Lectures 1 and 2 Matrix multiplication and matrix inversion
Scribe: Jessica Su **Date:** September 25, 2015
Editor: Kathy Cooper

This class is about matrix multiplication and how it can be applied to graph algorithms.

1 Prior work on matrix multiplication

Definition 1.1. *(Matrix multiplication) Let A and B be n-by-n matrices (where entries have $O(\log n)$ bits). Then the product C, where $AB = C$ is an n-by-n matrix defined by $([i,j]) = \sum_{k=1}^{n} A(i,k)B(k,j)$.*

We will assume that operations (addition and multiplication) on $O(\log n)$ integers takes $O(1)$ time, i.e. we'll be working in a word-RAM model of computation with word size $O(\log n)$.

There has been much effort to improve the runtime of matrix multiplication. The trivial algorithm multiplies $n \times n$ matrices in $O(n^3)$ time. Strassen ('69) surprised everyone by giving an $O(n^{2.81})$ time algorithm. This began a long line of improvements until in 1986, Coppersmith and Winograd achieved $O(n^{2.376})$. After 24 years of no progress, in 2010 Andrew Stothers, a graduate student in Edinburgh, improved the running time to $O(n^{2.374})$. In 2011, Virginia Williams got $O(n^{2.3729})$, which was the best bound until Le Gall got $O(n^{2.37287})$ in 2014. Many believe that the ultimate bound will be $n^{2+O(1)}$, but this has yet to be proven.

Today we'll discuss the relationship between the problems of matrix inversion and matrix multiplication.

2 Matrix multiplication is equivalent to matrix inversion

Matrix inversion is important because it is used to solve linear systems of equations. Multiplication is equivalent to inversion, in the sense that any multiplication algorithm can be used to obtain an inversion algorithm with similar runtime, and vice versa.

2.1 Multiplication can be reduced to inversion

Theorem 2.1. *If one can invert n-by-n matrices in $T(n)$ time, then one can multiply n-by-n matrices in $O(T(3n))$ time.*

Proof. Let A and B be matrices. Let

$$D = \begin{bmatrix} I & A & 0 \\ 0 & I & B \\ 0 & 0 & I \end{bmatrix}$$

where I is the n-by-n identity matrix. One can verify by direct calculation that

$$D^{-1} = \begin{bmatrix} I & -A & AB \\ 0 & I & -B \\ 0 & 0 & I \end{bmatrix}$$

Inverting D takes $O(T(3n))$ time and we can find AB by inverting C. Note that C is always invertible since its determinant is 1. □

2.2 Inversion can be reduced to multiplication

Theorem 2.2. *Let $T(n)$ be such that $T(2n) \geq (2+\varepsilon)T(n)$ for some $\varepsilon > 0$ and all n. If one can multiply n-by-n matrices in $T(n)$ time, then one can invert n-by-n matrices in $O(T(n))$ time.*

Proof idea: First, we give an algorithm to invert symmetric positive definite matrices. Then we use this to invert arbitrary invertible matrices.

The rest of Section 2 is dedicated to this proof.

2.2.1 Symmetric positive definite matrices

Definition 2.1. *A matrix A is symmetric positive definite if*

1. *A is symmetric, i.e. $A = A^t$, so $A(i,j) = A(j,i)$ for all i, j*
2. *A is positive definite, i.e. for all $x \neq 0$, $x^t A x > 0$.*

2.2.2 Properties of symmetric positive definite matrices

Claim 1. *All symmetric positive definite matrices are invertible.*

Proof. Suppose that A is not invertible. Then there exists a nonzero vector x such that $Ax = 0$. But then $x^t A x = 0$ and A is not symmetric positive definite. So we conclude that all symmetric positive definite matrices are invertible. □

Claim 2. *Any principal submatrix of a symmetric positive definite matrix is symmetric positive definite. (An m-by-m matrix M is a principal submatrix of an n-by-n matrix A if M is obtained from A by removing its last $n-m$ rows and columns.)*

Proof. Let x be a vector with m entries. We need to show that $x^t M x > 0$. Consider y, which is x padded with $n-m$ trailing zeros. Since A is symmetric positive definite, $y^t A y > 0$. But $y^t A y = x^t M x$, since all but the first m entries are zero. □

Claim 3. *For any invertible matrix A, $A^t A$ is symmetric positive definite.*

Proof. Let x be a nonzero vector. Consider $x^t(A^t A)x = (Ax)^t(Ax) = ||Ax||^2 \geq 0$. We now show $||Ax||^2 > 0$. For any $x \neq 0$, Ax is nonzero, since A is invertible. Thus, $||Ax||^2 > 0$ for any $x \neq 0$. So $A^t A$ is positive definite. Furthermore, it's symmetric since $(A^t A)^t = A^t A$. □

Claim 4. *Let n be even and let A be an $n \times n$ symmetric positive definite matrix. Divide A into four square blocks (each one $n/2$ by $n/2$):*

$$A = \begin{bmatrix} M & B^t \\ B & C \end{bmatrix}.$$

Then the Schur complement, $S = C - BM^{-1}B^t$, is symmetric positive definite.

The proof of the above claim will be in the homework.

2.2.3 Reduction for symmetric positive definite matrices

Let A be symmetric positive definite, and divide it into the blocks M, B^t, B, and C. Again, let $S = C - BM^{-1}B^t$. By direct computation, we can verify that

$$A^{-1} = \begin{bmatrix} M^{-1} + M^{-1}B^t S^{-1} B M^{-1} & -M^{-1}B^t S^{-1} \\ -S^{-1}BM^{-1} & S^{-1} \end{bmatrix}$$

Therefore, we can compute A^{-1} recursively, as follows: (let the runtime be $t(n)$)

Algorithm 1: Inverting a symmetric positive definite matrix

Compute M^{-1} recursively (this takes $t(n/2)$ time)
Compute $S = C - BM^{-1}B^t$ using matrix multiplication (this takes $O(T(n))$ time)
Compute S^{-1} recursively (this takes $t(n/2)$ time)
Compute all entries of A^{-1} (this takes $O(T(n))$ time)

The total runtime of the procedure is

$$t(n) \leq 2t(n/2) + O(T(n)) \leq O(\sum_j 2^j T(n/2^j))$$

$$\leq O(\sum_j (2/(2+\varepsilon))^j T(n)) \leq O(T(n)).$$

2.2.4 Reduction for any matrix

Suppose that inverting a symmetric positive definite matrix reduces to matrix multiplication. Then consider the problem of inverting an arbitrary invertible matrix A. By Claim 3, we know that A^tA is symmetric positive definite, so we can easily find $C = (A^tA)^{-1}$. Then $CA^t = A^{-1}A^{-t}A^t = A^{-1}$, so we can compute A^{-1} by multiplying C with A^t.

3 Boolean Matrix Multiplication (Introduction)

Scribe: Robbie Ostrow
Editor: Kathy Cooper

Given two $n \times n$ matrices A, B over $\{0, 1\}$, we define Boolean Matrix Multiplication (BMM) as the following:

$$(AB)[i, j] = \bigvee_k (A(i, k) \wedge B(k, j))$$

Note that BMM can be computed using an algorithm for integer matrix multiplication, and so we have BMM for $n \times n$ matrices is in $O(n^{\omega+O(1)})$ time, where $\omega < 2.373$ (the current bound for integer matrix multiplication).

Most theoretically fast matrix multiplication algorithms are impractical. Therefore, so called "combinatorial algorithms" are desirable. "Combinatorial algorithm" is loosely defined, but one has the following properties:

- Doesn't use subtraction
- All operations are relatively practical (like a lookup tables)

Remark 1. *No $O(n^{3-\epsilon})$ time combinatorial algorithms for matrix multiplication are known for $\varepsilon > 0$, even for BMM! Such an algorithm would be known as "truly subcubic."*

4 Four Russians

In 1970, Arlazarov, Dinic, Kronrod, and Faradzev (who seem not to have all been Russian) developed a combinatorial algorithm for BMM in $O(\frac{n^3}{\log n})$, called the Four-Russians algorithm. With a small change to the algorithm, its runtime can be made $O(\frac{n^3}{\log^2 n})$. In 2009, Bansal and Williams obtained an improved

algorithm running in $O(\frac{n^3}{\log^{2.25} n})$ time. In 2014, Chan obtained an algorithm running in $O(\frac{n^3}{\log^3 n})$ and then, most recently, in 2015 Yu, a Stanford graduate student, achieved an algorithm that runs in $O(\frac{n^3}{\log^4 n})$. Today we'll present the Four-Russians algorithm.

4.1 Four-Russians Algorithm

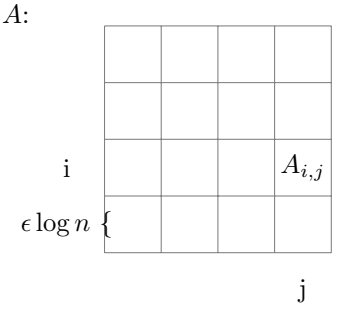
We start with an assumption:

- We can store a polynomial number of lookup tables T of size n^c where $c \leq 2 + \epsilon$, such that given the index of a table T, and any $O(\log n)$ bit vector x, we can look up $T(x)$ in constant ($O(1)$) time.

Theorem 4.1. *BMM for $n \times n$ matrices is in $O(\frac{n^3}{\log^2 n})$ time (on a word RAM with word size $O(\log n)$).*

Proof. We give the Four Russians' algorithm. (More of a description of the algorithm than a full proof of correctness.)

Let A and B be $n \times n$ boolean matrices. Choosing an arbitrary ϵ, we can split A into blocks of size $\epsilon \log n \times \epsilon \log n$. That is, A is partitioned into blocks $A_{i,j}$ for $i, j \in [\frac{n}{\epsilon \log n}]$. Below we give a simple example of A:



For each choice of i, j we create a lookup table $T_{i,j}$ corresponding to $A_{i,j}$ with the following specification:

For every bit vector v with length $\epsilon \log n$:

$T_{i,j}[v] = A_{i,j} \cdot v$.

That is, $T_{i,j}$ takes keys that are $\epsilon \log n$-bit sequences and stores $\epsilon \log n$-bit sequences. Also since there are n^ϵ bit vectors of $\epsilon \log n$ bits, and $A_{i,j} \cdot v$ is $\epsilon \log n$ bits, we have $|T_{i,j}| = n^\epsilon \epsilon \log n$.

The entire computation time of these tables is asymptotically

$$(\frac{n}{\log n})^2 n^\epsilon \log^2 n = n^{2+\epsilon},$$

since there are $(\frac{n}{\log n})^2$ choices for i, j, n^ε vectors v, and for each $A_{i,j}$ and each v, computing $A_{i,j}v$ take $O(\log^2 n)$ time for constant ε.

Given the tables that we created in subcubic time, we can now look up any $A_{ij} \cdot v$ in constant time.

We now consider the matrix B. Split each column of B into $\frac{n}{\epsilon \log n}$ parts of $\varepsilon \log n$ consecutive entries. Let B_j^k be the j^{th} piece of the k^{th} column of B. Each $A_{ij}B_j^k$ can be computed in constant time, because it can be accessed from $T_{ij}[B_j^k]$ in the tables created from preprocessing.

