1 Graph Transitive Closure and BMM

Definition 1.1. *The Transitive Closure (TC) of a directed graph $G = (V, E)$ on n nodes is an $n \times n$ matrix such that* $\forall u, v \in E$ $(T(u, v) = \begin{cases} 1 & \text{if } v \text{ is reachable from } u \\ 0 & \text{otherwise} \end{cases}$

Transitive Closure on an undirected graph is trivial in linear time– just compute the connected components. In contrast, we will show that for directed graphs the problem is equivalent to BMM.

Theorem 1.1. *Transitive closure is equivalent to BMM*

Proof. We prove equivalence in both directions.

Claim 1. *If TC is in $T(n)$ time then BMM on $n \times n$ matrices is in $O(T(3n))$ time.*

Proof. Consider a graph like the one below, where there are three rows of n vertices. Given two boolean matrices A and B, we can create such a graph by adding an edge between the i^{th} vertex of the first row and the j^{th} of the second row iff $A_{ij} == 1$. Construct edges between the second and third rows in a similar fashion for B. Thus, the product AB can be computed by taking the transitive closure of this graph. It is equivalent to BMM by simply taking the $\wedge$ of $ij \to jk$. Since the graph has $3n$ nodes, given a $T(n)$ algorithm for TC, we have an $O(T(3n))$ algorithm for BMM. (Of course, it takes n^2 time to create the graph, but this is subsumed by $T(n)$ as $T(n) \geq n^2$ as one must at least print the output of AB.)

□

Claim 2. *If BMM is in $T(n)$ time, such that $T(n/2) \leq T(n)/(2 + \epsilon)$, then TC is in $O(T(n))$ time.*

We note that the condition in the claim is quite natural. For instance it is true about $T(n) = n^c$ for any $c > 1$.

Proof. Let A be the adjacency matrix of some graph G. Then $(A + I)^n$ is the transitive closure of G. Since we have a $T(n)$ algorithm for BMM, we can compute $(A + I)^n$ using $\log n$ successive squarings of $A + I$ in $O(T(n) \log n)$ time. We need to get rid of the log term to show equivalence.

We do so using the following algorithm:

1. Compute the strongly connected components of G and collapse them to get G', which is a D.A.G. We can do this in linear time.
2. Compute the topological order of G' and reorder vertices according to it (linear time)
3. Let A be the adjacency matrix of G'. $(A + I)$ is upper triangular. Compute $C = (A + I)^*$, i.e. the TC of G'.
4. Uncollapse SCCs. (linear time)

All parts except (3) take linear time. We examine part (3).
Consider the matrix $(A + I)$ split into four sub-matrices M, C, B, and 0 each of size $n/2 \times n/2$.
$(A + I) = \begin{bmatrix} M & C \\ 0 & B \end{bmatrix}$

We claim that $(A+I)^* = \begin{bmatrix} M^* & M^*CB^* \\ 0 & B^* \end{bmatrix}$

The reasoning behind this is as follows. Let U be the first $n/2$ nodes in the topological order, and let V be the rest of the nodes. Then M is the adjacency matrix of the subgraph induced by U and B is the adjacency matrix induced by V. The only edges between U and V go from U to V. Thus, M^* and U^* represents the transitive closure restricted to $U \times U$ and $V \times V$. For the TC entries for $u \in U$ and $v \in V$, we note that the only way to get from u to v is to go from u to possibly another $u' \in U$ using a path within U, then take an edge (u', v') to a node $v' \in V$ and then to take a path from v' to v within V. I.e. the $U \times V$ entries of the TC matrix are exactly M^*CB^*.

Suppose that the runtime of our algorithm on n node graphs is $\text{TC}(n)$. To calculate the transitive closure matrix, we recursively compute M^* and B^*. Since each of these matrices have dimension $n/2$, this takes $2TC(n/2)$ time.

We then compute M^*CB^*, which takes $O(T(n))$ time, where $T(n)$ was the time to compute the boolean product of $n \times n$ matrices.

Finally, we have $TC(n) \leq 2TC(\frac{n}{2}) + O(T(n))$. If we assume that there is some $\varepsilon > 0$ such that $T(n/2) \leq T(n)/(2+\epsilon)$, then the recurrence solves to $TC(n) = O(T(n))$.

□

It follows from claim 1 and claim 2 that BMM of $n \times n$ matrices is equivalent in asymptotic runtime to TC of a graph on n nodes. □

2 Introduction to APSP

In this lecture, we will talk about the problem of computing all-pairs shortest paths (APSP) using matrix multiplication. Formally, given a graph $G = (V, E)$, the goal is to compute the distance $d(u, v)$ for all pairs of nodes $u, v \in V$. Given that G has n nodes and m edges, it is easy to come up with algorithms that run in $\tilde{O}(mn)$ time [1] – for example, on graphs with nonnegative edge weights, we can run Dijkstra's Algorithm from each node. When G is dense, this time is on the order of n^3; the natural question is can we do better?

