n^3/(sf(n)\log n)$; such an algorithm is still slow on sufficiently dense matrices.

1.2 Related Work

Closely related to our work is Feder and Motwani's ingenious method for compressing sparse graphs, introduced in STOC'91. In the language of Boolean matrices, their method runs in $\tilde{O}(mn^{1-\varepsilon})$ time and decomposes an $n \times n$ matrix A with m nonzeros into an expression of the form $A = (A_1 * A_2) \vee A_3$, where $*$ and $\vee$ are matrix product and pointwise-OR, A_1 is $n \times mn^{\varepsilon}$, A_2 is $mn^{\varepsilon} \times n$, A_1 and A_2 have $O(\frac{m \log(n^2/m)}{\log n})$ nonzeros, and A_3 has $O(n^{1+\varepsilon})$ nonzeros. Such a decomposition has many applications to speeding up algorithms.

While a $(\log n)/\log(n^2/m)$ factor of savings crops up in both approaches, our approach is markedly different from graph compression; in fact it seems orthogonal. From the above viewpoint, Feder-Motwani's graph compression technique can be seen as a method for preprocessing a sparse Boolean matrix so that multiplications of it with arbitrary vectors can be done in $O\left(\frac{m \log(n^2/m)}{\log n}\right)$ time. In contrast, our method preprocesses an *arbitrary matrix* so that its products with *sparse vectors* are faster, and updates to the matrix are not prohibitive. This sort of preprocessing leads to a Feder-Motwani style improvement for a new set of problems. It is especially applicable to problems in which an initially sparse graph is augmented and becomes dense over time, such as transitive closure.

Our data structure is related to one given previously by the third author [18], who showed how to preprocess a matrix over a constant-sized semiring in $O(n^{2+\varepsilon})$ time so that subsequent vector multiplications can be performed in $O(n^2/\log^2 n)$ time. Ours is a significant improvement over this data structure in two ways: the runtime of multiplication now varies with the sparsity of the vector (and is never worse than $O(n^2/\log^2 n)$), and our data structure also returns minimum witnesses for the multiplication. Both of these are non-trivial augmentations that lead to new applications.

2 Preliminaries and Notation

Define $H(x) = x \log_2(1/x) + (1-x)\log_2(1/(1-x))$. H is often called the binary entropy function. All logarithms are assumed to be base two. Throughout this paper, when we consider a graph $G = (V, E)$ we let $m = |E|$ and $n = |V|$. G can be directed or undirected; when unspecified we assume it is directed. We define $\delta_G(s, v)$ to be the distance in G from s to v. We assume G is always weakly connected, so that $m \geq n-1$. We use the terms *vertex* and *node* interchangeably. For an integer ℓ, let $[\ell]$ refer to $\{1, \ldots, \ell\}$.

We denote the typical Boolean product of two matrices A and B by $A \cdot B$. For two Boolean vectors u and v, let $u \wedge v$ and $u \vee v$ denote the componentwise AND and OR of u and v respectively; let $\neg v$ be the componentwise NOT on v.

A *minimum witness* vector for the product of a Boolean matrix A with a Boolean vector v is a vector w such that $w[i] = 0$ if $\vee_j (A[i][j] \cdot v[j]) = 0$, and if $\vee_j (A[i][j] \cdot v[j]) = 1$, $w[i]$ is the minimum index j such that $A[i][j] \cdot v[j] = 1$.

3 Combinatorial Matrix Products With Sparse Vectors

We begin with a data structure which after preprocessing stores a matrix while allowing efficient matrix-vector product queries and column updates to the matrix. On a matrix-vector product query, the data structure not only returns the resulting product matrix, but also a minimum witness vector for the product.

Theorem 1. *Let B be a $d \times n$ Boolean matrix. Let $\kappa \geq 1$ and $\ell > \kappa$ be integer parameters. Then one can create a data structure with $O(\frac{dn\kappa}{\ell} \cdot \sum_{b=1}^{\kappa} \binom{\ell}{b})$ preprocessing time so that the following operations are supported:*

- *given any $n \times 1$ binary vector r, output $B \cdot r$ and a $d \times 1$ vector w of* minimum witnesses *for $B \cdot r$ in $O(d \log n + \frac{d}{\log n}\left(\frac{n}{\ell} + \frac{m_r}{\kappa}\right))$ time, where m_r is the number of nonzeros in r;*
- *replace any column of B by a new column in $O(d\kappa \sum_{b=1}^{\kappa} \binom{\ell}{b})$ time.*

The result can be made to work on a pointer machine as in [18]. Observe that the naïve algorithm for $B \cdot r$ that simulates $\Theta(\log n)$ word operations on a pointer machine would take $O(\frac{dm_r}{\log n})$ time, so the above runtime gives a factor of κ savings provided that ℓ is sufficiently large. The result can also be generalized to handle products over any fixed size semiring, similar to [18].

