Randomized Algorithms

CMSC451

Mattias Ehatamm

October 14, 2026

Guest lecture for Runzhou Tao

Table of contents

1. Intro

2. A Randomized Min Cut Algorithm

3. Analysis of the Min Cut Algorithm

4. Polynomial Identity Testing

5. Contention Resolution

Introduction to Randomized Algorithms

Randomized Algorithms

You have probably seen some randomized algorithms before.

Examples?

What does it mean for an algorithm to be random?

Stricter Definition

Randomized algorithms are algorithms that do not need to be always correct. Or, correct most of the time.

Strictly, let $f(x)$ be the function we are trying to compute.

$$\forall x, \Pr[A(x) = f(x)] \geq \frac{2}{3}$$

Why $\frac{2}{3}$? Later.

Exploring that definition

$$\forall x, \Pr[A(x) = f(x)] \geq \frac{2}{3}$$

Note that the probability is within the x quantifier. Thus, for *all inputs* we must be correct *most* of the time, as opposed to being correct *most of the time* across *all inputs*

An incorrect algorithm

Suppose the problem is ”given a binary string, determine if this string has a 1 in it”. A linear-time deterministic algorithm would iterate over all indices.

What if we just output ”yes” no matter what?

An incorrect algorithm

Suppose the problem is ”given a binary string, determine if this string has a 1 in it”. A linear-time deterministic algorithm would iterate over all indices.

What if we just output ”yes” no matter what?

$$Pr_x[A(x) = f(x)] = 1 - \frac{1}{2^n} \geq \frac{2}{3}$$

This is not the same as the previous definition, because there exists an input (specifically, the all zero string) where we fail every time.

How can we take advantage of randomness?

Vaguely, we can ”spread out” bad outcomes across random trials. If, for a particular input, most deterministic algorithms work, we can randomize over deterministic algorithms to ensure that we *usually* get something that works.

How can we take advantage of randomness?

This gives a relatively clean general definition of what it means for an algorithm to be ”random”:

The randomized algorithm selects over some probability distribution of deterministic algorithms, and then implements a deterministic algorithm.

A Randomized Min Cut Algorithm

Cuts in Graphs

For this problem, all graphs are undirected and unweighted. A **cut** is a partition of the graph into two sets of vertices.

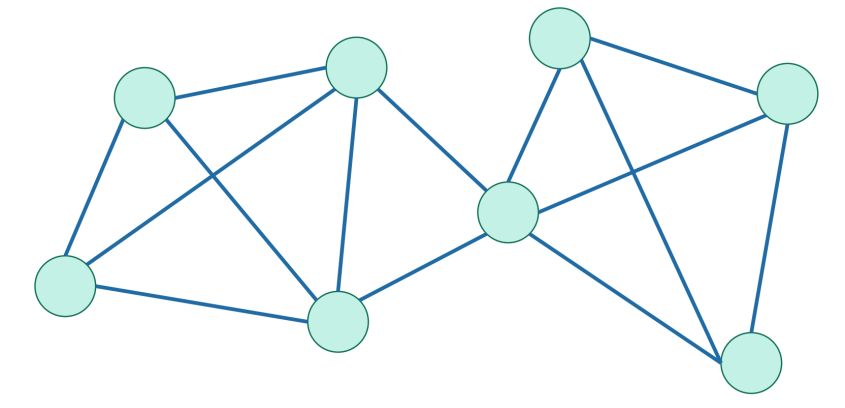


Figure 1: An undirected graph

Cuts in Graphs

For this problem, all graphs are undirected and unweighted. A **cut** is a partition of the graph into two sets of vertices.

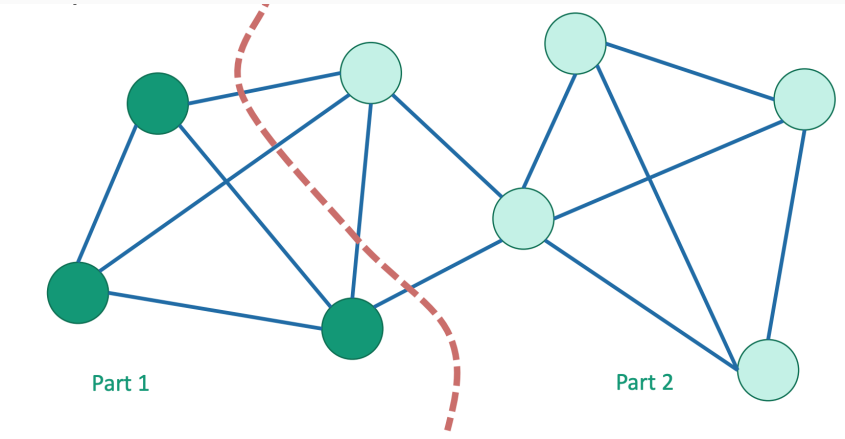


Figure 2: An undirected graph with a cut

Cuts in Graphs

The **size** of a cut is the number of edges crossing the partition.

$$\text{size}(A, B) = |\{uv \in E : u \in A, v \in B\}| \text{ for partition } A, B$$

If a graph is disconnected, there exists a cut of size 0.

