Lecture 1 (06.03.2013) Author: Jarosław Błasiok

Algebraic method for Shortest Paths problems

1 Introduction

In the following lecture we will see algebraic algorithms for various shortest-paths problems. As the base of many of them there is a matrix multiplication algorithm.

From now on we will denote ω to be the smallest constant s.t. two matrices of size $n \times n$ can be multiplied in time $\mathcal{O}(n^{\omega})$, as long as elements of those matrices form a ring. This means $\mathcal{O}(n^{\omega})$ operations on the underlying ring structure — if this operation could not be assumed to be constant time, whole algorithm complexity becomes $\mathcal{O}(n^{\omega}R)$ — where R is single operation.

Note that demanding from operations on elements to fulfill ring axioms is sometimes troublesome. Nevertheless, sometimes we will work out a way to reduce those degenerate matrix multiplying problem to a matrix multiplying over a ring — so we can use matrix multiplying algorithm running in time $\mathcal{O}(n^{\omega})$.

At the point of this lectures ω is proven to be less then 2.3727 (i.e. there exist an algorithm which can multiply two matrices in time $\mathcal{O}(n^{2.3727})$.

During this lecture we will focus on the following problems:

All Pairs Shortest Paths (APSP)
Input: Graph G (directed/undirected, weighted/unweighted)
Question: For every pair of vertices find the length of the cheapest/shortest path between them. Sometimes: find the first edge of this path

Remark 1. If an algorithm returns for each pair of vertices (v_1, v_2) the first edge on the shortest path from v_1 to v_2, this path itself could be easily reconstructed.

Classic way of various APSP problems is Floyd-Warshall algorithm. I will shortly describe it here, for the sake of completeness:

- Initialize a matrix: D_{ij} of size $n \times n$: $D_{ij} := w(v_i, v_j)$ if there is edge from v_i to v_j, $D_{ij} := \infty$ in the opposite case.
- For every vertex v_k, and for every pair v_i, v_j do relaxation: if $D_{ij} > D_{ik} + D_{kj}$, assign $D_{ij} := D_{ik} + D_{kj}$.

2 Boolean Matrix Multiplication

Before we will discuss further APSP problems and our approach, lets take a look on the following problem, which will be a useful tool for our algorithms.

Boolean Matrix Multiplication
Input: A, B — boolean matrices of size $n \times n$.
Question: C — boolean matrix of size $n \times n$, s. t. $c_{ij} = (A \cdot B)_{ij} = \bigvee_k a_{ik} \wedge b_{kj}$

Unfortunately $(\{\bot, \top\}, \vee, \wedge)$ is not ring — we cannot use $\mathcal{O}(n^\omega)$ matrix multiplication directly. Nevertheless this problem can be reduced to matrix multiplication over a ring: take $\tilde{A}$, s.t.

$$\tilde{a}_{ij} = \begin{cases} 0 & \text{for} \quad a_{ij} = \bot \\ 1 & \text{for} \quad a_{ij} = \top \end{cases}$$

$\tilde{B}$ is generated from B in same fashion, now take $\tilde{C} = \tilde{A} \cdot \tilde{B}$ (where $\tilde{A}$ and $\tilde{B}$ are matrices over $(\mathbb{Z}, \cdot, +)$). Here you can get C back from $\tilde{C}$ taking $\bot$ instead of 0, and $\top$ instead of integers greater than 0.

This leads us to the following corollary

Corollary 2. *There is an $\mathcal{O}(n^\omega)$ algorithm for the Boolean Matrix Multiplication problem.*

Now we can see a simple application of this tool.

Transitive closure
Input: Directed graph $G = (V, E)$
Question: $G^* = (V, \{(u, v) : u \rightsquigarrow v \text{ in } G\})$

Bruteforce algorithm involves n graph searching, each in time $\mathcal{O}(m)$, which gives $\mathcal{O}(mn)$ in total, that is $\mathcal{O}(n^3)$ for dense graphs. Can we do better than this?

Consider boolean matrix A, of size $n \times n$ defined:

$$A_{uv} = \begin{cases} \top & (uv) \in E \\ \bot & \text{in opposite case} \end{cases}$$

Now take $X = A \vee I$, so $X_{uv} = \top$ if and only if there is a path from u to v of size not exceeding one. It is easy to see by induction, that $X^k_{ij} = \top$ if and only if there is a path from u to v of length not exceeding k, i.e. for $k \geq n$, we know that X^k is the incidence matrix of G^* — we will denote this matrix as X^*. Take just: $X_0 = X, X_1 = X_0^2, X_2 = X_1^2, \cdots$ — repeating boolean matrix multiplication $\log n$ times, one can calculate X^*, and hence G^* itself in time $\mathcal{O}(n^\omega \log n)$.