Proof of Theorem 1. Let $0 < \varepsilon < 1$ be a sufficiently small constant in the following. Set $d' = \lceil \frac{d}{\lceil \varepsilon \log n \rceil} \rceil$ and $n' = \lceil n/\ell \rceil$. To preprocess B, we divide it into block submatrices of at most $\lceil \varepsilon \log n \rceil$ rows and ℓ columns each, writing B_{ji} to denote the j, i submatrix, so

$$B = \begin{bmatrix} B_{11} & \dots & B_{1n'} \\ \vdots & \ddots & \vdots \\ B_{d'1} & \dots & B_{d'n'} \end{bmatrix}.$$

Note that entries from the kth column of B are contained in the submatrices $B_{j\lceil k/\ell \rceil}$ for $j = 1, \dots, d'$, and the entries from kth row are in $B_{\lceil k/\ell \rceil i}$ for $i = 1, \dots, n'$. For simplicity, from now on we omit the ceilings around $\varepsilon \log n$.

For every $j = 1, \dots, d'$, $i = 1, \dots, n'$ and every ℓ length vector v with at most κ nonzeros, precompute the product $B_{ji} \cdot v$, and a minimum witness vector w which is defined as: for all $k = 1, \dots \varepsilon \log n$,

$$w[k] = \begin{cases} 0 & \text{if } (B_{ji} \cdot v)[k] = 0 \\ (i-1)\ell + w' & \text{if } B_{ji}[k][w'] \cdot v[w'] = 1 \text{ and } \forall w'' < w', B_{ji}[k][w''] \cdot v[w''] = 0. \end{cases}$$

Store the results in a look-up table. Intuitively, w stores the minimum witnesses for $B_{ji} \cdot v$ with their indices in $[n]$, that is, as if B_{ji} is construed as an $n \times n$ matrix which is zero everywhere except in the (j, i) subblock which is equal to B_{ji}, and v is construed as a length n vector which is nonzero only in its ith block which equals v. This product and witness computation on B_{ji} and v takes $O(\kappa \varepsilon \log n)$ time. There are at most $\sum_{b=1}^{\kappa} \binom{\ell}{b}$ such vectors v, and hence this precomputation takes $O(\kappa \log n \sum_{b=1}^{\kappa} \binom{\ell}{b})$ time for fixed B_{ji}. Over all j, i the preprocessing takes $O(\frac{dn}{\ell \log n} \cdot \kappa \log n \sum_{b=1}^{\kappa} \binom{\ell}{b}) = O(\frac{dn\kappa}{\ell} \sum_{b=1}^{\kappa} \binom{\ell}{b})$ time.

Suppose we want to modify column k of B in this representation. This requires recomputing $B_{j\lceil k/\ell \rceil} \cdot v$ and the witness vector for this product, for all $j = 1, \ldots, n'$ and for all length ℓ vectors v with at most κ nonzeros. Thus a column update takes only $O(d\kappa \sum_{b=1}^{\kappa} \binom{\ell}{b})$ time.

Now we describe how to compute $B \cdot r$ and its minimum witnesses. Let m_r be the number of nonzeros in r. We write $r = [r_1 \ \cdots \ r_{n'}]^T$ where each r_i is a vector of length ℓ. Let m_{ri} be the number of nonzeros in r_i.

For each $i = 1, \ldots, n'$, we decompose r_i into a union of at most $\lceil m_{ri}/\kappa \rceil$ disjoint vectors r_{ip} of length ℓ and at most κ nonzeros, so that r_{i1} contains the first κ nonzeros of r_i, r_{i2} the next κ, and so on, and $r_i = \vee_p r_{ip}$. Then, for each $p = 1, \ldots, \lceil m_{ri}/\kappa \rceil$, r_{ip} has nonzeros with larger indices than all $r_{ip'}$ with $p' < p$, *i.e.* if $r_{ip}[q] = 1$ for some q, then for all $p' < p$ and $q' \geq q$, $r_{ip'}[q'] = 0$.

