2MMD30: Graphs and Algorithms Lecture 13 by Jesper Nederlof, 30/03/2016

Matrix Multiplication and applications

In this lecture we'll look at matrix multiplication and its applications to several graph problems.

Note: Since it's relevant for this lecture let us remark something about the computational model often used: it is most common to assume we are working in the RAM model with word size $O(\log(n))$; i.e. elementary operations such as additions and comparisons on objects represented with $O(\log(n))$ bits take constant time. One motivation for this is that otherwise it is not even clear whether running a for-loop with empty body and n iterations takes $O(n)$ time since we need to increase the counter of the iteration every time.

13.1 Integer and Matrix Multiplication

13.1.1 Integers

The usual algorithm for multiplying two b-digit[1] integers as illustrated in Figure 13.1 takes $O(b^2)$ time.

Figure 13.1: Multiplying large numbers as learned in primary school

$$
\begin{array}{rr}
 & 9\,6\,2\,5\,8\,7\,2\,3\,4 \\
\times & 9\,8\,2\,4\,5\,1\,6\,5\,3 \\
\hline
 & 2\,8\,8\,7\,7\,6\,1\,7\,0\,2 \\
 & 4\,8\,1\,2\,9\,3\,6\,1\,7\,0 \\
 & 5\,7\,7\,5\,5\,2\,3\,4\,0\,4 \\
 & 9\,6\,2\,5\,8\,7\,2\,3\,4 \\
 & 4\,8\,1\,2\,9\,3\,6\,1\,7\,0 \\
 & 3\,8\,5\,0\,3\,4\,8\,9\,3\,6 \\
 & 1\,9\,2\,5\,1\,7\,4\,4\,6\,8 \\
 & 7\,7\,0\,0\,6\,9\,7\,8\,7\,2 \\
8\,6\,6\,3\,2\,8\,5\,1\,0\,6 & \\
\hline
9\,4\,5\,6\,9\,5\,4\,1\,9\,1\,9\,9\,9\,7\,8\,0\,2 &
\end{array}
$$

[1] Of course, the integers will actually be represented in binary, so it will require $\log_2 10^b$ bits to represent b but note that for us there's not much difference since this change of representation will only influence constant factors in the running time

This can be quite costly when we have long integers. Note that the output has at most $2b$ digits, so there is no reason we can do significantly faster than this[2].

We'll now see a very elegant algorithm that actually does better. If we want to multiply two b-bit integers x, y, we can write $x = x_1 + 10^{b/2}x_2$, $y = y_1 + 10^{b/2}y_2$ where x_1, x_2, y_1, y_2 are integers on at most $b/2$ digits and have

$$xy = (x_1 + 10^{b/2}x_2)(y_1 + 10^{b/2}y_2) = m_1 + m_2 10^{b/2} + m_3 10^b,$$

where $m_1 = x_1y_1$ $m_2 = x_1y_2 + x_2y_1$, $m_3 = x_2y_2$. So we reduced the computation of one product of b-digit integers to 4 products of integers on $b/2$ digits. If we use a recursive algorithm based on this observation, its running time $T(b)$ thus would satisfy $T(b) = 4T(\lceil b/2 \rceil) + O(b)$ time[3], so $T(b) = O(b^2)$ and we obtained yet another quadratic time algorithm.

However, note that $m_2 = (x_1 + x_2)(y_1 + y_2) - m_1 - m_3$, so we only need three multiplications (x_1y_1,x_2y_2 and $(x_1 + x_2)(y_1 + y_2)$), so only three recursive calls! Then we get a running time of $T(b) = 3T(\lceil b/2 \rceil) + O(b)$ which solves to $O(3^{\log_2(b)})$ which is $O(b^{\log_2(3)}) \approx O(b^{1.58})$ time[4].