The **minimum cut** is the cut with the smallest size.

Thus, the minimum cut may also be seen as the minimum number of edges required to disconnect the graph.

Cuts in Graphs

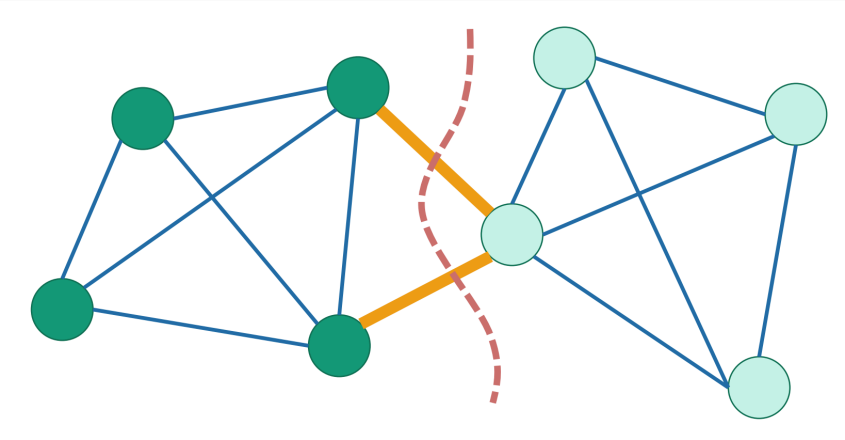


Figure 3: The minimum cut

Randomized Algorithm for the Minimum Cut

Using network flow algorithms which you will see in a later unit, this has a deterministic polynomial time algorithm.

However, with the help of randomized computation, we may present an absurdly simple randomized algorithm (known in the literature as Karger's Algorithm and published in 1993, and thus by far the newest content you have seen so far in this course)

Types of Randomness

Karger's algorithm is known as a **Monte Carlo Algorithm**

1. It is always fast
2. It is sometimes wrong

Compare to Quicksort, which is a **Las Vegas Algorithm**

1. It is sometimes slow
2. It is never wrong

Description of Karger's Algorithm

The **Contraction Operation** on an edge $e = (a, b) \in E(G)$ works as follows: a and b are combined into a new node ab, e is deleted, and the new node gains all edges of a, b.

We will treat contraction as creating multigraphs, so if $(a, v), (b, v)$ were both edges beforehand, then $(w, v), (w, v)$ will be two different nodes afterwards. However, if there were two (a, b) edges, they will both be deleted in the contraction operation.