For $j = 1, \ldots, d'$, let $B^j = [B_{j1} \ldots B_{jn'}]$. We shall compute $v_j = B^j \cdot r$ separately for each j and then combine the results as $v = [v_1, \ldots, v_{d'}]^T$. Fix $j \in [d']$. Initially, set v_j and w_j to be the all-zeros vector of length $\varepsilon \log n$. The vector w_j shall contain minimum witnesses for $B^j \cdot r$.

For each $i = 1, \ldots, n'$ in increasing order, consider r_i. In increasing order for each $p = 1, \ldots, \lceil m_{ri}/\kappa \rceil$, process r_{ip} as follows. Look up $v = B_{ji} \cdot r_{ip}$ and its witness vector w. Compute $y = v \wedge \neg v_j$ and then set $v_j = v \vee v_j$. This takes $O(1)$ time. Vector y has nonzeros in exactly those coordinates h for which the minimum witness of $(B^j \cdot r)[h]$ is a minimum witness of $(B_{ji} \cdot r_{ip})[h]$; since over all i' and p' the nonzeros of $r_{i'p'}$ partition the nonzeros of r, this minimum witness is not a minimum witness of any $(B_{ji'} \cdot r_{i'p'})[h]$ with $i' \neq i$ or $p' \neq p$. In this situation we say that the minimum witness is in r_{ip}. In particular, if $y \neq 0$, r_{ip} contains some minimum witnesses for $B^j \cdot r$ and the witness vector w_j for $B^j \cdot r$ needs to be updated. Then, for each $q = 1, \ldots, \varepsilon \log n$, if $y[q] = 1$, set $w_j[q] = w[q]$. This ensures that after all i, p iterations, $v = \bigvee_{i,p} (B_{ji} \cdot r_{ip}) = B^j \cdot r$ and w_j is the product's minimum witness vector. Finally, we output $B \cdot r = [v_1 \cdots v_{d'}]^T$ and $w = [w_1 \ldots w_{d'}]$. Updating w_j can happen at most $\varepsilon \log n$ times, because each $w_j[q]$ is set at most once for $q = 1, \ldots, \varepsilon \log n$. Each individual update takes $O(\log n)$ time. Hence, for each j, the updates to w_j take $O(\log^2 n)$ time, and over all j, the minimum witness computation takes $O(d \log n)$. Computing $v_j = B^j \cdot r$ for a fixed j takes $O(\sum_{i=1}^{n'} \lceil m_{ri}/\kappa \rceil) \leq O(\sum_{i=1}^{n'} (1 + m_{ri}/\kappa))$ time. Over all $j = 1, \ldots, d/(\varepsilon \log n)$, the computation of $B \cdot r$ takes asymptotically

$$\frac{d}{\log n} \sum_{i=1}^{n/\ell} \left(1 + \frac{m_{ri}}{\kappa}\right) \leq \frac{d}{\log n} \left(\frac{n}{\ell} + \frac{m_r}{\kappa}\right).$$

In total, the running time is $O(d \log n + \frac{d}{\log n}(\frac{n}{\ell} + \frac{m_r}{\kappa}))$. □

Let us demonstrate what the data structure performance looks like with a particular setting of the parameters ℓ and k. From Jensen's inequality we have:

Fact 1 *$H(a/b) \leq 2a/b \log(b/a)$, for $b \geq 2a$.*

Corollary 1. *Given a parameter m and $0 < \varepsilon < 1$, any $d \times n$ Boolean matrix A can be preprocessed in $O(dn^{1+\varepsilon})$ time, so that every subsequent computation of AB can be determined in $O(de \log n + \frac{md \log(ne/m)}{\log^2 n})$ time, for any $n \times e$ Boolean matrix B with at most m nonzeros.*

Proof. When $m \geq \frac{en}{2}$, the runtime in the theorem is $\Omega(end/\log^2 n)$, and can be achieved via Four Russians processing. If $m \leq en^{1-\varepsilon}$ then $\varepsilon \log n \leq \log \frac{en}{m}$ and running a standard sparse matrix multiplication with $\log n$ bit operations in $O(1)$ time achieves $O(md/\log n) \leq O(md\frac{\log(en/m)}{\log^2 n})$ time. If $en^{1-\varepsilon} < m < \frac{en}{2}$, apply Theorem 1 with $\ell = \varepsilon\frac{en}{m} \cdot (\log n)/\log(\frac{en}{m})$, and $\kappa = \varepsilon(\log n)/\log(\frac{en}{m}) \geq 1$. Then by Fact 1 and $m < \frac{en}{2}$, we can show that $\binom{\ell}{\kappa} \leq n^{\varepsilon}$. Hence the preprocessing step takes $O(dn^{1+\varepsilon}m/(ne)) = O(dn^{1+\varepsilon})$ time. Matrix-vector multiplication with a vector of m_i nonzeros takes $O\left(d\log n + \frac{d}{\log n}\left(\frac{m\log(en/m)}{e\log n} + \frac{m_i\log(en/m)}{\log n}\right)\right)$ time. If m_i is the number of nonzeros in the ith column of B, as $\sum_i m_i = m$, the full matrix product $A \cdot B$ can be done in $O\left(de\log n + \frac{md\log(en/m)}{\log^2 n}\right)$ time. □