For weighted graphs, the best known algorithm for APSP runs in $O\left(\frac{n^3}{2^{\Omega(\sqrt{\log n})}}\right)$ time, by Ryan Williams (2014). A major open problem is whether there exist "truly subcubic" algorithms for this version of APSP, namely, algorithms running in time $O(n^{3-\varepsilon})$ for some constant $\varepsilon > 0$.

For unweighted graphs, we know of algorithms that achieve this subcubic performance. In particular, for undirected graphs there is an algorithm running in $O(n^\omega \log n)$ time by Seidel (1992), and for directed graphs there is an algorithm running in $O(n^{2.575})$ time by Zwick (2002). This difference in runtime persists even with improvements in the matrix multiplication exponent, ω. For instance, if $\omega = 2$, then Seidel's algorithm would run in $\tilde{O}(n^2)$ time, whereas Zwick's algorithm would run in $\tilde{O}(n^{2.5})$ time. In this lecture, we will talk about the algorithms for unweighted graphs. In particular, we discuss a baseline algorithm using a hitting set which will work for directed or undirected graphs, and then describe Seidel's algorithm for undirected graphs.

3 Hitting Set Algorithm

Given G, and particularly the adjacency matrix A which represents G, we would like to compute distances between all pairs of nodes. A natural first algorithm would be to compute the distances by successive boolean matrix multiplication of A with itself. The (i, j)th entry in A^k is 1 if and only if i has a path of length k to j. Thus, if the graph has finite diameter, then for all i, j, $d(i, j) = \min\{k \mid A^k[i, j] = 1\}$.

Fact 3.1. *If G has diameter D we can compute APSP in $O(Dn^\omega)$ time.*

[1] $\tilde{O}(T(n)) = O(T(n) \cdot \text{polylog } n)$. In other words, the poly-logarithmic terms have been dropped.

If the diameter of G is small, then we have found a fast algorithm for computing APSP, but D can be $O(n)$ in which case we have no improvement from $O(n^3)$ run time. Nevertheless, we can compute all short distances less than some k in $O(kn^\omega)$ time, and then employ another technique to compute longer distances. The key idea is to use a "hitting set".

Lemma 3.1. *(Hitting Set) Let S be a collection of m sets of size $\geq k$ over $V = [n]$. Fix any constant $c \geq 1$. With probability at least $1 - 1/m^c$, a uniformly random subset $T \subseteq V$ of size $(c+1)\frac{n}{k} \log m$ intersects all the sets in S nontrivially.*

With this hitting set lemma in mind, we can use Algorithm 1 to compute distances that are greater than or equal to k.

Algorithm 1: LONGDIST(V, E)

Pick $T \subseteq V$ randomly s.t. $|T| = c \cdot \frac{n}{k} \log n$ for large enough constant c
foreach $t \in T$ **do**
  Compute $\text{BFS}_G(t)$ and $\text{BFS}_{G^{rev}}(t)$ [2]
foreach $u, v \in V$ **do**
  Compute $d(u, v) = \min_{t \in T} \; d(u, t) + d(t, v)$

With high probability, this algorithm will compute distances $\geq k$ correctly (as these paths involve at least k nodes, so with high probability T hits the path). The algorithm requires running BFS from and to $O(\frac{n}{k} \log n)$ nodes so takes $\tilde{O}(\frac{n}{k} n^2)$ time. If we use this algorithm to compute long distances and the iterative matrix multiplication to compute short distances, we have an algorithm for all-pairs shortest paths.

Theorem 3.1. *Let G be an unweighted directed or undirected graph on n nodes. APSP in G can be computed in $\tilde{O}(kn^\omega + \frac{n}{k} n^2)$ time.*

When we optimize for a choice of k and set it to $n^{(3-\omega)/2}$, the runtime comes out to be $\tilde{O}\left(n^{\frac{3+\omega}{2}}\right)$ which is roughly $\tilde{O}(n^{2.69})$.

We will later see how to vastly improve this algorithm obtaining Zwick's algorithm for APSP.

4 Seidel's Algorithm

While this first algorithm gives us a fast algorithm for computing APSP, the question remains, can we do better? In particular, can we avoid computing short and long distances separately? Can we leverage matrix multiplication to compute all the shortest paths?

In fact, we can improve on the hitting set algorithm for undirected graphs as follows. Given a graph G with adjacency matrix A, consider its boolean square $A^2 = A \cdot A$, where $\cdot$ represents boolean matrix multiplication. Consider a graph G' with adjacency matrix $A' = A^2 \vee A$.

Fact 4.1. $d_{G'}(s, t) = \left\lceil \frac{d(s,t)}{2} \right\rceil$

To see this fact, note that edges in A^2 represent paths of length 2 in the original graph G, and G' also contains the edges of G. Thus, any path of length $2k$ in G induces a path of length k in G' using only edges of A^2, and also any path of length $2k + 1$ induces a path of length k (from A^2) followed by a single original edge, thus forming a path of length $k + 1$.

Now suppose that we have a way of determining the parity of the distance between all pairs of nodes. Then we can use the following recursive strategy to compute APSP.

