CS 598CCI: Topics in Graph Algorithms	Lecture date: 22 January, 2020
Instructor: Chandra Chekuri	Scribe: Vasilis Livanos

1 Introduction

Graphs and graph algorithms are ubiquitous in Computer Science and many other areas. A number of graph algorithms are designed and analyzed from a "combinatorial" point of view. On the other hand many interesting and powerful results have and can be obtained by using methods and techniques from continuous optimization, (linear) algebra and other approaches that are less discrete and combinatorial. These techniques are not new but have come to the fore recently due to a number of developments. In this course we will explore some of these topics by combining some old and new results. The choice of topics and problems will be somewhat adhoc and based on the instructor's taste. Some of the topics that we are hoping to cover are the following.

-
- Matrix multiplication based algorithms including matchings via determinants.
- Spanning trees and connection to matroids. Spanning tree polytope, bounded degree spanning trees via iterated rounding, and matrix tree theorem for counting number of spanning trees.
- Introduction to semi-definite programming (SDP) via Max Cut, perfect graphs, Grothendieck constant
- Sparsest cut. Via LPs and embeddings. Lower bounds on flow-cut gap via expanders.
- Spectral graph partitioning and Cheeger inequalities
- SDP approach to graph partitioning and ARV theorem/algorithm
- Graph sparsification.
- Time permitting some intro to improved max-flow algorithms via electric flows and related ideas.

See webpage for links to various references, lecture notes, and related courses.

2 Matrix Multiplication and variants

Let A and B be two $n \times n$ matrices over some field $\mathcal{F}$ (for the most part we will work with the rationals $\mathbb{Q}$, the reals $\mathbb{R}$, the complex numbers $\mathbb{C}$ or the finite field $\mathbb{Z}_p$ for prime p). The *matrix product* $C = A \cdot B$ is defined as an $n \times n$ matrix C where

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}.$$

It is easy to see that the definition allows one to compute C in $O(n^3)$ arithmetic operations. Surprisingly, [Strassen, 1969] showed that one can do better. He developed a $O\left(n^{2.81}\right)$-time algorithm using a clever divide and conquer strategy. First, consider the following obvious divide and conquer scheme

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

where

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

In this divide and conquer scheme, to compute C we need to perform 8 multiplications of $\frac{n}{2} \times \frac{n}{2}$-sized matrices, along with 4 addition of $\frac{n}{2} \times \frac{n}{2}$-sized matrices. This yields a recurrence of the form

$$T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2)$$

which can be easily shown to yield $T(n) = O(n^3)$.

However, Strassen was able to reduce the number of multiplications to 7 using subtraction

$$\begin{aligned} C_{11} &= M_1 + M_4 - M_5 + M_7 \\ C_{12} &= M_3 + M_5 \\ C_{21} &= M_2 + M_4 \\ C_{22} &= M_1 - M_2 + M_3 + M_6 \end{aligned}$$

where

$$\begin{aligned} M_1 &= (A_{11} + A_{12})(B_{11} + B_{12}) \\ M_2 &= (A_{21} + A_{22})B_{11} \\ M_3 &= A_{11}(B_{12} - B_{22}) \\ M_4 &= A_{22}(B_{21} - B_{11}) \\ M_5 &= (A_{11} + A_{12})B_{22} \\ M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ M_7 &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

This changes the recurrence to

$$T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2)$$

and we get

$$T(n) = O\left(n^{\log_2 7}\right) = O(n^{2.81}).$$

Several subsequent improvements led to the CoppersmithWinograd algorithm that runs in time $O(n^{2.376})$ [Coppersmith and Winograd, 1990]. This bound stood for a long time until some recent improvements due to [Davie and Stothers, 2013, Williams, 2012, Le Gall, 2014], with the current state-of-the-art running time being $O(n^{2.37287})$.

Open Problem. *Can matrix multiplication be done in* $O\left(n^{2+o(1)}\right)$ *time?*

The above question has very important connections to symmetry, groups, etc. See [Cohn and Umans, 2003, Cohn et al., 2005, Alman and Williams, 2018] and their approaches for finding improved upper and lower bounds.

