Algorithms M2 IF

Introduction to Randomized Algorithms

Michael Lampis

Fall 2019

Class Overview

This is an **Advanced Algorithms** class. We will care about:

- Time complexity (and also space complexity) of our algorithms as a function of n, the input size.
- We will pay close attention to the asymptotics. We distinguish between $O(n)$ and $O(n^2)$
- Performance Guarantees. We only care about an algorithm if we can **prove mathematically** that it “works well”.
- Possible definitions of “works well”: solves the problem always or with high probability, its time complexity is below a certain bound always, or with high probability.

Class Overview

This is an **Advanced Algorithms** class. We will care about:

- Time complexity (and also space complexity) of our algorithms as a function of n, the input size.
- We will pay close attention to the asymptotics. We distinguish between $O(n)$ and $O(n^2)$
- Performance Guarantees. We only care about an algorithm if we can **prove mathematically** that it “works well”.
- Possible definitions of “works well”: solves the problem always or with high probability, its time complexity is below a certain bound always, or with high probability.
  - With high probability (whp) is a precise mathematical statement $\rightarrow$ with probability $\geq 1 - o(1)$.

Class Overview

This is an **Advanced Algorithms** class. We will care about:

- Time complexity (and also space complexity) of our algorithms as a function of n, the input size.
- We will pay close attention to the asymptotics. We distinguish between $O(n)$ and $O(n^2)$
- Performance Guarantees. We only care about an algorithm if we can **prove mathematically** that it "works well".
- Possible definitions of "works well": solves the problem always or with high probability, its time complexity is below a certain bound always, or with high probability.

  Topics that will be covered (this may be updated during the semester):

  - Randomized Algorithms
  - Dynamic Programming (vs. Recursion and Divide-and-Conquer)
  - (*) Sub-linear Algorithms – Property Testing
  - (*) On-line Algorithms

Administration

- Course taught in English.
- Web page:
  `https://www.lamsade.dauphine.fr/~mlampis/Algo/`
- Regularly check web page (and Dauphine planning) for updates!
- Grading:
  - 30% Homework assignments (CC)
  - 70% Final exam
- Course organization:
  - 1h30 of lecture
  - 1h30 of exercises (TD)
  - Homeworks will be of same spirit as TD.
- Reading material (including these slides) found on the web page.
- If in doubt, email me!

Randomized Algorithms

Introduction

- A **randomized** algorithm is an algorithm which may at any step produce a random bit (say, by flipping a coin) and use this bit in its calculations.

Introduction

- A **randomized** algorithm is an algorithm which may at any step produce a random bit (say, by flipping a coin) and use this bit in its calculations.
  - Example: Polling for elections. Given n voters, the algortihm selects $k << n$ voters at random and uses their preferences to predict the outcome of the election.

Introduction

- A **randomized** algorithm is an algorithm which may at any step produce a random bit (say, by flipping a coin) and use this bit in its calculations.

  Main applications/advantages of randomized algorithms:

  - Simpler to describe
  - Faster to run (if we have access to random bits!)
  - Performance guarantee depends on **our own** random bits, applies **to all inputs**

  On a basic level, randomized algorithms make it easy to "find hay in a haystack". Same problem not obvious for deterministic algorithms (think serial search).

  Disadvantages:

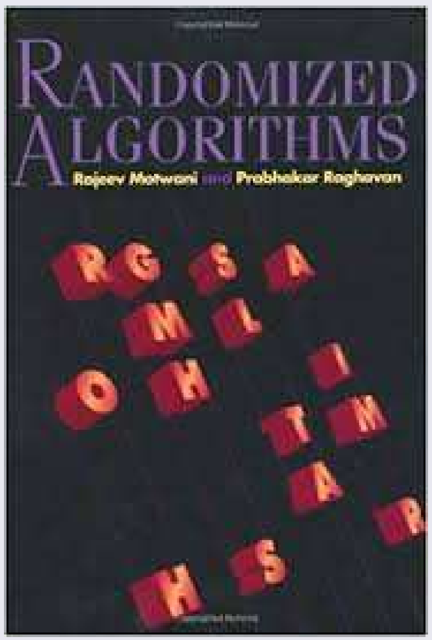
  - Math is usually harder!
  - Producing random bits is not obvious.

