CS 267 Lecture 11 Dynamic Transitive Closure
Scribe: Michael P. Kim **Date:** November 8, 2016

In this lecture, we discuss dynamic algorithms for computing the transitive closure of a graph. We describe the static transitive closure problem briefly and then discuss approaches to tackling the dynamic problem.

1 Transitive Closure

Formally, we define the transitive closure (TC) problem as follows. Given a directed graph $G = (V, E)$ with $|V| = n, |E| = m$, we aim to output an $n \times n$ matrix where $C(u, v) \neq 0$ iff v is reachable from u.

For the static version of the problem, there are two natural algorithms. Using depth-first search from every ndoe, we can compute TC in $O(mn)$ time. Using matrix multiplication we can successively square the adjacency matrix A of the graph defined as $A(u, v) = 1$ if $u = v$ or $(u, v) \in E$ and $A[u, v] = 0$ otherwise. After $\log_2 n$ squarings we will have A^n where $A^n(u, v) \neq 0$ iff v is reachable from u. Thus, we can compute the transitive closure in $O(n^\omega \log n)$ time for $\omega = 2.38$.

We don't discuss the algorithm here, but in 1971, Fisher and Meyer gave an $O(n^\omega)$ algorithm (without the $\log n$ dependence). It is not hard to show that any $T(n)$ time algorithm that computes the TC of an n-node graph can compute the Boolean product of two $n \times n$ matrices in $T(3n)$ time. Thus TC is asymptotically equivalent to Boolean matrix multiplication (BMM).

2 Dynamic Transitive Closure

In the dynamic version of transitive closure, we must maintain a directed graph $G = (V, E)$ and support the operations of deleting or adding an edge and querying whether v is reachable from u as quickly as possible. Recall that the best possible algorithm for the static case can also multiply Boolean matrices, then it would run in $O(n^{\omega'})$ time, where ω' is the exponent of BMM. Hence the best update time for TC we could hope for is $O(\frac{n^{\omega'}}{m}) = O(n^{\omega'-2})$. Currently the best bound on ω' is the same as for ω and is 2.38, and so the best known update times for TC can't be better than $O(n^{0.38})$.

Below we include a table of known algorithms and their amortized run times.

update(u, v)	*query*(u, v)	Paper	Notes
$O(n^2)$	$O(1)$	King/Sagert	Only for DAGs
		Sankowski	general graphs
$O(n^{1.575})$	$O(n^{0.575})$	Demetrescu/Italiano	Only for DAGs
		Sankowski	general graphs
$O(m\sqrt{n})$	$O(\sqrt{n})$	Rodity/Zwick	
$O(m + n \log n)$	$O(n)$	Rodity/Zwick	

Table 1: The best known bounds for dynamic transitive closure.

In this lecture, we will discuss the algorithms proposed by King/Sagert and Demetrescu/Italiano for DAGs. Cycles complicate the problem, so for the time being, we exclude them from our discussion.

3 King and Sagert

The key idea behind King/Sagert's strategy is to maintain a full transitive closure matrix C and update it as necessary. Clearly, then the query time is constant.

Notice however, that any approach that explicitly stores a transitive closure matrix cannot do better than $\Omega(n^2)$ time for updates. To see this, consider a graph consisting of an edge $e = (u, v)$ and where v points to $\Omega(n)$ nodes B, and $\Omega(n)$ nodes A point to u. Here e connects $\Omega(n^2)$ pairs of nodes. Repeatedly removing and inserting e would correspond to updating $\Omega(n^2)$ entries of the TC matrix in each update. In this sense, King and Sagert's algorithm is optimal for algorithms that explicitly store a TC matrix.

Operations

Let G be a DAG. We will maintain a matrix C where $C(u, v) =$ the number of paths from u to v.

insert(x, y): For all pairs (u, v), update $C(u, v) \leftarrow C(u, v) + C(u, x) \cdot C(y, u)$.

delete(x, y): For all pairs (u, v), update $C(u, v) \leftarrow C(u, v) - C(u, x) \cdot C(y, u)$.

query(x, y): Return REACHABLE iff $C(u, v) \neq 0$.