We would like to get rid of this logarithm factor, and provide $\mathcal{O}(n^\omega)$ algorithm for transitive closure.

Assume for simplicity that n is a power of two (one can always add at most n isolated vertices to make this assumption true). Let $X = \left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right]$, and $X^* = \left[\begin{array}{c|c} E & F \\ \hline G & H \end{array}\right]$.

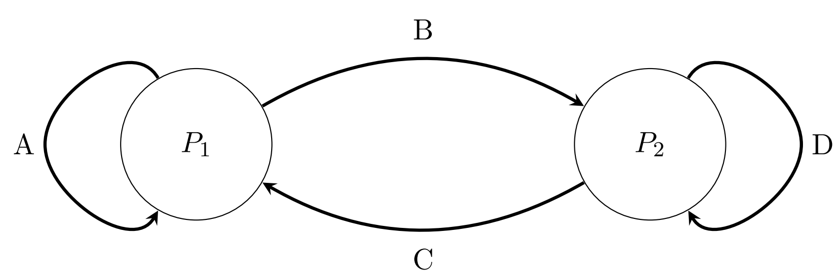

Figure 1: Graph decomposition corresponding to the block matrix $\begin{bmatrix} A & B \\ C & D \end{bmatrix}$

We will prove that

$$X^* = \begin{bmatrix} (A \vee BD^*C)^* & EBD^* \\ D^*CE & D^* \vee GBD^* \end{bmatrix} \tag{1}$$

Indeed, let P_1 be the set of vertices corresponding to the first $n/2$ rows (and columns as well), P_2 — corresponds to the last $n/2$ rows (columns). By abuse of notation A, B, C, D, E, F, H, G will be treated sometimes as sets of edges/paths in G.

Consider for example a path from v_1 in P_1 to v_2 in P_2. It can be decomposed into parts which are either edges from A (inside P_1) or consists of: one edge in B (leading to P_2), then some number of edges from D (inside P_2), then an edge from C back to P_1. This exactly corresponds to the block $(A \vee BD^*C)^*$ of the matrix X.

Now any path from P_1 to P_2 may be decomposed as a path from E (from the first vertex, to the last vertex on the path which is in P_1, then an edge from B, and lastly a path inside P_2 of the form D^*. A similar argument holds for the last two blocks of the matrix X.

In order to compute X^* for a matrix of size $n \times n$ it suffices to compute D^*, and then $(A \vee BD^*C)*$ — transitive closure of matrices of size $\frac{n}{2} \times \frac{n}{2}$ (plus constant number of BMM). Recursive formula for the time complexity of this algorithm is:

$$T(n) = 2T(n/2) + 6\text{BMM}(n/2) + \mathcal{O}(n^2)$$

This recursion leads to $T(n) = \mathcal{O}(\text{BMM(n)}) = \mathcal{O}(n^\omega)$ (the last step proven during exercises).

3 APSP

3.1 Undirected, unweighted case

Consider now the All Pairs Shortest Paths problem on an undirected, unweighted connected graph G. Our goal is to achieve $\tilde{\mathcal{O}}(n^\omega)$ time complexity. This algorithm was first given by Seidel in [1].

Definition 3. For an unweighted graph $G = (V, E)$ we denote $G^k = (V, \{(u, v) : d_G(u, v) \leq k\})$.

Note that, if $A(G)$ is boolean adjacency matrix of a graph G, we have the following property: $A(G^k) \vee I = (A(G) \vee I)^k$.

Now the outline of the algorithm could be summarized as follows:

- Compute G^2.
- Count all pairs shortest paths in G^2.
- Fix them a little, to get APSP in G.

Remark 4. Recursion depth here is at most $\log n$, as $G^{2^{\log n}} = G^n = K_n$ (we assume G is connected), and APSP is trivial there.

Remark 5. The "Compute G^2" step can be done in $\mathcal{O}(n^\omega)$ time, by boolean matrix multiplication.