13.1.2 Matrices

As usual, we denote matrices with capital letters. We'll refer to entries of the matrices with lower case letters. If $A \in \mathbb{Z}^{l \times m}$ and $B \in \mathbb{Z}^{m \times n}$, recall the matrix product $C = AB \in \mathbb{Z}^{l \times n}$ is defined by $c_{ik} = \sum_{j=1}^{m} a_{ij}b_{jk}$, i.e. c_{ij} equals the inner-product of the vector $(a_{i1}, \ldots, a_{im})$ and $(b_{1k}, \ldots, b_{mk})$. Let us focus on the situation where $A, B \in \mathbb{Z}^{n \times n}$ and all entries are $O(n)$ so every arithmetic operation takes $O(1)$ time. How fast can we compute AB? If we just follow the definition, we can do this in $O(n^3)$ time, and several scientists thought for a long time that one cannot do much faster that this. However, in 1969 Strassen surprised everybody by showing one can do asymptotically faster. The idea is very similar to Karatsuba! Divide the matrices into 4 submatrices of size at most $\lceil n/2 \rceil \times \lceil n/2 \rceil$, multiply the submatrices, then recombine the resulting matrices:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \cdot \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

Thus, instead of multiplying two $n \times n$ matrices we only need to multiply 8 pairs of $\lceil n/2 \rceil \times \lceil n/2 \rceil$ matrices. A recursive algorithm based on this would have a running time $T(n)$ satisfying $T(n) \leq 8T(\lceil n/2 \rceil) + O(n^2)$ which solves to $O(n^3)$ again.

As in the algorithm by Karatsuba the surprising part comes from saving a multiplication in a base case. Strassen managed to come up with an approach to multiply 2×2 matrices with only 7 multiplications! The multiplications are as follows:

$$M1 : (A + D)(E + H) \qquad M5 : (A + B)H$$
$$M2 : (C + D)E \qquad M6 : (C - A)(E + F)$$
$$M3 : A(F - H) \qquad M7 : (B - D)(G + H)$$
$$M4 : D(G - E)$$

[2]This is exactly what the famous scientist Kolmogorov was wondering, but in 1952 he conjectured one cannot compute the product in $\Omega(b^2)$ time. In 1960 he stated this as open problem and a week later the 23-years-old Karatsuba came up with his $O(b^{\log_2(3)})$ algorithm!

[3]$T(b) = \sum_{i=0}^{\lceil \lg b \rceil} 4^i O(b)/2^{i=O(b)} \sum_{i=1}^{n} 2^i = O(b^2)$, or one could use the 'Master Theorem'`https://en.wikipedia.org/wiki/Master_theorem`

[4]There are much faster algorithms known relying on the famous 'Fast Fourier Transformation'.

Then, it can be verified[5] that

$$\begin{bmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{bmatrix} = \begin{bmatrix} M1+M4-M5+M7 & M3+M5 \\ M2+M4 & M1-M2+M3+M6 \end{bmatrix}$$

Thus, instead of multiplying two $n \times n$ matrices we only need to multiply 7 pairs of $\lceil n/2 \rceil \times \lceil n/2 \rceil$ matrices. A recursive algorithm based on this would have a running time $T(n)$ satisfying $T(n) \leq 7T(\lceil n/2 \rceil) + O(n^2)$ which solves to $O(n^{\log_2(7)}) \approx O(n^{2.81})$.

There has been a long line of research after Strassen's breakthrough, and since we still do not know how fast exactly we can multiply matrices it is common to denote ω for the matrix multiplication exponent: ω is the smallest real number such that two $n \times n$ matrices can be multiplied in $O(n^\omega)$ time. Note that trivially $\omega \geq 2$, and by reducing the number of multiplications needed for multiplying matrices of finite dimension combined with the above divide&conquer scheme, researchers managed to prove that $\omega \leq 2.37$. Many researchers believe that $\omega = 2$, but this is unclear still.

13.2 Counting and Finding Triangles

For a square $n \times n$ matrix, define $\mathsf{trace}(A) = \sum_{i=1}^{n} a_{ii}$. If $G = (V, E)$, recall that a triangle of G is a triple of vertices $u, v, w \in V$ such that $(u, v), (u, w), (v, w) \in E$.

