Ling Li, `ling@cs.caltech.edu`

F.1 Perfect matching with min max weight

Assume the number of vertices in G is even, i.e., $|V| = 2n$. Otherwise there doesn't exist a perfect matching. Let $m = |E|$. A naive algorithm to do the job is

Init: Sort the weights $w(e)$ for $e \in E$ in ascending order. Let $k = n$, and $G' = (V, E')$, where E' only contains k edges with the first k smallest weights. Go to **Match**.

Match: Apply the algorithm of Micali and Vazirani for finding maximum matching in general graphs to G'. If the maximum matching is a perfect matching, output it and halt. Otherwise go to **Next**.

Next: If $k \geq m$, output no perfect matching and halt. Otherwise, $k \leftarrow k + 1$, and add the edge with the k^{th} smallest weight (which is the edge with smallest weight which is not in E') into E'. Go to **Match**.

The correctness of this algorithm is very obvious. The worst time is $(m - n)$ times $O(m\sqrt{n})$ which is the time of the algorithm of Micali and Vazirani, plus the time for sorting and adding edges, which is $O(m \log m)$. Thus the total time is $O(m^2\sqrt{n})$.

Another way is to convert the problem into a minimum weighted perfect matching problem, by setting the weight of edge e as $n^{w(e)}$. Let G' denote the transformed graph. A minimum weighted perfect matching M in G' corresponds to a perfect matching M in G with minimum $\max_{e \in M} w(e)$, since the weights sum is solely decided by the maximum w in the matching. We use n as the base in case that there are (at most) $n - 1$ edges with the same w. The Edmonds' blossom algorithm can find a minimum weighted perfect matching in $O(n^2 m)$. Thus the total time for this algorithm is also $O(n^2 m)$.

F.2 Directed matching

Idea: Construct a bipartite graph with twice number of vertices as in G and reduce the problem to a perfect matching in that bipartite graph.

Algorithm:

Trans: $L = \emptyset$, $R = \emptyset$, $E' = \emptyset$. For each vertex $v \in V$, $L \leftarrow L \cup \{\mathtt{l}_v\}$, $R \leftarrow R \cup \{\mathtt{r}_v\}$. For every edge $(u,v) \in E$, $E' \leftarrow E' \cup \{(\mathtt{l}_u, \mathtt{r}_v)\}$. We get a bipartite graph $G' = (L, R, E')$. Go to **Match**.

Match: Apply the algorithm of Hopcroft and Karp for unweighted matching in bipartite graphs to G'. If there is no perfect matching, declare there is no directed matching in G and halt. Otherwise go to **TransBack**.

TransBack: Let M' denote the perfect matching found in **Match**. $M = \emptyset$. For every edge $(\mathtt{l}_u, \mathtt{r}_v) \in M$, $M \leftarrow M \cup \{(u,v)\}$. Declare $H = (V, M)$ is a directed matching (subgraph) in G and halt.

Correctness proof: After the step **Trans**, we have $L = \{\mathtt{l}_v : v \in V\}$, $R = \{\mathtt{r}_v : v \in V\}$, and $E' = \{(\mathtt{l}_u, \mathtt{r}_v) : (u,v) \in E\}$.

- For any directed matching $H = (V, M)$ in G, the in-degree and out-degree of every vertex in H is 1. Thus we have

  Properties M: For every vertex $u \in V$, there exists one and only one vertex $v \in V$ such that $(u,v) \in M$; For every vertex $v \in V$, there exists one and only one vertex $u \in V$ such that $(u,v) \in M$.

  Construct $M' = \{(\mathtt{l}_u, \mathtt{r}_v) : (u,v) \in M\}$. Thus from $M \subseteq E$, $M' \subseteq E'$. And by the construction of L and R, we have properties similar to those stated above:

  Properties M': For every vertex $\mathtt{l}_u \in L$, there exists one and only one vertex $\mathtt{r}_v \in R$ such that $(\mathtt{l}_u, \mathtt{r}_v) \in M$; For every vertex $\mathtt{r}_v \in R$, there exists one and only one vertex $\mathtt{l}_u \in L$ such that $(\mathtt{l}_u, \mathtt{r}_v) \in M$.

  Hence M is a perfect matching in the bipartite graph G'.