It is an open question what is the best-possible running time for matrix multiplication. Therefore, we use ω to refer to the smallest possible exponent for matrix multiplication. In other words, matrix multiplication can be done in $O(n^{\omega})$, but not in $O(n^{\omega-\epsilon})$ for any $\epsilon > 0$. By our discussion above, we know that $2 \leq \omega \leq 2.37287$.

Strassen's 1969 paper [Strassen, 1969] not only showed that matrix multiplication can be done in subcubic time, but also that many other matrix computations can be performed in $O(n^{\omega})$ time. Specifically, for an $n \times n$ matrix A, we can, in $O(n^{\omega})$ time

- Compute the inverse A^{-1}
- Compute the determinant $\det(A)$
- Solve $Ax = b$, given an $n \times 1$ vector b
- Compute the characteristic polynomial $\det(A - \lambda I)$
- Compute the rank, $rank(A)$, and a corresponding submatrix

See Zwick's slides [Zwick, 2015] on the course website for additional information on how some of the above are accomplished.

In combinatorial applications we encounter two other forms of matrix multiplication.

Definition 1 (Boolean Matrix Multiplication (BMM))**.** *Given* $A, B \in \{0,1\}^{n \times n}$*, we define* $C = A \cdot B$ *as*

$$C_{ij} = \bigvee_{k=1}^{n} (A_{ik} \wedge B_{kj})$$

In other words, $C_{ij} = 1$ if and only if there exists an index k such that both A_{ik} and B_{kj} are 1.

It is easy to see that BMM can be reduced to standard integer multiplication.

Open Problem. *Can Boolean Matrix Multiplication be done in* $O\left(n^{2+o(1)}\right)$ *time?*

Definition 2 (Min-Plus Product (MPP))**.** *Given* A, B*, we define* $C = A \star B$ *as*

$$C_{ij} = \min_{k=1}^{n} (A_{ik} + B_{kj})$$

It is not clear how to compute $A \star B$ using standard products, but one can achieve some results via a complicated reduction. See Zwick's slides [Zwick, 2015] for further information on this.

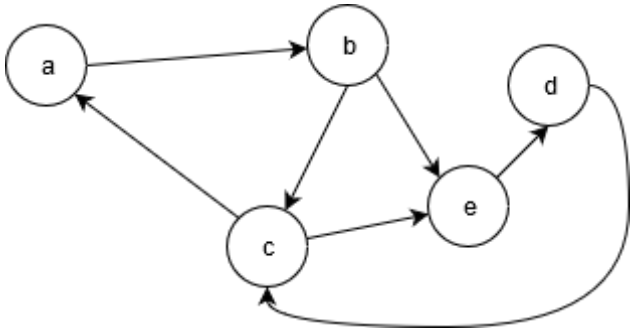
3 Applications

We aim to give a short overview of some simple and surprising applications of fast matrix multiplication. There are many more applications which you can find in the references.

There are several matrices that one can associate with a graph, but perhaps the most common and natural one is the *adjacency matrix*, which is commonly denoted by A_G, but we will shorten this to A when the graph G is clear from the context.

For a directed simple graph $G = (V, E)$ on n vertices, A is a $n \times n$ matrix with $A_{ij} = 1$ if $(i, j) \in E$ and $A_{ij} = 0$ otherwise. Note that A is not necessarily symmetric. For an undirected graph, we define A by bidirecting each edge and taking the adjacency matrix of the resulting directed graph, which is the same as setting $A_{ij} = A_{ji} = 1$ if $\{i, j\} \in E$ and $A_{ij} = A_{ji} = 0$ otherwise.

Example 1. *Let G be the following graph.*



The adjacency matrix of G is

$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Let G be a directed graph and A be its adjacency matrix. An interesting question is what does A^2 represent. We have

$$A^2_{ij} = \sum_{k=1}^{n} A_{ik} A_{kj}$$

It is easy to see that $A_{ik}A_{kj}$ is 1 only if $(i, k), (k, j) \in E$, and thus A^2_{ij} is equal to the number of paths from i to j of length 2. It turns out that this does not generalize for all $k \geq 2$, however, as the product also counts walks from i to j, in which vertices can be repeated.