It follows that Boolean matrix multiplication for $n \times n$ matrices can be done in $O(n^{2+\varepsilon} + mn \log \frac{n^2}{m}/\log^2 n)$ time, provided that at least one of the matrices has only m nonzeros. We note that such a result could also be obtained by construing the sparse matrix as a bipartite graph, applying Feder-Motwani's graph compression to represent the sparse matrix as a product of two sparser matrices (plus a matrix with $n^{2-\delta}$ nonzeros), then using an $O(mn/\log n)$ Boolean matrix multiplication algorithm on the resulting matrices. However, given the complexity of this procedure (especially the graph compression, which is involved) we believe that our method is more practical. Theorem 1 also leads to two other new results almost immediately.

Minimum Witnesses for Matrix Multiplication. The *minimum witness* product of two $n \times n$ Boolean matrices A and B is the $n \times n$ matrix C defined as $C[i][j] = \min_{k=1}^{n}\{k \mid A[i][k] \cdot B[k][j] = 1\}$, for every $i, j \in [n]$. This product has applications to several graph algorithms. It has been used to compute all pairs least common ancestors in DAGs [12, 7], to find minimum weight triangles in node-weighted graphs [17], and to compute all pairs bottleneck paths in a node weighted graph [16]. The best known algorithm for the minimum witness product is by Czumaj, Kowaluk and Lingas [12, 7] and runs in $O(n^{2.575})$ time. However, when one of the matrices has at most $m = o(n^{1.575})$ nonzeros, nothing better than $O(mn)$ was known (in terms of m) for combinatorially computing the minimum witness product. As an immediate corollary of Theorem 1, we obtain the first asymptotic improvement for sparse matrices: an algorithm with $O(mn\log(n^2/m)/\log^2 n)$ running time.

Corollary 2. *Given $n \times n$ Boolean matrices A and B, where B has at most m nonzero entries, all pairs minimum witnesses of $A \cdot B$ can be computed in $O(n^2 + mn\log(n^2/m)/\log^2 n)$ time.*

Minimum Weighted Triangles. The problem of computing for all pairs of nodes in a node-weighted graph a triangle (if any) of minimum weight passing through both nodes can be reduced to finding minimum witnesses as follows ([17]). Sort the nodes in order of their weight in $O(n \log n)$ time. Create the adjacency matrix A of the graph so that the rows and columns are in the sorted order of the vertices. Compute the minimum witness product C of A. Then, for every edge $(i, j) \in G$, the minimum weight triangle passing through i and j is $(i, j, C[i][j])$. From this reduction and Corollary 2 we obtain the following.

Corollary 3. *Given a graph G with m edges and n nodes with arbitrary node weights, there is an algorithm which finds for all pairs of vertices i, j, a triangle of minimum weight sum going through i, j in $O(n^2 + mn \log(\frac{n^2}{m})/\log^2 n)$ time.*

4 Transitive Closure

The transitive closure matrix of an n node graph G is the $n \times n$ Boolean matrix A for which $A[i][j] = 1$ if and only if node i is reachable from node j in G. In terms of n, the complexity of computing the transitive closure of an n node graph is equivalent to that of multiplying two Boolean $n \times n$ matrices [1], thus the best known algorithm in terms of n is $O(n^\omega)$. However, when the sparsity of G is taken into account, it is unclear how to use an algorithm for sparse matrix multiplication to solve transitive closure in the same runtime. While Feder and Motwani's [9] graph compression implies an $O(mn \log(\frac{n^2}{m})/\log^2 n)$ algorithm for sparse matrix product, this result gives little insight on how to compute the transitive closure of a sparse graph—since the number of edges in the transitive closure is independent of m in general, maintaining a graph compression will not suffice. In contrast, the data structure of Theorem 1 is perfectly suited for use with a dynamic programming algorithm for transitive closure.

