Randomness and Computation

or, “Randomized Algorithms”

Heng Guo

(Based on slides by M. Cryan)

Tuesday's lecture: Verification of polynomial identities

On Tuesday we considered the problem of taking two polynomials of degree d, $F(x)$ written as a product of "monomials" and $G(x)$ as an expansion of x^i terms, and testing whether $F(x)$ is identical to $G(x)$.

The basic algorithm takes a single uniform random sample x_1 from the set $\{1, \ldots, 100d\}$ and calculates whether $F(x_1)$ and $G(x_1)$ are equal. This testing algorithm gives a correct answer with probability at least $\frac{1}{100}$ ("one-sided" error).

- The sample drawn to perform the test is just a single value chosen uniformly from $\{1, \ldots, 100d\}$ ... easy probability distribution to understand.
- To refine the algorithm, we can do k trials and "power up" the error to just $\frac{1}{100}^k$.

Matrix multiplication verification

Given three $n \times n$ matrices A, B, C, we want to verify *whether*

$$AB \stackrel{?}{=} C.$$

Recall that the "high-school/Uni" algorithm for evaluating AB would take $\Theta(n^3)$ time. The best algorithm is $\Theta(n^{2.3728639})$ or so.

We will show how to verify (with high probability) in $\Theta(n^2)$ time.

We will exploit the fact that Av can be computed in $O(n^2)$ time for a matrix A and a vector v.

Matrix multiplication verification

Assume that the values in the matrix are integers over some field like $\mathbb{F}_2$ (also known as $GF(2)$), or indeed any $\mathbb{F}_p$ for prime $p > 2$, or even the standard field over $\mathbb{Z}$.

The algorithm is parametrized by some natural number $k > 1$.

Algorithm MMVERIFY(n, A, B, C)

```
for j = 1, ..., k do
        Generate x̄ uniformly at random from {0, 1}^n
        Calculate ȳ_B = B · x̄ in Θ(n^2) time.
        Calculate ȳ_AB = A · ȳ_B in Θ(n^2) time.
        Calculate ȳ_C = C · x̄ in Θ(n^2) time.
        if ȳ_AB ≠ ȳ_C
                return "no"
return "yes"
```

Analysing MMVerify

For the analysis, we will show ...

"One-sided error"

- $AB \equiv C$: In this case, we know that $AB \cdot \bar{x} = C\bar{x}$ for every $\bar{x} \in \{0,1\}^n$. Hence MMVERIFY is guaranteed to return the value "yes".
- $AB \not\equiv C$: We will now show that in this case, when a vector $\bar{x}$ is drawn u.a.r. from $\{0,1\}^n$, the probability that $AB \cdot \bar{x} = C \cdot \bar{x}$ is at most $1/2$.

After this analysis, we will calculate the effect of doing k trials.

Analysing MMVerify: $AB \neq C$

Consider the two $n \times n$ matrices AB and C. They are non-identical, so there must be *at least* one cell (i^*, j^*) such that the values $(AB)_{i^* j^*}$ and $C_{i^* j^*}$ are different.

Let $D = (AB - C)$. Then equivalently, we have $D_{i^* j^*} \neq 0$.

Consider row i^* of D, and consider its product with a vector $\bar{x} \in \{0, 1\}^n$:

$$\sum_{j=1}^{n} D_{i^* j} \cdot x_j.$$

This gives the value for *position* i^* in the length-n vector computed by $D \cdot \bar{x}$.

We will show that this will be 0 with probability at most $1/2$.

Analysing MMVerify: $AB \neq C$

When drawing a random $\bar{x} \in \{0,1\}^n$ uniformly at random (u.a.r.), each $\bar{x}$ has equal probability ($1/2^n$).

This is equivalent to choosing the value $x_i \in \{0,1\}$ independently with probability $1/2$, for each $i \in [n] = \{1,\ldots,n\}$.

Use this in the analysis (*principle of deferred decisions*).
Write $\sum_{j=1}^{n} D_{i^*j} \cdot x_j$ as

$$\left(\sum_{j \in [n] \setminus \{j^*\}} D_{i^*j} \cdot x_j \right) + D_{i^*j^*} \cdot x_{j^*}$$

Think about sampling $\bar{x}$ (*deferred decisions*) as $\{0,1\}^{n-1}$ vector first, followed by the value for x_{j^*} last.