To calculate the product $Q = AB$, we can do the following.

From $j = 1$ to $\frac{n}{\epsilon \log n}$: $Q_{ik} = Q_{ik} \vee (A_{ij} \wedge B_j^k)$, by the definition. With our tables T, we can calculate the bitwise "and" (or *) in constant time, but the "or" (or sum) still takes $O(\log n)$ time. This gives us an algorithm running in time $O(n \cdot \frac{n}{\log n}^2 \cdot \log n) = O(\frac{n^3}{\log n})$ time, the original result of the four Russians.

How can we get rid of the extra $\log n$ term created by the sum?

We can precompute all possible pairwise sums! Create a table S such that $S(u, v) = u \vee v$ where $u, v \in \{0,1\}^{\epsilon \log n}$. This takes us time $O(n^{2\epsilon} \epsilon \log n)$, since there are $n^{2\epsilon}$ pairs u, v and each component takes only $O(\log n)$ time.
This precomputation allows us constant time lookup of any possible pairwise sum of $\epsilon \log n$ bit vectors.

Hence, each $Q_{ik} = Q_{ik} \vee (A_{ij} \wedge B_j^k)$ operation takes $O(1)$ time, and the final algorithm asymptotic runtime is

$$n \cdot (n/\varepsilon \log n)^2 = n^3 / \log^2 n,$$

where the first n counts the number of columns k of B and the remaining term is the number of pairs i, j.

Thus, we have a combinatorial algorithm for BMM running in $O(\frac{n^3}{\log^2 n})$ time.

□

Note: can we save more than a log factor? This is a major open problem.

5 Transitive Closure

Definition 5.1. *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 5.1. *Transitive closure is equivalent to BMM*

Proof. We prove equivalence in both directions.

Claim 5. *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 6. *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. □

6 Other Notes

BMM can also solve problems that look much simpler. For example: does a directed graph have a triangle? We can easily solve with an algorithm for BMM by taking the adjacency matrix, cubing it, and checking if the resulting matrix contains a 1 in the diagonal.

Somewhat surprisingly, it is also known that for any $\epsilon > 0$, an $O(n^{3-\epsilon})$ time combinatorial algorithm for triangle finding also implies an $O(n^{3-\epsilon/3})$ time combinatorial algorithm for BMM. Hence in terms of combinatorial algorithms BMM and triangle finding are "subcubic"-equivalent.

CS 367 Lecture 2 (part 1) All-Pairs Shortest Paths
Scribe: Michael P. Kim **Date:** October 8, 2015

1 Introduction

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.

2 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 2.1. *If G has diameter D we can compute APSP in $O(Dn^\omega)$ time.*

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 2.1. *(Hitting Set) Let S be a collection of n^2 sets of size $\geq k$ over $V = [n]$. With high probability, a random subset $T \subseteq V$ of size $O(\frac{n}{k} \log n)$ hits all the sets in S.*

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

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 Dijkstra's algorithm from $O(\frac{n}{k} \log n)$ 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 2.1. *Let G be a directed or undirected graph on n nodes, with unit weights. APSP of G can be computed in $\tilde{O}(kn^\omega + \frac{n}{k} n^2)$ time.*

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

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 Dijkstra(t)
foreach $u, v \in V$ **do**
  Compute $d(u, v) = \min_{t \in T} \; d(u, t) + d(t, v)$

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 $\tilde{O}(n^{2.69})$.

3 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 3.1. $d_{G'}(s, t) = \left\lceil \frac{d(s,t)}{2} \right\rceil$

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.

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
    $d(u, v) = 2d'(u, v) - 1$

Note that in each recursive call, the diameter of the graph decreases by 2, and that after $\log n$ iterations, A will be the all 1s matrix with 0s along the diagonal, which we can detect. Thus, if we can find a way to determine the parity of a u, v-path efficiently, we should 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 1. *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 2. *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 $d(i,k') < d(i,j)$. □

Claim 3. *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.

Claim 4. *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?