Recall that a *walk* in a directed graph is a sequence of vertices $v_1, v_2, \ldots, v_\ell$, where $(v_k, v_{k+1}) \in E$ for all $1 \leq k < \ell$. Notice that, unlike in a path, the same vertex can appear multiple times in a walk. Also, we denote the k-fold product of A with itself is denoted by A^k, i.e. $A^k = A \cdot A \cdot \cdots \cdot A$ where the right-hand side product has k factors.

Lemma 3. *Let G be a directed graph and A its adjacency matrix. The number of* walks *from vertex i to vertex j of length k is equal to A^k_{ij}.*

Proof: Run induction on k. For the base case, let $k = 1$. Clearly, the number of walks from i to j of length 1 is equal to 1 if and only if $(i, j) \in E$ and 0 otherwise, therefore A_{ij} is equal to the number of walks from i to j of length 1. Assume that, for all $\ell < k$, A^ℓ_{ij} is equal to the number of walks of length ℓ from i to j.

For the inductive step, we know that

$$A^k_{ij} = \sum_{p=1}^{n} A^{k-1}_{ip} A_{pj}$$

Notice that every walk of length k from i to j visits a neighbor of k after traversing $k-1$ edges. Therefore, every walk of length k from i to j is equivalent to a walk of length $k-1$ from i to some neighbor p of j, along with the edge (p, j). Equivalently, every walk of length $k-1$ from i to p can be extended to a walk of length k from i to j, by adding the edge (p, j).

By our inductive hypothesis, A^{k-1}_{ip} is equal to the number of walks from i to p of length $k-1$, and thus, by the equation above, A^k_{ij} counts the number of walks from i to p of length $k-1$ for all p such that $(p, j) \in E$, which is equal to the number of walks from i to j of length k, and the Lemma holds. □

By the above, it is clear that A^k_{ii} counts the number of closed walks that start and end at vertex i of length k. While walks are not as useful as paths for algorithmic purposes, we can still obtain a useful application from the following special case.

Let $k = 3$, and consider a simple directed graph G. A^3_{ii} counts the number of closed walks that start and end at vertex i of length 3, and since G has no multiple edges or self-loops, and vertices cannot be repeated in a closed walk of length 3, a closed walk of length 3 in G is a *triangle* (i.e. a cycle of length 3). Thus, A^3_{ii} counts the number of triangles of length 3 that contain i.

Suppose we would like to compute the number of triangles in an **undirected** graph G. Then, we can bidirect G to get a directed graph $\overrightarrow{G}$, compute $\sum_{i \in V(\overrightarrow{G})} A^3_{ii}$, where A is the adjacency matrix of $\overrightarrow{G}$, and obtain the number of triangles in G, which is equal to $\frac{1}{6} \sum_{i \in V(\overrightarrow{G})} A^3_{ii}$, since each triangle in G is counted in $\overrightarrow{G}$ six times; for each of its three vertices, and for both orientations of the directed edges.

Theorem 4. *The total number of triangles in an n-vertex undirected graph and also the number of triangles incident to each vertex can be computed in $O(n^\omega)$ time.*

Interestingly, triangle counting is quite useful in social network analysis. For details on this, we refer the reader to [Roughgarden, 2014a, Roughgarden, 2014b]. For sparse graphs, an algorithm with running time $O\left(m^{\frac{3}{2}}\right)$ is known, where m is the number of edges in the graph.

3.1 Transitive Closure

A very fundamental problem in graphs is reachability. That is, given a graph G and two nodes s, t, does there exist a path from s to t. Reachability can be solved in linear time via standard graph search methods.

Definition 5 (Transitive Closure of a graph)**.** *The* transitive closure *of a directed graph $G = (V, E)$ is a graph $G^* = (V, E^*)$ where $(i, j) \in E^*$ if and only if there exists a (directed) path form i to j in G.*

In other words, G^* encodes all-pair reachability. Therefore, it is easy to see that $A_{G^*,ij} = 1$ if and only if i can reach j in G^*.