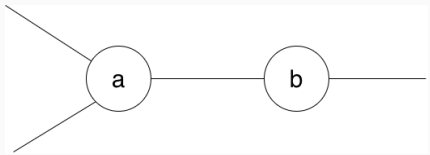


Figure 4: a, b before contraction

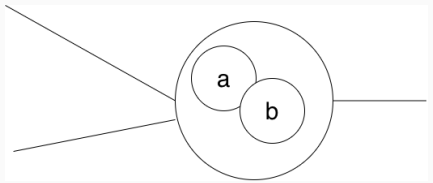


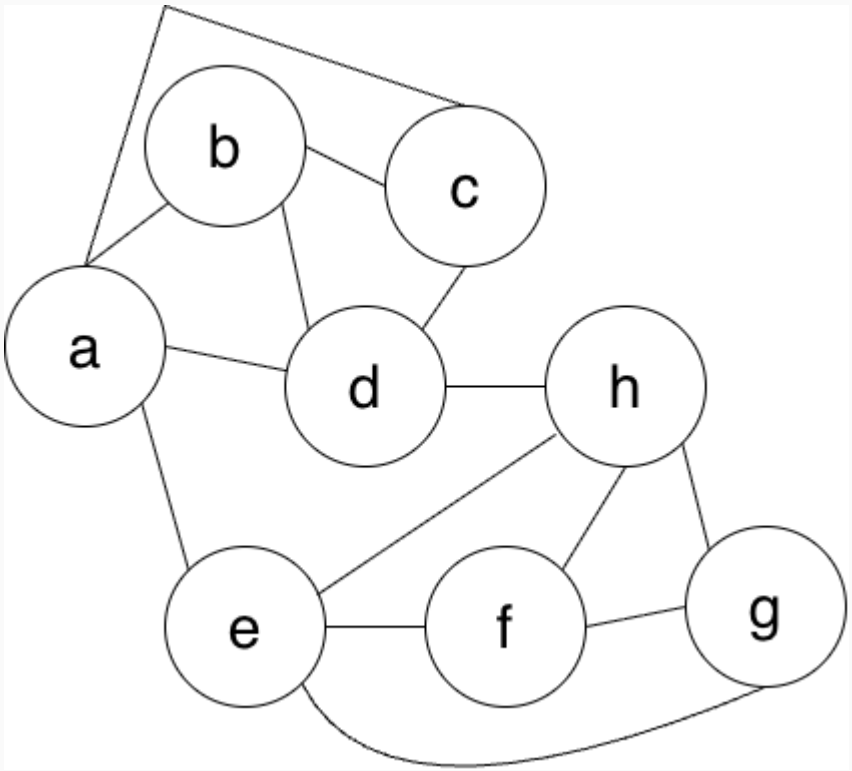
Figure 5: a, b after contraction

Description of Karger's Algorithm

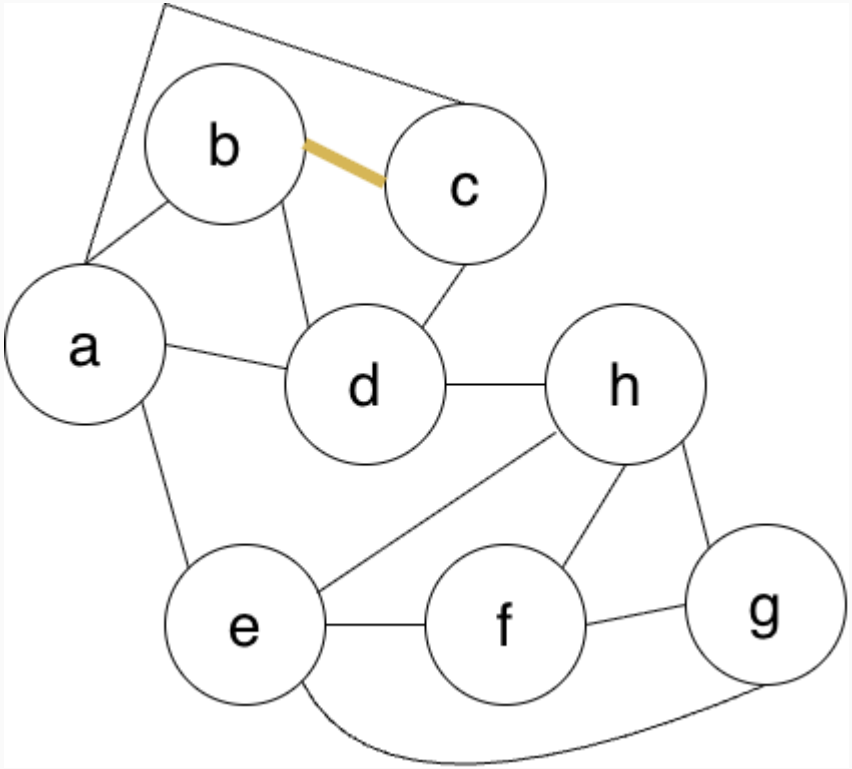
For a graph $G = (V, E)$, pick a random edge $e \in E$. Contract that edge. Repeat until there are only two vertices in the graph. The vertices composing these two super-vertices make up a partition, output the cut produced by those vertices.

A Worked Example:

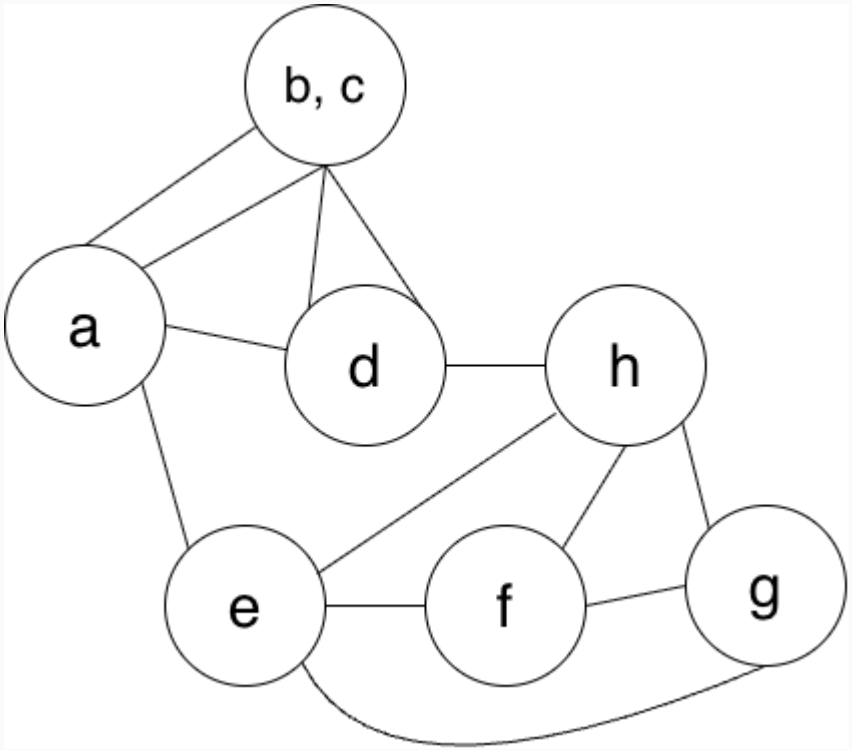
Consider the following graph: It has a minimum cut of size 2, found by removing $\{(ae), (dh)\}$.