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
```

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).

The goal of this week is to show the following Theorem of Zwick:

Theorem 0.1. *All-Pairs Shortest Paths (APSP) on unweighted directed graphs can be solved in $\widetilde{O}(n^{2+1/(4-\omega)})$ time, where ω is the matrix multiplication exponent.*

Zwick's algorithm computes distances between close nodes (nodes u, v with $d(u, v) < P$) and far-away nodes (nodes (u, v) with $d(u, v) \geq P$) separately.

Both cases, however, take advantage of the Hitting Set Lemma, which can be proven using standard probability arguments.

Lemma 0.1. *Suppose we have poly(n) sets $S_1, \ldots, S_k$ of $\{1, \ldots, n\}$, each of size $\geq L$. Then a random sample $S \subseteq \{1, \ldots, n\}$ with $|S| = c(n/L \lg n)$ for a sufficiently large constant c hits all S_i in at least one element with high probability.*

We will start by addressing nodes of distance P or more away from each other.

Proposition 1. *Let $G = (V, E)$ be an unweighted directed graph. Fix a parameter P. In $\widetilde{O}(n^3/P)$ time, one can compute $d(u, v)$ for every u, v with $d(u, v) \geq P$.*

Proof. For every pair u, v of nodes at least P apart in V, let $S_{u,v}$ be a set containing the nodes in some shortest path from u to v. Pick a hitting set S of size $\Theta(n/P \lg n)$ so that S hits every $S_{u,v}$ (with high probability).

For each $s \in S$, compute $d(s, v)$ for all $v \in V$ using breadth-first-search in $O(n^2)$ time. Similarly, for each $s \in S$, also compute $d(v, s)$ for all $v \in V$ using breadth-first-search (on G with its edges inverted) in $O(n^2)$ time.

Since S hits each $S_{u,v}$ for every u, v with $d(u, v) \geq P$, it follows that for some $s \in S$ we have $d(u, v) = d(u, s) + d(s, v)$. Thus in $O(n^2|S|)$ time we can compute

$$d'(u, v) = \min_{s \in S} d(u, s) + d(s, v),$$

which is the correct distance for all u, v at least P apart. □

In order to handle shortest paths of length less than P, we introduce the *distance product*.

Definition 0.1. *Let A, B be $n \times n$ matrices. Define the distance product by*

$$(A \star B)[i, j] = \min_k \{A(i, k) + B(k, j)\}.$$

Although we will not prove it, a theorem of Fisher, Mayer, et al. states that if $A \star B$ can be computed in $T(n)$ time, then APSP in weighted graphs can be done in $O(T(n))$ time, and vice-versa.

It turns out that distance products can be computed relatively quickly.

Theorem 0.2. *if A, B are $n \times n$ matrices with entries in $\{-M, \ldots, M\}$, then $A \star B$ can be computed in $\widetilde{O}(Mn^\omega)$ time.*

Proof. Define matrices A' and B' with entries

$$A'[i, j] = (n + 1)^{M - A(i,j)}$$
$$B'[i, j] = (n + 1)^{M - B(i,j)}$$

Computing the integer product of A' and B' we obtain C' with entries

$$C'[i,j] = \sum_k (n+1)^{2M-(A(i,k)+B(k,j))}.$$

For a given i, j, we can then compute $(A \star B)[i,j] = \min_k A(i,k) + B(k,j)$, which we will refer to as L, as follows.

Observe that $(n+1)^{2M-L} \leq C'[i,j]$ because $(n+1)^{2M-L}$ is a summand in $C'[i,j]$. At the same time, $C'[i,j] \leq (n+1)^{2M-L} \cdot n$ because $(n+1)^{2M-L}$ is the largest summand in $C'[i,j]$ and $C'[i,j]$ has only n summands. Therefore, we can set L to be the smallest integer such that $C'[i,j] \geq (n+1)^{2M-L}$.

Note that we are dealing with integers having $O(M \lg n)$ bits in C', for which operations take $\widetilde{\mathrm{O}}(M)$ time. Bearing this to mind, it is straightforward to see that the above method computes $A \star B$ in $\widetilde{\mathrm{O}}(Mn^\omega)$ time. □

By combining fast computations of distance products with the idea of a hitting set, we can now obtain a fast algorithm for computing distances between close-together nodes.

Proposition 2. *Let $G = (V, E)$ be an unweighted directed graph, and P be a fixed parameter. We can compute $d(u,v)$ for pairs of nodes less than P apart in time*

$$\widetilde{O}\left(n^\omega P^{3-\omega}\right).$$

Proof. We will have $\lceil \lg_{3/2} P \rceil$ stages. Let V_j be the set of pairs of vertices (u,v) such that $d(u,v) \in [(3/2)^{j-1}, (3/2)^j)$, and let $V_{\leq j}$ denote $\cup_{i=1}^{j} V_i$. In stage j, we will compute every $d(u,v)$ for every $(u,v) \in V_j$. More specifically, we will compute a matrix D_j such that for all $(x,y) \in V_{\leq j}$, $D_j[x,y] = d(x,y)$; and $D_j[x,y] = \inf$ for all $(x,y) \notin V_{\leq j}$. Note that D_1 can easily be obtained from the adjacency matrix of G.

One could easily obtain a valid D_j from D_{j-1} by simply computing and cleaning up $D_{j-1} \star D_{j-1}$. However, we cannot afford to compute $n \times n$ matrix distance products. Instead, we will take advantage of hitting sets.

For every $(u,v) \in V_j$, consider a shortest path $P_{u,v}$ from u to v. The *middle third* of $P_{u,v}$ is a set of $\lfloor (3/2)^{j-1} \rfloor$ nodes appearing consecutively in $P_{u,v}$ such that at most $(3/2)^{j-1}$ nodes precede them, and at most $(3/2)^{j-1}$ follow them.

At stage j, take a random $S_j \subseteq V$ with $|S_j| \in \Theta(\frac{n}{(3/2)^{j-1}} \lg n)$ so that for all $(u,v) \in V_j$ with high probability V hits a node $s_{u,v}$ in the middle third of $P_{u,v}$. Observe that because $s_{u,v}$ is in the middle third of $P_{u,v}$, we get that $(u, s_{u,v}), (s_{u,v}, v) \in D_{\leq j-1}$.

It follows that with high probability, for all $(u,v) \in V_j$,

$$d(u,v) = \min_{s \in S_j} D_j[u,s] + D_j[s,v].$$

Thus we can compute $D_j(u,v)$ to be

$$\min(D_{j-1}[u,v], \min_{s \in S_j} D_j[u,s] + D_j[s,v]).$$

This is easy to do in n^2 time once we have already computed each $\min_{s \in S} D_j[u,s] + D_j[s,v]$, which can be obtained by computing the product $X \star Y$ where X contains the columns in D_{j-1} corresponding with elements of S_j, and Y contains the rows in D_{j-1} corresponding with elements of S_j. In other words, by selecting a hitting set S_j, we are able to use the distance product of matrices much smaller than D_{j-1} in order to compute D_j.

Breaking X and Y into square (or smaller) blocks of side-length approximately $(3/2)^j$, we can use the distance products of all $(3/2)^{2j}$ pairs of blocks to easily recover $X \star Y$. By theorem 0.2, since D_j has entries in $\{0, \ldots, (3/2)^j\}$ (as well as entries with value inf which Theorem 0.2 can easily be adapted to handle) this takes time

$$\widetilde{\mathrm{O}}\left((3/2)^{2j}\left(\frac{n}{(3/2)^j}\right)^\omega (3/2)^j\right) = \widetilde{\mathrm{O}}\left(n^\omega ((3/2)^{3-\omega})^j\right).$$

Summing over the $\lceil \lg_{3/2} P \rceil$ stages, we get time

$$\widetilde{\mathrm{O}}\left(n^{\omega} \sum_{j:(3/2)^j < P} ((3/2)^j)^{3-\omega}\right) \leq \widetilde{\mathrm{O}}\left(n^{\omega} P^{3-\omega}\right).$$

□

We are now in a position to complete the proof of Zwick's Theorem. Indeed, combining Proposition 1 and Proposition 2 and optimizing for P (at $P = n^{(3-\omega)/(4-\omega)}$), we get total runtime of $\widetilde{\mathrm{O}}(n^{2+1/(4-\omega)})$. Observe that both the algorithm from Proposition 1 and from Proposition 2 compute either correct distances or overestimates for distances between pairs of nodes (that are correct for far nodes); thus minimizing the outputted distances of the two, one can obtain the exact $d(u, v)$ for all $u, v \in V$.

1 The Distance Product

Last time we defined the distance product of $n \times n$ matrices:

$$(A \star B)[i,j] = \min_k \{A(i,k) + B(k,j)\}$$

Theorem 1.1. *Given two $n \times n$ matrices A, B over $\{-M, M\}$, $A \star B$ can be computed in $\tilde{O}(Mn^\omega)$ time.*

2 Oracle for All-Pairs Shortest Paths

Theorem 2.1 (Yuster, Zwick '05)**.** *Let G be a directed graph with edge weights in $\{-M, M\}$ and no negative cycles. Then in $\tilde{O}(Mn^\omega)$ time, we can compute an $n \times n$ matrix D such that for every $u, v \in V$,* ***with high probability****:*

$$(D \star D)[u,v] = d(u,v)$$

It is called distance product because the shortest path from node u to node v can be created by first going on the shortest path from u to an intermediate node k and then from k to v. We are searching for the most convenient node k that will minimize the sum of the two.

Note that this does not immediately imply a fast APSP algorithm, because D may have large entries, making computing $D \star D$ expensive. However, for *single source shortest paths* we have the following corollary:

Corollary 2.1. *Let $G = (V, E)$ be a directed graph with edge weights in $\{-M, M\}$ and no negative cycles. Let $s \in V$. Then single-source shortest path from s can be computed in $\tilde{O}(Mn^\omega)$ time.*

Proof. By Theorem 2.1, we can compute an $n \times n$ matrix D such that $D \star D$ is the correct all-pairs shortest-paths matrix, in $\tilde{O}(Mn^\omega)$ time.

Then for all $v \in V$, we know that:

$$d(s,v) = \min_k D[s,k] + D[k,v]$$

Computing this for all $v \in V$ only takes $O(n^2)$ time. Since $\omega \geq 2$, this entire computation is in $\tilde{O}(Mn^\omega)$ time. □

Similarly, we can show that detecting negative cycles is fast since any negative cycle contains a simple cycle of negative weight, and thus corresponds to a path from i to i for some i of length $\leq n$.

Corollary 2.2. *Let G be a directed graph with edge weights in $\{-M, M\}$. Then negative cycle detection can be computed in $\tilde{O}(Mn^\omega)$ time.*

Note: For notational convenience, suppose that A is an $n \times n$ matrix and that $S, T \subseteq \{1, \ldots, n\}$. Then $A[S,T]$ is the submatrix of A consisting of rows indexed by S and columns indexed by T.

We now prove our main theorem:

The main algorithm uses again randomness and the hitting set lemma but now we do not take freshly random samples every time, but instead we take B_{j+1} to be a random sample from B_j.

Proof of Theorem 2.1. Let $\ell(u,v)$ be the number of nodes on a shortest u to v path.

Algorithm 1: YZ(A)

A is a weighted adjacency matrix;
Set $D \leftarrow A$;
Set $B_0 \leftarrow V$;
for $j = 1, \ldots, \log_{3/2} n$ **do**
  Let D' be D but with all entries larger than $M(3/2)^j$ replaced by ∞;
  Choose B_j to be a random set of B_{j-1} of size $S_j = \frac{c \cdot n}{(3/2)^j} \log n$;
  Compute $D_j \leftarrow D'[V, B_{j-1}] \star D'[B_{j-1}, B_j]$;
  Compute $\overline{D}_j \leftarrow D'[B_j, B_{j-1}] \star D'[B_{j-1}, V]$;
  foreach $u \in V, b \in B_j$ **do**
    Set $D[u,b] = \min(D[u,b], D_j[u,b])$;
    Set $D[b,u] = \min(D[b,u], \overline{D}_j[b,u])$;
return D;

We claim that Algorithm 1 is our desired algorithm. (desired running time and correctness)

Running Time: In iteration j, we multiply an $n \times \tilde{O}\left(\frac{n}{(3/2)^{j-1}}\right)$ matrix by a $\tilde{O}\left(\frac{n}{(3/2)^{j-1}}\right) \times \tilde{O}\left(\frac{n}{(3/2)^j}\right)$ matrix, where all entries are at most $(3/2)^j M$ (we will show iteration j only needs to consider paths with at most $(3/2)^j$ nodes).