Theorem 2. *Transitive closure can be computed in $O(n^2 + mn \log(\frac{n^2}{m})/\log^2 n)$ time on graphs with n nodes and m edges.*

Proof. We first compute the strongly connected components of G in $O(m + n)$ time. We then contract them in linear time to obtain a DAG G'. Clearly, if we have the transitive closure matrix of G', we can recover the transitive closure matrix of G with an extra $O(n^2)$ additive overhead: for every edge (u, v) in the transitive closure graph of G', go through all pairs of vertices (i, j) such that i is in u and j is in v and add 1 to the transitive closure matrix of G. Hence it suffices for us to compute the transitive closure of a DAG G'.

First, we topologically sort G' in $O(m + n)$ time. Our transitive closure algorithm is based on a dynamic programming formulation of the problem given by Cheriyan and Mehlhorn [5], later also mentioned by Chan [3]. The algorithm proceeds in n iterations; after the kth iteration, we have computed the transitive closure of the last k nodes in the topological order. At every point, the current transitive closure is maintained in a Boolean matrix R, such that $R[u][v] = 1$ iff u is reachable from v. Let $R[\cdot][v]$ denote column v of R. Let p be the $(k+1)$st node in reverse topological order. We wish to compute $R[\cdot][p]$, given all the vectors $R[\cdot][u]$ for nodes u after p in the topological order. To do this, we compute the componentwise OR of all vectors $R[\cdot][u]$ for the neighbors u of p.

Suppose R is a matrix containing columns $R[\cdot][u]$ for all u processed so far, and zeros otherwise. Let v_p be the *outgoing neighborhood* vector of p: $v_p[u] = 1$ iff there is an arc from p to u. Construing v_p as a column vector, we want to compute $R \cdot v_p$. Since all outgoing neighbors of p are after p in the topological order, and we process nodes in reverse order, correctness follows.

We now show how to implement the algorithm using the data structure of Theorem 1. Let R be an $n \times n$ matrix such that at the end of the algorithm R is the transitive closure of G'. We begin with R set to the all zero matrix. Let $\kappa \geq 1$, and $\ell > \kappa$ be parameters to be set later. After $O((n^2\kappa/\ell)\sum_{b=1}^{\kappa}\binom{\ell}{b})$ preprocessing time, the data structure for R from Theorem 1 is complete.

Consider a fixed iteration k. Let p be the kth node in reverse topological order. As before, v_p is the neighborhood column vector of p, so that v_p has $outdeg(p)$ nonzero entries. Let $0 < \varepsilon < 1$ be a constant. We use the data structure to compute $r_p = R \cdot v_p$ in $O(n^{1+\varepsilon} + n^2/(\ell \log n) + outdeg(p) \cdot n/(\kappa \log n))$ time. Then we set $r_p[p] = 1$, and $R[\cdot][p] := r_p$, by performing a column update on R in $O(n\kappa \sum_{b=1}^{\kappa}\binom{\ell}{b})$ time. This completes iteration k. Since there are n iterations and since $\sum_p outdeg(p) = m$, the overall running time is asymptotic to

$$\frac{n^2\kappa}{\ell}\sum_{b=1}^{\kappa}\binom{\ell}{b} + n^{2+\varepsilon} + \frac{n^3}{\ell \log n} + \frac{mn}{\kappa \log n} + n^2\kappa\sum_{b=1}^{\kappa}\binom{\ell}{b}.$$

If for some $\varepsilon' > 0$, $m \leq n^{2-\varepsilon'}$, then we can ignore the $O(n^{2+\varepsilon})$ preprocessing step and execute the original algorithm, in $O(mn/\log n) \leq O(mn \log \frac{n^2}{m}/\log^2 n)$ time. Otherwise, we set ℓ and κ to minimize the running time. Similar to Corollary 1, we set $\frac{n^3}{\ell \log n} = \frac{mn}{\kappa \log n}$, (implying $\ell = \kappa n^2/m$), and $n^{\varepsilon} = \sum_{b=1}^{\ell m/n^2}\binom{\ell}{b}$. For $m \leq n^2/2$, $\sum_{b=1}^{m\ell/n^2}\binom{\ell}{b} = O(2^{\ell H(m/n^2)}) = O(2^{\ell \frac{m}{n^2}\log\frac{n^2}{m}})$. Hence we want $\ell\frac{m}{n^2}\log\frac{n^2}{m} = \log n^{\varepsilon}$, and $\ell = \varepsilon' \frac{n^2 \log n}{m \log(n^2/m)}$, $\kappa = \varepsilon' \frac{\log n}{\log(n^2/m)}$ for $\varepsilon' < \varepsilon$ suffices. Since for all $\varepsilon' > 0$, $m \geq n^{2-\varepsilon'}$, we can pick any $\varepsilon' < \varepsilon$ and we will have $\kappa \geq 1$. For sufficiently small ε' the runtime is $\Omega(n^{2+\varepsilon})$, and the final runtime is $O(n^2 + mn\log(n^2/m)/\log^2 n)$. □