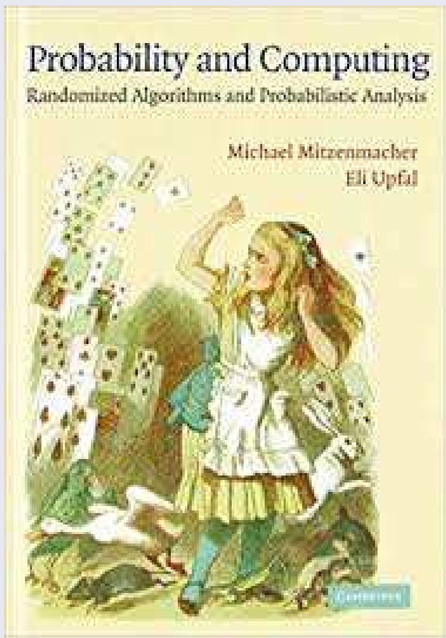
Randomized Algorithms – This course

- We want to prove theorems of the form “With high probability, (randomized) algorithm A does X”
  - Implied $\rightarrow$ **for any input**.
- We assume that random bits are given for free.
  - Not necessarily realistic (pseudo-random bit generators are hard!)
- Type of performance guarantee we want:
  - Whp algorithm A is “fast”
  - Whp algorithm A is correct.
    - If not, what kind of error could we have?
  - Algorithm A is **expected** to be fast/good/correct.
    - Will discuss how to transform expectation guarantees to whp guarantees.

References

- Refs:
  - Mitzenmacher and Upfal, Probability and Computing [MU]
  - Motwani and Raghavan, Randomized Algorithms [MR]





Average-Case Analysis of (Deterministic) Algorithms

- Probabilities are also important for “normal” (deterministic) algorithms.
- Example: algorithm A works great “most of the time”.
  - Meaning what?
- One possible interpretation:
  - Define a natural probability distribution over inputs (uniform?)
  - **Prove** that if input follows this distribution, then algorithm A is “good”.
  - $\rightarrow$ algorithm A is good with high probability!

Example Theorem:

- (Deterministic) Quicksort takes time $O(n \log n)$ on average.

Worst-Case Analysis of Randomized Algorithms

- In this course we are less interested in **average-case** guarantees, and more in **worst-case** (i.e. all cases) guarantees.
- Problems with average-case guarantees:
  - What is the average case? Uniform? Sparse? Gaussian?
  - Hard to analyze.
  - Still may fail badly sometimes (though not often).
- We prefer theorems which prove a statement **for all inputs**, and may rely on probabilities on bits **picked by the algorithm**.
- Think that the input is selected by an adversary, but the random bits by the referee.

  Example Theorem:
- Randomized Quicksort takes $O(n \log n)$ time on average.

Worst-Case Analysis of Randomized Algorithms

- In this course we are less interested in **average-case** guarantees, and more in **worst-case** (i.e. all cases) guarantees.
- Problems with average-case guarantees:
  - What is the average case? Uniform? Sparse? Gaussian?
  - Hard to analyze.
  - Still may fail badly sometimes (though not often).
- We prefer theorems which prove a statement **for all inputs**, and may rely on probabilities on bits **picked by the algorithm**.
- Think that the input is selected by an adversary, but the random bits by the referee.

Example Theorem:

- Randomized Quicksort takes $O(n \log n)$ time on average.
- Can you tell the difference with the previous slide? Which is better?

Problem:

- Input: an array of n **distinct** integers.
- Operations: Compare, Swap, in unit time.
- Output: the same numbers sorted in increasing order.

Quicksort

- If $n \leq 1$ Done!
- Partition the array into $L = \{x \mid x < A[1]\}$, $R = \{x \mid x > A[1]\}$
  - We are using $A[1]$ as the **pivot**
- Output $\mathrm{QSort}(L)$, $A[1]$, $\mathrm{QSort}(R)$.