Hence the runtime for iteration j is $\tilde{O}\left(M(3/2)^j (3/2)^j (\frac{n}{(3/2)^j})^\omega\right) = \tilde{O}\left(\frac{Mn^\omega}{(3/2)^{j(\omega-2)}}\right)$. The term $((3/2)^j (\frac{n}{(3/2)^j})^\omega)$ is due to the blocking that we use when computing D_j and $\overline{D}_j$. Over all iterations, the running time is, asymptotically, ignoring polylog factors,

$$Mn^\omega \sum_j ((3/2)^{\omega-2})^j \leq \tilde{O}(Mn^\omega).$$

If $\omega > 2$, one of the log factors in the $\tilde{O}$ can be omitted.

Correctness: We will prove the correctness by proving two claims.

Claim 1: For all $j = 0, \ldots, \log_{3/2} n$, $v \in V$, $b \in B_j$, if $\ell(v,b) < (3/2)^j$ then w.h.p. after iteration j, $D[v,b] = d(v,b)$

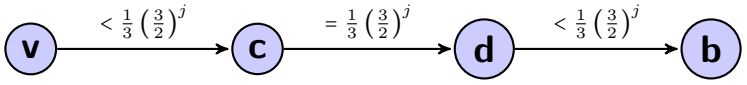
Proof of Claim 1: We will prove it via induction. The base case ($j = 0, \ell(v,b) < (3/2)^0 = 1$) is trivial, since the distance is for one-hop paths is exactly the adjacency matrix. Now, assume the inductive hypothesis is true for $j-1$, that is we have stored correctly $D[u,b] = d[v,b]$ if the shortest path (v,b) has length $\ell(v,b) < (3/2)^{j-1}$. We will show correctness for j. Consider some $v \in V$ and $b \in B_j$. We consider two possible cases depending on how far is node b from v.

Case I: $\ell(v,b) < (3/2)^{j-1}$ (b is near)

But then $b \in B_j \subset B_{j-1}$. By our inductive hypothesis, $D[v,b] = d(v,b)$ w.h.p.!

Case II: $\ell(v,b) \in [(3/2)^{j-1}, (3/2)^j)$ (b is far)

We will need to use our "middle third" technique we saw from last lecture.



We can choose $c, d \in V$ such that:

$$\ell(v,c) < \frac{1}{3}\left(\frac{3}{2}\right)^j$$
$$\ell(d,b) < \frac{1}{3}\left(\frac{3}{2}\right)^j$$
$$\ell(c,d) = \frac{1}{3}\left(\frac{3}{2}\right)^j < \left(\frac{3}{2}\right)^{j-1}$$

By a hitting set argument, if c is a large enough constant, $B_{j-1} \cap$ "middle third" $\neq \varnothing$ (w.h.p. depending on c) since $|B_{j-1}| = c\frac{n}{(3/2)^{j-1}} \log n$.

Let x in $B_{j-1} \cap$ "middle third". Then $\ell(v,x) \leq \ell(v,c) + \ell(c,d) < \frac{2}{3}(\frac{3}{2})^j = (\frac{3}{2})^{j-1}$. Since $x \in B_{j-1}$, by induction $D[v,x] = d(v,x)$ w.h.p. at iteration j. By a similar argument we get that w.h.p. $D[x,b] = d(x,b)$ at iteration j (at the beginning of iteration j).

Hence after this iteration, $D[v,b] \leq D[v,x] + D[x,b] = d(v,b)$.

As a small technical note, we will need to actually remove entries larger than $(3/2)^j M$ from D before multiplying, but they are not needed.

Claim 2: For all $u, v \in V$, w.h.p. $(D \star D)[u,v] = d(u,v)$.

Proof of Claim 2: Fix $u, v \in V$, and let j be such that $\ell(u,v) \in [(3/2)^{j-1}, (3/2)^j)$. Look at a shortest path between u and v. Its middle third hence has a length of $(1/3)(3/2)^j$.

But then w.h.p. B_j hits this path at some $x \in V$ such that $\ell(u,x), \ell(x,v) < (3/2)^{j-1}$. By Claim 1, $D(u,x) = d(u,x)$ and $D(x,b) = d(x,b)$. Hence:

$$d(u,v) \leq (D \star D)[u,v] \leq min_{x \in B_{j-1}} D(u,x) + D(x,v) \leq d(u,v)$$

This completes the proof.

□

3 Node-Weighted All-Pairs Shortest Paths

Now we will see an interesting variant of the APSP which is called node-weighted all pairs shortest paths problem (now the weights are associated with the nodes instead of the edges) for which we can have a truly subcubic solution despite the fact that for APSP no known truly subcubic solution exists. This gap might be inherent since it is difficult to map $\approx n^2$ weight values to only n and still maintain the pairwise shortest paths information.

Here we prove a theorem by Chan [1]

Theorem 3.1. *APSP with node weights can be computed in $O(n^{\frac{9+\omega}{4}})$ or $O(n^{2.84})$ time.*

The idea is to compute long paths ($> s$ hops) via a hitting set argument and running multiple calls to Dijkstra's algorithm, in a running time of $\tilde{O}(\frac{n^3}{s})$. Then, handle short paths ($\leq s$ hops) in $O(sn^{\frac{3+\omega}{2}})$ time via a specialized matrix multiplication.

Let G be a directed graph with node weights $w : V \to Z$. Suppose we just wanted to compute distances over paths of length two.

Let A be the *unweighted* adjacency matrix. Notice that $d_2(u,v) = w(u) + w(v) + \min\{w(j) \mid A[u,j] = A[j,v] = 1\}$ (we are looking for our cheapest neighbour to go through).

Suppose we made two copies of A, and sorted one's columns by $w(j)$ in nondecreasing order, and the others rows by $w(j)$ in nondecreasing order.

Then it would suffice to compute $\min\{j \mid A[i,j] = A[j,k] = 1\}$, or the "minimum witnesses" matrix product. We use an algorithm provided by Kowaluk and Lingas [3]:

Lemma 3.1 (Kowaluk, Lingas '05)**.** *Minimum witnesses of A, B ($n \times n$ matrices) is in $O(n^{2.616})$ or $O(n^{2+\frac{1}{4-\omega}})$ time.*

Note that this algorithm has been improved on by Czumaj, Kowaluk, and Lingas [2]

Proof. Let p be some parameter that we will choose later. Bucket A by columns into buckets of size p. Bucket B by rows into buckets of size p.

For every bucket $b \in \{1, \ldots, \frac{n}{p}\}$, compute $A_b \cdot B_b$ (boolean matrix product). This takes $O((\frac{n}{p})^2 p^\omega)$ time each, or $O(n^2 p^{\omega-2})$ time each. But there are $\frac{n}{p}$ of these, so this takes $O(\frac{n^3}{p^{3-\omega}})$ time total.

Then for all $i, j \in \{1, \ldots, n\}$, do the following. Let b_{ij} be the smallest b such that $(A_b \cdot B_b)[i,j] = 1$. Hence we can just try all the choices of k in bucket b_{ij}, and return the smallest k such that $A_b[i,k]B_b[k,j] = 1$. This is just n^2 exhaustive searches, so this step runs in $O(n^2 p)$ time. The intuition for this is that after the sorting, the minimum witness k for which we have $A_b[i,k]B_b[k,j] = 1$ is the most convenient neighbour of both i and j to go through if we are seeking for the shortest path between i, j.

Setting the above running times equal ($\frac{n^3}{p^{3-\omega}} = n^2 p$) and balancing, we get that we should set $p = n^{\frac{1}{4-\omega}}$ to make the overall time $O(n^{2+\frac{1}{4-\omega}})$. □

Intuition: The blocking we do to our matrices after the sorting has the following interpretation: On the sorted adjacency matrix, when we do blocking is like grouping together the nodes according to their weight values. So if we had 2 blocks then we would have 2 node groups: cheap nodes, expensive nodes. Then by the multiplication process we are trying to figure out which nodes we can reach passing through the different node groups.

Now that we saw how to deal with distance 2 we can proceed with longer paths. How can we compute distances for paths that are longer than two hops?

We will have two parameters p, s that we will choose later so that we minimise the runtime. Parameter s is for distinguishing the "short" paths from the "long" paths. Parameter p is again related to the blocking of the matrices that is convenient. For each $\ell \leq s$, we want to compute D_ℓ such that:

$$D_\ell[u,v] = d(u,v) - w(u) - w(v) \text{ if } \ell(u,v) = \ell$$
$$D_\ell[u,v] = \min_{j \in N(u)} \{w(j) + D_{\ell-1}[j,v]\}$$

This gives rise to a new matrix product! Suppose we are given $D_{\ell-1}$. Let $\overline{D}_{\ell-1}[u,v] = w(u) + D_{\ell-1}[u,v]$. Then we are interested in $(A \odot \overline{D}_{\ell-1})[u,v] = \min\{\overline{D}_{\ell-1}[j,v] \mid A[u,j] = 1\}$.

We can compute this product as follows. Again, let p be a parameter that we will choose later. Sort the columns of $\overline{D}_{\ell-1}$, using $O(n^2 \log n)$ time. Then partition each column into blocks of length p.

Let $D_b[u,v] = 1$ if $\overline{D}_{\ell-1}[u,v]$ is between the $\left(b\frac{n}{p}\right)^{th}$ and the $\left((b+1)\frac{n}{p}\right)^{th}$ element of column v.

Compute the boolean matrix product of A and D_b for all b. Notice that $(A \cdot D_b)[u,v] = 1$ iff there exists an x such that $A[u,x] = 1$ and $\overline{D}_{\ell-1}[x,v]$ is among the b^{th} block of p elements in the sorted order of the v^{th} column. We can finish via an exhaustive search, trying all j such that $D_{\ell-1}[j,v]$ is in the b^{th} block of column v.

This takes $O(\frac{n}{p} n^\omega)$ time for multiplications, and $O(n^2 p)$ time for the exhaustive search. This yields $O(n^{\frac{3+\omega}{2}})$ time after balancing. However, we need to do this s times.

The overall runtime is hence $O(n^{\frac{3+\omega}{2}} s + n^3/s)$, which becomes $O(n^{\frac{9+\omega}{4}})$ time after balancing.

Today we will present and solve two variants of All Pairs Shortest Paths (APSP) in $O(n^{3-\delta})$ time for some constant $\delta > 0$. In doing so, we will also introduce two more matrix products, namely the $(\min, \leq)$ product and the dominance product.

4 Earliest Arrivals

The first variant of APSP we will study is the Earliest Arrivals problem. We are given a set V consisting of n airports and a set F of n flights. Each flight $f \in F$ consists of a a source airport $s \in V$, a destination airport $t \in V$, a departure time, and an arrival time.

Definition 4.1. *A valid itinerary from s to t is a sequence of flights $f_1, \dots f_k$ such that, for all $i \in \{1, \dots, k\}$, $source(f_{i+1}) = destination(f_i)$ and $departure(f_{i+1}) \geq arrival(f_i)$.*

The Earliest Arrivals problem is to compute, for all airports $u, v \in V$, the earliest arrival time over all valid itineraries. This problem has a natural graph interpretation. Consider a bipartite graph $G = (V \cup F, E)$. For each flight $f \in F$, we add a directed edge $(source(f), f)$ to E with weight $departure(f)$. Then, we add another directed edge $(f, destination(f))$ with weight $arrival(f)$.

On this graph, a valid itinerary is a $s \to t$ path such that all of the edges form a nondecreasing sequence, and the arrival time is given by the last edge weight. Therefore, Earliest Arrivals is equivalent to finding, $\forall s, t \in V$, the minimum last edge weight over all nondecreasing $s \to t$ paths.

Definition 4.2. *Let A, B be $n \times n$ matrices. The $(\min, \leq)$ product of A and B, denoted $A \circledcirc B$ is given by*

$$(A \circledcirc B)(i, j) = \min_k \{B(k, j) \mid A(i, k) \leq B(k, j)\}$$

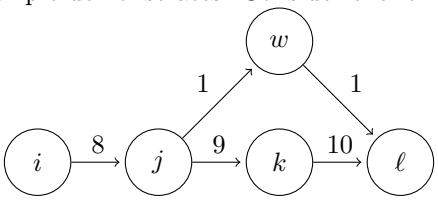
or ∞ if no such k exists.

If we define the adjacency matrix A of G in the natural way,

$$A(i, j) = \begin{cases} w(i, j) & (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$$

(here we assume that all weights are positive) then we find that $(A \circledcirc A)(i, j)$ is the minimum last edge weight over paths of length 2. Iterating this relationship, we find that $\underbrace{(A \circledcirc \cdots \circledcirc A)}_{\ell - 1 \text{ times}} \circledcirc A(i, j)$ is the minimum last edge over all paths of length ℓ.

We must be careful, however, because the $(\min, \leq)$ product is not associative in general, as the following example demonstrates. Consider the following graph.