- For any perfect matching M' in G', construct $M = \{(u,v) : (\mathtt{l}_u, \mathtt{r}_v) \in M'\}$. Since E' is constructed from E, it is obviously $M \subseteq E$. And from M' is a perfect matching, we have properties M' above. Thus we also get properties M above. Thus $H = (V, M)$ is a directed matching in G.

Hence finding a directed matching in G is equivalent to finding a perfect matching in G'.

Runtime analysis: Let $n = |V|$ and $m = |E|$. The runtime of **Trans** is $O(n+m)$ and that of **Match** is $O(m\sqrt{n})$. The step **TransBack** takes time $O(n)$, since there are exactly n edges in a perfect matching. Thus the total time is $O(n + m\sqrt{n})$.

Ling Li, ling@cs.caltech.edu

F.3 Vertex-disjoint paths

Main idea: Transform the graph G into a unit capacity graph G' such that any flow in G' consists of vertex-disjoint path flows. And the value of the max flow in G' is the maximum number of vertex-disjoint paths in G.

Algorithm:

Trans: Initially $V' = \emptyset$, $E' = \emptyset$. For every vertex $v \in V$, add two vertices $\mathtt{i}_v$ and $\mathtt{o}_v$ into V', and add an edge $(\mathtt{i}_v, \mathtt{o}_v)$ into E'. For every edge $(u, v) \in E$, add an edge $(\mathtt{o}_u, \mathtt{i}_v)$ into E'. $G' = (V', E', c)$, where $c = 1$ for all edges in E'. Go to **Maxflow**.

Maxflow: Use Dinic's algorithm to get a max flow f from $\mathtt{o}_s$ to $\mathtt{i}_t$ in G'. Output $|f|$ as the maximum number of vertex-disjoint paths from s to t in G.

Correctness proof: After **Trans**, we get $V' = \{\mathtt{i}_v, \mathtt{o}_v : v \in V\}$ and $E' = \{(\mathtt{i}_v, \mathtt{o}_v) : v \in V\} \cup \{(\mathtt{o}_u, \mathtt{i}_v) : (u, v) \in E\}$.

- Let P be any set of vertex-disjoint paths from s to t in G. For each path $(u_0, u_1, \ldots, u_k) \in P$ with $u_0 = s$, $u_k = t$, there is a path $p' = (\mathtt{o}_{u_0}, \mathtt{i}_{u_1}, \mathtt{o}_{u_1}, \ldots, \mathtt{i}_{u_{k-1}}, \mathtt{o}_{u_{k-1}}, \mathtt{i}_{u_k})$ in G' with $\mathtt{o}_{u_0} = \mathtt{o}_s, \mathtt{i}_{u_k} = \mathtt{i}_t$. Let P' be the set of those p' paths. Since paths in P are vertex-disjoint (they do not share vertices other than s, t), paths in P' are also vertex-disjoint. Then P' can be regarded as a collection of vertex-disjoint path flows in G', each path flow having value 1. Thus we get a flow in G' from $\mathtt{o}_s$ to $\mathtt{i}_t$, with value $|P'| = |P|$, the number of paths in P.
- In Homework 13.2, we have shown that for a unit capacity graph with a max flow f, there are $|f|$ edge-disjoint paths from s to t. In G', any edge must have $\mathtt{o}_u$ as one end and $\mathtt{i}_v$ as the other end, for some u and v. Thus any path flow from $\mathtt{o}_s$ to $\mathtt{i}_t$ must be $p' = (\mathtt{o}_s, \mathtt{i}_{u_1}, \mathtt{o}_{u_1}, \ldots, \mathtt{i}_{u_{k-1}}, \mathtt{o}_{u_{k-1}}, \mathtt{i}_t)$, for some u_i. By the construction in **Trans**, there is only one edge from $\mathtt{i}_{u_i}$ to $\mathtt{o}_{u_i}$, thus the 'edge-disjoint' paths in G' are also 'vertex-disjoint'.* Thus for any max flow f in G', there are $|f|$ vertex-disjoint paths from $\mathtt{o}_s$ to $\mathtt{i}_t$ in G'. Those paths correspond to $|f|$ vertex-disjoint paths in G, with the inverse mapping mentioned in the above paragraph.

Thus the maximum number of vertex-disjoint paths in G is just the value of max flow in G'.