An example: the complexity of Quicksort

Problem:

- Input: an array of n **distinct** integers.
- Operations: Compare, Swap, in unit time.
- Output: the same numbers sorted in increasing order.

Quicksort

- If $n \leq 1$ Done!
- Partition the array into $L = \{x \mid x < A[1]\}$, $R = \{x \mid x > A[1]\}$
  - We are using $A[1]$ as the **pivot**
- Output $\mathrm{QSort}(L)$, $A[1]$, $\mathrm{QSort}(R)$.
- Correctness?
- Worst-case complexity: $O(n^2)$ operations. (Why?)

Theorem: (Det.) Quicksort on average

Time complexity on n elements:

$$T(n) \leq T(q) + T(n - q - 1) + O(n)$$

where $q = |L|$.

Theorem: (Det.) Quicksort on average

Time complexity on n elements:

$$T(n) \leq T(q) + T(n - q - 1) + O(n)$$

where $q = |L|$.

This gives

- $T(n) = O(n \log n)$ if $q = n/2$ always (unlikely!)
- $T(n) = O(n \log n)$ if $q \in [n/4, 3n/4]$ always (more likely)
- $T(n) = O(n^2)$ if $q = O(1)$. (Tight example?)

Theorem: (Det.) Quicksort on average

Time complexity on n elements:

$$T(n) \leq T(q) + T(n - q - 1) + O(n)$$

where $q = |L|$.

This gives

- $T(n) = O(n \log n)$ if $q = n/2$ always (unlikely!)
- $T(n) = O(n \log n)$ if $q \in [n/4, 3n/4]$ always (more likely)
- $T(n) = O(n^2)$ if $q = O(1)$. (Tight example?)

Would like to prove:

- If A is in a (uniformly) random permutation, then the expected time complexity of Quicksort is $O(n \log n)$.

Theorem: (Det.) Quicksort on average continued

- $T(n)$ now denotes **expected** number of steps. (We are using linearity of expectations.)
- Assume that $T(n)$ is increasing, and in fact super-linear ($\Omega(n \log n)$).
- Say $A[1]$ is a good pivot if $q \in [n/4, 3n/4]$.

Then:

$$\begin{aligned}
T(n) &\leq \frac{1}{2}(T(q_{good}) + T(n - q_{good})) + \frac{1}{2}T(n) + c \cdot n \\
T(n) &\leq T(3n/4) + T(n/4) + 2c \cdot n \\
T(n) &\leq O(n \log n)
\end{aligned}$$

Theorem: (Det.) Quicksort on average continued

- $T(n)$ now denotes **expected** number of steps. (We are using linearity of expectations.)
- Assume that $T(n)$ is increasing, and in fact super-linear ($\Omega(n \log n)$).
- Say $A[1]$ is a good pivot if $q \in [n/4, 3n/4]$.

Then:

$$
\begin{aligned}
T(n) &\leq \frac{1}{2}(T(q_{good}) + T(n - q_{good})) + \frac{1}{2}T(n) + c \cdot n \\
T(n) &\leq T(3n/4) + T(n/4) + 2c \cdot n \\
T(n) &\leq O(n \log n)
\end{aligned}
$$

- We use the fact that $T(n)$ is increasing (so in case of bad pivot we assume we spend another $T(n)$ steps).
- $T(n)$ is super-linear $\to T(q) + T(n-q) \leq T(n/4) + T(3n/4)$.
- Final recurrence can be solved with standard techniques (or verified with induction).

Theorem: Rand. Quicksort with high probability

Alternative algorithm:

1. Pick a random element x of A as pivot.
2. If x is a good pivot, partition, recurse.
3. If not, go back to step 1.

Theorem: Rand. Quicksort with high probability

Alternative algorithm:

1. Pick a random element x of A as pivot.
2. If x is a good pivot, partition, recurse.
3. If not, go back to step 1.

Notes:

- Can check if x is a good pivot in $O(n)$ time. (How?)
- Probability that x is a good pivot is $\frac{1}{2}$.
- $\rightarrow$ Expected number of times going back to 1 is 2.