Observe that $(A \circledcirc A) \circledcirc A(i, \ell) = 10$, but $A \circledcirc (A \circledcirc A)(i, \ell) = \infty$. Consequently, we cannot simply use successive squaring to solve the Earliest Arrivals problem. Instead, our approach will be to compute Earliest Arrivals for "short paths" and use the random sampling technique developed in previous lectures to handle "long paths." The rough idea is as follows.

Algorithm 2: Earliest Arrivals(G)

Form adjacency matrix A
Set $D = A$
for $i := 1$ **to** s **do**
  Compute $D = D \otimes A$
end for
Compute a random sample, S, of size $c * \frac{n}{s} \log n$
for all $x \in S$ **do**
  Compute All Pairs Earliest Arrivals for paths through x
end for
for all $i, j \in V$ **do**
  $EA(i,j) = \min_{x \in S}$ min last edge weight over valid itineraries of the form $i \to x \to j$
  $EA(i,j) = \min\{EA(i,j), D(i,j)\}$
end for
Return EA

It is left as a homework exercise to show that, for any node $x \in S$ we can compute all pairs earliest arrival for paths through x in $O(n^2 \log n)$ time.

Lemma 4.1. *If the* $(\min, \leq)$ *product of* $n \times n$ *matrices can be computed in* $O(n^c)$ *time, then we can solve Earliest Arrivals in* $O(n^{\frac{3+c}{2}})$ *time.*

Proof of Lemma 4.1. Using the algorithm sketched above, we obtain a runtime of $O(\frac{n^3}{s} + s(n^c))$. Optimizing over s, we set $s = n^{\frac{3-c}{2}}$ and obtain a total runtime of $O(n^{\frac{3+c}{2}})$, as required. □

5 All Pairs Bottleneck Paths

(we skipped that in class but you should still study it)

Let graph $G = (V, E)$ be a graph with edge weights given by $w : E \to \mathbb{Z}$.

Definition 5.1. *Given a path* p *in* G*, its bottleneck edge is the edge of minimum weight.*

Definition 5.2. *The All Pairs Bottleneck Paths problem (APBP) is to find, for all pairs* $u, v \in V$*, the maximum bottleneck weight over all* $u \to v$ *paths.*

In order to tackle this problem, we need to define another matrix product.

Definition 5.3. *Let* A *and* B *be* $n \times n$ *matrices. The* $(\max, \min)$ *product of* A *and* B*, denoted* $A \circledast B$ *is given by*

$$(A \circledast B)(i,j) = \max_k \min(A(i,k), B(k,j))$$

Observe that that $(\max, \min)$ product is precisely the bottleneck path problem in graphs with diameter 2. It is left as an exercise to verify that $\circledast$ is associative. Since $\circledast$ is associative and $A \circledast A$ gives the maximum bottleneck for length 2 paths, we can solve All Pairs Bottleneck Paths using successive squaring. This gives us the following lemma.

Lemma 5.1. *If the* $(\max, \min)$ *product of two* $n \times n$ *matrices can be computed in* $\tilde{O}(n^c)$ *time, then we can solve All Pairs Bottleneck Paths in* $\tilde{O}(n^c)$ *time.*

In fact, we will show that computing $\circledast$ is equivalent to two $\otimes$ product computations.

Lemma 5.2. *If there is an* $O(n^c)$ *algorithm for computing* $(\min, \leq)$ *products, there is an* $O(n^c)$ *algorithm for computing* $(\max, \min)$ *products.*

Proof. Consider the matrix product defined by $(A \circledcirc B)(i,j) = \max_k\{A(i,k) \mid A(i,k) \le B(i,k)\}$. Note that this product is in fact a $(\min, \le)$ product. In particular, it is the product $-B \circledcirc -A$ obtained by negating all of the entries $a_{i,j}$ in A and $b_{i,j}$ in B and then swapping matrices A and B, i.e. $(A \circledcirc B)(i,j) = -(-B \circledcirc -A)(i,j)$. Using this product, we can compute

$$(A \circledast B)(i,j) = \max\{(A \circledcirc B)(i,j), (B \circledcirc A)(i,j)\}.$$

Therefore, we can compute $A \circledast B$ using two $(\min, \le)$ computations, as required. □

By the above discussion, we can solve both the All Pairs Earliest Arrivals problem and the All Pairs Bottleneck Path problem with a fast algorithm for computing $(\min, \le)$ products. The rest of this writeup is dedicated to finding such an algorithm.

6 A Fast Algorithm for Computing $(\min, \le)$ Products

We will use another special matrix product in our algorithm for computing $(\min, \le)$.

Definition 6.1. *The dominance product of $n \times n$ matrices A and B, denoted $A \odot B$, is given by*

$$(A \odot B)(i,j) = |\{k \mid A(i,k) \le B(k,j)\}|$$

Theorem 6.1. *(Matousek) The dominance product of two $n \times n$ matrices can be computed in $O(n^{\frac{3+\omega}{2}})$ time.*

Theorem 6.2. *If dominance product can be computed in $O(n^d)$ time, then the $(\min, \le)$ product can be computed in $O(n^{\frac{3+d}{2}})$ time.*

Assuming 6.1, we first prove 6.2.

Proof of Theorem 6.2. Let A, B be two $n \times n$ matrices. We will compute $A \circledcirc B$ as follows.

1. Sort each column j of matrix B
2. Fix parameter p. Partition each sorted column into $\frac{n}{p}$ consecutive buckets of p elements each. Name the buckets so that for all buckets $b \le b'$, $\forall B(i,j)$ in bucket b of column j, and $\forall B(\ell,j)$ in bucket b' of j, we have $B(i,j) \le B(\ell,j)$.
3. For each $b \in \{1, \ldots, \frac{n}{p}\}$, create an $n \times n$ matrix B_b such that
$$B_b(i,j) = \begin{cases} B(i,j) & \text{if } B(i,j) \text{ in bucket } b \text{ of column } j \\ -\infty & \text{otherwise} \end{cases}$$
4. Compute for all buckets b, $A \odot B_b$, which is
$$(A \odot B_b)(i,j) = \begin{cases} \neq 0 & \text{if } \exists k \text{ such that } B_b(k,j) \neq -\infty \text{ and } A(i,k) \le B(k,j) \\ 0 & \text{otherwise} \end{cases}$$
5. For all i,j determine $b_{i,j}$ = smallest b such that $(A \odot B_b)(i,j) \neq 0$. This is equivalent to
$$\min\{B[k,j] \mid B(k,j) \text{ in bucket } b(i,j) \text{ and } A(i,k) \le B(k,j)\}.$$
Therefore, we can use brute force, as follows. For all i,j examine each $B(k,j)$ in bucket $b_{i,j}$ of j, compare it with $A(i,k)$ and output the minimum $B(k,j)$ for which $A(i,k) \le B(k,j)$. Observe that this is $(A \circledcirc B)(i,j)$.

The running time of this algorithm is dominated by computing the dominance product in step 4 and brute force in step 5. Using 6.1, we can compute dominance product in $O(n^d)$ time. Therefore, it takes $O(\frac{n^{d+1}}{p})$ time to compute the required $\frac{n}{p}$ dominance products. The brute force step takes $O(n^2 p)$ time. Choosing $p = n^{\frac{d-1}{2}}$, we obtain a total runtime of $O(n^{\frac{3+d}{2}})$, as desired. □

It remains to prove 6.1

Proof of Theorem 6.1. Let A, B be $n \times n$ matrices. We compute $A \odot B$ as follows.

1. For all j, sort the set of entries of column j of A and row j of B together. This produces a list of $2n$ elements.
2. Partition this list into buckets of p elements each. There are $\frac{2n}{p}$ buckets for each j.
3. For all $b \in \{1, \ldots, \frac{2n}{p}\}$, create $n \times n$ matrices

$$A_b(i,j) = \begin{cases} 1 & \text{if } A(i,j) \text{ is in bucket } b \text{ of } j \\ 0 & \text{otherwise} \end{cases}$$

$$B_b(j,k) = \begin{cases} 1 & \text{if } \exists b' > b \text{ such that } B(j,k) \text{ is in bucket } b' \text{ of } j \\ 0 & \text{otherwise} \end{cases}$$

4. For distinct buckets b, compute the integer matrix product

$$(A_b B_b)(i,j) = |\{k \mid A(i,k) \in b, A(i,k) \leq B(k,j), \text{ and } B(k,j) \notin b\}$$

   We handle identical buckets b using brute force search. For all i, j and buckets b, compare $A(i,k)$ with all $B(k,j)$ in the same bucket as $A(i,k)$ and update the sum in the output.

The brute force step requires $O(n^2 p)$ time. Then, we require $\frac{n}{p} n^\omega$ time to perform matrix multiplications. We minimize p by taking $p = n^{\frac{3-\omega}{2}}$ to obtain a final running time of $O(n^{\frac{3+\omega}{2}})$, as desired. □

7 Conclusions

In this lecture we saw many different matrix products that are useful for many applications. The distance product is useful for APSP oracles and a slight variant of it is useful in the Node-Weighted APSP problem (NW-APSP). The $(\min, \leq)$ product (which is not associative) is useful when searching for non-decreasing paths and has applications in the All Pairs Earliest Arrivals problem (APEA). The (max,min) product is used when searching for All Pairs Bottleneck Paths (APBP). Finally, we defined the dominance product. Using all these, we concluded that NW-APSP, APEA, APBP are truly subcubic, having $O(n^{2.84})$,$O(n^{2.9})$,$O(n^{2.8})$ time algorithms respectively, if we use the current value of ω. However, it remains a major open question to find a truly subcubic algorithm for general APSP.

References

[1] T.M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. *SIAM J. Comput.*, 39(5):20752089, 2010.

[2] Artur Czumaj, Miroslaw Kowaluk, and Andrzej Lingas. Faster algorithms for finding lowest common ancestors in directed acyclic graphs. *Theoretical Computer Science*, 380(1):3746, 2007.

[3] Miroslaw Kowaluk and Andrzej Lingas. Lca queries in directed acyclic graphs. In *Automata, Languages and Programming*, pages 241248. Springer, 2005.

[4] Raphael Yuster and Uri Zwick. Answering distance queries in directed graphs using fast matrix multiplication. In *FOCS*, pages 389396, 2005.

CS367 Lecture 8 Perfect Matching
Scribe: Jaehyun Park **Date:** April 28, 2014
Edited by: Ilan Goodman **Updated:** October 16, 2015

1 Successor Matrix for Shortest Paths

So far, we studied how to compute the shortest path distances under various matrix products, but never showed how one can compute the paths corresponding to these shortest distances. These notes will focus on how to find the actual paths, by modifying the algorithms we previously studied.

The only type of matrix product we will use throughout the notes is the $(\min, \odot)$ product, defined as $(A \odot B)[i,j] = \min_k(A[i,k] \odot B[k,j])$ for some operation $\odot : \mathbb{Z} \to \mathbb{Z}$. We will only consider operations $\odot$ that makes this matrix product associative. Note that many problems we considered in previous lectures, such as Boolean matrix multiplication or all-pairs shortest paths on directed, weighted graphs, fall under this framework. Also, in these notes, we will use the notation $i \rightsquigarrow j$ to denote a path from i to j, and whether it represents the shortest path or not should be clear from the context.

To motivate the discussion, suppose that we want to compute shortest paths between *all* pairs of nodes. However, in general, the size of the output can be $\Omega(n^3)$, and as a result, many related problems become uninteresting. Thus, we will look for a *successor matrix* instead, which essentially has the same representational power as the list of all shortest paths, but using only $\tilde{O}(n^2)$ memory.

Definition A *successor matrix* is an $n \times n$ matrix S, where we define $S[i,j] = k$ such that k is the next node after i on the $i \rightsquigarrow j$ shortest path. For the degenerate case $i = j$, then $S[i,i] = i$.

This definition immediately gives a procedure to retrieve a shortest path between any given pair of nodes using the successor matrix, assuming that the path is simple[1]; start from a node, proceed to the next one that is given by the successor matrix, and repeat. This shows that finding a successor matrix is basically as useful as having the entire list of shortest paths.