Suppose we have a graph G on n vertices with adjacency matrix A. If $B = A^2$, we have that $b_{ik} = \sum_{j=1}^{n} a_{ij} a_{jk}$, so b_{ik} equals the number of j such that $(v_i, v_j), (v_j, v_k) \in E$, i.e. the number of walks on 2 edges from v_i to v_k. Similarly, if $C = A^3$ then $c_{il} = \sum_{j,k=1}^{n} a_{ij} a_{jk} a_{kl}$ equals the number of walks on 3 edges from v_i to v_l. Thus the trace of A^3 equals the number of walks on 3 edges that end where they started, and this is exactly six times the number of triangles (since we can start on any of the three vertices and can traverse the triangle in two directions). So the following algorithm counts the number of triangles in $O(n^\omega)$ time:

Algorithm #`triangle`(G) — The adjacency matrix of G is A
Output: The number of triangles of G
1: Compute A^3 with two matrix multiplications
2: **return** $\mathsf{trace}(A^3)/6$

Algorithm 1: Detecting / Counting triangles in $O(n^\omega)$ time.

Sometimes we also would like to know the weight of a maximum weight triangle. Suppose we are given an undirected graph $G = (V, E)$ with weights $\omega : E \to \{1, \ldots, W\}$. Suppose we define $a_{i,j} = n^{3\omega(v_i, v_j)}$, and let $C = A^3$ as before.

Then $c_{ii} = \sum_{j,k=1}^{n} n^{3\omega(v_i,v_j)+3\omega(v_j,v_k)+3\omega(v_k,v_i)}$. We see that if a triangle containing v_i of weight OPT exists $c_{ii} > n^{3OPT}$, and if no such triangle exists, $c_{ii} < n^2 n^{3(OPT-1)} = n^{3OPT-1}$. Thus the maximum weight of a triangle containing v_{ii} can be read off from c_{ii}. Thus the following algorithm determines the weight of a maximum weight triangle:

[5] Indeed, finding the minimum number of multiplications is an instance of an NP-complete problem. It takes us a few minutes to check what Strassen came up with, but it is quite an achievement that he managed to find these multiplications!

Algorithm `maxTriangle`$(G = (V, E), \omega : E \to \{1, \ldots, W\})$
Output: The maximum weight of a triangle of G, -1 if none exists.
1: Let $V = \{v_1, \ldots, v_n\}$
2: **for** $(v_i, v_j) \in E$ **do**
3: $\quad a_{i,j} = n^{3\omega(v_i, v_j)}$
4: Compute $C = A^3$ using two matrix multiplications
5: Set $max = -1$
6: **for** $i = 1, \ldots, n$ **do**
7: $\quad$ Let c be the max integer such that $c_{i,i} \geq n^c$ $O(\log_2^2(c_{i,i}))$ time by trying all c
8: $\quad$ **if** $c/3 > max$ **then** set $max = c/3$
9: **return** c

Algorithm 2: Finding the weight of a maximum weight triangle in $O^*(n^\omega (W \log_2(n))^2)$ time.

All integers in this computation will be at most $n^{O(W)}$, so they can be represented with $O(W)\log_2(n)$ bits so arithmetic operations on these integers can be performed in at most $O(W^2 \log_2^2 n)$ time, so the running time of this algorithm can be upper bounded by $O(n^\omega (W \log_2(n))^2)$ time.

13.3 Exponential Time Algorithm for Max 2-Sat

In the Max 2-SAT problem, we are given a 2-CNF formula φ on variables $v_1, \ldots, v_n$ and we would like to find an assignment $x \in \{0,1\}^n$ that satisfies as many clauses of φ as possible. Can we solve this problem in $O^*((2-\epsilon)^n)$ time, for some $\epsilon > 0$? In turns out we can, but curiously the only currently known algorithm relies on matrix multiplication:

Algorithm `max2Sat`(φ) Assume the number n of variables of φ divides 3.
Output: An assignment maximizing the number of satisfying clauses.
1: Partition the variables of φ in V_1, V_2, V_3, $|V_1| = |V_2| = |V_3|$.
2: Create an instance $(G = (V, E), \omega)$ of maximum weight triangle as follows:
3: Add vertices $\{v_x^i\}_{x \in \{0,1\}^{n/3}}$, for $i = 1, \ldots, 3$, to V
4: **for** $x, y \in \{0,1\}^{n/3}$ and $1 \leq i \leq 3$ **do**
5: $\quad$ Let $j = (i+1)\%3$
6: $\quad$ Let p be the partial assignment obtained by assigning x to variables V_i and y to variables V_j
7: $\quad$ Set $c_1 =$ number of clauses with only variables in V_i satisfied by p
8: $\quad$ Set $c_2 =$ number of clauses with a variable in V_i and in V_j satisfied by p
9: $\quad$ Set $\omega(v_x^i, v_y^j) = c_1 + c_2$
10: **return** `maxTriangle`(G, ω)

Algorithm 3: Solving Max 2-Sat in $O^*(2^{\omega n/3})$ time.