Runtime analysis: Let $n = |V|$ and $m = |E|$. The runtime of **Trans** is $O(n+m)$ and after that, $|V'| = 2n$, $|E'| = n+m$. The time for Dinic's algorithm is $O(|E'|\,|V'|^2) = O((n+m)n^2)$. Thus the total runtime is $O(n^2(n+m))$. Or, if the MPM algorithm is used in the step **Maxflow**, the total runtime is $O(n^3 + m)$.

*Even taking into consideration that there may exist path $(\mathtt{o}_s, \mathtt{i}_t)$, that is right since we do not count in $\mathtt{o}_s$ and $\mathtt{i}_t$ as shared vertices for vertex-disjoint paths.

Ling Li, ling@cs.caltech.edu

F.4 Identification and square root

(a) For any quadratic residue $x \in Z_n^*$, x has 4 different square roots in Z_n^*. If we get two of them, r_1, r_2 and $r_1 \not\equiv \pm r_2 \bmod n$, then from $(r_1 + r_2)(r_1 - r_2) = r_1^2 - r_2^2 \equiv 0 \bmod n$, we know $(r_1 + r_2) \bmod n$ is one of p and q, and $(r_1 - r_2) \bmod n$ is the other. Let A be the algorithm assumed in the problem which can compute a square root of $x \bmod n$ in time p, where p is a polynomial of $\log n$. Thus we have the algorithm below:

Loop: Select $r \in_U \{1, 2, \ldots, n-1\}$. If $\gcd(n, r) \neq 1$, output r as p, n/r as q, and halt the algorithm. Otherwise go to **Root**.

Root: Calculate $x \equiv r^2 \bmod n$. Use A to get a square root r' of x. If $r \not\equiv r' \bmod n$, output $(r + r') \bmod n$ as p and $(r - r') \bmod n$ as q, and halt. Otherwise go to **Loop**.

For any selected r, with probability no more than $\frac{1}{2}$ (since the algorithm may halt in **Loop**), the algorithm will halt in **Root** without going back to **Loop**. Thus the expected runtime of this algorithm is no more than

$$p + \frac{p}{2} + \frac{p}{4} + \cdots = 2p,$$

which is also polynomial in $\log n$. (WLOG, we assume that $p > \log^2 n$. Thus the time of gcd and division and multiplication of numbers of $\log n$ bits can be omitted compared to p.)

(b) For $b = 0$, Maggie can select r and compute $x \equiv r^2 \bmod n$. For $b = 1$, Maggie can select y and compute $x \equiv y^2 u^{-1} \bmod n$. Thus if Maggie knew which bit b Victor would send, she could fool Victor. However, she can not know in advance which b Victor will send. Thus to fool Victor no matter what x she sent, Maggie must have the ability to get a pair of y and r in polynomial time such that $x \equiv r^2 \bmod n$ and $u \equiv y^2 x^{-1} \bmod n$. Thus by calculating $a \equiv y r^{-1} \bmod n$, she get $u \equiv y^2 x^{-1} \equiv (y r^{-1})^2 \equiv a^2 \bmod n$. That is, Maggie can compute a square root of u. The total time to calculate a is still polynomial in $\log n$ since to get a from y and r can be done in $O(\log^2 n)$.

(c) I have two readings for this question. One is that Maggie always chooses an r and compute $x \equiv r^2 \bmod n$. The other is that Maggie can use either way in (b) to compute x. For the first case, Maggie can always fool Victor when he chooses $b = 0$. But for $b = 1$, she has to get y such that $u \equiv y^2 x^{-1} \bmod n$ in polynomial time in order to fool Victor. For the second case, she can fool Victor if b is the 'correct' bit with respect to her choice of x. That is, if she chooses $x \equiv y^2 u^{-1} \bmod n$ and $b = 1$, or if she chooses $x \equiv r^2 \bmod n$ and $b = 0$, she can fool Victor. For the other b, she also has to get the pair of y and r such that $x \equiv r^2 \bmod n$ and $u \equiv y^2 x^{-1} \bmod n$. Thus for either reading, in order to fool Victor, the probability that Maggie has to know y and r simultaneously is $\frac{1}{2}$.[†]

From the analysis in (b), knowing y and r simultaneously leads to solving a square root of u. Since the probability that Maggie can fool Victor is at least $\frac{3}{4}$, then the probability

[†] Here we assume b is uniformly chosen by Victor. If b is not uniformly chosen, for example, $P(b = 0) > \frac{3}{4}$, then under this situation, Maggie can always fool Victor with probability larger than $\frac{3}{4}$ by using $x \equiv r^2 \bmod n$, without the ability to compute a square root of u.