Proposition 1.1. *Given S, for any i, j, we can output a shortest $i \rightsquigarrow j$ path in time linear in the number of edges of that path.*

Proof. Look at the appropriate entries in the matrix and keep following until we find the destination node. □

2 Witness Matrix

Definition A *witness matrix* for the boolean matrix product of A and B is a matrix W such that $\forall i, j$, $W[i,j] =$ some k such that $A[i,k] = B[k,j] = 1$ if such a k exists, or $= \infty$ otherwise.

Remark A witness matrix is not unique necessarily: we only need to compute one such k for each i, j, but many may exist.

Let's assume that one can compute a witness matrix for the Boolean product of two $n \times n$ matrices in $\tilde{O}(n^\omega)$ time. We will show how to compute successor matrices for:

1. Transitive closure
2. Zwick's algorithm
3. Seidel's algorithm

[1]The shortest paths problems we consider all have the property that there is always a simple shortest path.

2.1 Transitive Closure

Recall we can compute transitive closure by successive squarings of the adjacency matrix (under the Boolean product). Boolean matrix multiplication (BMM) can actually be viewed as $(\max, \times)$ restricted to $\{0, 1\}$ entries, where $\times$ is the ordinary multiplication; by negating the entries of one of the matrices, BMM can also be represented as a $(\min, \odot)$ product.

If there is a path from i to j, then the (i, j) entry of the successor matrix S should give the node next to i in some simple path $i \rightsquigarrow j$. If there is no path, then $S[i, j]$ can be arbitrary (this is consistent with the definition of successor matrices). Then we can use the following algorithm:

Algorithm 1: $\mathsf{TC}(A)$

$A^0 \leftarrow A$
$\forall i, j,\ S^0[i, j] \leftarrow j$ if $(i, j) \in E$
foreach $k \in \{1, \ldots, \log n\}$ **do**
  $A^{(k)} \leftarrow A^{(k-1)} \cdot A^{(k-1)}$
  $W^{(k)} \leftarrow$ witness matrix of above product
  $\forall i, j,\ S^{(k)} \leftarrow S^{(k-1)}[i, W^{(k)}[i, j]]$
return $A^{(\log n)}, S^{(\log n)}$

In this algorithm, $A^{(k)}[i, j] = 1$ iff there's an $i \rightsquigarrow j$ path of length $\leq 2^k$. Algorithm 1 returns the transitive closure T, along with the successor matrix S. To see its correctness, we argue that $S^{(k)}$ is a correct successor matrix for all pairs of nodes i, j such that j is reachable from i via a path of length $\leq 2^k$. If there is a path $i \rightsquigarrow j$ of length at most 2^k, then the corresponding entry of $W^{(k)}$ gives a midpoint node w so that there are paths $i \rightsquigarrow w$ and $w \rightsquigarrow j$, both of which have length at most 2^{k-1}. The successor of path $i \rightsquigarrow j$ can then be retrieved from the successor of path $i \rightsquigarrow w$, which is already computed in $S^{(k-1)}$. The base case $k = 0$ is obtained directly from the adjacency matrix. This shows that $S^{(k)}$ is indeed a correct successor matrix.

2.2 Zwick's Algorithm

In Zwick's algorithm, we look at a shortest path such that the shortest path has between $\left(\frac{3}{2}\right)^{k-1}$ and $\left(\frac{3}{2}\right)^{k}$ edges. The idea was that the middle part has length $\sim \left(\frac{3}{2}\right)^{k-1}$, so we choose s from this random sample and the tails (parts of the path) on either side are each of length $\leq \left(\frac{3}{2}\right)^{k-1}$.

In step j, we compute some $(\min, +)$ product on matrices of the following dimensions: $n \times \frac{n}{\left(\frac{3}{2}\right)^k} \star \frac{n}{\left(\frac{3}{2}\right)^k} \times n$. We want witnesses for these products.

Definition A $(\min, \odot)$-*product of* A, B is C such that $C[i, j] = \min_k A[i, k] \odot B[k, j]$.

Remark Boolean product is a $(\min, +)$ product: $\bar{A} \leftarrow$ integer $\{0, 1\}$ corresponding to boolean A and $\bar{B}[i, j] = -1$ if $B[i, j] = 1$ and $= 0$ otherwise.

Definition A *witness matrix* for $(\min, \odot)$ is W such that $\forall i, j,\ W[i, j] = \arg\min_k A[i, k] \odot B[k, j]$.

If witnesses for $(\min, +)$ on matrices with entries in $\{-M, \ldots, M\}$ can be computed in $\tilde{O}(Mn^\omega)$ time, then successor matrices for Zwick's algorithm can be found in $\tilde{O}(M^{-.68}n^{2.575})$.

Remark The purpose of these witness matrices is that rather than finding the length of the shortest path, we want to obtain the actual shortest path. It costs a little more to do this, but not appreciably more.

2.3 Seidel's Algorithm

We presented a nice algorithm by Seidel for APSP in undirected graphs. For Seidel's algorithm, however, the above approach for computing the successor matrix does not apply as Seidel's algorithm runs recursively on a new graph and it computes the distances in a special way, using the counting power of integer matrix multiplication. Nevertheless, we can show that for undirected graphs, just being given the matrix of pairwise distances is sufficient to obtain the successor matrix in $O(n^\omega)$ time.

One of the main ideas in Seidel's algorithm was that for all i, j, and a neighbor k of i, $d[i,j]$ and $d[k,j]$ differ by at most 1, where d is the shortest distance matrix. Moreover, any k that satisfies $d[k,j] = d[i,j]-1$ is a valid successor of i in path $i \rightsquigarrow j$. Now, because $d[k,j]$ can be only 1 away from $d[i,j]$, we know that $d[k,j] \equiv d[i,j]-1 \pmod 3$ implies that k is a successor. This fact can be used to compute the successor matrix, given the matrix D of distances $D[i,j] = d[i,j]$, as follows.

Compute $d(i,j)$ for all i,j in $\tilde{O}(n^\omega)$ time (e.g., via Seidel). For all $s \in \{0,1,2\}$, set

$$D^{(s)}[k,j] = \begin{cases} 1 & \text{if } D[k,j] \equiv s-1 \pmod 3 \\ 0 & \text{otherwise.} \end{cases}.$$

Let A be the adjacency matrix. Compute $A \cdot D^{(s)}$ (boolean product, takes $O(n^\omega)$ time) for each choice of s and the witness matrix $W^{(s)}$ for this product. For all i,j, let $s_{ij} \equiv d(i,j) \mod 3$ and set $S[i,j] = W^{(s_{ij})}[i,j]$. To see why this is correct, fix i and j, and consider the (i,j) entry of the product $A \cdot D^{(s_{ij})}$. The index k that contributes to this entry satisfies $A[i,k] = D^{(s_{ij})}[k,j] = 1$. By construction, this means that $D[k,j] \equiv D[i,j]-1 \pmod 3$, and from the previous observation, we can conclude that k is a correct successor of i in path $i \rightsquigarrow j$. The witness is some k such that $A[i,k] = 1$ and $d(k,j) = d(i,j)-1 \mod 3$, i.e., $W^{(s_{ij})}[i,j]$ is the successor.

3 Computing Witness Matrices

In this section, we show an algorithm for computing witness matrices associated with the matrix product $A \odot B$, so that the successor matrix can be computed as described in the previous part.

Before handling the general case, we first focus on an easier variant of the problem, in which the witnesses are unique. From there, we will show we can easily find any witness matrix.

Definition A *unique witness matrix* for a $(\min, \odot)$-product of A and B is a matrix U such that $\forall i,j$, $U[i,j] = k$ such that k is the unique column such that $(A \odot B)[i,j] = A[i,k] \odot B[k,j]$ and $= \infty$ if no unique witness exists.

Special case: unique witness for BMM. Given A, B, create A' such that $A'[i,j] = j$ if $A[i,j] = 1$ and $= 0$ otherwise. Multiply $A' \cdot B$ (integer product). If k is a unique witness for i,j, then $\sum_l A'[i,l]B[l,j] = k$.

Strategy: $C \leftarrow A' \cdot B$ and for all i,j check if $A[i,C[i,j]] = B[C[i,j],j] = 1$. If so, $U[i,j] \leftarrow C[i,j]$, otherwise $U[i,j] \leftarrow \infty$.

Theorem 3.1. *If one can compute the* $(\min, \odot)$*-product of* $n \times n$ *matrices in* $T(n)$ *time, then computing* U *can be done in* $O(T(n) \log n)$ *time.*

Notation: for $S \subseteq [n]$, $A[\cdot, S]$ is the submatrix of A composed of the columns in S.

Proof of theorem 3.1. For all b from 1 to $\log n$ define $S_b = \{j | b\text{th bit of } j \text{ is } 1\}$. Let $C \leftarrow A \odot B$. Compute $(\min, \odot)$: $C_b \leftarrow A(\cdot, S_b) \odot B(S_b, \cdot)$. For all i,j,b, if $C[i,j] = C_b[i,j]$, then set the bth bit of $U[i,j]$ to 1, otherwise set it to 0. At the end, check the result $\forall i,j$, if $A[i,U[i,j]] \odot B[U[i,j],j] \neq C[i,j]$, then $U[i,j] \leftarrow \infty$ (if the witness is not unique we could get garbage).

This procedure computes the $(\min, \odot)$ product $\log n$ times, so the total running time is $O(T(n) \log n)$. It is also easy to see why this algorithm is correct: fix i and j that have a unique witness k. Then at every b, the bth bit of $W[i,j]$ will be set correctly. □

Finally, we show how to compute the more general witness matrix, given an algorithm for computing unique witnesses (get W from the ability to compute U). To do this, we do random samplings and use the algorithm above. First, we claim the following:

Lemma 3.2. *Let s be some value between 0 and $\log n$. Let (i,j) have c witnesses (in the $(\min,\odot)$-product of A and B) such that $\frac{n}{2^{s+1}} \le c \le \frac{n}{2^s}$. Let S be a random sample of $\{1,\dots,n\}$ of size 2^s. Then S contains a* unique *witness for (i,j) with probability $\ge \frac{1}{2e}$.*

If lemma 3.2 holds, for all $\log n$ values of s, repeat the following $d\log n$ times (for some constant $d > 3$): pick a random sample $S \subseteq [n]$ with $|S| = 2^s$, find $U \leftarrow$ unique witness matrix for $A(\cdot,S) \odot B(S,\cdot)$, and for all i,j, if $U[i,j] < \infty$, set $W[i,j] \leftarrow U[i,j]$.

For any (i,j) and the s such that the number of witnesses for (i,j) is in $\left(\frac{n}{2^{s+1}}, \frac{n}{2^s}\right)$, the probability that (i,j) doesn't have a unique witness in any of the $d\log n$ samples is $\le \left(1-\frac{1}{2e}\right)^{d\log n} \sim \frac{1}{n^d}$. By a union bound, the probability all (i,j) get a witness is $\ge 1 - \frac{1}{n^{d-2}}$.

Proof of lemma 3.2. Let W be the c witnesses for (i,j). Since there are 2^s elements in S and we have probability $\frac{c}{n}$ of hitting an element in W and we want to hit one element but not the others:

$$\begin{aligned} Pr[|W \cap S| = 1] &\sim \left(\frac{c}{n}\right) \cdot 2^s \left(1-\frac{c}{n}\right)^{2^s-1} \\ &\ge 2^s \cdot \frac{1}{2^{s+1}} \cdot \left(1-\frac{1}{2^s}\right)^{2^s-1} \\ &\ge \frac{1}{2e}. \end{aligned}$$

□

As stated between lemma 3.2 and its proof, this fact can be used to design a randomized algorithm that outputs a correct witness matrix. We do not know the number of witness for each individual (i,j) pair, but we know that it has to be contained in the intervals $\left[\frac{n}{2^{s+1}}, \frac{n}{2^s}\right]$ for some $s = 0,1,\dots,\log n$. The idea is to loop over all possible values of s, and determine the witnesses for all the (i,j) index pairs for which the number of witnesses fall into the right interval. Formally, the algorithm can be written as below.

Algorithm 2: WitnessMatrix(A, B)