Lemma 6. $2d_{G^2}(u, v) - 1 \leq d_G(u, v) \leq 2d_{G^2}(u, v)$

Proof. If there is a path of length p in G^2 it induces a path of length at most $2p$ in G (every edge from G^2 becomes a path of length at most two). On the other hand if there is a path of length $2k$ or $2k - 1$ in G, one can find a corresponding path of length k in G^2: for a path $v_1v_2v_3 \ldots v_p$ in G, $v_1v_3v_5 \ldots v_p$ is a path in G^2. □

One needs to know for every u, v in G, whether this length is $2d_{G^2}(u,v)$ or $2d_{G^2}(u,v) - 1$. Following lemmas give us simple criteria in terms of distances in G^2.

Lemma 7. *If $d_G(u,v) = 2d_{G^2}(u,v)$, then for every $w \in N_G(v)$ we have $d_{G^2}(u,w) \geq d_{G^2}(u,v)$.*

Proof. It follows simply from Lemma 6. Indeed: for w being a neighbour of v, surely from triangle inequality for d we have $d_G(u,w) \geq d_G(u,v) - 1$. Now, using Lemma 6, one can conclude:

$$d_{G^2}(u,w) \geq \frac{1}{2}d_G(u,w) \geq \frac{1}{2}(d_G(u,v) - 1) = \frac{1}{2}(2d_{G^2}(u,v) - 1) = d_{G^2}(u,v) - \frac{1}{2}$$

As $d_{G^2}(u,w)$ is an integer, it has to be at least $d_{G^2}(u,v)$ — this is what was claimed in the statement of the lemma. □

Lemma 8. *If $d_G(u,v) = 2d_{G^2}(u,v) - 1$, then for every $w \in N_G(v)$ we have $d_{G^2}(u,w) \leq d_{G^2}(u,v)$, furthermore there exist a vertex $w \in N_G(v)$, s. t. $d_{G^2}(u,w) < d_{G^2}(u,v)$*

Proof. For the first part of the lemma: for every $w \in N_G(v)$ we have:

$$d_{G^2}(u,w) \leq \frac{1}{2}(d_G(u,w) + 1) \leq \frac{1}{2}(d_G(u,v) + 1 + 1) = \frac{1}{2}(2d_{G^2}(u,v) + 1) = d_{G^2}(u,v) + \frac{1}{2}$$

Again, as both or those distances are integers, we have the desired inequality.

For the existence part of lemma: take w as first vertex on the shortest path from v to u. Now $d_G(u,w) = d_G(u,v) - 1$, hence $d_G(u,w) = 2d_{G^2}(u,v) - 2$, and from Lemma 6, we have $d_{G^2}(u,w) = d_{G^2}(u,v) - 1$. □

Now we need a way to determine for every pair (u,v) whether we are in the case from Lemma 7, or in the case from Lemma 8. We will see that it can be done again using matrix multiplication.

Let D_2 be a matrix representing distances in G^2, A — the adjacency matrix of G. Take $Y = AD_2$, and observe that

$$Y_{vu} = \sum_{w \in N(v)} d_{G^2}(w,u).$$

Now it is easy to distinguish between vertices of even and odd distances in G. Namely: if $Y_{uv} < \deg_G(v)(D_2)_{uv}$ we know that $d_G(u,v) = 2d_{G^2}(u,v) - 1$, and $d_G(u,v) = 2d_{G^2}(u,v)$ in the opposite case.

3.2 Undirected, weighted case

Seidel algorithm has been generalized to weighted graph G, were each edge has assigned integer weight from the set $\{0, \ldots, M\}$. This generalization is due to Shoshan and Zwick [2]. On those graphs one can solve APSP in time $\tilde{\mathcal{O}}(Mn^\omega)$. There is no known algorithm for that problem running in $\mathcal{O}(n^{3-\epsilon})$ for general weights.

3.3 Min-Plus product

In order to solve the APSP problem efficiently it would suffice to calculate the so called Min-Plus product: $(A * B)_{ij} = \min_k(A_{ik} + B_{kj})$. Indeed: if A were a matrix with edge weights as its elements (and 0 on diagonal), $A * A$ would have shortest paths of length not exceeding 2. The question of APSP in a given graph reduces to solving $\mathcal{O}(\log n)$ Min-Plus product instances.

Unfortunately $(\mathbb{Z}, \min, +)$ does not form a ring, so we cannot take the standard matrix multiplication algorithm to calculate the min-plus product of two matrices right away.

