# Matching Extensions

## 1 Red-Blue Matchings

We are given a bipartite graph $G$ where each is either red or blue. Does $G$ have a perfect matching with exactly $k$ red edges and $n-k$ blue edges? Clearly, assigning cost 1 to red edges and 0 to blue edges and computing a min/max-cost perfect matching does not tell us about intermediate $k$.

We extend the idea of randomly drawn edge costs from before. For each $(i,j)$ draw $c_{ij}$ i.u.a.r. from $\{1,\ldots,2m\}$. If $(i,j)$ is red, set $x_{ij} = y \cdot 2^{c_{ij}}$. If $(i,j)$ is blue, set $x_{ij} = 2^{c_{ij}}$.

Define $B$ to be the matrix of all $x_{ij}$. We have

$$\begin{aligned}
\mathrm{perm}(B) &= \sum_{\sigma \text{ permutations}} \prod_i x_{i,\sigma(i)} \\
&= \sum_{\sigma \text{ perfect matchings}} \prod_i x_{i,\sigma(i)} \\
&= \sum_{\sigma \text{ permutations}} \prod_i \begin{cases} y \cdot 2^{c_{i,\sigma(i)}} & \text{red edges} \\ 2^{c_{i,\sigma(i)}} & \text{blue edges} \end{cases} \\
&= \sum_{\sigma \text{ perfect matchings}} y^{\mathrm{red}(\sigma)} 2^{\mathrm{cost}(\sigma)} \\
\det(B) &= \sum_{\sigma \text{ perfect matchings}} (-1)^{\mathrm{sgn}(\sigma)} y^{\mathrm{red}(\sigma)} 2^{\mathrm{cost}(\sigma)}
\end{aligned}$$

Combining terms to isolate each occurrence of $y$,

$$\begin{aligned}
\mathrm{perm}(B) &= \sum_{j=0}^{n} y^j \sum_{\sigma \text{ perfect matching}, \mathrm{red}(\sigma) = j} 2^{\mathrm{cost}(j)} \\
&= \sum_{j=0}^{n} y^j \alpha_j \\
\det(B) &= \sum_{j=0}^{n} y^j \sum_{\sigma \text{ perfect matching}, \mathrm{red}(\sigma) = j} (-1)^{\mathrm{sgn}(\sigma)} 2^{\mathrm{cost}(j)} \\
&= \sum_{j=0}^{n} y^j \beta_j
\end{aligned}$$

A perfect matching with exactly $k$ red edges exists iff $\alpha_k > 0$. If $\beta_j \neq 0$ then $\alpha_j > 0$ for all $j$ (and in particular, $j = k$). If $\alpha_j \neq 0$ then $\beta_j$ may still be 0. But if the min-cost matching among matchings with exactly $j$ red edges is unique, then $\beta_j \neq 0$. So, the proofs from last time are easily adapted to show:

**Lemma 1.** *The min-cost matching with exactly $j$ red edges is unique with probability at least $1/2$.*

Assume we are in this case (and repeat for higher probability). It is sufficient to test if $\beta_k = 0$ (*i.e.,* compute $\beta_k$). Notice that $\det(B(y)) \coloneqq p(y)$ is a univariate polynomial in $y$ of degree at most $n$. We have

$$p(y) = \sum_{j=0}^{n} \beta_j y^j.$$

If we plug in any $n+1$ distinct values for $y$ and evaluate $p(y)$, we obtain a system of $n+1$ linear equations for $n+1$ variables $\beta_0, \ldots, \beta_n$. Thus, we can solve the system to get all $\beta_j$.

# 2 String Comparisons and Pattern Matching

## 2.1 String Comparisons

We have two strings (could be files, archives, etc) $x$ and $y$ of length $n$ with $n$ very large. They are located in different places. We would like to find out if $x = y$ (with high enough probability) without communicating too much.

The idea is to interpret $x$ and $y$ as $n$-bit numbers. We choose a random (small) prime $p$. Send $p$ and $x \mod p$. The recipient computes $y \mod p$ and compares it to $x \mod p$. If they are unequal, then $x \neq y$; otherwise, $x = y$ with TBD probability.

This goes wrong when $x \neq y$ and $p = y \mod p$. Equivalently, $p \mid (x - y)$. How many distinct $p$ can divide $x - y$? Since $|x - y| \leq 2^n$ (because both are length $\leq n$), $|x - y|$ can have at most $n$ distinct prime factors. So, there are at most $n$ "bad" choices.

If we draw $p$ uniformly from the first $n^3$ prime numbers, we succeed with probability at least $1 - n^{-2}$.