Theorem: Rand. Quicksort with high probability

Alternative algorithm:

1. Pick a random element x of A as pivot.
2. If x is a good pivot, partition, recurse.
3. If not, go back to step 1.

Notes:

- Can check if x is a good pivot in $O(n)$ time. (How?)
- Probability that x is a good pivot is $\frac{1}{2}$.
- $\rightarrow$ Expected number of times going back to 1 is 2.

$$
\begin{aligned}
T(n) &\leq T(n/4) + T(3n/4) + 2 \cdot c \cdot n \\
T(n) &\leq O(n \log n)
\end{aligned}
$$

Summary

Important lessons to remember.

- "Alg A is good on most inputs" is NOT THE SAME as "Alg A is good most of the time"
  - For the former we need input to be random.
  - For the latter we need random bits to be random. Much more realistic.
  - Example: for Quicksort, second algorithm is provably expected $O(n \log n)$, no matter the input.
- Only proved expected performance (because it's easier). How to get "with high probability" guarantee?

Summary

Important lessons to remember.

- "Alg A is good on most inputs" is NOT THE SAME as "Alg A is good most of the time"
  - For the former we need input to be random.
  - For the latter we need random bits to be random. Much more realistic.
  - Example: for Quicksort, second algorithm is provably expected $O(n \log n)$, no matter the input.
- Only proved expected performance (because it's easier). How to get "with high probability" guarantee?
  - Here, use Markov's inequality. $Prob[X > aE[X]] \leq \frac{1}{a}$.

Summary

Important lessons to remember.

- “Alg A is good on most inputs” is NOT THE SAME as “Alg A is good most of the time”
  - For the former we need input to be random.
  - For the latter we need random bits to be random. Much more realistic.
  - Example: for Quicksort, second algorithm is provably expected $O(n \log n)$, no matter the input.
- Only proved expected performance (because it’s easier). How to get “with high probability” guarantee?
- This algorithm ALWAYS produces the correct answer.
  - $\rightarrow$ Las Vegas algorithm

Testing Matrix Multiplication

Problem:

- Input: Three $n \times n$ matrices A, B, C.
- Operations: Addition, multiplication over scalars.
- Question: Is it true that $AB = C$?

Example:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 3 & 4 \\ 1 & 2 \end{bmatrix} \stackrel{?}{=} \begin{bmatrix} 5 & 8 \\ 13 & 20 \end{bmatrix}$$

- Important note: we do not need to calculate C from scratch! It is given to us and we want to verify if it is correct (or find an error).

Testing Matrix Multiplication

Problem:

- Input: Three $n \times n$ matrices A, B, C.
- Operations: Addition, multiplication over scalars.
- Question: Is it true that $AB = C$?

Example:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 3 & 4 \\ 1 & 2 \end{bmatrix} \stackrel{?}{=} \begin{bmatrix} 5 & 8 \\ 13 & 20 \end{bmatrix}$$

- Important note: we do not need to calculate C from scratch! It is given to us and we want to verify if it is correct (or find an error).
- Can we do this in linear time?
  - **Linear in what**? Here, the input has size $\Theta(n^2)$ (if we assume numbers take constant space). Hence, we are looking for an $O(n^2)$ algorithm.

Testing Matrix Multiplication – Algorithm 1

Naive algorithm:

- Calculate AB from scratch.
- Compare each element of AB with the corresponding element of C.

What is the complexity of this algorithm?

Testing Matrix Multiplication – Algorithm 1

Naive algorithm:

- Calculate AB from scratch.
- Compare each element of AB with the corresponding element of C.

What is the complexity of this algorithm?

- Step 1 takes time:
  - $O(n^3)$ if done trivially.
  - About $O(n^{2.3})$ if we use state of the art MM algorithms.
  - HUGE open problem if it can be done in $O(n^2)$.
- Step 2 takes $O(n^2)$ and this is obviously tight (why?)
- $\rightarrow$ algorithm runs in more than linear time.