5 APSP on Unweighted Undirected Graphs

Our data structure can also be applied to solve all pairs shortest paths (APSP) on unweighted undirected graphs, yielding the fastest algorithm for sparse graphs to date. Chan [4] gave two algorithms for this problem, which constituted the first major improvement over the $O(mn\frac{\log(n^2/m)}{\log n} + n^2)$ obtained by Feder and Motwani's graph compression [9]. The first algorithm is deterministic running in $O(n^2 \log n + mn/\log n)$ time, and the second one is Las Vegas running in $O(n^2 \log^2 \log n/\log n + mn/\log n)$ time on the RAM. To prove the following Theorem 3, we implement Chan's first algorithm, along with some modifications.

Theorem 3. *The All Pairs Shortest Paths problem in unweighted undirected graphs can be solved in* $O(n^{2+\varepsilon} + mn\log(n^2/m)/\log^2 n)$ *time, for any* $\varepsilon > 0$.

By running our algorithm when $m = \Omega(n^{1+\varepsilon}\log^2 n)$ for some $\varepsilon > 0$ and Chan's second algorithm otherwise, we obtain the following result.

Theorem 4. *On the probabilistic RAM, the APSP problem on unweighted undirected graphs can be solved in* $O(n^2\log^2\log n/\log n + mn\log(n^2/m)/\log^2 n)$ *time, with high probability.*

To be able to prove Theorem 3, let us focus on a particular part of Chan's first algorithm that produces the bottleneck in its runtime. The algorithm contains a subprocedure $P(d)$, parameterized by an integer $d < n$. The input to $P(d)$ is graph $G = (V, E)$, a collection of n/d vertices $\{s_1, \ldots, s_{n/d}\}$, and a collection of n/d *disjoint* vertex subsets $\{S^1, \ldots, S^{n/d}\}$, such that $\forall\ i \in [n/d]$, every $s \in S^i$ has distance at most 2 from s_i. $P(d)$ outputs the $|\cup_i S^i| \times n$ matrix D, such that for every $s \in \cup_i S^i$ and $v \in V$, $D[s][v] = \delta_G(s, v)$. Notice, $|\cup_i S^i| \leq n$. This procedure $P(d)$ is significant for the following reason:

Lemma 1 (Implicit in Chan [4]). *If* $P(d)$ *can be implemented in* $O(T(P(d)))$ *time, then APSP on unweighted undirected graphs is in* $O(T(P(d)) + n^2 \cdot d)$ *time.*

Setting $d = n^\varepsilon$, Theorem 3 is obtained from the following lemma.

Lemma 2. $P(d)$ *is implementable in* $O(n^{2+\varepsilon} + \frac{mn}{d} + \frac{mn\log(n^2/m)}{\log^2 n})$ *time,* $\forall\varepsilon > 0$.

Proof. First, we modify Chan's original algorithm slightly. As in Chan's algorithm, we first do breadth-first search from every s_i in $O(mn/d)$ time overall. When doing this, for each distance $\ell = 0, \ldots, n-1$, we compute the sets $A^i_\ell = \{v \in V \mid \delta_G(s_i, v) = \ell\}$. Suppose that the rows (columns) of a matrix M are in one-to-one correspondence with the elements of some set U. Then, for every $u \in U$, let $\bar{u}$ be the index of the row (column) of M corresponding to u.

Consider each (s_i, S^i) pair separately. Let $k = |S^i|$. For $\ell = 0, \ldots, n-1$, let $B_\ell = \cup_{j=\ell-2}^{\ell+2} A^i_\ell$, where $A^i_j = \{\}$ when $j < 0$, or $j > n-1$.