Analysing MMVerify: $AB \neq C$

After sampling the $\{0,1\}^{n-1}$ vector for positions $\{x_j \mid j \in [n] \setminus j^*\}$, we now have a fixed value for

$$Y := \sum_{j \in [n] \setminus \{j^*\}} D_{i^* j} \cdot x_j.$$

Then no matter which field we are in ($\mathbb{Z}$ with standard arithmetic, $\mathbb{F}_p$ for some prime $p > 2$, even $\mathbb{F}_2 \ldots$) there is at most one value (namely $-Y$) which could be added to this to get 0 (maybe 0, maybe 1, maybe some other non-zero value).

Also, we know $D_{i^* j^*} \neq 0$. Sampling x_{j^*} last, we will get $D_{i^* j^*} \cdot x_{j^*} = D_{i^* j^*}$ (which is non-zero) with prob. $1/2$, and $D_{i^* j^*} \cdot x_{j^*} = 0$ with prob. 1/2.

Law of total probability

Theorem (1.6)

Suppose the probability space Ω is partitioned into mutually disjoint events $E_1, E_2, \ldots E_n$. (Namely $E_i \cap E_j = \emptyset$ and $\cup_{i=1}^{n} E_i = \Omega$.)
For any event B,

$$\Pr[B] = \sum_{i=1}^{n} \Pr[B \cap E_i] = \sum_{i=1}^{n} \Pr[B \mid E_i] \Pr[E_i].$$

Analysing MMVerify: $AB \neq C$

As $D_{i^*j^*} \neq 0$,

$$D_{i^*j^*} \cdot x_{j^*} = \begin{cases} D_{i^*j^*} & \text{w.p. } 1/2; \\ 0 & \text{w.p. } 1/2. \end{cases}$$

For a fixed Y, let E_Y be the event that $Y = \sum_{j \in [n] \setminus \{j^*\}} D_{i^*j} \cdot x_j$. Hence, by the law of total probability,

$$\begin{aligned} & \Pr\left[\sum_{j=1}^{n} D_{i^*j} \cdot x_j = 0\right] \\ = & \sum_{Y} \Pr\left[D_{i^*j^*} \cdot x_{j^*} = -Y \middle| E_Y\right] \Pr[E_Y] \\ \leq & \; 1/2 \sum_{Y} \Pr[E_Y] \\ = & \; 1/2. \end{aligned}$$

All trials of MMVerify: $AB \neq C$

Previous slides present the analysis of what happens ($AB \neq C$ case) on a single sample from $\{0, 1\}^n$ (tested in lines 2-7 of MMVERIFY).

- The Algorithm is set up to **return** “no” (and terminate) on the first trial where it discovers a mismatch between $AB \cdot \bar{x}$ and $C \cdot \bar{x}$.
- It only **return**s “yes” if it passed through all iterations of the loop with all trials giving a match.
- “Every trial gives a match” is the bad event for analysing the $AB \neq C$ case.

All trials of MMVerify: $AB \neq C$

Notice that the k repeated trials fit into the paradigm of “sampling with replacement”.

Since all of the trials are independent, the probability that all k repeated trials fail is at most 2^{-k}.

(note: we will skip the Bayes stuff in the book)

More generally . . .

Selecting random inputs is a good way for testing / verification.

Min-Cut

Given an undirected graph $G = (V, E)$, we want to find a "min cut"; that is, a partition of E into two non-empty sets S, $V \setminus S$, such that the following quantity is minimized:

$$|\{e = (u, v) : u \in S, v \in V \setminus S\}|$$

There are many deterministic algorithms which can solve this problem in polynomial-time.

Algorithms for Min-Cut

Let $n = |V|$ and $m = |E|$.

- Classical "network flow" algorithm solves the (s,t)-cut problem.

  To solve min-cut, we can run (s,t)-cut algorithm $n-1$ times. Namely fix a "source" s and run through all possible t.

  This would take $O(mn^2)$ time.

- Best deterministic algorithm pre Karger is due to Stoer and Wagner (1997) in time $O(mn + n^2 \log(n))$.

- Karger (1993) gave a beautiful $O(mn^2 \log n)$ randomised (and parallel) algorithm, and in 1998 a $O(m(\log n)^3)$ randomised algorithm.

