# Homework II, Advanced Algorithms 2019

Due on Friday May 24 at 17:00 (upload one solution per group on moodle). Solutions to many homework problems, including problems on this set, are available on the Internet, either in exactly the same formulation or with some minor perturbation. It is *not acceptable* to copy such solutions. It is hard to make strict rules on what information from the Internet you may use and hence whenever in doubt contact Ola Svensson. You are, however, allowed to discuss problems in groups of up to three students; it is sufficient to hand in one solution per group.

**The last problem (Problem 5) is not graded but is a very good exercise!**

**1 Weighted Frequency Estimation.** *(25 pts)* Recall the following notation from the lecture notes. Let $\sigma = \langle \sigma_1, \sigma_2, \ldots, \sigma_m \rangle$ be a stream consisting of $m$ elements, where each element takes a value from the universe $[n] = \{1, 2, \ldots, n\}$. Then the *frequency vector* is defined to be $\mathbf{f} = (f_1, f_2, \ldots, f_n)$ where $f_i = |\{j : \sigma_j = i\}|$ is the number of items of value $i$.

In class, we saw a one-pass streaming algorithm for estimating $F_2 = \sum_{i=1}^{n} f_i^2$. In this problem, we are going to generalize this result to the weighted case. That is, every value $i \in [n]$ has a non-negative weight $w_i$ and the goal is to estimate

$$W_2 = \sum_{i=1}^{n} w_i f_i^2 \,.$$

Design and analyze a one-pass streaming algorithm, parameterized by $\epsilon, \delta > 0$, such that:

1. It uses memory at most $O\left(\log(n) \cdot \frac{1}{\epsilon^2} \cdot \log(1/\delta)\right)$.
2. It outputs an estimate $\hat{W}_2$ of $W_2$ satisfying

$$\Pr[(1-\epsilon)W_2 \leq \hat{W}_2 \leq (1+\epsilon)W_2] \geq 1 - \delta \,.$$

**2 Submodular function maximization.** *(25 pts)* In this problem we will analyze a streaming algorithm for monotone submodular function maximization subject to a cardinality constraint $k$. Assuming knowledge of the optimal value OPT[1], it processes the stream of elements as follows:

**Initialization:** $S = \emptyset$

**Process item** $e$**:** At the arrival of item $e$, if

$$|S| < k \qquad \text{and} \qquad f(e|S) \geq \frac{\text{OPT}}{2k}$$

then add it to our (partial) solution $S$, i.e., set $S = S \cup \{e\}$.

Assuming that $f : 2^N \to \mathbb{R}$ is a monotone submodular function that is normalized (i.e., $f(\emptyset) = 0$), your task is to prove that this simple algorithm produces a set $S$ satisfying $f(S) \geq \text{OPT}/2$ by considering two cases:

**2a** *(10 pts)* If we have $|S| = k$ at the end of the stream, then $f(S) \geq \text{OPT}/2$.

**2b** *(15 pts)* If we have $|S| < k$ at the end of the stream, then $f(S) \geq \text{OPT}/2$.

*(Interesting fact: it is an open problem whether the above is the best possible streaming algorithm for monotone submodular function maximization subject to a cardinality constraint.)*

[1] Assuming knowledge of OPT can be removed by paying a logarithmic factor using standard tricks. We omit those details here and assume that the algorithm knows the exact value of OPT.

**3 Exact Matching in Bipartite Graphs.** *(25 pts)* Design and analyze a randomized polynomial-time algorithm for the following problem:

| **The Exact Matching Problem on Bipartite Graphs** |
|---|
| **Input:** An $n$-by-$n$ bipartite graph $G = (V, E)$, an integer $k$, a subset $R \subseteq E$ of "red" edges. |
| **Output:** A perfect matching $M$ with exactly $k$ red edges, i.e., $\lvert M \cap R \rvert = k$. If no such matching exists, output "NONE". |

Since your algorithm will be randomized it is allowed to fail to find a matching $M$ (even if it exists) with a small probability say 1/10. Note: if you manage to devise a *deterministic* polynomial-time algorithm, call me because you have solved a long standing open problem!

To devise your randomized algorithm you may use the following facts without giving a proof:

1. Let $A$ be the $n \times n$ matrix of indeterminates $(X_e)_{e \in E}$ and $Y$ defined by

$$A_{u,v} = \begin{cases} Y \cdot X_{\{u,v\}} & \text{if } \{u,v\} \in R, \\ X_{\{u,v\}} & \text{if } \{u,v\} \in E \setminus R, \\ 0 & \text{otherwise.} \end{cases}$$

   Then

$$\det(A) = \sum_{i=0}^{n} Y^i \left( \sum_{M \in \mathcal{M}_i} \operatorname{sgn}(M) \prod_{e \in M} X_e \right),$$

   where $\operatorname{sgn}(M) \in \{\pm 1\}$ for each perfect matching and $\mathcal{M}_i$ contains those perfect matchings $M$ with exactly $i$ red edges, i.e., $\mathcal{M}_i = \{M : M \text{ is a perfect matching with } \lvert M \cap R \rvert = i\}$.