The algorithm proceeds in n iterations. Each iteration $\ell = 0, \ldots, n-1$ produces a $k \times |B_\ell|$ matrix D_ℓ, and a $k \times n$ matrix OLD (both Boolean), such that for all $s \in S^i$, $u \in B_\ell$ and $v \in V$,

$$D_\ell[\bar{s}][\bar{u}] = 1 \text{ iff } \delta_G(s, u) = \ell \text{ and } \mathrm{OLD}[\bar{s}][\bar{v}] = 1 \text{ iff } \delta_G(s, v) \leq \ell.$$

At the end of the last iteration, the matrices D_ℓ are combined in a $k \times n$ matrix D^i such that $D[\bar{s}][\bar{v}] = \ell$ iff $\delta_G(s, v) = \ell$, for every $s \in S^i$, $v \in V$. At the end of the algorithm, the matrices D^i are concatenated to create the output D.

In iteration 0, create a $k \times |B_0|$ matrix D_0, so that for every $s \in S^i$, $D_0[\bar{s}][\bar{s}] = 1$, and all 0 otherwise. Let OLD be the $k \times n$ matrix with $\mathrm{OLD}[\bar{s}][\bar{v}] = 1$ iff $v = s$.

Consider a fixed iteration ℓ. We use the output $(D_{\ell-1}, \mathrm{OLD})$ of iteration $\ell - 1$. Create a $|B_{\ell-1}| \times |B_\ell|$ matrix N_ℓ, such that $\forall u \in B_{\ell-1}, v \in B_\ell,\ N_\ell[\bar{u}][\bar{v}] = 1$ iff v is a neighbor of u. Let N_ℓ have m_ℓ nonzeros. If there are b_ℓ edges going out of B_ℓ, one can create N_ℓ in $O(b_\ell)$ time: Reuse an $n \times n$ zero matrix N, so that N_ℓ

will begin at the top left corner. For each $v \in B_\ell$ and edge (v, u), if $u \in B_{\ell-1}$, add 1 to $N[\bar{u}][\bar{v}]$. When iteration ℓ ends, zero out each $N[\bar{u}][\bar{v}]$.

Multiply $D_{\ell-1}$ by N_ℓ in $O(k \cdot |B_{\ell-1}|^{1+\varepsilon} + m_\ell k \log(\frac{n^2}{m}) / \log^2 n)$ time (applying Corollary 1). This gives a $k \times |B_\ell|$ matrix A such that for all $s \in S^i$ and $v \in B_\ell$, $A[\bar{s}][\bar{v}] = 1$ iff there is a length-ℓ path between s and v. Compute D_ℓ by replacing $A[\bar{s}][\bar{v}]$ by 0 iff $\mathrm{OLD}[\bar{s}][\bar{v}] = 1$, for each $s \in S^i, v \in B_\ell$. If $D_\ell[\bar{s}][\bar{v}] = 1$, set $\mathrm{OLD}[\bar{s}][\bar{v}] = 1$.

Computing the distances from all $s \in S^i$ to all $v \in V$ can be done in $O(m + \sum_{\ell=1}^{n-1} (k \cdot |B_{\ell-1}|^{1+\varepsilon} + m_\ell k \log(n^2/m) / \log^2 n + b_\ell))$ time. However, since every node appears in at most $O(1)$ sets B_ℓ, $\sum_{\ell=0}^{n-1} |B_\ell| \leq O(n)$, $\sum_\ell b_\ell \leq O(m)$ and $\sum_\ell m_\ell \leq O(m)$. The runtime becomes $O(kn^{1+\varepsilon} + mk \log(n^2/m) / \log^2 n + m)$. When we sum up the runtimes for each (s_i, S^i) pair, since the sets S^i are disjoint, we obtain asymptotically

$$\sum_{i=1}^{n/d} \left(m + |S^i| \cdot \left(n^{1+\varepsilon} + \frac{m \log(n^2/m)}{\log^2 n} \right) \right) = \frac{mn}{d} + n^{2+\varepsilon} + \frac{mn \log(n^2/m)}{\log^2 n}.$$

As the data structure returns witnesses, we can also return predecessors. □

6 All Pairs Least Common Ancestors in a DAG

To our knowledge, the best algorithm in terms of m and n for finding all pairs least common ancestors (LCAs) in a DAG, is a dynamic programming algorithm by Czumaj, Kowaluk and Lingas [12, 7] which runs in $O(mn)$ time. We improve the runtime of this algorithm to $O(mn \log(n^2/m) / \log n)$ using the following generalization of Theorem 1, the proof of which is omitted. Below, for an $n \times n$ real matrix B and $n \times 1$ Boolean vector r, $B \odot r$ is the vector c with $c[i] = \max_{k=1,\ldots,n}(A[i][k] \cdot r[k])$ for each $i \in [n]$. This product clearly generalizes the Boolean matrix-vector product.