The algorithm arbitrarily partitions the variables into three parts, creates an exponentially sized graph with a vertex for every partial assignment to one of the three variables sets. An edge then encodes a partial assignment with $2n/3$ variables set, and the weight of an edge is exactly the number of clauses this partial assignment satisfies. We see there is a one-to-one mapping between the triangles of G and the total assignments of φ. Moreover, every satisfied clause is accounted for

exactly once in in the weight of a triangle since the variables of any clause are either contained in V_1, V_2, V_3 or if they are not in either $V_1 \cup V_2$, $V_2 \cup V_3$ or $V_3 \cup V_1$.

The running time of the algorithm is $O^*(2^{\omega n/3})$: Line 10 clearly is the bottleneck and since $|V| = 2^{n/3}$ and W is $\texttt{poly}(m)$, `maxTriangle` runs in $O^*(2^{\omega n/3})$ time.

13.4 All Pairs Distances

In this sections and the next two sections, we'll focus on the computing the shortest paths between all pairs of vertices of a given connected undirected unweighted graph. This will improve over Exercise 6.8 in this more restricted case.

Given a graph $G = (V, E)$ with $V = \{v_1, \ldots, v_n\}$ we want to compute for every $1 \leq i < j \leq n$ the distance $d_{i,j}$ which is the number of edges on a shortest path from v_i to v_j.

Let G' be the graph obtained from G by placing an edge between every pair of vertices that are at distance 1 or 2 in G. We directly see that the adjacency matrix A' of G' can be computed in $O(n^\omega)$ time since a'_{ij} is 1 is $a_{ij} = 1$ or if $c_{ij} > 0$ where $C = A^2$. Motivated by the close relation between the adjacency matrices of G and G', G' is often called the 'square' of G.

The *diameter* $\mathsf{diam}(G)$ of a graph G is the number of edges on the longest shortest path between two vertices. We see that $\mathsf{diam}(G') = \lceil \mathsf{diam}(G)/2 \rceil$. Moreover, if $\mathsf{diam}(G) \leq 2$ we see that $D = 2A' - A$. This suggests a recursive approach by 'repeated squaring' to reduce the diameter: if we would be able to compute D from D' quickly, we can recurse to reduce to $\mathsf{diam}(G) \leq 2$ in only $\log_2(n)$ recursive steps.

If we let d'_{ij} denote the number of edges on a shortest path between v_i and v_j in G' we see that

$$d_{ij} = \begin{cases} 2d'_{ij}, & \text{if } d_{ij} \text{ is even} \\ 2d'_{ij} - 1, & \text{if } d_{ij} \text{ is odd,} \end{cases}$$

or equivalently put, $d'_{ij} = \lceil d_{ij}/2 \rceil$. Thus if we would know the parities of d_{ij} for every i, j we would be able to compute d_{ij} from d'_{ij}, so we can restrict our attention to this!

Since $d_{kj} - 1 \leq d_{ij} \leq d_{kj} + 1$, if d_{ij} is odd there must be some k such that $d_{kj} < d_{ij}$ and $d'_{ij} = \lceil d_{ij}/2 \rceil$ we note the following:

Observation 13.1. *For every $1 \leq i < j \leq n$ we have that*

- *If d_{ij} is even, then $d'_{kj} \geq d'_{ij}$ for every neighbor k of i in G,*
- *If d_{ij} is odd, then $d'_{kj} \leq d'_{ij}$ for every neighbor k of i in G. Moreover, there exists a neighbor k of i such that $d'_{kj} < d'_{ij}$.*

If we sum these inequalities over all neighbors k we obtain

$$d_{ij} \text{ is even if and only if } \sum_{k \in N_G(v_i)} d'_{kj} \geq d'_{ij} |N_G(v_i)|. \tag{13.1}$$

Here $N_G(v_i)$ and $d_G(v_i)$ denote the neighborhood and degree of v_i in G, just to be explicit about which graph we are looking at and avoid confusion with the distances.