Ling Li, ling@cs.caltech.edu

that she can compute a is at least $\frac{1}{2}$ (otherwise the probability of fooling Victor is less than $\frac{1}{2} + \frac{1}{2} \times \frac{1}{2} = \frac{3}{4}$).

Thus she can use an algorithm similar to that in (a), to randomly select x and compute a. The expected runtime is twice the time she uses to calculate a in one run, which is polynomial. Thus she can compute a square root of u in expected polynomial time.

An extension of this question is that if Maggie can compute a square root of u in polynomial time with probability at least $\frac{1}{p}$, where p is any polynomial in $\log n$, then she can calculate a square root of u in expected polynomial time.

(d) Since we are pretty sure that there is no algorithm to factor n in time polynomial in $\log n$, the probability that Maggie can compute a square root of u in polynomial time is not nonnegligible (otherwise from (c) and (a), n can be factorized in expected polynomial time.) Maggie can only guess a b, select a strategy to compute x according to b, and hope that Victor will also choose that b. Thus the probability that Maggie can fool Victor in one trial is at most $\frac{1}{2}$. Hence for T trials, the probability that Maggie fools Victor is at most 2^{-T}.

However, we assumed in above discussion that the probability of Maggie fooling Victor in those T trials are independent. This is true if the x used in every trial is different, which requires T is small relative to n (such as $T = \log^c n$ for some constant c). If T is really large, say $T > n$, then during those trials some x and b pairs would appear several times and thus Maggie can reuse some of her answers in previous trials. Hence the probability that Maggie fools Victor would be larger than 2^{-T}, for T large relative to n.

Ling Li, ling@cs.caltech.edu

F.5 SOP and PRG

Since f is a strong one-way function, by definition there exists a PPT F such that $F(x) = f(x)$. Thus we can design a statistical test T such that $T(x_1 \cdots x_n x_{n+1} \cdots x_{2n}) = 1$ if and only if $F(x_1 \cdots x_n) = x_{n+1} \cdots x_{2n}$. Obviously T is a PPT.

For $h(x) = (f(x), f(f(x)))$, we have $T(h(x))$ is always 1, i.e.,

$$P_{x \leftarrow U_n}(T(h(x)) = 1) = 1.$$

However, it is obvious that

$$P_{x \leftarrow U_{2n}}(T(x) = 1) = \frac{1}{2^n}.$$

Thus $h(x)$ is not a PRG.

By the way, there is a theorem saying $h(x) = (f(x), b(x))$ is a PRG, if f is an SOP and b is a hard-core bit for f.

Ling Li, ling@cs.caltech.edu

F.6 Yet another random walk

(a) Let P be the set of all primes p less than n. Then any ℓ ($1 \le \ell < n$) can be written as

$$\ell = \prod_{p \in P} p^{m(p)}.$$

By the fundamental theory of arithmetic (unique factorization), to get $L = \ell$, the random walk must stay $m(p)$ time steps at p for every $p \in P$. Thus the probability of $L = \ell$ is

$$\prod_{p \in P} \frac{p-1}{p^{m(p)+1}} = \prod_{p \in P} \frac{1}{p^{m(p)}} \prod_{p \in P} \frac{p-1}{p} = \frac{1}{\ell} B(n).$$

(b) Step (i) consists of a random walk and multiplying of $p^{m(p)}$ over all primes p less than n. As proved in Homework 16.4(d), the random walk takes expected time $1 + H(n-1)$. Since $H(n-1) < 1 + \log(n-1) < 2\log n$ for $n \ge 2$, the time of this part is just $O(\log n)$.