Testing Matrix Multiplication – Algorithm 2

Let's use randomness!

- Pick a random element $C[i, j]$
- Calculate the product of row i of A with column j of B.
- If not equal, we have found an error.
- Otherwise, accept as "probably equal".

This algorithm has

- One-sided error (can only be wrong if it accepts that $AB = C$). :-)
  - Monte Carlo algorithm
- Running time $O(n)$ (sub-linear!) :-)
- Probability of success?

Testing Matrix Multiplication – Algorithm 2

Let's use randomness!

- Pick a random element $C[i, j]$
- Calculate the product of row i of A with column j of B.
- If not equal, we have found an error.
- Otherwise, accept as "probably equal".

This algorithm has

- One-sided error (can only be wrong if it accepts that $AB = C$). :-)
  - Monte Carlo algorithm
- Running time $O(n)$ (sub-linear!) :-)
- Probability of success?
  - Suppose C is incorrect in just 1 element.
  - With probability $1 - \frac{1}{n^2}$ algorithm picks another element $\rightarrow$ error. :-(
  - Even if we repeat n times prob of error $(1 - \frac{1}{n^2})^n \rightarrow 1$. :-(

Testing Matrix Multiplication – Algorithm 3

Let's use randomness in a more clever way!

- Pick d to be an $n \times 1$ vector.
  - Each element is $\{0, 1\}$ independently with probability $1/2$.
- Check if $ABd = Cd$.
  - If no, we have a proof that $AB \neq C$.
  - If yes, say "probably equal".

Testing Matrix Multiplication – Algorithm 3

Let's use randomness in a more clever way!

- Pick d to be an $n \times 1$ vector.
  - Each element is $\{0, 1\}$ independently with probability $1/2$.
- Check if $ABd = Cd$.
  - If no, we have a proof that $AB \neq C$.
  - If yes, say "probably equal".

Analysis:

- Calculating Bd takes $O(n^2)$ (trivial). Same for Cd.
- Given Bd, calculating $A(Bd) = ABd$ takes $O(n^2)$.
- Checking if $ABd = Cd$ takes $O(n^2)$. Total time = $O(n^2)$.
- Probability of success?

Algorithm 3 continued

Let $D = AB - C$. If $D \neq 0$ then what is the probability that $Dd = 0$?

- Note: if $Dd = 0$ the algorithm is wrong! We want this probability to be low.
- Suppose that $D \neq 0$, so D contains a non-zero element. Without loss of generality $D[1,1] \neq 0$.
- If $Dd = 0$ then

$$
\begin{aligned}
D[1,1]d[1] \quad &+ \quad \sum_{j=2}^{n} D[1,j]d[j] = 0 \Rightarrow \\
d[1] \quad &= \quad -\frac{\sum_{j=2}^{n} D[1,j]d[j]}{D[1,1]}
\end{aligned}
$$

- Note: we have used that $D[1,1] \neq 0$
- Prob that $d[1]$ takes the rhs value is at most $1/2$.

Summary

Important lessons to remember.

- Randomized algorithms are great for finding hay in a haystack.
- If we want to find a needle in a haystack (here: one out of n^2 elements) we need to do some work to "spread it around" so that it's easy to find.
- Probability of success is $\frac{1}{2}$. Can be improved:
  - Repeat the algorithm k times, independently. Because one-sided error, error probability becomes 2^{-k}.
  - Important here: randomness is over our own bits!
  - Alternative: set d a random vector over $\{0, \ldots, k\}$. (Problem-specific solution).

Testing Polynomial Identities

Problem:

- Input: Two polynomials on one variable x
- Operations: Normal arithmetic
- Output: Are the two polynomials equal for all x?

Examples:

$$(x+1)(x+2) \stackrel{?}{=} x^2 + 2x + 1$$

Testing Polynomial Identities

Problem:

- Input: Two polynomials on one variable x
- Operations: Normal arithmetic
- Output: Are the two polynomials equal for all x?