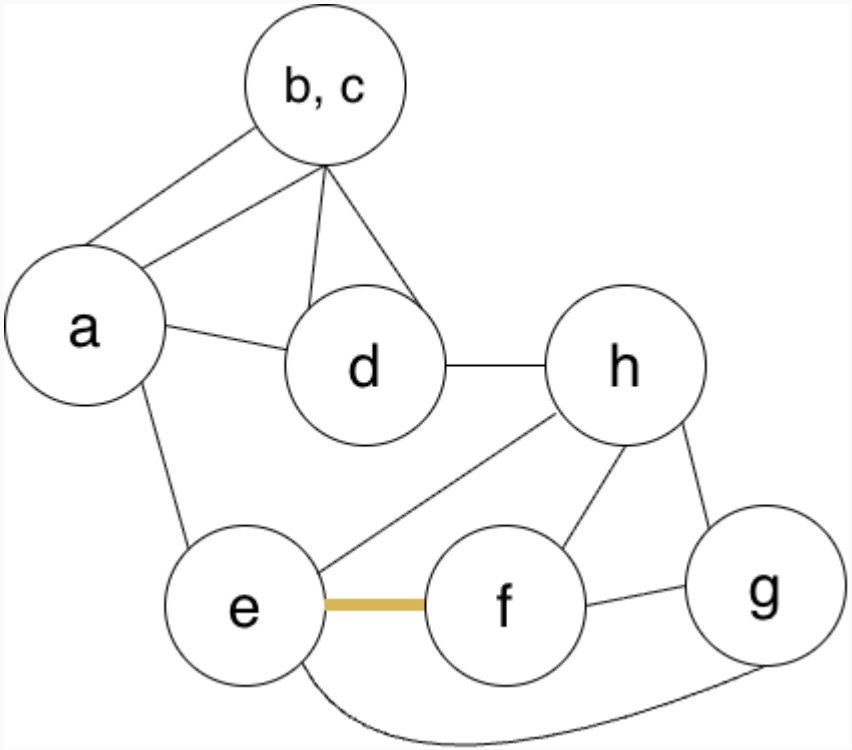
A Worked Example:



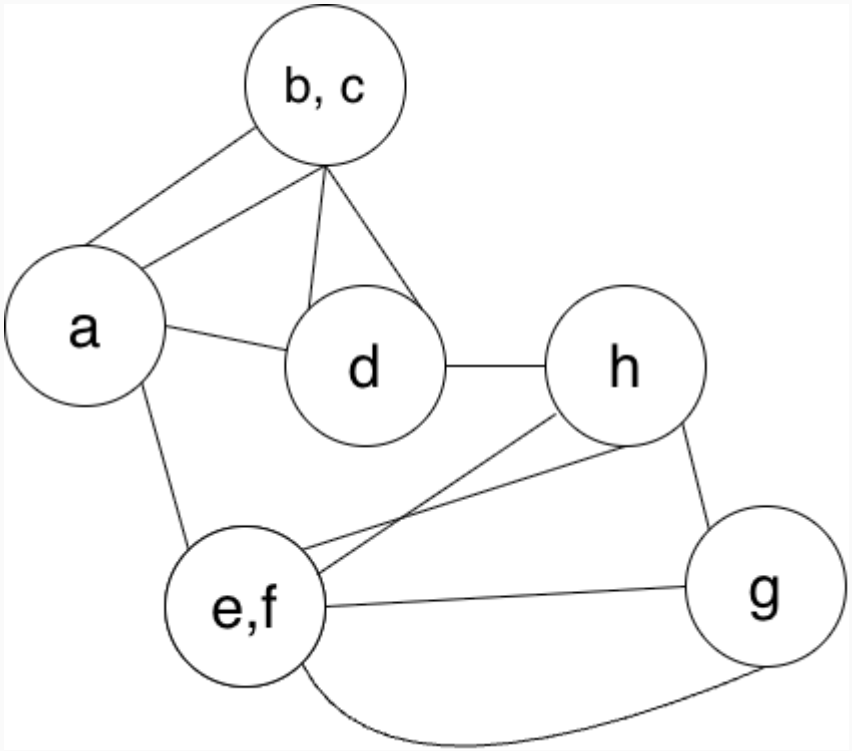
A Worked Example:



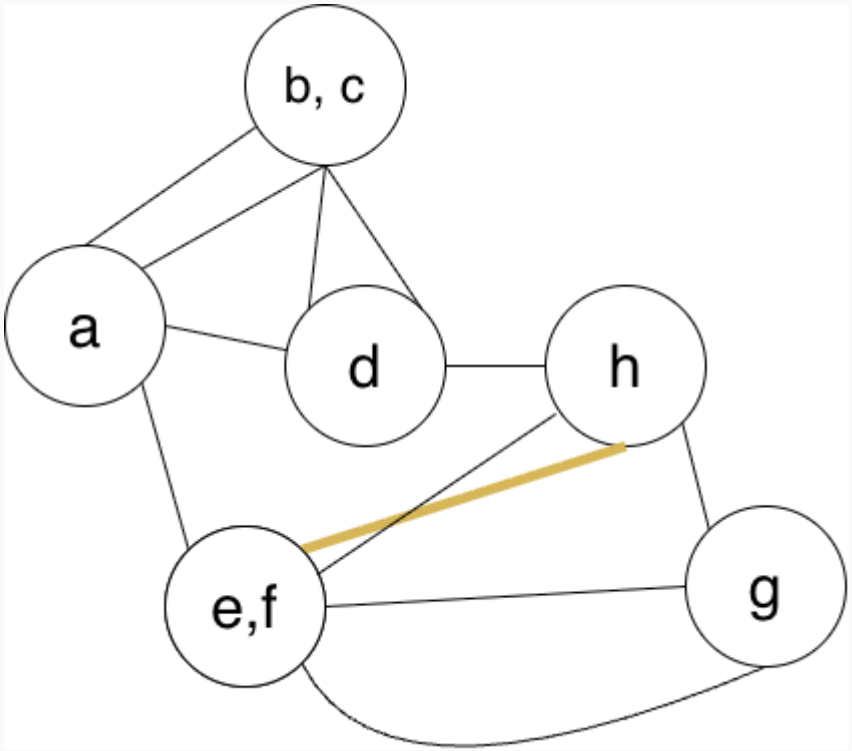
A Worked Example:



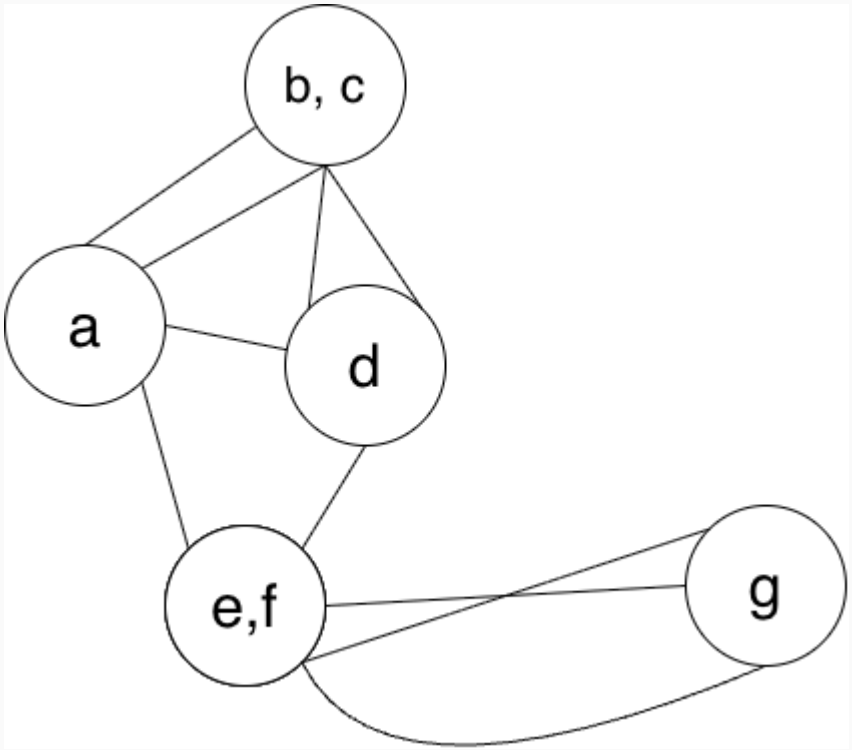
A Worked Example:



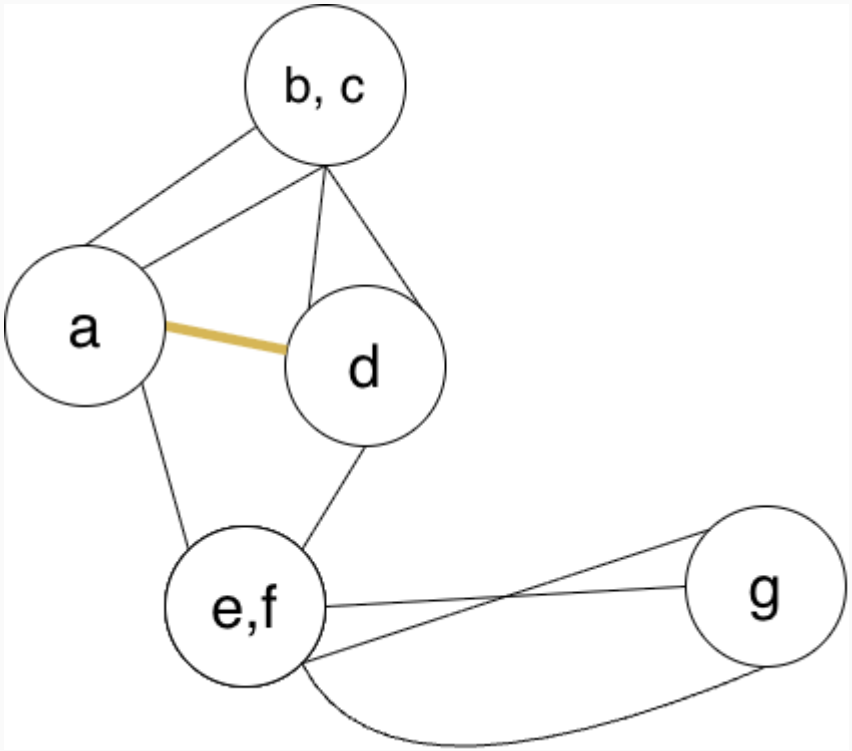
A Worked Example:



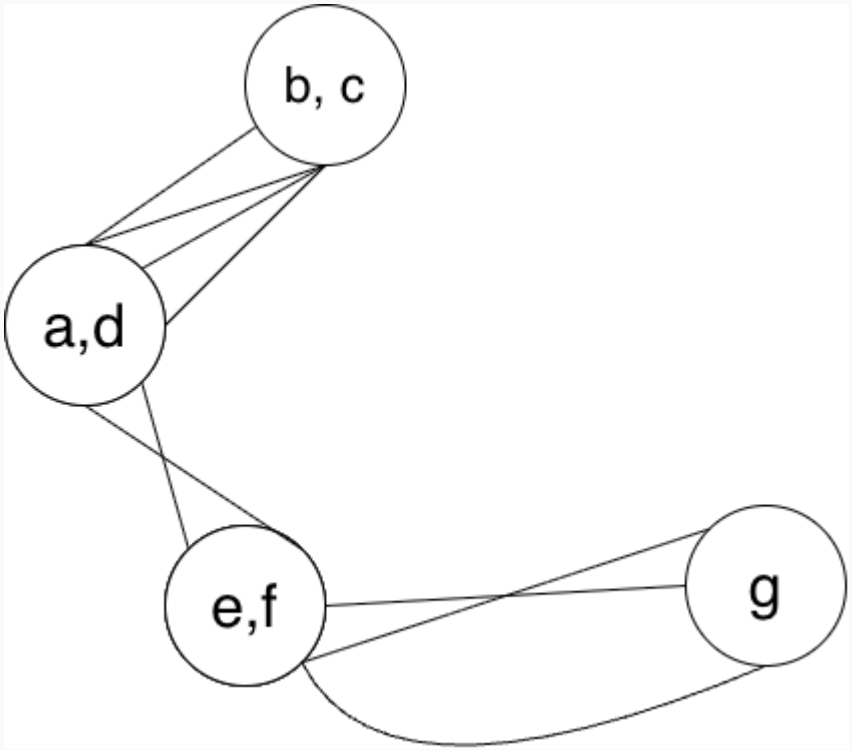
A Worked Example:



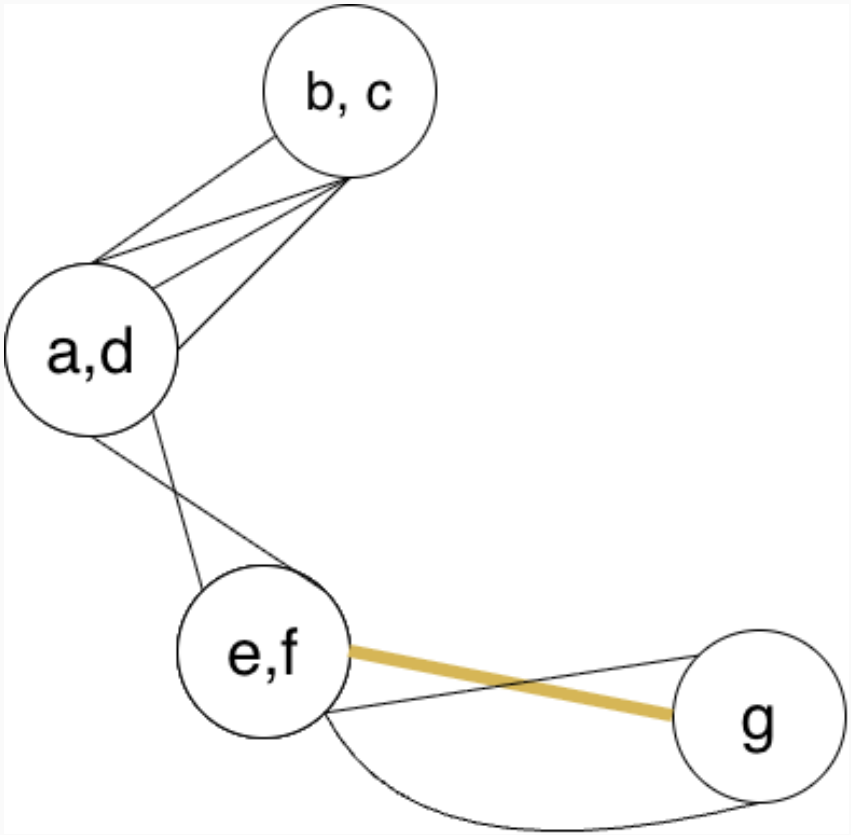
A Worked Example:



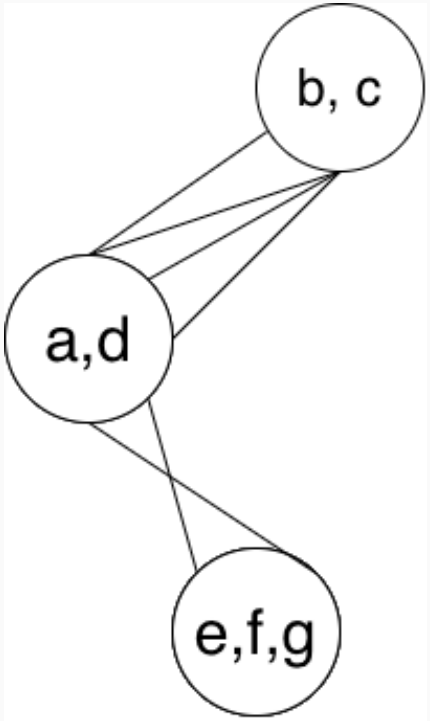
A Worked Example:



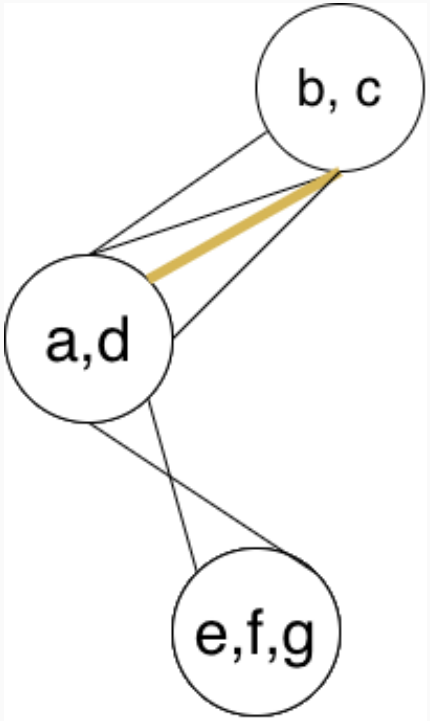
A Worked Example:



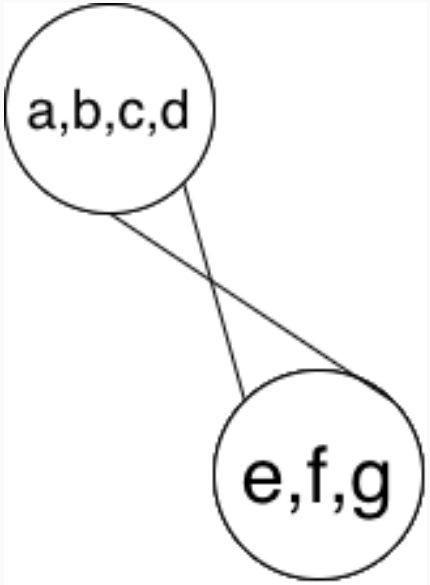
A Worked Example:



A Worked Example:



A Worked Example:



Analysis of the Min Cut Algorithm

Analysis

At first glance, it seems like there's no way this algorithm can be likely to get the right answer.

How unlikely is it to get the right answer?

Analysis

Let K be the set of edges for a min cut of size k.

Recall that K is made up of edges between the partition (A_K, B_K). The algorithm fails when it contracts along K, as two vertices which were meant to be in different sets end up in the same super-vertex.

If the algorithm never contracts along K, then the final partition will be exactly (A_K, B_K). Therefore, the algorithm fails to find this min cut whenever it contracts along K.

Analysis

The algorithm makes $n - 2$ contractions. How many edges are in the graph?

Suppose that G had a vertex with degree $< k$. Then, that vertex itself would be a smaller minimum cut. Thus, $\forall v, \deg(v) \geq k$.

$$|E| \geq \frac{1}{2} n * k$$

On the first contraction, the probability of messing up is $\leq \frac{k}{\frac{1}{2}nk} = \frac{2}{n}$

Analysis

Let G' be the graph after j iterations. Because G' is just a contraction of G, cuts in G' correspond to cuts in G. Thus, if any super-vertex in G' has degree $< k$, that corresponds to a cut of size $< k$. We may also then conclude:

$$\forall v \in G', \deg(v) \geq k$$

If no edges in K have been contracted, the probability of failure on iteration $j + 1$ is then:

$$\leq \frac{k}{\frac{1}{2}(n-j)k} = \frac{2}{n-j}$$

Analysis

We are interested in probability of success. This is the product of the probabilities of success at each step, conditioned on not having already failed.

$$\Pr[\text{success}] \geq (1 - \frac{2}{n})(1 - \frac{2}{n-1}) \cdots (1 - \frac{2}{3})$$

Analysis

We are interested in probability of success. This is the product of the probabilities of success at each step, conditioned on not having already failed.

$$\Pr[\text{success}] \geq (1 - \frac{2}{n})(1 - \frac{2}{n-1}) \cdots (1 - \frac{2}{3})$$

$$\geq \frac{n-2}{n}\frac{n-3}{n-1}\frac{n-4}{n-2} \cdots \frac{2}{4}\frac{1}{3}$$

This product is telescoping, match up numerator and denominator terms and eliminate to obtain:

$$\frac{2}{n * (n-1)} = \binom{n}{2}^{-1}$$

Repetition

It seems like we have failed, because $\binom{n}{2}^{-1}$ is a significantly smaller probability of success than $\frac{2}{3}$.

However, this value is only polynomially small in n.