Question. *How fast can we compute G^* given G?*

Notice that G being directed is crucial, since for undirected graphs there exist simple and fast algorithms that compute G^*.

Exercise 1. *Given an undirected graph G, give an algorithm that runs in $O(n^2)$ time and computes the reachability of all pairs of vertices in G.*

The easy solution to the question above is to compute the set of vertices that are reachable from each vertex, which takes $O(mn)$ time. Can we do better?

Consider G and add a self-loop to every vertex. This corresponds to setting $A_{ii} = 1$ for each i. Now what does A^2 represent? We know that A^2_{ij} counts the number of walks of length 2 from i to j, in the graph with self-loops. Thus, $A^2_{ij} > 0$ if and only if i can reach j with at most two edges.

In fact, we can do Boolean Matrix Multiplication. We can write the adjacency matrix of the new graph with added self-loops as $A \vee I$, where I is the $n \times n$ identity matrix.

Lemma 6. *There exists a path from i to j in G if and only if $(A \vee I)^{n-1}_{ij} = 1$, where the multiplication is Boolean Matrix Multiplication.*

Proof: Firstly, suppose that $(A \vee I)^{n-1}_{ij} = 1$. This implies that there exist $v_1 = i, v_1, \ldots, v_n = j$ such that

$$(A \vee I)_{iv_2} \wedge (A \vee I)_{v_2v_3} \wedge \cdots \wedge (A \vee I)_{v_{n-1}j} = 1.$$

Due to the added self-loops, it could be that v_i and v_{i+1} are the same vertex for some i. This implies that there exists a walk from i to j with at most $n - 1$ edges, and thus i can reach j.

On the other hand, assume that i can reach j in G. This implies that there exists a path from i to j, and we know that any path in G has length at most $n - 1$, since no two vertices can be repeated in a path. Let $P = (i = v_1, v_2, \ldots, v_k = j)$ be this path, where k denotes it length. Its existence implies that

$$A_{iv_2} \cdot A_{v_2v_3} \cdot \cdots \cdot A_{v_{k-1}j} = 1. \tag{1}$$

Notice that, if we consider $A \vee I$ however, instead of A, we can create a walk from P which traverses the self-loop at v_{k-1} a number of times to make the total edge traversals equal to $n - 1$. In other words, from P we can create the walk $W = (i = v_1, v_2, \ldots, v_{k-1}, v_k = v_{k-1}, v_{k+1} = v_{k-1}, \ldots, v_{n-1} = v_{k-1}, v_n = j)$. Since the vertex v_{k-1} has a self-loop, we know that $(A \vee I)_{v_{k-1}v_{k-1}} = 1$, and thus, by (1), we get

$$A_{iv_2} \cdot A_{v_2v_3} \cdot \cdots \cdot A_{v_{k-1}v_k} \cdot A_{v_kv_{k+1}} \cdot \cdots \cdot A_{v_{n-1}j} = 1.$$

Thus, at least one clause of the "or" operation in $(A \vee I)^k_{ij}$ is equal to 1, which implies that $(A \vee I)^k_{ij} = 1$. □

We can compute $(A \vee I)^{n-1}_{ij} = 1$ with only $O(\log n)$ Boolean Matrix Multiplications by repeated squaring. First, notice that, since any path from i to j requires at most $n - 1$ edges, for all $k \geq n$,

Algorithm 1: Computing $(A \vee I)^{n-1}$ via Repeated Squaring

$B = A \vee I$;
for $i = 1$ *to* $\lceil \log n \rceil$ **do**
  $B = B \cdot B$ (Boolean MM)
end
Output B

$(A \vee I)^k_{ij} = 1$ if and only if $(A \vee I)^{n-1}_{ij} = 1$. Therefore, even if $n-1$ is not a power of 2, it is enough to compute $(A \vee I)^k$ where k is the smallest power of 2 which is bigger than $n-1$.