```
C ← A ⊙ B
repeat
    for s = 0, 1, ..., log n do
        S ← random subset of {1, ..., n} of size 2^s
        Attempt to find the unique witness matrix W' of the matrix product A[·, S] ⊙ B[S, ·]
        for i, j = 1, ..., n do
            if A[i, W'[i, j]] ⊙ B[W'[i, j], j] = C[i, j] then
                W[i, j] ← W'[i, j]
until all elements of W are determined;
return W
```

For every outermost iteration, each $W[i,j]$ is filled with a correct witness with some constant probability, and once it is determined, it never changes. There are n^2 entries to be determined, so the expected number of outermost iterations is $O(\log n)$. We conclude by this final result.

Theorem 3.3. *If the $(\min,\odot)$-product is computable in $T(n)$ time, then finding the witness matrix takes $O(T(n)\log^3 n)$ time.*

CS367 Lecture 8
Scribe: Amir Abboud
Edited by: Cynthia Day

Perfect Matching
Date: April 28, 2014
Updated: October 23, 2015

1 Matchings in graphs

This week we will be talking about finding matchings in graphs: a set of edges that do not share endpoints.

Definition 1.1 (Maximum Matching)**.** *Given an undirected graph $G = (V, E)$, find a subset of edges $M \subseteq E$ of maximum size such that every pair of edges $e, e' \in M$ do not share endpoints $e \cap e' = \emptyset$.*

Definition 1.2 (Perfect Matching)**.** *Given an undirected graph $G = (V, E)$ where $|V| = n$ is even, find a subset of edges $M \subseteq E$ of size $n/2$ such that every pair of edges $e, e' \in M$ do not share endpoints $e \cap e' = \emptyset$. That is, every node must be covered by the matching M.*

Obviously, any algorithm for Maximum Matching gives an algorithm for Perfect Matching. It is an exercise to show that if one can solve Perfect Matching in $T(n)$ time, then one can solve Maximum Matching in time $\tilde{O}(T(2n))$. The idea is to binary search for the maximum k for which there is a matching M with $|M| \geq k$. To check whether such M exists, we can add a clique on $n - 2k$ nodes to the graph and connect it to the original graph with all possible edges. The new graph will have a perfect matching if and only if the original graph had a matching with k edges.

We will focus on Perfect Matching and give algebraic algorithms for it. Because of the above reduction, this will also imply algorithms for Maximum Matching. The idea will be to define some matrix such that the determinant of this matrix is non-zero if and only if the graph has a perfect matching.

1.1 The Tutte Matrix

Definition 1.3. *For a graph $G = (V, E)$ with $|V| = n$, the following $n \times n$ matrix T is the Tutte matrix of G:*

$$T[i,j] = \begin{cases} 0 & \text{if } i = j \text{ or if } (i,j) \notin E \\ x_{i,j} & \text{if } (i,j) \in E \text{ and } i < j \\ -x_{i,j} & \text{if } (i,j) \in E \text{ and } i > j \end{cases}$$

The following theorem is at the core of all the algorithms for Perfect Matching that we will discuss.

Theorem 1.1 (Tutte)**.** *For any graph $G = (V, E)$, the determinant of the Tutte matrix T is non-zero if and only if G contains a perfect matching.*

$$\det(T) \neq 0 \iff G \text{contains a perfect matching.}$$

Proof. By the definition of the determinant:

$$\det(T) = \sum_{\sigma \in S_n} (-1)^{sign(\sigma)} \cdot \underbrace{\prod_{i=1}^{n} T[i, \sigma(i)]}_{f_\sigma} \tag{1}$$

where S_n is the set of permutations of $[n]$ and $sign(\sigma)$ is the parity of inversions for σ, i.e. the number of pairs $x < y$ for which $\sigma(x) > \sigma(y)$.

We break the proof into three claims:

Claim 1 Permutations with odd cycles cancel out in $det(T)$.

Let P be the set of permutations in S_n that contain at least 1 odd cycle. For each $\sigma \in P$, let C_σ be the odd cycle in σ with minimum element, and let σ' be σ with C_σ reversed.

For example, if $\sigma = (1,5)(2,3,4)(6,7,8)$, then $\sigma' = (1,5)(4,3,2)(6,7,8)$.

$sign(\sigma) = sign(\sigma')$, so it follows that

$$\prod_{i=1}^{n} T(i,\sigma(i)) = -\prod_{i=1}^{n} T(i,\sigma'(i))$$

because the odd cycle $C_\sigma = (z_1,\ldots,z_r)$ leads to entries $T(z_1,z_2)\cdot\cdots\cdot T(z_r,z_1)$ in the first term and entries $T(z_2,z_1)\cdot\cdots\cdot T(z_1,z_r) = (-1)^r \cdot T(z_1,z_2)\cdot\cdots\cdot T(z_r,z_1)$.

Claim 2 Any σ with only even cycles corresponds to a perfect matching.

For any even cycle $C = (z_1,\ldots,z_{2r})$ in σ, we can select edges $(z_1,z_2),(z_3,z_4),\ldots,(z_{2r-1},z_{2r})$ to be in the matching. These edges are node-disjoint and cover all the vertices.

Note that Claim 1 in conjunction with Claim 2 demonstrates that whenever $det(T) \neq 0$, there is a σ with only even cycles so a perfect matching exists.

Claim 3 If G has a perfect matching, then $det(T) \neq 0$.

Say $(a_1,b_1),\ldots,(a_{n/2},b_{n/2})$ is a perfect matching. Consider the permutation $\sigma = (a_1,b_1)\ldots(a_{n/2},b_{n/2})$. Then

$$\prod_{i=1}^{n} T(i,\sigma(i)) = \prod_{i=1}^{n/2} T(a_i,b_i)T(b_i,a_i) = \prod_{i=1}^{n/2} -(x_{a_i,b_i})^2$$

No other permutation has the same variables, so this term can't cancel out. It follows that $det(T) \neq 0$.

□

The determinant $\det(T)$ is an n^2-variate polynomial of degree n and therefore can be expensive to compute.

Theorem 1.2 (Lovasz). *If we pick values v_{ij} for each x_{ij} uniformly at random from $\{1,\ldots,n^2\}$ and let $T(\{v_{ij}\})$ be the matrix obtained from T by these substitutions, then* $\det(T(\{v_{ij}\})) \neq 0$ *iff* $\det(T) \neq 0$*, with high probability.*

This gives a polynomial time algorithm for Perfect Matching that works with high probability. To prove this theorem we use:

Lemma 1.1 (Schwartz-Zippel). *Let P be a non-zero polynomial over $\{x_1,\ldots,x_N\}$ of degree d over a field $\mathbb{F}$. If we pick values $v_1,\ldots,v_N$ randomly from a finite set $S \subseteq \mathbb{F}$ and let $P(\{v_i\})$ be the value obtained by setting $x_1 = v_1,\ldots,x_N = v_N$ in P, then $P(\{v_i\}) \neq 0$ with probability at least $1 - \frac{d}{|S|}$.*

For $\det(T)$ we have $\deg(\det(T)) = n$ and therefore it is enough to pick $|S| = n^2$. However, if we work over $\mathbb{Z}$ the entries of this determinant could be very large and we only get a running time of $O(n^{\omega+1})$. Instead, pick a prime $p \geq n^3$ and work over $\mathbb{Z}_p$. If G has a perfect matching M then the polynomial $\det(T) \mod p$ contains the non-zero term f_{σ_M} and is therefore a non-zero polynomial and we can apply the Schwartz-Zippel lemma to check whether the determinant is zero in $O(n^\omega)$ time.

2 Finding the matching

The above algorithm tells us in $O(n^\omega)$ time whether the graph contains a perfect matching. In the rest of this lecture (and the next one) we will discuss algorithms that can find the perfect matching for us.

There is a simple $O(n^{\omega+2})$ solution: for every edge $e \in E$, remove it from the graph and check if there is still a perfect matching in $O(n^\omega)$ time. If the graph does not contain a perfect matching any more, put the edge back and move on to the next edge, otherwise leave the edge out of the graph. What we get in the end is a graph with $n/2$ edges that contains a perfect matching and we're done.

Today we will see an $O(n^{\omega+1})$ algorithm and next week we'll see an $O(n^\omega)$ one.

2.1 The Rabin-Vazirani Algorithm

We will prove this thorem.

Theorem 2.1 (Rabin-Vazirani)**.** *A perfect matching can be found in $O(n^{\omega+1})$ time.*

Consider Algorithm 1.

Algorithm 1: RV(G)

$T \leftarrow T(\{v_{ij}\})$: a random substitution of the Tutte matrix modulo a large enough prime;
if $\det(T) = 0$ **then**
  return no perfect matching;
Set $M = \emptyset$;
while $|M| < n/2$ **do**
  Compute $N = T^{-1}$ in $O(n^\omega)$ time;
  Find j such that $N[1, j] \neq 0$ and $(1, j) \in E$;
  $M \leftarrow M \cup \{(1, j)\}$;
  $T \leftarrow T_{\{1,j\},\{1,j\}}$ i.e. remove rows 1 and j and columns 1 and j from T;

We use the notation $T_{X,Y}$ for subsets $X, Y \subseteq [n]$ to denote the matrix obtained from T by removing the rows indexed by X and columns indexed by Y.

Clearly, the algorithm performs $O(n)$ computations that take $O(n^\omega)$ time and therefore runs in $O(n^{\omega+1})$ time. In fact, the algorithm is choosing some $e = (1, j)$ in some perfect matching and recursing on $G \setminus \{1, j\}$. We show the correctness below.

Recall the adjoin formula:

$$T^{-1}[i, j] = (-1)^{i+j} \cdot \frac{\det(T_{\{i\},\{j\}})}{\det(T)},$$

and therefore in the algorithm we have that $N[1, j] \neq 0$ iff $\det(T_{\{i\},\{j\}}) \neq 0$.

By the definition of the determinant:

$$\det(T) = \sum_{j=1}^{n} (-1)^{1+j} \cdot T[1, j] \cdot \det(T_{\{1\}\{j\}}),$$

and therefore if $\det(T) \neq 0$ then there exists $j \in [n]$ such that $T[1, j] \cdot \det(T_{\{1\}\{j\}}) \neq 0$ and therefore $(1, j) \in E$ and $\det(T_{\{1\}\{j\}}) \neq 0$. Therefore, to show the correctness of the algorithm, it is enough to show that the latter also implies that $\det(T_{\{1,j\}\{1,j\}}) \neq 0$ (i.e. when removing $\{1, j\}, \{1, j\}$ instead of just $\{1\}\{j\}$.)

To prove this, we need to use properties of the Tutte matrix. Note that T is a *skew symmetric* matrix: $T = -T^t$.

Proposition 1. *Let A be an $n \times n$ skew symmetric matrix, then:*

1. *A^{-1} is skew symmetric.*

2. *If n is odd, then* $\det(A) = 0$.

3. *(Frobenius) Let $Y \subseteq [n]$ s.t. $|Y| = rank(A)$ and the column rank of $A[[n], Y]$ is $rank(A)$, then* $\det(A[Y, Y]) \neq 0$.

Proof of 2: $\det(A) = \det(-A^t) = (-1)^n \det(A)$. We will use 3 without proof.

Lemma 2.1. *If* $\det(T_{\{1\},\{j\}}) \neq 0$ *then* $\det(T_{\{1,j\},\{1,j\}}) \neq 0$.

Proof. Assume without loss of generality that $j = 2$. By property 2 we know that $A = T_{\{1\},\{1\}} = 0$, so its rank is at most $n - 2$. By our assumption, $\det(T_{\{1\},\{2\}}) \neq 0$ so $\det(T_{\{1\},\{2\}})$ has rank $n - 1$. Therefore the column rank of $T_{\{1\},\{1,2\}}$ is $n - 2$ and the rank of A is $n - 2$. A is skew-symmetric, so it follows from the Frobenius property that for $Y = \{3, \ldots, n\}$, $det(A[Y, Y] = det(T_{\{1\},\{2\}} \neq 0$. □

Lecture 10 More on Matching
Scribe: Michael P. Kim **Date:** April 30, 2014
Edited by: Cynthia Day **Updated:** October 23, 2015

1 Introduction

In the last lecture, we discussed two algorithms for finding perfect matchings both of which leveraged a random substitution of the Tutte Matrix to find edges in a pefect matching. Recall the Tutte Matrix is defined as follows.

$$T(i,j) = \begin{cases} 0 & \text{if } (i,j) \notin E \\ x_{ij} & \text{if } i < j \\ -x_{ij} & \text{if } i > j \end{cases}$$

The naive algorithm uses the determinant of the Tutte Matrix as an oracle for whether a perfect matching exists, while the Rabin-Vazirani Algorithm finds edges to include in the matching by leveraging properties of the Tutte Matrix.

Recall the naive algorithm.

Algorithm 1: NaiveMatch(G)

for $e \in E$ **do**
  if $\det T_{G\setminus\{e\}} \neq 0$ **then**
    Remove e from G;

Because we can compute the determinant of T with an $\tilde{O}(n^\omega)$ randomized algorithm, this gives us an $O(n^{\omega+2})$ benchmark. Now, recall the Rabin -Vazirani algorithm.

Algorithm 2: RV(G)

$T \leftarrow$ random substitution of Tutte Matrix mod some prime $p > n^3$;
$M \leftarrow \emptyset$;
while $|M| < \frac{n}{2}$ **do**
  $N \leftarrow T^{-1}$ `// run time bottleneck to invert T`
  Find j s.t. $N(1,j) \neq 0,\ T(1,j) \neq 0$;
  $M \leftarrow M \cup \{(1,j)\}$;
  $T \leftarrow T_{\{1,j\},\{1,j\}}$

Remember that the runtime of RV is $O(n^{\omega+1})$, primarily due to the bottleneck in the while loop to compute the inverse of the Tutte Matrix in each iteration. In this lecture, we will improve this to $O(n^3)$ by using a specialized algorithm to invert the Tutte Matrix in $O(n^2)$ time.

2 Improving Rabin-Vazirani to $O(n^3)$

Claim 1 (Mucha, Sankowski)**.** RV *can be implemented so that updating N takes $O(n^2)$ time.*

Corollary 2.1. *Perfect matchings can be found in $O(n^3)$ time.*

To see Claim 1, we will start by proving a lemma, which we will use to recompute the necessary parts of the Tutte Matrix in $O(n^2)$ time.

Lemma 2.1. *Let M be an $n \times n$ invertible matrix. Let $N = M^{-1}$. Let M and N be of the following form.*

$$M = \begin{array}{c|c|c} & k & n-k \\ \hline k & X & Z \\ \hline n-k & Y & W \\ \hline \end{array} \qquad N = \begin{array}{c|c|c} & k & n-k \\ \hline k & \widehat{X} & \widehat{Z} \\ \hline n-k & \widehat{Y} & \widehat{W} \\ \hline \end{array}$$

If $\widehat{X}$ is invertible, so is W, with $W^{-1} = \widehat{W} - \widehat{Y} \cdot \widehat{X}^{-1} \cdot \widehat{Z}$.

Proof. We know $M \cdot N = I$. This means that $Y\widehat{X} + W\widehat{Y} = 0$. By assumption, X is invertible, so $Y = -W\widehat{Y}\widehat{X}^{-1}$. We also know that $Y\widehat{Z} + W\widehat{W} = I$. We can combine these facts to obtain the following result.

$$\begin{aligned} (-W\widehat{Y}\widehat{X}^{-1})\widehat{Z} + W\widehat{W} &= I \\ \implies W \cdot (\widehat{W} - \widehat{Y}\widehat{X}^{-1}\widehat{Z}) &= I \end{aligned}$$

Thus, W is invertible and its inverse is $\widehat{W} - \widehat{Y} \cdot \widehat{X}^{-1} \cdot \widehat{Z}$. □

Note that this lemma also holds for permutations of the columns/rows of M. In particular, this means we can apply the lemma to the Tutte matrix, where X is the 2×2 matrix formed by rows 1 and j and columns 1 and j and W is $T_{\{1,j\},\{1,j\}}$. WLOG, we'll assume $j = 2$. Then, we get the following result.

$$T^{-1}_{\{1,2\},\{1,2\}} = N_{\{1,2\},\{1,2\}} - N_{\{3:n\},\{1,2\}} \cdot N^{-1}_{\{1,2\},\{1,2\}} \cdot N_{\{1,2\},\{3:n\}}$$

We know that $N^{-1}_{\{1,2\},\{1,2\}}$ exists because N is skew-symmetric.

Additionally, we claim that we can compute this matrix inverse in $O(n^2)$ time. The inverse requires the subtraction of two $(n-2)\times(n-2)$ matrices which takes $O(n^2)$ time, and requires computing an $(n-2)\times(n-2)$ matrix by the multiplication of an $(n-2) \times 2$ matrix by a 2×2 matrix, and then a $(n-2) \times 2$ matrix by a $2 \times (n-2)$ matrix. These multiplications also take $O(n^2)$ time. Thus, with minor modifications to the Rabin-Vazirani Algorithm, we come up with an $O(n^3)$ algorithm for finding a perfect matching. This algorithm can be further modified to compute perfect matchings in bipartite graphs in $O(n^\omega)$ time, but this result is not generalizable to other graphs.

3 Improving Naive Search to $O(n^\omega)$

Claim 2 (Harvey)**.** *There is an $\tilde{O}(n^\omega)$ time algorithm for perfect matching in general graphs.*

The algorithm by Harvey reimplements the naive algorithm for perfect matching. The naive version uses the determinant of the Tutte Matrix as an oracle for whether a perfect matching exists, and incrementally removes unnecessary edges. The key idea to the improved version is to choose a way to access the edges such that checking if $G \setminus \{e\}$ has a perfect matching is cheap. Consider the following lemma.

Lemma 3.1. *Let M be an invertible $n\times n$ matrix and $S \subseteq [n]$ be some small subset of the entries. Let $\tilde{M}$ be an $n\times n$ matrix such that if $\tilde{M}(i,j) \neq M(i,j)$, then $i,j \in S$. Finally, let $\Gamma = I_{|S|} + (\tilde{M}[S,S] - M[S,S])M^{-1}[S,S]$. Then the following statements are true.*

1. *$\tilde{M}$ is invertible if and only if*

$$\det(\Gamma) \neq 0$$

2. If $\tilde{M}$ is invertible, then its inverse is

$$\tilde{M}^{-1} = M^{-1} - M^{-1}(:, S)\Gamma^{-1}(\tilde{M}[S, S] - M[S, S])M^{-1}[S, :]$$

An immediate corollary of *1* is that we can check if $\tilde{M}$ is invertible in $O(|S|^\omega)$ time. Also, suppose we're only interested in $\tilde{M}^{-1}[S, S]$. Then, we can compute this submatrix by using a $|S| \times |S|$ submatrix of M^{-1} in each submatrix in *2* of the lemma. In this case, the overall time to compute $\tilde{M}^{-1}[S, S]$ is $O(|S|^\omega)$ given M and M^{-1}.

Let $N = T^{-1}$ and $\tilde{T}$ be T with $\tilde{T}(i, j) = \tilde{T}(j, i) = 0$.

Claim 3. *Checking if* $\det \tilde{T} \neq 0$ *is in* $O(|S|^\omega) = O(1)$ *time.*

To see this claim, note that we need to check if $\det \Gamma \neq 0$ where in this case Γ is as follows.

$$\begin{aligned}\Gamma &= \begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} + \begin{bmatrix}0 & -T(i,j)\\ T(i,j) & 0\end{bmatrix} \cdot \begin{bmatrix}0 & N(i,j)\\ -N(i,j) & 0\end{bmatrix}\\ &= \begin{bmatrix}T(i,j)N(i,j)+1 & 0\\ 0 & T(i,j)N(i,j)+1\end{bmatrix}\end{aligned}$$

Thus, $G \setminus \{(i, j)\}$ has a perfect matching if and only if $T(i, j) \cdot N(i, j) \neq -1$.

Thus, we can detect the presence of a perfect matching quickly, and we have that $\tilde{T}$ is the substituted Tutte Matrix of $G \setminus \{(i, j)\}$. Now all that remains to be seen is how to recompute $\tilde{T}^{-1}$, the updated version of N. We could naively do this in $O(n^2)$ time, by the previous lemma but this would only give us an $O(n^4)$ algorithm. We need to do something more sophisticated to obtain the desired $O(n^\omega)$ run time.

Let $S_1, S_2 \subseteq V$ where $|S_1| = |S_2|$. Let's try removing edges from $S_1 \times S_2$ according to Algorithm 3.

Algorithm 3: DeleteCross(S_1, S_2)