2. Recall polynomial interpolation: for a univariate polynomial $p(x)$ of degree $d$, given $d+1$ evaluations $(x_0, p(x_0)), (x_1, p(x_1)), \ldots, (x_d, p(x_d))$ where no two $x_i$'s are the same, we can in polynomial-time find the unique coefficients $\alpha_0, \alpha_1, \ldots, \alpha_d$ such that

$$p(x) = \sum_{i=0}^{d} \alpha_i x^i .$$

**4** *(25 pts)* **Karger's min-cut algorithm.** In this problem you are supposed to implement Karger's min-cut algorithm. Specifically, given a graph as input you should output the size and the number of min cuts. It is important that you implement your own version of Karger's algorithm and do not simply copy an implementation from the web.

**Implementation Hint:** If edge contractions are properly implemented using a suitable data structure like UNION-FIND, a single round of Karger's can be implemented in $O(n \log n + m)$ time where $m$ is the number of edges. Although this is comparable to $O(n^2)$ for dense graphs, it is significantly faster for sparse graphs. For this problem, all input graphs will satisfy $n \leq 100$ and $m \leq 400$, and consequently, all graphs with a relatively large number of vertices will be sparse.

We have prepared an automatic correction tool at CodeForces where you can submit and test your code. Please see the separate instructions document for this problem to see how to access this system. In case of any questions regarding this, please contact TA buddhima.gamlath@epfl.ch.

**5 (NOT GRADED) Tutte matrix.**

In this problem, you will implement a randomized algorithm to check whether a given graph has a perfect matching.

Let $G = (V, E)$ be an undirected graph where $V = \{1, 2, \ldots, n\}$. We define *Tutte's matrix* of the graph $G$ as the symbolic matrix $A$ defined as follows:

$$A_{ij} = \begin{cases} x_{ij} & \text{if edge } \{i,j\} \in E \text{ and } i < j \\ -x_{ij} & \text{if edge } \{i,j\} \in E \text{ and } j < i \\ 0 & \text{otherwise.} \end{cases}$$

The matrix $A$ is said to be *skew-symmetric* – that is, $A^\top = -A$.

Recall the following theorem from Lecture 12:

**Theorem 1.** *The graph $G$ has a perfect matching if and only if* $\det(A)$ *is not identically zero.*

Also recall that the determinant of an $n \times n$ matrix $A$ is given by the following formula, where the summation is over all permutations of $[n] = \{1, 2, \ldots, n\}$:

$$\det(A) = \sum_{\sigma:[n]\to[n]} \operatorname{sgn}(\sigma) \prod_{i\in[n]} A_{i,\sigma(i)}.$$

Here $\operatorname{sgn}(\sigma)$ is a function that can take values $+1$ or $-1$. Notice that when $A$ is Tutte's matrix of $G$, $\det(A)$ is a polynomial in variables $x_{ij}$ for $1 \le i < j \le n$ whose degree is at most $n$.

Let $S$ be a set of at least $kn$ distinct real numbers for some $k > 1$. If you substitute all variables $x_{ij}$ with values $s_{ij}$ chosen independently and uniformly at random from the set $S$ and evaluate $\det(A)$, by Theorem 1 and the Schwartz-Zippel lemma, you have the following:

$$\Pr\left[\det(A)|_{x_{ij}=s_{ij}} = 0 \middle| G \text{ has no perfect matching}\right] = 1$$

and

$$\Pr\left[\det(A)|_{x_{ij}=s_{ij}} = 0 \middle| G \text{ has a perfect matching}\right] \le \frac{1}{k}.$$

For any $0 < \delta < 1$, by repeating this $\log_k(1/\delta)$ times independently, you can conclude whether the graph $G$ has a perfect matching or not with a success probability of at least $(1 - \delta)$. I.e., if the graph $G$ has a perfect matching, then with probability at least $(1-\delta)$, at least one evaluation of $\det(A)$ will be non-zero.

## Computational Considerations

**Evaluating the Determinant** If you try to evaluate the determinant naively using the recursive definition (i.e., as a linear combination of the determinants of minors), it will take exponential time. An easier way to check whether the determinant is nonzero is to check whether all the diagonal entries of the row echelon form of the matrix (obtained using Gauss-Jordan elimination) are nonzero. If at least one of the diagonal entries of its row echelon form is zero, then the matrix is non-invertible and its determinant is zero.

**Precision Issues** In theory, if $\det(A)$ is identically zero, no matter what real numbers you substitute, it must evaluate to zero. However, computers have finite precision. As a result, when a result of a certain arithmetic expression should be exactly zero, it might still have a small non-zero value when it is computed using a finite precision.

To get rid of precision issues, it is a good idea to consider integer arithmetic (in contrast to double precision) over a finite field. For this problem, you can think of doing all computations modulo some large prime (say 1 000 000 009 – the favorite prime of many competitive programmers).