Therefore, the total running time of Algorithm 1 is $O(n^\omega \log n)$.

One can improve this running time and get rid of the $\log n$ factor for a total running time of $O(n^\omega)$. For additional information on how this can be done, see [Fischer and Meyer, 1971, Macii, 1995]. *Hint: First assume that G is a DAG. How can you improve the running time then?*

Open Problem. *Is there a combinatorial algorithm to solve the all-pairs reachability problem that runs in $O(n^{3-\delta})$ time for some fixed $\delta > 0$?*

Matrix multiplication can also be used for *dynamic* maintenance of the transitive closure of a graph. See [Zwick, 2015] and the references there for more information.

3.2 Seidel's algorithms for APSP

Given a graph with non-negative edge-weights, *All-Pairs-Shortest-Path (APSP)* is the problem of computing the shortest path between all pairs of vertices in the graph. Using Dijkstra's shortest path algorithm n times one can achieve a running time of $O(nm + n^2 \log n)$ for this problem. While, for $m = O(n \log n)$, this is $O(n^2 \log n)$, which is subcubic, for dense graphs this running time becomes $O(n^3)$. Can we do better?

It is not clear how to achieve a subcubic running time for dense graphs in the general case of integer weights, but for some special cases we can improve the running time via matrix multiplication. For the general case, the state-of-the-art is a running time of $O\left(\frac{n^3}{exp(\sqrt{\log n})}\right)$. The actual complexity of APSP is a very important question in fine-grained complexity and has a lot of interesting connections to the exact time complexity of many other problems. For more information, see Section 5 of [Williams, 2018].

Theorem 7 ([Seidel, 1995])**.** *Given a simple undirected graph G with unit edge lengths, the APSP problem can be solved in $O(n^\omega \log n)$ time.*

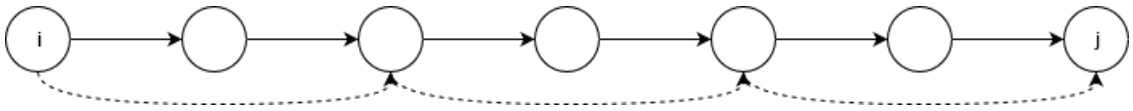
Definition 8 (Square of a graph)**.** *Let $G = (V, E)$ be an undirected simple graph. The* square *of G, denoted by G^2, is the graph $G^2 = (V, E')$, where $\{i, j\} \in E'$ if and only if $\{i, j\} \in E$ or there exists a node $k \in V$ such that $\{i, k\}, \{k, j\} \in E$.*

We have already seen that G^2 can be computed in $O(n^\omega)$ time by Boolean Matrix Multiplication. Can we perhaps compute APSP in G by computing APSP in G^2? In other words, suppose we have solved the APSP problem in G^2. Can we then use the APSP in G^2 to compute the APSP in G? If so, then we can recursively solve the APSP on G^2 for a total $O(\log n)$ blowup in time. For this, we need to understand some properties of shortest paths in G and G^2. Let $d_G(i, j)$ (resp. $d_{G^2}(i, j)$ denote the shortest-path distance between i and j in G (resp. in G^2).

Claim 9. *If $d_G(i, j)$ is even, then*

$$d_{G^2}(i, j) = \frac{1}{2} d_G(i, j).$$

Proof:



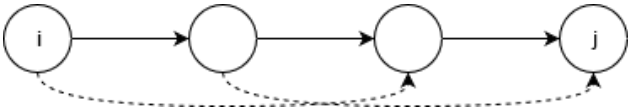
It is easy to see that $d_{G^2}(i,j) \leq \frac{1}{2}d_G(i,j)$. Let $d_G(i,j) = 2k$ and consider a shortest path $P = (i = v_0, v_1, v_2, \ldots, v_{2k-1}, v_{2k} = j)$ from i to j in G. By definition, G^2 also contains the edges $(i, v_2), (v_2, v_4), \ldots, (v_{2k-2}, j)$, and thus there exists a path $P' = (i = v_0, v_2, v_4, \ldots, v_{2k-2}, v_k = j)$ in G^2. Notice that $|P'| = k$, and thus $d_{G^2}(i,j) \leq \frac{1}{2}d_G(i,j)$.