Recall we were trying to compute the d_{ij}'s from the d'_{ij} and G. Then the right hand side of (13.1) can easily be computed in $O(n^2)$ time for all i, j but what about the left hand side?

Specifically, we would like to know $l_{ij} = \sum_{k \in N_G(v_i)} d'_{kj}$ for every i, j. We recognize that $L = AD'$, so we can compute these values with a matrix multiplication!

Summarizing, the algorithm is as follows:

Algorithm $\texttt{APD}(A)$ Assumes A is the adjacency matrix of an undirected graph G
Output: Matrix D with d_{ij} being the distance from i to j in G.
1: $Z = A^2$
2: Construct A' such that $a'_{ij} = 1$ if and only if $i \neq j$ and $a_{ij} = 1$ or $Z_{ij} = 1$
2: $a'_{ij} = 1$ if and only if $d(v_i, v_j) \leq 2$
3: **if** $a'_{ij} = 1$ for every $i \neq j$ **then return** $2A' - A$ $\mathsf{diam}(G) \leq 2$
4: $D' = \texttt{APD}(A')$
5: $L = AD'$
6: **for** $1 \leq i < j \leq n$ **do**
6: $l_{ij} \geq d'_{ij}|N_G(v_i)|$ iff d_{ij} even
7: **if** $l_{ij} \geq d'_{ij}|N_G(v_i)|$ **then**
8: $d_{ij} = 2d'_{ij}$
9: **else**
10: $d_{ij} = 2d'_{ij} - 1$
11: **return** D

Algorithm 4: Compute APD matrix.

Now the running time $T(n, d)$ of $\texttt{APD}(A)$ where d is the diameter of the graph represented by A satisfies

$$T(n, d) \leq 2n^\omega + T(n, \lceil d/2 \rceil) + O(n^2),$$

since the involved integers will never exceed n^2 (n^2 may happen in AD'). This recursion resolves to $T(n, d) \leq O(n^\omega \lg(n))$.

13.5 Boolean Matrix Product and Witness Matrices

If $A, B \in \{0, 1\}^{n \times n}$ we define $A \odot B$ to be the matrix C with $c_{ij} = \bigvee_k (a_{ik} \wedge b_{kj})$. Then we can compute C from A, B in $O(n^\omega)$ time by computing $C' = AB$ and setting $c_{ij} = 1$ if $c'_{ij} > 0$ and 0 otherwise. But ideally, we would also like to have a *witness* k ready for every ij such that $c_{ij} = 1$, where a witness is an integer k such that $a_{ik} \wedge b_{kj}$. The matrix that contains in every cell a 0 if no witness exists and a witness otherwise is called a *witness matrix*. We'll focus now on computing such a witness matrix fast

Suppose that for every i, j there is at most one witness. Then the witness matrix is just $\hat{A}B$ where $\hat{a}_{ij} = 0$ if $a_{ij} = 0$ and $\hat{a}_{ij} = j$ otherwise. But of course in general this may not be the case. However, somewhat similar to the isolation lemma we saw previous time, we can use randomization to reduce to this case. Our isolation lemma is somewhat more elementary now:

Lemma 13.1. *Suppose $X \subseteq U$, $|U| = n$ and $|X| = w$. If $R \subseteq U$ is picked uniformly from the set of all subsets of U of size r, and $n/2 \leq wr \leq n$ then* $\Pr[|X \cap R| = 1] \geq 1/8$.