Suppose we repeated the algorithm $2 * \binom{n}{2}$ times. The probability that we would not find a minimum cut is at most:

$$(1 - \binom{n}{2}^{-1})^{2*\binom{n}{2}} \approx \frac{1}{e^2} \leq \frac{1}{3}$$

This is shown by basic calculus (recall that $(1 - \frac{1}{n})^n$ converges to $\frac{1}{e}$)

Repetition

A single contraction takes (naively) $O(deg(v))$ time and a run of the algorithm performs contraction $O(n)$ times, so the complexity of a full contraction is $O(m)$. If repeated $O\binom{n}{2} = O(n^2)$ times, the algorithm would have a final runtime of $O(n^2 m)$.

This can be improved using better data structures to represent the contraction.

Repetition

This reptition is known as **amplification** and is a standard technique among randomized algorithms, frequently using a statistical tool called **Chernoff Bounds**

The reason why $\frac{2}{3}$ is chosen as the success probability is because the number is mostly arbitrary. It could be $\frac{3}{5}$ or $\frac{4}{7}$ or $\frac{1}{2} + \frac{1}{n}$

In general, if you have a procedure which succeeds or fails with some probability, you can run it a bunch of times and take the majority vote. This will ”amplify” the probability of the correct outcome exponentially with respect to the number of iterations

Polynomial Identity Testing

Polynomials over $\mathbb{F}_p$

Recall that if p is a prime number, modular multiplication over p has unique inverses.

This allows you to define a **Field** known as $\mathbb{F}_p$, which defines values $\{0, \cdots, p-1\}$ and has addition, subtraction, multiplication, and addition well-defined.

A **polynomial over** $\mathbb{F}_p$ $P(x)$ takes inputs on $\mathbb{F}_p$ and is defined by some coefficients $c_0, \cdots, c_{n-1}$ where $P(x) = c_0 + c_1 x^1 + \cdots + c_{n-1} x^{n-1}$.

PIT Setup

Suppose you are given access to two polynomials $f(x)$ and $g(x)$ of degree at most d, in the sense that you are given a "black box" in which you may "query" the value of $f(x)$ and $g(x)$ on elements of $\mathbb{F}_p$. You wish to determine if they are the same polynomial. Suppose that $d \leq \frac{n}{2}$

By defining a new polynomial $h(x) = f(x) - g(x)$, which may be queried by querying $f(x)$ and $g(x)$, we may reduce this problem to determining if $h(x) = 0$. $h(x)$ will still have degree at most d.

PIT Setup

Recall that by the **Fundamental Theorem of Algebra**, polynomials are exactly defined by their roots. $h(x)$ has degree at most d, so there may be at most d roots. Query $h(x)$ n times at some deterministic series of inputs, and if it returns 0 every time it must be zero. This is $O(d) = O(n)$ time.

PIT Algorithm

Query $h(x)$ twice on random x values. If it returns zero both times, return that the polynomial is 0. If not, return that the polynomial is nonzero. This algorithm runs in $O(1)$ time.

PIT Algorithm Analysis

The algorithm only fails when $h(x)$ is nonzero and we accidentally query its zeroes twice. Because $d \leq \frac{n}{2}$, there are at most $\frac{n}{2}$ zeroes out of n values. Thus, the probability two independent random queries both hit a zero is $\leq \frac{1}{4}$. Therefore, our algorithm succeeds with probability $\frac{3}{4}$ and runs in $O(1)$ time.

We believe there exists a deterministic, efficient algorithm for polynomial identity testing. Study of this problem is one of the central focuses of derandomization in complexity theory.

Contention Resolution

Problem Setup... If we get here

Suppose there is some resource R and a number of actors $a_1, \cdots, a_k$. On each of n rounds, some number of actors declare that they wish to use the resource. If multiple declare they wish to use the resource, they are all locked out for that round.

Actors **cannot communicate** before rounds.

Problem Setup

Suppose you need to devise an algorithm for actors such that the most actors get resource access. If determinism required and all actors are given the same algorithm, you can **never** come up with an algorithm where some actor gets access to the resource.

Randomized Algorithm

On each of n rounds, actors individually decide whether or not they will access R with probability p.

Analyzing this

In a single round, when do we succeed? If a single actor a_i decides to access R, and every other actor decides not to access R.

$$p * (1 - p)^k$$

We therefore need to maximize this value.

Analyzing this

How do we maximize $p * (1-p)^k$? Take the derivative and set to 0:

$$(1-p)^{k-1} - (k-1) * p * (1-p)^{k-2} = 0$$

$$(1-p) - (k-1) * p = 0$$

$$p = \frac{1}{k}$$

Analyzing this

This makes sense, because we expect 1 actor to be accessing the resource if we set the probability to $\frac{1}{k}$. Thus, for a particular round, the probability of a particular actor getting to access is:

$$\frac{1}{n}(1-\frac{1}{n})^{n-1}$$

Which is asymptotically equal to $\frac{1}{n}$.

Going further:

Analyze how many rounds need to occur for every actor to access the resource at least once with high probability: Union Bounding, could be on the quiz

Textbook Readings

The contention resolution and min cut algorithms are sections 13.1 and 13.2 in Kleinberg and Tardos.

PIT: not in textbooks just google it