Those two problems are actually related more closely: as one can consider MPP as a special case of directed, weighted APSP. Indeed: given a matrix A of size $n \times m$ and a matrix B of size $m \times l$, one can consider a graph with $n + m + l$ vertices as on Figure 2, with vertices: $\{u_1, \ldots u_n, v_1, \ldots v_m, w_1, \ldots w_l\}$. Take an edge with weight a_{ij} from the vertex u_i to v_j, and an edge of weight b_{ij} from v_i to w_j for every i and j. Now the length of the shortest paths from u_i to w_j is exactly the value of the corresponding element in the min-plus product: $(A * B)_{ij}$.

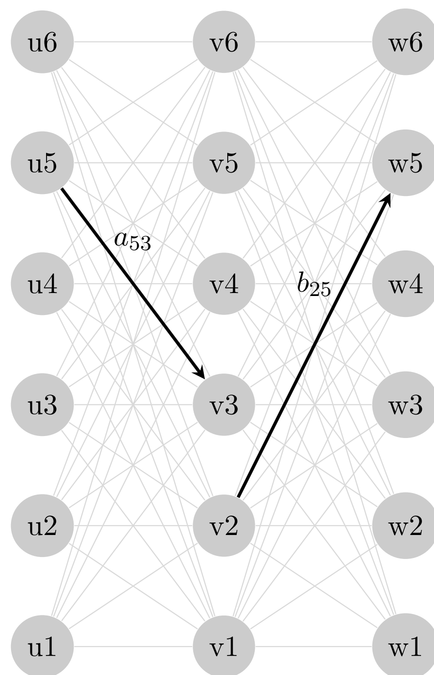

Figure 2: Graph for min-plus product $A * B$

Claim 1. *If elements of matrices A and B are integers from $\{0, \ldots, M\}$ or ∞, one can find min-plus products of those matrices in time $\mathcal{O}(Mn^\omega)$.*

Consider matrix A', where

$$a'_{ij} = \begin{cases} x^{a_{ij}} & \text{for} \quad a_{ij} < \infty \\ 0 & \text{for} \quad a_{ij} = \infty \end{cases}$$

Let matrix B' be construdcted from B in similar manner. Now every entry of $A'B'$ is a polynomial, say: $(A'B')_{ij} = c_{n_1}x^{n_1} + c_{n_2}x^{n_2} + \ldots$, for $n_1 < n_2 < \ldots$. It is easy to check, straight from definition that that lowest degree of non-vanishing monomial (i.e. n_1) is in fact demanded value of corresponding element in min-plus product $(A * B)_{ij}$.

If elements of input matrices are polynomials of degree not exceeding M it is possible to compute product of those matrices in time $\tilde{\mathcal{O}}(Mn^\omega)$ — all appearing polynomials are of degree at most $2M$, and one can add those two in linear time (just as-is), and multiply them in $\mathcal{O}(M \log M)$ by Fast Fourier Transform. Now using $\mathcal{O}(n^\omega)$ ring operations one can multiply two matrices over that ring.

Unfortunately this does not give us immediately $\tilde{\mathcal{O}}(Mn^{\omega})$ time algorithm for undirected weighted (with bounded weights) APSP. If we take $A * A$, then the resulting matrix has elements of value not exceeding $2M$, after the next step we get a matrix with entries of value $4M$ — this blows up rapidly, and we can solve MPP problem only when guarantied that the value of elements is bounded. To overcome this difficulty, Shoshan and Zwick calculated simultaneously $\lfloor \log_2 n \rfloor$ most significant bits of every distance, and remainders of this distance modulo M — they used min-plus product for both those tasks. It obviously is enough to determine distance itself, as for every u, v $d(u, v) \le (n-1)M$. For more information check [2].

3.4 Directed unweighted case

The following approach for directed, unweighted APSP problem has been proposed by Zwick in [3].

Given graph $G = (V, E)$, let A be matrix defined as follows:

$$a_{ij} = \begin{cases} 0 & \text{for} & i = j \\ 1 & \text{for} & v_i v_j \in E \\ \infty & \text{in opposite case} & \end{cases}$$

For a matrix D, and $P, T \subset \{1, 2, \ldots, n\}$ let $D_{P,T}$ be the matrix created from D by taking rows P and columns T.