Obviously, *query*(x, y) returns in constant time. At first, it seems that both update operations would run in $O(n^2)$ time because there are n^2 updates to perform, but what if the number of paths between nodes is large? Could multiplying these large numbers push the run time beyond $O(n^2)$? How many paths can there be in a DAG between two nodes u and v. The following Claim shows that the complete DAG on n nodes has 2^{n-2} paths between the first and last node in the topological order.

Claim 1. *Let G be the DAG on $n \geq 2$ nodes where for every $i < j$ there is a directed edge (i, j). Then there are 2^{n-2} distinct paths between nodes 1 and n.*

Proof. We will prove this by induction. The base case is $n = 2$. Then there is $1 = 2^{2-2}$ path between nodes 1 and 2. Suppose that the complete DAG on $n - 1$ nodes has 2^{n-3} paths between its first and last node. Then consider the complete DAG on n nodes. The paths from 1 to n are of two types: those that go through node 2 and those that don't. The first type are exactly the paths between nodes 2 and n in the DAG excluding node 1, and the latter type are the paths between 1 and n in the DAG excluding node 2. Both of these by the inductive hypothesis are exactly 2^{n-3} and hence the number of paths between 1 and n is $2 \times 2^{n-3} = 2^{n-2}$. $\square$

Claim 1 shows that for any n-node DAG and any u, v, $C(u, v) \leq 2^{n-2}$, and there exists DAGs in which for some u, v, $C(u, v) = 2^{n-2}$. The numbers can indeed be exponential! Representing these values could, thus, require $\Omega(n)$ bits. Multiplications on b bit integers take $\Omega(b)$ time, so in the worst case, the run time of each update could be $\Omega(n^3)$.

However, we promised an $O(n^2)$ algorithm. The key idea to attaining this improvement is to use a random prime modulus. We will let c be a constant and p be a random prime $p \in (n^c, 2n^c)$. Then we will do all computations modulo p. Since p can be represented using $O(\log n)$ bits, all operations will be fast.

We will support up to n^k operations, where we pick c to be large in terms of k, say $c > k - 1$. (In the world there are only polynomially many operations :) .)

Claim 2. *With high probability, $\forall u, v,\ C(u, v) \neq 0 \iff v$ is reachable from u.*

Proof. Fix $C(u, v)$. We know $C(u, v) < 2^n$ and that a query will be incorrect if and only if $C(u, v) \neq 0$ but p divides $C(u, v)$. The number of primes in $[n^c, 2n^c]$ that divide $C(u, v)$ must be less than $\log_{n^c}(C(u, v)) \leq O(\frac{n}{\log n})$. By the prime number theorem there are $\Omega(\frac{n^c}{\log n})$ primes over the range $[n^c, 2n^c]$. Thus, we get the following expression for the probability that the prime we choose divides a fixed $C(u,v)$.

$$Pr[p \text{ divides } C(u,v)] \le O\left(\frac{1}{n^{c-1}}\right).$$

We performed $\le n^k$ operations on every $C(\cdot,\cdot)$. Thus, the overall probability that some operation was incorrect can be expressed by a union bound as follows.

$$Pr[\text{algorithm is incorrect over } n^k \text{ operations}] \le O\left(\frac{1}{n^{c-k-1}}\right) \le O\left(\frac{1}{n}\right).$$

So the probability that all queries are correct is greater than or equal to $1 - O(\frac{1}{n})$. □

4 Demetrescu and Italiano

What if we don't maintain C explicitly? Can we improve update time while still maintaining fast query time? Consider what happens in the first algorithm after B updates. Let C_i be the matrix C after the ith update. C_0 is the initial state of C. On the ith update, for all pairs of nodes $u, v \in V$, we perform

$$C_i(u,v) \leftarrow C_{i-1}(u,v) \pm C_{i-1}(u,x) \cdot C_{i-1}(y,v),$$

where we perform $+$ for inserts and $-$ for deletes.

We can model each update i with a matrix addition and multiplication.

$$C_i \leftarrow C_{i-1} \pm C_{i-1}^{col}(x) \cdot C_{i-1}^{row}(y),$$

where $C_{i-1}^{col}(x)$ is the xth column vector of C_{i-1} (hence of dimensions $n \times 1$) and $C_{i-1}^{row}(y)$ is the yth row vector of C_{i-1} (hence a $1 \times n$ vector).