Proof.

$$\begin{aligned}
\frac{\binom{w}{1}\binom{n-w}{r-1}}{\binom{n}{r}} &= w\frac{r!}{(r-1)!}\frac{(n-w)!}{n!}\frac{(n-r)!}{(n-w-r+1)!} && \binom{n}{d} = \tfrac{n!}{d!(n-d)!} \\
&= wr\left(\prod_{i=0}^{w-1}\frac{1}{n-i}\right)\left(\prod_{j=0}^{w-2}(n-r-j)\right) \\
&= \frac{wr}{n}\prod_{j=0}^{w-2}\frac{n-r-j}{n-1-j} \\
&= \frac{wr}{n}\prod_{j=0}^{w-2}\left(\frac{n-1-j}{n-1-j}-\frac{r-1}{n-1-j}\right) \\
&\geq \frac{wr}{n}\prod_{j=0}^{w-2}\left(1-\frac{r-1}{n-w}\right) && \text{using } (r-1)w \leq n-w \text{ since } wr \leq n;\ wr \geq n/2 \\
&= \frac{1}{2}\left(1-\frac{1}{w}\right)^{w-1} \geq \frac{1}{2}\left(1-\frac{1}{w}\right)^{w} && \text{using } 1-x/2 \geq 2^{-x} \text{ for } x \in [0,1] \text{ with } x = 2/w \\
&\geq 1/8.
\end{aligned}$$

□

Note that of course we do not know w, but since it is good enough to have $n/(2w) \leq r \leq n/w$ we only need to try powers of 2 for r:

Algorithm $\texttt{wm}(A, B)$ Assume A, B are $n \times n$ matrices with entries from $\{0, 1\}$.
Output: Witness matrix for the Boolean matrix product $A \odot B$, with constant probability.

1: **for** $t = 0, \ldots, \lceil \log_2 n \rceil$ **do**
2: $r = 2^t$
3: **for** $i = 1, \ldots, 32\lceil \log_2 n \rceil$ **do**
4: Pick random set $R \subseteq \{1, \ldots, n\}$ uniformly from the set of all r-sized subsets of $\{1, \ldots, n\}$
5: Compute $\hat{A}^R$ with $\hat{a}^R_{ij} = j$ if $j \in R$ and $a_{ij} = 1$ and $\hat{a}^R_{ij} = 0$ otherwise.
6: Compute $Z = \hat{A}^R B$
7: **for** $1 \leq i, j \leq n$ **do**
8: **if** $z_{i,j} > 0$ and z_{ij} is witness **then** set $w_{ij} = z_{i,j}$
9: **return** W

Algorithm 5: Compute the witness matrix, with constant error probability.

Note that if ij has a unique witness k in R, then $z_{ij} = \hat{a}^R_{ik} b_{kj} = k$ so indeed conditioned on this event we correctly set w_{ij} on Line 8. So we just need to lower bound the probability that for every pair ij where a witness exists, there will be a unique witness in R at some iteration. Line 3 is repeated $16\lceil \log n \rceil$ times, and we are successful for fixed ij with probability at least $1/8$ by Lemma 13.1 when $n/(2w) \leq r \leq n/w$, which happens for some iteration since for every x there is a power of two between x and $2x$. Thus, the probability that we never have a unique witness for

fixed i, j is at most

$$\left(1 - \frac{1}{8}\right)^{32 \log n} \leq e^{-4 \log n} \leq 2^{-4 \log n} = n^{-4}.$$

Thus by the union bound, the probability that we do not find a witness for some i, j is at most n^{-2}, and hence $\texttt{wm}(A, B)$ constructs a witness matrix for $A \odot B$ with probability at least $1 - \frac{1}{n^2}$. The running time is easily upper bounded by $O(n^\omega \log_2^2(n))$.

13.6 Computing the Successor Matrix

Given undirected graph $G = (V, E)$ with $V = \{v_1, \ldots, v_n\}$, the successor matrix S of G is an $n \times n$ matrix with entries from $\{1, \ldots, n\}$ in which s_{ij} is the index k of a neighbor v_k of v_i that lies on a shortest path from v_i to v_j.

We will now combine the algorithms from the last two sections two compute the successor matrix fast. A sensible first step is to compute $\texttt{APD}(A)$, since it already provides a lot of information on the shortest paths:

$$d_{ij} = d_{kj} + 1 \text{ and } (i, j) \in E \leftrightarrow k \text{ is on shortest path from } i \text{ to } j.$$