```
if |S1| = |S2| = 1 then
    S1 = {s}, S2 = {r};
    if T(s,r) · N(s,r) ≠ −1 then
        T(s,r) = T(r,s) = 0    // remove (s,r)
        UpdateN;
else
    S1 ← S11 ∪ S12,  S2 ← S21 ∪ S22    // partition S1 and S2 each into two equal subsets
    for i, j ∈ {1,2} do
        DeleteCross(S1i, S2j);
        UpdateN;
```

While this recursive procedure will work, the question that remains is how to update N efficiently. The solution will be to update only $N[S_1 \cup S_2, S_1 \cup S_2]$ after each DeleteCross call within DeleteCross(S_1, S_2).

In particular, we will maintain the invariant that within DeleteCross(S_1, S_2), $N[S_1 \cup S_2, S_1 \cup S_2]$ will be correct. Then at the base case when $|S_1| = |S_2| = 1$, we will have the correct values $T(s, r)$ and $N(s, r)$ and we can correctly figure out whether (r, s) can be deleted.

To update N after a call to DeleteCross(S_1, S_2), we have the old T before the call and the new $\tilde{T}$ of changes within the call. All the changes will be in $(S_1 \cup S_2) \times (S_1 \cup S_2)$, so the updates will be sufficient. Updating N will take $O((|S_1| + |S_2|)^\omega)$ time using our observations above and Lemma 3.1.

Thus, the overall runtime of DeleteCross(S_1, S_2) is given by the following recurrence relation, where $n = |S_1| = |S_2|$.

$$\begin{aligned} T(n) &\leq 4T(\frac{n}{2}) + 4O(n^\omega)\\ \implies T(n) &= \tilde{O}(n^\omega)\end{aligned}$$

We also need to handle edges within a set S in order to appropriately partition it into S_1 and S_2. Consider the final algorithm.

(Below we assume that $|S|$ is a power of 2. If it is not, we can add a large enough matching Y of new nodes to the graph: the size of the graph at most doubles, and the new graph has a perfect matching if and only if the old one did.)

Algorithm 4: DeleteWithin(S)

if $|S| = 1$ **then**
  Return;
$S \leftarrow S_1 \cup S_2$ `// such that |S1| = |S2| = |S|/2`
DeleteWithin(S_1), UpdateN(S, S);
DeleteWithin(S_2), UpdateN(S, S);
DeleteCross(S_1, S_2), UpdateN(S, S);

The runtime of Algorithm 4 is given by the following recurrence relation.

$$\begin{aligned} T(n) &\leq 2T(\frac{n}{2}) + T(\text{DeleteCross}(\frac{n}{2})) \\ &= 2T(\frac{n}{2}) + T(\tilde{O}(n^\omega)) \\ \implies T(n) &= \tilde{O(n^\omega)} \end{aligned}$$

The final algorithm for finding a matching is

Algorithm 5: Matching(G)

DeleteWithin(V);
Return remaining edges;

References

[1] Marcin Mucha and Piotr Sankowski, *Maximum Matchings via Gaussian Elimination*, FOCS, 248-255 (2004).

[2] Nicholas J.A. Harvey, *Algebraic Algorithms for Matching and Matroid Problems*, SIAM Journal on Computing, 39(2):679-702, (2009).