**Arithmetic Modulo p** For a prime $p$, modulo arithmetic operations of addition, subtraction, and multiplication are similar to their usual counterparts except that we take the modulo-$p$ remainder of the result. A few examples of modulo 7 arithmetic are as follows:

$$\begin{aligned} 5 + 6 \bmod 7 &= 11 \bmod 7 = 4 \\ 15 - 1 \bmod 7 &= 14 \bmod 7 = 0 \\ 5 \times 6 \bmod 7 &= 30 \bmod 7 = 2. \end{aligned}$$

For division, recall that it can be viewed as multiplication by the multiplicative inverse. Hence, $a \div b \bmod p$ is defined as $(a \times (b^{-1} \bmod p) \bmod p)$, and it is defined only when $b \neq 0 \mod p$. For $b \neq 0 \mod p$, $b^{-1} \bmod p$ is the (unique) integer $x \in \{0, \ldots, p-1\}$ such that $b \cdot x = 1 \mod p$. Note that for $b \neq 0 \mod p$, $b^{-1} \bmod p = b^{(p-2)} \bmod p$ by Fermat's Little Theorem.

**Schwartz-Zippel Lemma in a Finite Field** You may have already noticed that, if we restrict the computations to a finite field, the Schwartz-Zippel Lemma as we learned in Lecture 12 no longer applies. Luckily, there is a version of the same lemma for finite fields.

**Lemma 1.** *Let $p(x_1, \ldots, x_n)$ be a non-zero polynomial of $n$ variables with degree $d$ over a finite field $F$. Let $S \subseteq F$ be a nonempty set. If we independently assign values from $S$ to $x_1, \ldots, x_n$ uniformly at random, then* $\Pr[p(x_1, \ldots, x_n) = 0] \leq d/|S|$.

## Implementation Guide

At this point, you have all the ingredients needed to implement a randomized algorithm for checking whether a given graph has a perfect matching or not. Here are some useful implementation hints.

- The graphs that will be given as inputs will have at most 100 vertices. Hence, the maximum degree of $\det(A)$ is at most 100, and using a finite field of size at least 200 is sufficient to get 1/2 success probability. Remember that you can always repeat your algorithm several times (with independently chosen random values in each iteration) to boost the success probability.
- Choose a sufficiently large prime for your finite field arithmetic. Using a larger finite field allows you to choose your random values from a larger set and consequently increase the success probability of a single round. A field modulo a prime $p$ has $p$ elements in it. Two large primes that you can use are 1 000 000 007 and 1 000 000 009.
- First implement the modulo multiplicative inverse using Fermat's Little Theorem. I.e., write a function `mod_inv(x, p)` that computes $x^{-1} \bmod p$. You can also Google other methods to implement this. Keep in mind that, for a large $p$, calculating $x^{p-2} \bmod p$ can be done efficiently using the repeated squaring technique (`https://en.wikipedia.org/wiki/Exponentiation_by_squaring`).

- When multiplying three or more numbers modulo $p$, to avoid overflows, always perform the mod operation after each multiplication. Also make sure that the magnitudes of the initial numbers are smaller than $p$, specially when $p$ is large. Notice that $p^2$ should fit into the range of the integer data type that you're using.
  - To implement $c = a * b \bmod p$, make sure that $-p < a, b < p$. If not, make $a = a\%p$ and $b = b\%p$. Then $c = (a * b)\%p$.
  - To implement $c = x * y * z$, do $c = (((x * y)\%p) * z)\%p$.
- Whenever a division is involved, remember to use multiplication by the modulo multiplicative inverse instead. I.e., Instead of $c = (a/b)\%p$, use $c = (a * \texttt{mod\_inv}(b, p))\%p$.
- Using the modulo arithmetic, write a function that performs Gaussian elimination to compute the row echelon form of a Matrix $A$ (`https://en.wikipedia.org/wiki/Row_echelon_form`) modulo $p$. The determinant of $A$ is non-zero if and only if all diagonal entries of its row echelon form are non-zero.
- Finally, independently substitute variables $x_{ij}$ for $1 \leq i < j \leq n$ with elements chosen uniformly at random from (a subset of) the field, and check whether the determinant of Tutte's matrix is zero or not using the function you wrote in the previous step. You may repeat this step several times to boost the probability of success.
- **Warning** to Python programmers: Since integers in Python have arbitrary precision, you may be tempted not to worry about overflow issues, and decide not to take modulo $p$ value after each arithmetic operation. Although this might not cause errors, the results of the arithmetic operations may get very large, and consequently, the subsequent arithmetic operations on those numbers might take a long time to complete! As such, it is safer to take the modulo $p$ after each arithmetic operation and keep the values small at all times.
- **Warning** to Java, C, C++ programmers: Beware of overflows! For example, `long long x = 1000000 * 1000000;` may produce an incorrect result in $C$ although the variable `x` can clearly hold the result. The reason is that the multiplication is done on 32-bit integers. The solution is to make sure at least one of the operand is considered as a 64-bit integer. I.e., `long long x = 1000000LL * 1000000;` or `long long x = (long long)1000000*1000000;` will do. In Java, the same will happen with data type `long`.

We have prepared an automatic correction tool at CodeForces where you can submit and test your code. Please see the separate instructions document for this problem to see how to access this system. In case of any questions regarding this, please contact TA buddhima.gamlath@epfl.ch.