In particular, this means that if we define $D^{(l)}$ by letting $d_{ij}^{(l)} = 1$ if $d_{ij} = l$ and 0 otherwise, then in the witness matrix of $A \odot D^{(l-1)}$ we can find successors for every i, j such that $d_{i,j} = l$. However, iterating for every l will be too slow for our purposes. This is fixed by observing that since $d_{kj} - 1 \leq d_{ij} \leq d_{kj} + 1$, $d_{ij} = d_{kj} + 1$ is equivalent with $d_{ij} \equiv_3 d_{kj} + 1$:

$$d_{ij} = d_{kj} + 1 \text{ and } (i, j) \in E \leftrightarrow d_{ij} \equiv_3 d_{kj} + 1 \text{ and } (i, j) \in E \leftrightarrow k \text{ is on shortest path from } i \text{ to } j.$$

Therefore, we can still use the above idea but only need to do 3 iterations, except we use a mild variant of $D^{(s)}$:

Algorithm $\texttt{succ}(A)$ — Adjacency matrix of a graph G
Output: The successor matrix of G with constant probability.
1: $D = \texttt{APD}(A)$
2: **for** $s = 1, 2, 3$ **do**
3: Construct $D^{(s)}$ with $d_{ij}^{(s)} = 1$ iff $d_{ij} \equiv_3 s$
4: Compute $W^{(s)} = \texttt{wm}(A, D^{(s)})$
5: Construct S with $s_{ij} = w_{ij}^{((d_{ij}-1)\%3)}$

Algorithm 6: Compute the successor matrix in $O(n^\omega \log_2^2 n)$ time.

The algorithm clearly runs in $O(n^\omega \log_2^2 n)$ time (with $\texttt{wm}$ being the bottleneck) given the previous sections, and S is a successor matrix since for every i, j we have that $k = w_{ij}^{((d_{ij}-1)\%3)}$ satisfies $d_{kj} \equiv_3 d_{ij} - 1$ (since $d_{kj}^{(s)} = 1$ which is because k is a witness for the entry ij in $A \odot D^{(s)}$) so $d_{kj} = d_{ij} - 1$ and $a_{ik} = 1$ since k (because, again, k is a witness for the entry ij in $A \odot D^{(s)}$).

13.7 Exercises

Excercise 13.1. The successor matrix is an implicit representation of the shortest paths. A more explicit representation of the shortest paths would be a sequence of vertices describing the shortest path for every pair. Give a graph with $\Omega(n^2)$ pairs of distance $\Omega(n)$ from each other to show that such a representation will require $\Omega(n^3)$ to compute.

Excercise 13.2. In Exercise 5.3 you gave an algorithm for determining whether a clique of size k exists with running time $O(n^k k^2)$. Unfortunately, researchers believe that clique parameterized by k is not FPT, and in this exercise you are asked to find the currently asymptotically fastest algorithm for this problem for the case when k is a multiple of 3: show how to determine whether a clique on k vertices exists in $O(n^{\omega k/3}\texttt{poly}(k))$ time in this case. Hint: start with $k = 3, 6$.

Excercise 13.3. In the Max-Cut problem we are given an undirected graph $G = (V, E)$ and need to find a partition of V into V_1, V_2 maximizing $E \cap (V_1 \times V_2)$. It is known that Max-Cut is NP-complete. Show that MAX-2-SAT is NP-complete and solve Max-Cut in $O^*(2^{\omega n/3})$ time.

Excercise 13.4. It is a big open problem to solve MAX-3-SAT (which has the same definition of MAX-2-SAT except we are given a 3-CNF formula) in $O^*((2-\epsilon)^n)$ time for some $\epsilon > 0$. Why can we not use the approach from Section 13.3 for MAX-2-SAT for this?

Excercise 13.5. The n'th Fibonacci number f_n is defined as follows: $f_1 = 1$,$f_2 = 1$ and for $n > 2$, $f_n = f_{n-1} + f_{n-2}$. Show that $\begin{pmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$. Show how to compute the n'th Fibonacci number using $O(\log_2(n))$ arithmetic operations. Why is this 'running time' misleading?

Excercise 13.6. The transitive closure of a directed acyclic graph $G = (V, E)$ is the graph $G^* = (V, E^*)$ where $(u, v) \in E^*$ whenever there is a path from u to v in G. Compute the transitive closure of a directed acyclic graph in $O(n^\omega \log(n))$ time.

Excercise 13.7. Why doesn't Algorithm `succ` work for directed graphs?