Also notice that $d_{G^2}(i,j) \geq \frac{1}{2}d_G(i,j)$, since if $d_{G^2}(i,j) < k$, then this would imply a path from i to j in G of length less than $2k$, which contradicts the fact that $d_G(i,j) = 2k$. We conclude that $d_{G^2}(i,j) = \frac{1}{2}d_G(i,j)$. □

Claim 10. *If $d_G(i,j)$ is odd, then*

$$d_{G^2}(i,j) \leq \left\lceil \frac{1}{2}d_G(i,j) \right\rceil.$$

Proof:



Again, let $d_G(i,j) = 2k+1$ and consider a shortest path $P = (i = v_0, v_1, v_2, \ldots, v_{2k}, v_{2k+1} = j)$ from i to j in G. By definition, G^2 also contains the edges $(i, v_2), (v_2, v_4), \ldots, (v_{2k-2}, v_{2k})$ and the edge (v_{2k}, j) which is also in G, and thus there exists a path $P' = (i = v_0, v_2, v_4, \ldots, v_{2k}, j)$ in G^2. Notice that $|P'| = k+1$, and thus $d_{G^2}(i,j) \leq \frac{2k+2}{2} = \lceil \frac{1}{2}d_G(i,j) \rceil$. □

Claims 9 and 10 imply that either

$$d_G(i,j) = 2d_{G^2}(i,j)$$
$$\text{or}$$
$$d_G(i,j) = 2d_{G^2}(i,j) - 1$$

and if we can figure out which of the two cases we are in then we can compute d_G from d_{G^2}.

Lemma 11. *Suppose that $d_G(i,j)$ is even. Then, for every neighbor k of i, we have*

$$d_{G^2}(k,j) \geq d_{G^2}(i,j)$$

Proof: First, assume that k is not on any shortest path from i to j. This implies that $d_G(k,j) \geq d_G(i,j)$, since if $d_G(k,j) < d_G(i,j)$, then the shortest path from k to j along with the edge (i,k) would imply a path from i to j that goes through k of length at most $d_G(i,j)$. Since $d_G(i,j)$ is the length of the shortest path from i to j, this would imply that k lies on a shortest path from i to j, contradicting our assumption. However, $d_G(k,j) \geq d_G(i,j)$ directly implies that $d_{G^2}(k,j) \geq d_{G^2}(i,j)$, and we are done in this case.

Suppose that k is on a shortest path from i to j, and let $d_G(i,j) = 2\ell$. Then, $d_G(k,j) = 2\ell - 1$, and, by Claim 9 we have $d_{G^2}(i,j) = \ell$. However, notice that $d_{G^2}(k,j) = \ell$, as if $d_{G^2}(k,j) < \ell$, this would imply $d_G(i,j) \leq 2\ell - 1$. We conclude that, even when k lies on a shortest path from i to j, we have $d_{G^2}(k,j) \geq d_{G^2}(i,j)$. □

Lemma 12. *Suppose that $d_G(i,j)$ is odd and $d_G(i,j) > 1$. Then, for every neighbor k of i, we have*

$$d_{G^2}(k,j) \leq d_{G^2}(i,j).$$

Furthermore, for every neighbor k of i that lies on a shortest path from i to j, we have

$$d_{G^2}(k,j) < d_{G^2}(i,j).$$

Therefore, the above inequality holds for at least one neighbor k^* of i.

Proof: Let $d_G(i,j) = 2\ell + 1$. We know that $d_G(k,j) \leq d_G(i,j) + 1$, as k is a neighbor of i, and thus $d_{G^2}(i,j) = \lceil \frac{2\ell+1}{2} \rceil = \ell + 1$, while $d_{G^2}(k,j) \leq \frac{2\ell+2}{2} = \ell + 1$, and thus $d_{G^2}(k,j) \leq d_{G^2}(i,j)$.