But how big could $p$ get? Equivalently, how large is the $(n^3)^{th}$ prime? Well, how many primes are $\leq k$? By the prime number theorem, it is about $k/\ln(k)$. So the $(n^3)^{th}$ prime is approximately $n^3 \log n^3 = \Theta(n^3 \log n)$. (Whereas $|x - y| \leq 2^n$).

In particular, both $p$ and $x \mod p \leq p$ can be encoded in at most

$$\log(\Theta(n^3 \log n)) = \Theta(\log n)$$

bits. This is exponentially better than sending $n$ bits. To boost the probability, we can pick several $p$ independently.

## 2.2 Pattern Matching

Given a long string $x$ with $|x| = n$ and shorter pattern $y$ with $|y| = m$, is there an occurrence of $y$ in $x$? Equivalently, if $x = x_1, \ldots, x_n$, find if there is a $k$ with $x_{k+1} = y_i$ for $i = 1, \ldots, m$.

The naive algorithm is $\mathcal{O}(nm)$ where we check each candidate $k$ and compare all $m$ characters. Suffix tree and related algorithms give $\mathcal{O}(n + m)$, but they are not very easy! We'll come up with a much simpler randomized algorithm instead.

Define $X(k) \coloneqq x_{k+1} x_{k+2} \ldots x_{k+m}$. The naive randomized algorithm compares $X(k)$ to $y$ for all $k$. We can use the previous technique to compute $X(k) \mod p$ and compare to $y \mod p$. However, this takes $\Theta(m)$ for each computation, so the total is still $\Theta(mn)$.

For binary strings, we can do this faster. If we already have $X(k)$, then computing $X(k+1)$ should be faster. Mod $p$, we can subtract $x_k 2^{m-1}$ from $X(k)$ to get rid of the first bit, then multiply by two to shift left by one bit. Then we add $x_{k+m}$. This is a constant number of operations mod $p$, so this is $\Theta(\log p)$ or $\Theta(1)$ depending on the model.

This leads to our algorithm: choose $p$ randomly from a range TBD, compute $X(0) \mod p$ and $y \mod p$, then compare all $X(i) \mod p$ to $y \mod p$ using the efficient update for $X(i)$. If a match is ever indicated, check it in time $\Theta(m)$. If verified, return it; otherwise, start over.

For the analysis, we want a union bound over $n$ steps, so we need a false match probability of at most $1/n^3$. So, draw $p$ from the first $n^3$ primes. Therefore, $\log(p) = \mathcal{O}(\log n)$. This makes the running time $\Theta(n \log n + m)$.

# 3 Random Walks and Markov Chains

Given an undirected graph, the random walk is at some node in each timestep. It randomly chooses a neighbor to move to next, according to some distribution. Usually, the distribution depends only on the current node, not on previous positions.

In the formal model, a Markov chain is characterized by a state space (in this class, finite) $\{1, \ldots, n\}$ and transition probabilities $p_{ij}$ from state $i$ to state $j$. We write $P = (p_{ij})_{i,j}$ for the matrix. $p_{ij}$ is the probability of moving to $j$ given that the walk is currently at $i$. Thus $\sum_i p_{ij} = 1$ for all $i$, so $P$ is stochastic.

The Markov chain starts at time 0 in some state $x_0$, and at each discrete timestep $t$ moves to $x_t$ according to the distribution $p_{x_{t-1},\ldots}$. While continuous time versions exist (Brownian motion), we will not study them in this class.

The key Markovian property is

$$\Pr[X_{t+1} = j \mid X_0 = i_0, X_1 = t_1, \ldots, X_t = i_t] = \Pr[X_{t+1} = j \mid X_t = i_t].$$

That is, the earlier history does not matter.

A random walk on a graph $G$ is a Markov chain with

$$p_{ij} = \frac{1}{\text{out-degree of } i}.$$

That is, we choose a uniformly random edge. Some natural questions to ask include

1. What fraction of time is spent at each node?
2. What is the probability of ever making it to $j$ if you are currently at $i$?
3. How long in expectation does it take to get from $i$ to $j$?
4. How long until all nodes have been visited at least once?

A state $j$ for which the probability under (2) is not equal to 1 is called transient – otherwise, it is called persistent. A state can be transient iff the chain can be trapped somewhere else. These are difficult to deal with, so we want to define them away.

**Definition 2.** A Markov chain is irreducible if the graph with edges $\{(i,j) : p_{ij} > 0\}$ is strongly connected.

Another undesirable behavior is oscillations, where we bounce back and forth between nodes. That is, the distribution of states is different at odd or even times $t$. The periodicity of state $i$ is the gcd of the lengths of all cycles including $i$. State $i$ is aperiodic iff its periodicity is 1, and periodic otherwise. A Markov chain is aperiodic iff all of its states are aperiodic.