Further, we can express all B updates as a matrix addition and multiplication.

$$C_B \leftarrow C_0 + \begin{bmatrix} \pm C_0^{col}(x_1) & \pm C_1^{col}(x_2) & \dots & \pm C_{B-1}(x_B) \end{bmatrix} \cdot \begin{bmatrix} C_0^{row}(y_1) \\ C_1^{row}(y_2) \\ \vdots \\ C_{B-1}(y_B) \end{bmatrix}.$$

Note that this matrix operation could be done as a batch update by multiplying an $n \times B$ matrix with a $B \times n$ matrix. We can treat this as $(\frac{n}{B})^2$ $B \times B$ matrix multiplications. If we knew, a priori, $C_i^{col}(x_{i+1})$ and $C_i^{row}(y_{i+1})$ for all i then we could multiply $C_{cols} \cdot C_{rows}$ in $(\frac{n}{B})^2 B^\omega = n^2 B^{\omega-2}$ run time. Remember though, that this update time is for B updates, which gives an amortized runtime of $O(\frac{n^2 B^{\omega-2}}{B}) = O(\frac{n^2}{B^{3-\omega}})$. This is an improvement over the $O(n^2)$ update time because $\omega < 3$.

What remains is to show how to compute the columns and rows of each C_i as necessary. We will maintain C_0 as well as a buffer of up to B operations $\{C_0^{col}(x_1) \cdot C_0^{row}(y_1), \dots C_{B-1}^{col}(x_B) \cdot C_{B-1}^{row}(y_B)\}$. We can then define the $O(B)$ operation for querying as follows.

Algorithm 1: *query*(u, v)

```
Look up $C_0(u,v)$;
$c_{uv} \leftarrow C_0(u,v) + \sum_{i=1}^{B} C_{i-1}^{col}(x_i)[u] \cdot C_{i-1}^{row}(y_i)[v]$;
Return REACHABLE iff $c_{uv} \neq 0$;
```

To insert an edge, we will need to update the buffer and potentially reset C_0 and the buffer. The pseudocode is below. To delete an edge the same operations are performed, except that $-C_{j+1}^{col}(x) \cdot C_{j+1}^{row}(y)$ is inserted in the buffer instead.

Algorithm 2: *insert*(x, y)

```
Let the buffer have $j$ elements;
Compute $C^{col}_{j+1}(x), C^{row}_{j+1}(y)$ as follows:;
    For each $u$, compute and set $C^{col}_{j+1}(x)[u] \leftarrow query(u, x)$ and $C^{row}_{j+1}(y)[u] \leftarrow query(y, u)$;
if $j + 1 \leq B$ then
    insert $C^{col}_{j+1}(x) \cdot C^{row}_{j+1}(y)$ into the buffer and return.
else
    perform the batch update operation to compute a new $C_0$ and reset the buffer.
```

Note that each of the $O(n)$ calls to *query*() costs $O(B)$ time, and we already argued that the amortized run time of the batch update operation is $O(\frac{n^2}{B^{3-\omega}})$, so the overall amortized run time is $O(nB + \frac{n^2}{B^{3-\omega}})$. If we set $B = n^{1/(4-\omega)}$ then we see the update time is $O(n^{1+1/(4-\omega)}) = O(n^{1.616})$. The query time is $O(B) = O(n^{0.616})$. With fast rectangular matrix multiplication, we can improve the update and query runtime as stated in Table 1, but this is beyond the scope of this lecture.

We note that just as in the King and Sagert algorithm we need to do all operations modulo a suitably large random prime.

References

[1] M.J. Fischer and A.R. Meyer. Boolean matrix multiplication and tranisitive closure. SWAT 1971.

[2] V. King and G. Sagert. A fully dynamic algorithm for maintaining the transitive closure. STOC 1999.

[3] C. Demetrescu and G.F. Italiano. Trade-offs for Fully Dynamic Transitive Closure on DAGs: Breaking Through the $O(n^2)$ Barrier. FOCS 2000.