Next, consider a neighbor k^* of i which lies on a shortest path from i to j. We know that $d_G(k,j) = d_G(i,j) - 1$, and thus $d_G(k,j) = 2\ell$. This implies, by Claim 9, that $d_{G^2}(k,j) = \ell < d_{G^2}(i,j)$. □

Let $N(i)$ denote the set of neighbors of i. Notice that $\sum_{k \in N(i)} d_{G^2}(k,j) \leq \deg(i) \cdot d_{G^2}(i,j)$, and Lemmas 11 and 12 give us a nice criterion that distinguishes between $d_G(i,j)$ being even or odd.

Corollary 13. *If $\sum_{k \in N(i)} d_{G^2}(k,j) < deg(i) \cdot d_{G^2}(i,j)$, then*

$$d_G(i,j) = 2d_{G^2}(i,j) - 1$$

and if $\sum_{k \in N(i)} d_{G^2}(k,j) = deg(i) \cdot d_{G^2}(i,j)$, then

$$d_G(i,j) = 2d_{G^2}(i,j)$$

The question now becomes: How do we compute $\sum_{k \in N(i)} d_{G^2}(k,j)$ for all $i,j \in V$? Let B denote the distance matrix of G^2, i.e. $B_{ij} = d_{G^2}(i,j)$. Notice that B is not necessarily a boolean matrix. We have

$$(AB)_{ij} = \sum_{k=1}^{n} A_{ik}B_{kj} = \sum_{k \in N(i)} B_{kj}$$

Thus, we can check for each pair i,j if $\deg(i) \cdot B_{ij} > (AB)_{ij}$ and conclude whether $d_G(i,j) = 2d_{G^2}(i,j)$ or $d_G(i,j) = 2d_{G^2}(i,j) - 1$. Notice that Seidel's algorithm requires one Boolean Matrix Multiplication and one Integer Matrix Multiplication in each recursive call. We leave it as an exercise to formalize the details and conclude that the total running time of Seidel's algorithm is $O(n^\omega \log n)$.

Exercise 2. *Use only Boolean Matrix Multiplication to solve the APSP problem in undirected graphs with unit edge lengths.*

There exist several interesting results and many open problems on APSP. As an example, for undirected, weighted graphs, if we know that the maximum edge weight in G is M, then there exists a $O(Mn^\omega)$-time algorithm for the APSP problem. For more information, see Zwick's excellent slides [Zwick, 2015].

It turns out that the APSP problem is captured by the Min-Plus Product (MPP). Let W be a matrix that encodes the edge weights of G. In other words, W_{ij} is equal to the weight of the edge $\{i,j\}$, if $\{i,j\} \in E$, and ∞ otherwise. For practical applications, one can replace ∞ by a sufficiently large number. Also, let $W^{\star k}$ denote the k-th power of W, where the multiplication is MPP.

Lemma 14. *For any pair $i,j \in V$, $W^{\star k}_{ij}$ is equal to the shortest walk between i and j using at most k edges.*

Proof: This can be shown easily via induction on k. For the base case let $k = 1$, and the Lemma holds since, if $\{i, j\} \in E$, then W_{ij} is equal to the weight of the edge $\{i, j\}$. Assume that the Lemma is true for all $\ell < k$.

For the inductive step, notice that $W^{\star k} = W^{\star k-1} \star W$, and thus

$$W_{ij}^{\star k} = \min_{p=1}^{n} \left(W_{ip}^{\star k-1} + W_{pj} \right)$$

Using our inductive hypothesis, $W_{ip}^{\star k-1}$ is equal to the shortest walk between i and p using at most $k-1$ edges. Therefore, by the above equation, $W_{ij}^{\star k}$ is equal to the shortest path between i and j using at most k edges and the Lemma holds. □

Using Lemma 14 and repeated squaring, one can solve the APSP problem in the general case of integer weights in $O(MPP(n) \log n)$ time, where $MPP(n)$ is the time it takes to compute the Min-Plus Product between two matrices A and B of size $n \times n$. With some extra work, one can do it in $O(MPP(n))$ time.