Theorem 5. *Let B be a $d \times n$ matrix with β-bit entries. Let $0 < \varepsilon < 1$ be constant, and let $\kappa \geq 1$ and $\ell > \kappa$ be integer parameters. Then one can create a data structure with $O(\frac{dn\kappa}{\ell} \cdot \lceil \frac{\beta}{\log n} \rceil \cdot \sum_{b=1}^{\kappa} \binom{\ell}{b})$ preprocessing time so that the following operations are supported on a pointer machine:*

- *given any $n \times 1$ binary vector r, output $B \odot r$ in $O(dn^\varepsilon + \frac{d\beta}{\log n} (\frac{n}{\ell} + \frac{m_r}{\kappa}))$ time, where m_r is the number of nonzeros in r;*
- *replace any column of B by a new column in $O(d\kappa \lceil \frac{\beta}{\log n} \rceil \sum_{b=1}^{\kappa} \binom{\ell}{b})$ time.*

Due to space limitations, the proof of the following is also omitted.

Theorem 6. *The all pairs least common ancestors problem on n node and m edge DAGs can be solved in $O(mn \log(n^2/m) / \log n)$ time.*

7 Conclusion

We have introduced a new combinatorial data structure for performing matrix-vector multiplications. Its power lies in its ability to compute sparse vector products quickly and tolerate updates to the matrix. Using the data structure, we gave new running time bounds for four fundamental graph problems: transitive closure, all pairs shortest paths on unweighted graphs, maximum weight triangle, and all pairs least common ancestors.

Acknowledgments. This research was supported in part by NSF ITR grant CCR-0122581 (The Aladdin Center).

References

1. A. V. Aho, J. E. Hopcroft, and J. Ullman. The design and analysis of computer algorithms. *Addison-Wesley Longman Publishing Co., Boston, MA*, 1974.
2. V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradzev. On economical construction of the transitive closure of an oriented graph. *Soviet Math. Dokl.*, 11:1209–1210, 1970.
3. T. M. Chan. All-pairs shortest paths with real weights in $O(n^3/\log n)$ time. *Proc. WADS*, 3608:318–324, 2005.
4. T. M. Chan. All-pairs shortest paths for unweighted undirected graphs in $o(mn)$ time. *Proc. SODA*, pages 514–523, 2006.
5. J. Cheriyan and K. Mehlhorn. Algorithms for dense graphs and networks on the random access computer. *Algorithmica*, 15(6):521–549, 1996.
6. D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *J. Symbolic Computation*, 9(3):251–280, 1990.
7. A. Czumaj, M. Kowaluk, and A. Lingas. Faster algorithms for finding lowest common ancestors in directed acyclic graphs. *TCS*, 380(1–2):37–46, 2007.
8. A. Czumaj and A. Lingas. Finding a heaviest triangle is not harder than matrix multiplication. *Proc. SODA*, pages 986–994, 2007.
9. T. Feder and R. Motwani. Clique partitions, graph compression and speeding-up algorithms. *Proc. STOC*, pages 123–133, 1991.
10. M. J. Fischer and A. R. Meyer. Boolean matrix multiplication and transitive closure. *Proc. FOCS*, pages 129–131, 1971.
11. Z. Galil and O. Margalit. All pairs shortest paths for graphs with small integer length edges. *JCSS*, 54:243–254, 1997.
12. M. Kowaluk and A. Lingas. LCA queries in directed acyclic graphs. *Proc. ICALP*, 3580:241–248, 2005.
13. J. I. Munro. Efficient determination of the transitive closure of a directed graph. *Inf. Process. Lett.*, 1(2):56–58, 1971.
14. W. Rytter. Fast recognition of pushdown automaton and context-free languages. *Information and Control*, 67(1–3):12–22, 1985.
15. R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *JCSS*, 51:400–403, 1995.
16. A. Shapira, R. Yuster, and U. Zwick. All-pairs bottleneck paths in vertex weighted graphs. *Proc. SODA*, pages 978–985, 2007.
17. V. Vassilevska, R. Williams, and R. Yuster. Finding the smallest H-subgraph in real weighted graphs and related problems. *Proc. ICALP*, 4051:262–273, 2006.
18. R. Williams. Matrix-vector multiplication in sub-quadratic time: (some preprocessing required). *Proc. SODA*, pages 995–1001, 2007.