To show that step (i) can be done in expected time polynomial in $\log n$, we need to show that deciding whether a number p is a prime or not could be done in expected time polynomial in $\log n$. This can be implied by the fact that PRIMES is in **co-RP** $\cap$ **RP**. PRIMES $\in$ **co-RP** means there exists a randomized algorithm, running in expected polynomial time (in $\log n$, which is the number of bits in p), which for p is a prime, announces p is a prime, and for p is not a prime, announces with probability at least $\frac{1}{2}$ that p is not a prime. And symmetrically, PRIMES $\in$ **RP** means there exists a randomized algorithm, running in expected polynomial time (in $\log n$), which for p is not a prime, announces p is not a prime, and for p is a prime, announces with probability at least $\frac{1}{2}$ that p is a prime. Thus we can run both randomized algorithms simultaneously, to decide whether p is a prime.[‡] Thus the time for step (i) is really polynomial in $\log n$.

However, this can't be implied only from that PRIMES is in **co-RP** $\cap$ **NP**, since we need both algorithms together to decide the primality of p.

(c) Assume that there is a positive c such that $B(n) \ge \frac{1}{c \lg n}$. The algorithm reaches step (iii) iff $L \le n-1$. From (a), the probability is

$$\sum_{\ell=1}^{n-1} \frac{1}{\ell} B(n) = H(n-1)B(n) \ge \frac{H(n-1)}{c \lg n}. \tag{1}$$

For $n \ge 3$, we have $(n-1)^2 > n$. Thus $H(n-1) > \log(n-1) > \frac{1}{2}\log n$, and that probability for $n \ge 3$ is at least

$$\frac{\frac{1}{2}\log n}{c \lg n} = \frac{1}{2c \lg e}.$$

[‡]For a prime p, the first algorithm will say p is a prime, and the second algorithm may say p is a not a prime. For p is not a prime, the first algorithm may say p is a prime, and the second algorithm will say p is not a prime. Thus we can not tell with full confidence that p is a prime or not. However, p can be decided if the first algorithm say it is not a prime, or the second algorithm say it is a prime. Then if thinking in the expected time, we can use those two algorithms to decide the primality of p with full confidence.

Ling Li, `ling@cs.caltech.edu`

(d) When the algorithm reaches step (iii), L can be any of $1, 2, \ldots, n-1$. For a specific ℓ in the range $1 \leq \ell < n$, the probability that ℓ is generated in step (i) and then passes step (iii) is

$$\frac{1}{\ell} B(n) \frac{\ell}{n-1} = \frac{B(n)}{n-1}. \tag{2}$$

Hence the probability that the algorithm goes on to step (iv) rather than returning to (i) is

$$\sum_{\ell=1}^{n-1} \frac{B(n)}{n-1} = B(n).$$

The probability that the algorithm can reach (iii) is $H(n-1)B(n)$ (see (1)).Thus the conditional probability that the algorithm goes on to (iv) given it has reached (iii) is

$$\frac{B(n)}{H(n-1)B(n)} = \frac{1}{H(n-1)}.$$

(e) From (b), step (i) can be implemented in expected time polynomial in $\log n$. Denote that time by T. From (c), the algorithm reaches (iii) with probability at least $\frac{1}{2c \lg e}$. Then the expected time to reach (iii) is at most

$$T + T\left(1 - \frac{1}{2c \lg e}\right) + T\left(1 - \frac{1}{2c \lg e}\right)^2 + \cdots = \frac{T}{1 - \left(1 - \frac{1}{2c \lg e}\right)} = (2c \lg e)T.$$

From (d), when the algorithm reaches step (iii), it goes on to step (iv) with probability $\frac{1}{H(n-1)}$. By similar computation, we get the expected time for the algorithm to reach step (iv), that is, to terminate, is at most (since $H(n-1) < 2 \log n$ for $n \geq 2$)

$$2c \lg e H(n-1)T \leq (4c \lg e \log n)T,$$

which is also a polynomial in $\log n$.

When the algorithm halts and outputs ℓ, ℓ must be in the range $1 \leq \ell < n$, and the probability of ℓ only depends on the last run of the algorithm. From (2), the probability that ℓ is generated in step (i) and then passes step (iii) (and then is outputted) is $\frac{B(n)}{n-1}$, which is independent of ℓ. Thus the algorithm outputs an integer uniformly between 1 and $n-1$.

We can also calculate the probability of each integer produced as the conditional probability of ℓ being generated and outputted given the algorithm halts, which is

$$\frac{\frac{B(n)}{n-1}}{\sum_{l=1}^{n-1} \frac{B(n)}{n-1}} = \frac{1}{n-1}.$$