- Heavily inspired by Karger's work, Kawarabayashi and Thorup (2014) found a deterministic algorithm that runs in $O(m(\log n)^{12})$ time.

- This is still a very active research area. For example, in Nov '19, Gawrychowski, Mozes, and Weimann claimed a $O(m(\log n)^2)$ randomised algorithm.

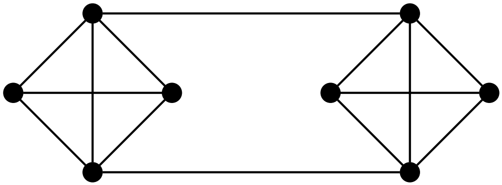
Karger's contraction algorithm

Karger (1993) uses random sampling:

> *Repeatedly, choose an edge uniformly at random (from the not-yet contracted edges) and contract its end-points.*
> *When there are just two "vertices" left, return that cut.*

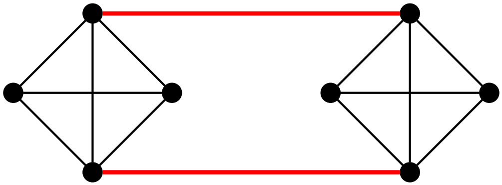
We will show that this algorithm finds the minimum cut with high probability in time $O(n^2 \log n)$.

Example



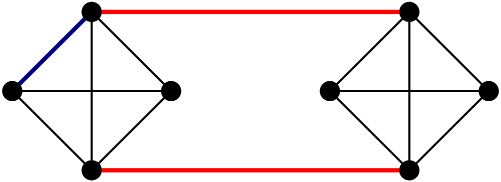
The min cut has size 2.

Example



The min cut has size 2.

Example

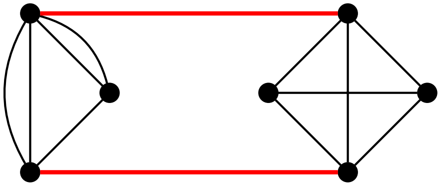


The algorithm randomly picks one edge out of 14.

We hope to avoid the min cut.

In this case the “bad” thing happens with probability $\frac{2}{14}$.

Example

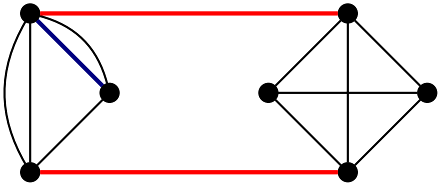


Contraction:

merge the endpoints of an edge into one.

Parallel edges are preserved, and self-loops removed.

Example

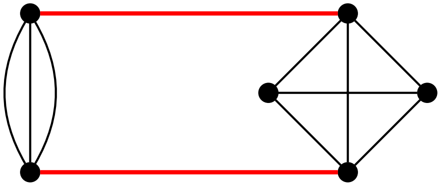


Contraction:

merge the endpoints of an edge into one.

Parallel edges are preserved, and self-loops removed.

Example

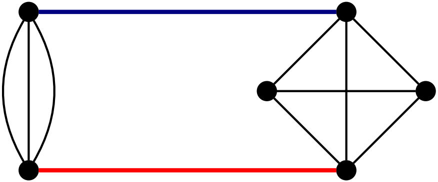


Contraction:

merge the endpoints of an edge into one.

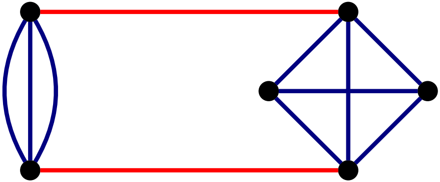
Parallel edges are preserved, and self-loops removed.

Example



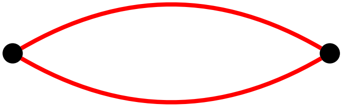
If we contract a cut edge, then we will not find that cut.

Example



Ideally, we should contract all edges except the min cut.

Example



Ideally, we should contract all edges except the min cut.

Implementation of the algorithm

Naive implementation of contractions would require complicated data structures to keep track of everything.

An equivalent way of looking at the algorithm is to pick a random permutation of all edges first, and then contracting edges from the first to the last.

Thus, what we need to find is the shortest prefix of the permutation such that they induce two connected components.

Finding connected components takes $O(m)$ time. Thus, by a binary search, this will take time

$$O(m) + O(m/2) + O(m/4) + \cdots = O(m).$$