Also $\text{crop}(D, s)$ will denote D with substituted ∞ instead of elements larger then s. That is $D' = \text{crop}(D, s)$ if

$$d'_{ij} = \begin{cases} d_{ij} & \text{for} & d_{ij} \le s \\ \infty & \text{for} & d_{ij} > s \end{cases}$$

Consider the following algorithm:

$D \leftarrow A$
for $i := 1 \to \lceil \log_{3/2} n \rceil$ **do**
  $s := \lceil (3/2)^i \rceil$
  B :=sample of $9(n \log n)/s$ vertices from V, taken uniformly at random
  $D := \min(D, D_{V,B} * D_{B,V})$
  $D := \text{crop}(D, (3/2)^{i+1})$
end for

Note that in the matrix $D_{V,B}$ there might be ∞ elements, which can be simulated by big enough integers - the details are left to the reader.

In the time complexity analysis of our algorithm we strongly depend on the following result by Coppersmith:

Theorem 9. *One can multiply a matrix of size $n \times p$ by a matrix of size $p \times n$ in time*

$$\mathcal{O}(n^{1.85} p^{0.54} + n^{2+o(1)})$$

In particular, that means that if $p < n^{0.29}$ one can multiply two matrices of that size in time $n^{2+o(1)}$.

Now for the min-plus product $D_{V,B} * D_{B,V}$, you can either solve it by brute-force algorithm (in time $n^2 \cdot (n \log n)/s$) or by our reduction to fast matrix multiplication, and then using Coppersmith method (in time $\tilde{\mathcal{O}}(n^{1.85} \cdot (n \log n/s)^{0.54} \cdot s)$). Note that the factor s comes from our reduction of MPP to matrix multiplication — time complexity of solving MPP depends on an upper bound on values of elements, but here elements does not exceed s (unless they are infinite) thanks to cropping those elements.

Now the time complexity of a single step is

$$\mathcal{O}(\min(n^3 \log n/s, n^{1.85}(n \log n/s)^{0.54} s)) = \mathcal{O}(n^{2.58})$$

We are left to prove that the above algorithm computes APSP with high probability.

Lemma 10. *Given that after phase i all paths of length not exceeding $\left(\frac{3}{2}\right)^i$ are good, after phase $i+1$ all paths of length not exceeding $\left(\frac{3}{2}\right)^{i+1}$ are good with probability at least $1 - \frac{1}{n}$.*

Proof. Consider two vertices u, v such that $\left(\frac{3}{2}\right)^i \leq d(u,v) \leq \left(\frac{3}{2}^{i+1}\right)$, and take any shortest path from u to v. Take Q as "middle" $\frac{1}{2}(\frac{3}{2})^i$ vertices on this path, i.e. in a way that there is at most $\frac{1}{2}(\frac{3}{2})^i$ vertices from u to the first vertex in Q, and there is at most $\frac{1}{2}(\frac{3}{2})^i$ vertices from the last vertex in Q to v. It can be done, as the length of this path does not exceed $\left(\frac{3}{2}\right)^{i+1}$. Now we know that the distance from u to any of vertices in Q is good, and distances from Q to v is good as well. If only B had nonempty intersection with Q, we would get proper value of $d(u,v)$ by a min-plus product.

For a single vertex $q \in Q$, the probability that it is not taken into B is $1 - \frac{9 \log n}{s}$. Now the probability that none vertex of Q is taken is at most:

$$\left(1 - \frac{9 \log n}{s}\right)^{s/3} \leq e^{-3 \log n} = \frac{1}{n^3}$$

Now by a union bound over all pairs u, v, the probability that we fail in step i is at most $\frac{1}{n}$ (given that we haven't failed in previous steps). $\square$

Now correctness of this algorithm follows simply from the lemma above, namely:

$$\begin{aligned}
&\mathbb{P}[\text{algorithm failed to compute APSP}] = \\
&= \sum_i \mathbb{P}[\text{everything went good until phase } i\text{, and algorithm failed in phase } i] \\
&\leq \sum_i \mathbb{P}[\text{algorithm failed in phase } i | \text{everything was good until then}] \\
&\leq \sum_i \frac{1}{n} = \frac{\log_3 n}{n}
\end{aligned}$$

As usual, we can achieve arbitrary low probability of error by repeating whole algorithm c times. The above algorithm can be derandomized, the details can be found in [3].

References

[1] R. Seidel. On the all-pairs-shortest-path problem. In *STOC*, pages 745–749, 1992.

[2] A. Shoshan and U. Zwick. All pairs shortest paths in undirected graphs with integer weights. In *FOCS*, pages 605–615, 1999.

[3] U. Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *J. ACM*, 49(3):289–317, 2002.