One way to compute the MPP is via exponentiation. Let α denote a parameter. To compute $A \star B$, we instead compute $A' \cdot B'$, where

$$A'_{ij} = \alpha^{A_{ij}} \text{ and } B'_{ij} = \alpha^{B_{ij}}$$

Let $C = A' \cdot B'$. We have

$$C_{ij} = \sum_{k=1}^{n} \alpha^{A_{ik}+B_{kj}}$$

If α is sufficiently large, we can deduce $\min_{k=1}^{n} (A_{ik} + B_{kj})$ from C_{ij}, by looking at C_{ij} in base α, but we would need a very big α to be able to do this, and the actual entries of C become extremely large. A number of tricks are needed to exploit this basic idea when the maximum egde weight $M = \max i, j w_{ij}$ is not too large.

One can ask why we focused our attention only in undirected graphs for APSP. It turns out that, for directed graphs, we do not know whether the APSP problem can be solved in $O(n^\omega)$ time even for unweighted graphs.

Open Problem. *Can the APSP problem in unweighted, directed graphs be solved in $O(n^\omega)$ running time?*

See Zwick's slides [Zwick, 2015] for several other open problems.

References

[Alman and Williams, 2018] Alman, J. and Williams, V. (2018). Limits on all known (and some unknown) approaches to matrix multiplication. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 580–591, Los Alamitos, CA, USA. IEEE Computer Society.

[Cohn et al., 2005] Cohn, H., Kleinberg, R., Szegedy, B., and Umans, C. (2005). Group-theoretic algorithms for matrix multiplication. In *Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science*, FOCS 05, page 379388, USA. IEEE Computer Society.

[Cohn and Umans, 2003] Cohn, H. and Umans, C. (2003). A group-theoretic approach to fast matrix multiplication. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, FOCS 03, page 438, USA. IEEE Computer Society.

[Coppersmith and Winograd, 1990] Coppersmith, D. and Winograd, S. (1990). Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251 – 280. Computational algebraic complexity editorial.

[Davie and Stothers, 2013] Davie, A. M. and Stothers, A. J. (2013). Improved bound for complexity of matrix multiplication. *Proceedings of the Royal Society of Edinburgh: Section A Mathematics*, 143(2):351369.

[Fischer and Meyer, 1971] Fischer, M. J. and Meyer, A. R. (1971). Boolean matrix multiplication and transitive closure. In *12th Annual Symposium on Switching and Automata Theory (swat 1971)*, pages 129–131.

[Le Gall, 2014] Le Gall, F. (2014). Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation - ISSAC '14*, pages 296–303.

[Macii, 1995] Macii, E. (1995). A discussion of explicit methods for transitive closure computation based on matrix multiplication. In *Conference Record of The Twenty-Ninth Asilomar Conference on Signals, Systems and Computers*, volume 2, pages 799–801 vol.2.

[Roughgarden, 2014a] Roughgarden, T. (2014a). Beyond worst-case analysis: Three motivating examples. `http://timroughgarden.org/f14/l/l1.pdf`.

[Roughgarden, 2014b] Roughgarden, T. (2014b). Reading in algorithms: Triangle-dense graphs. `http://timroughgarden.org/s14/l/l4.pdf`.

[Seidel, 1995] Seidel, R. (1995). On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of Computer and System Sciences*, 51(3):400 – 403.

[Strassen, 1969] Strassen, V. (1969). Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356.

[Williams, 2012] Williams, V. V. (2012). Multiplying matrices faster than coppersmith-winograd. In *In Proc. 44th ACM Symposium on Theory of Computation*, pages 887–898.

[Williams, 2018] Williams, V. V. (2018). *On some fine-grained questions in algorithms and complexity*, pages 3447–3487. World Scientific.

[Zwick, 2015] Zwick, U. (2015). Lecture slides on graph algorithms via matrix multiplication. `http://www.cs.tau.ac.il/~zwick/Adv-Alg-2016/Matrix-Graph-Algorithms.pptx`.