Note that in each recursive call, the diameter of the graph halves, and that after $\log n$ iterations, A will be the all 1s matrix with 0s along the diagonal, which we can detect (assuming the graph is connected to begin with, which we can ensure by processing each connected component separately). Thus, if we can find

Algorithm 2: APSP Idea

Given an adjacency matrix A
Compute $A^2 \vee A$
Recursively compute $d' \leftarrow \text{APSP}(A^2 \vee A)$
foreach $u, v \in V$ **do**
  if $d(u, v)$ *is even* **then**
    $d(u, v) = 2d'(u, v)$
  else
    $d(u, v) = 2d'(u, v) - 1$

a way to determine the parity of the distances efficiently, we would obtain an efficient recursive algorithm for APSP.

Consider any pair of nodes $i, j \in V$ and another node which is a neighbor of j, $k \in N(j)$. By the triangle inequality (which holds in unweighted, undirected graphs), we know $d(i, j) - 1 \leq d(i, k) \leq d(i, j) + 1$.

Claim 3. *If* $d(i, j) \equiv d(i, k) \mod 2$*, then* $d(i, j) = d(i, k)$*.*

Proof. By the triangle inequality, $d(i, j)$ and $d(i, k)$ differ by at most 1. Thus, if their parity is the same, they must also be equal. □

Claim 4. *Let* $d_{G^2}(i, j)$ *be the distance between* i *and* j *in* G^2 *defined by* $A^2 \vee A$*. Then, (a) if* $d(i, j)$ *is even and* $d(i, k)$ *is odd then* $d_{G^2}(i, k) \geq d_{G^2}(i, j)$*. (b) If* $d(i, j)$ *is odd and* $d(i, k)$ *is even,* $d_{G^2}(i, k) \leq d_{G^2}(i, j)$ *and there exists a* $k' \in N(j)$ *such that* $d_{G^2}(i, k') < d_{G^2}(i, j)$*.*

Proof of (a).

$$d_{G^2}(i, j) = \frac{d(i, j)}{2}$$
$$d_{G^2}(i, k) = \frac{d(i, k) + 1}{2} \geq \frac{d(i, j)}{2}$$

so $d_{G^2}(i, k) \geq d_{G^2}(i, j)$. □
Proof of (b).

$$d_{G^2}(i, j) = \frac{d(i, j) + 1}{2} \geq \frac{d(i, k)}{2} = d_{G^2}(i, k)$$

so in general, $d_{G^2}(i, k) \leq d_{G^2}(i, j)$, and for the neighbor of j along the shortest path from i to j, which we call k', we know that $d(i, k') = d(i, j) - 1$ and since $d(i, j)$ is odd, $d_{G^2}(i, k') = d_{G^2}(i, j) - 1$. □

Claim 5. *If* $d(i, j)$ *is even, then*

$$\sum_{k \in N(j)} d_{G^2}(i, k) \geq deg(j) d_{G^2}(i, j)$$

and if $d(i, j)$ *is odd, then*

$$\sum_{k \in N(j)} d_{G^2}(i, k) < deg(j) d_{G^2}(i, j)$$

This third claim follows directly from the first two. Additionally, if we can compute the sums here in $O(n^\omega)$ time, then the overall runtime will be $O(n^\omega \log n)$ as desired. The right expression can be computed in $O(n^2)$ time, which will be subsumed by the $O(n^\omega)$ term.

Consider D, an $n \times n$ matrix where $D(i,j) = d_{G^2}(i,j)$. We want to decide for each i,j pair whether $\sum_{k \in N(j)} d_{G^2}(i,k) < deg(j) d_{G^2}(i,j)$. Consider the integer matrix product DA. Note that

$$(D \cdot A)[i,j] = \sum_{k \in N(j)} d_{G^2}(i,k)$$

so this matrix product allows us to compute the left expression.

Now we are ready to state Seidel's Algorithm in full.

Algorithm 3: SEIDEL(A)

```
if A is all 1s except the diagonal then
    return A
else
    Compute boolean product A^2
    D ← SEIDEL(A^2 ∨ A)
    Compute integer product D · A
    R ← 0^{n×n}
    foreach i, j ∈ V do
        if DA(i,j) < deg(j)D(i,j) then
            R(i,j) ← 2D(i,j) − 1
        else
            R(i,j) ← 2D(i,j)
    return R
```

Claim 6. *Seidel's Algorithm runs in $O(n^\omega \log d)$ time where d refers to the diameter of the graph.*

Proof. The run time can be expressed as the following recurrence relation.

$$T(n,d) \leq T(n, \frac{d}{2}) + O(n^\omega)$$
$$\implies T(n,d) \leq O(n^\omega \log d)$$

Because $d \leq n$, this run time is upper bounded by $O(n^\omega \log n)$. □

Note that Seidel's Algorithm relies on fast integer matrix multiplicaion, which runs in $O(n^\omega)$, but for which no known fast combinatorial algorithms exist. Some questions remain open whose answers could speed up the computation of APSP in theory and in practice: Is the integer matrix multiplication step avoidable? Are there fast combinatorial matrix multiplication algorithms over the integers?

References

[1] Raimund Seidel, *On the All-Pairs-Shortest-Path Problem in Unweighted Undirected Graphs*, Journal of Computer and System Sciences 51, pp. 400-403 (1995).