Examples:

$$\begin{aligned}(x+1)(x+2) &\stackrel{?}{=} x^2+2x+1 \\ (x+1)(x+2)(x-1)(x-2) &\stackrel{?}{=} (x^2-1)(x^2-4)\end{aligned}$$

Problem:

- Input: Two polynomials on one variable x
- Operations: Normal arithmetic
- Output: Are the two polynomials equal for all x?

Examples:

$$
\begin{aligned}
(x+1)(x+2) &\stackrel{?}{=} x^2+2x+1 \\
(x+1)(x+2)(x-1)(x-2) &\stackrel{?}{=} (x^2-1)(x^2-4) \\
(x^3+9x^2+23x+15)(x^3+12x^2+44x+48) &\stackrel{?}{=} \\
(x^2+3x+2)(x^2+7x+1)(x^2+11x+30) &
\end{aligned}
$$

Testing Polynomial Identities – Algorithm 1

- Every polynomial has a canonical form as a sum of monomials

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

- Could try to calculate canonical forms for both polynomials, compare.

Testing Polynomial Identities – Algorithm 1

- Every polynomial has a canonical form as a sum of monomials

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

- Could try to calculate canonical forms for both polynomials, compare.
- Problem: this form may be exponentially longer than the original input!!

$$\left(\left((x+1)^2+1\right)^2+1\right)^2+1\ldots$$

- Degree of this polynomial is 2^n
- However, we can use the fact that **evaluating** a polynomial on a given value of x is easy.

Testing Polynomial Identities – Algorithm 2

Recall MM verification algorithm:

- We prove that two objects are different by showing that they interact in different ways with a random object.

Testing Polynomial Identities – Algorithm 2

Recall MM verification algorithm:

- We prove that two objects are different by showing that they interact in different ways with a random object.
- Given $P_1(x), P_2(x)$, select a random value x_0.
- If $P_1(x_0) \neq P_2(x_0)$, reject.
- Otherwise, accept as "probably equal".

Testing Polynomial Identities – Algorithm 2

Recall MM verification algorithm:

- We prove that two objects are different by showing that they interact in different ways with a random object.
- Given $P_1(x), P_2(x)$, select a random value x_0.
- If $P_1(x_0) \neq P_2(x_0)$, reject.
- Otherwise, accept as “probably equal”.

Problem: even if $P_1(x) \neq P_2(x)$, they could still agree on x_0!

- If $P_1(x) \neq P_2(x)$, in how many values could P_1, P_2 agree?

Testing Polynomial Identities – Algorithm 2

Recall MM verification algorithm:

- We prove that two objects are different by showing that they interact in different ways with a random object.
- Given $P_1(x), P_2(x)$, select a random value x_0.
- If $P_1(x_0) \neq P_2(x_0)$, reject.
- Otherwise, accept as "probably equal".

Problem: even if $P_1(x) \neq P_2(x)$, they could still agree on x_0!

- If $P_1(x) \neq P_2(x)$, in how many values could P_1, P_2 agree?
  - Let $Q(x) = P_1(x) - P_2(x)$. The degree of Q is at most the degree of P_1, P_2, say n.
  - $\Rightarrow Q$ has at most n roots.

Testing Polynomial Identities – Algorithm 2

- Calculate the degrees of the two polynomials n.
- Pick a random number x_0 in $\{0, \ldots, 2n\}$.
- Check if $P_1(x_0) = P_2(x_0)$.
    - If no, reject.
    - If yes, say "probably equal"

Analysis:

- Probability of success at least $1/2$.
- Can be increased by repeating the algorithm.
- Derandomizing this algorithm is a major open research problem.

Min-Cut

Problem:

- Input: Graph $G = (V, E)$
- Output: A minimum cut of G

- A cut is a set of edges whose removal creates at least two connected components.
- Problem solvable in polynomial time using max flow techniques.
- Goal: simple polynomial-time (randomized) algorithm.
- Note: linear-time probably very hard to do!

Min-Cut Algorithm

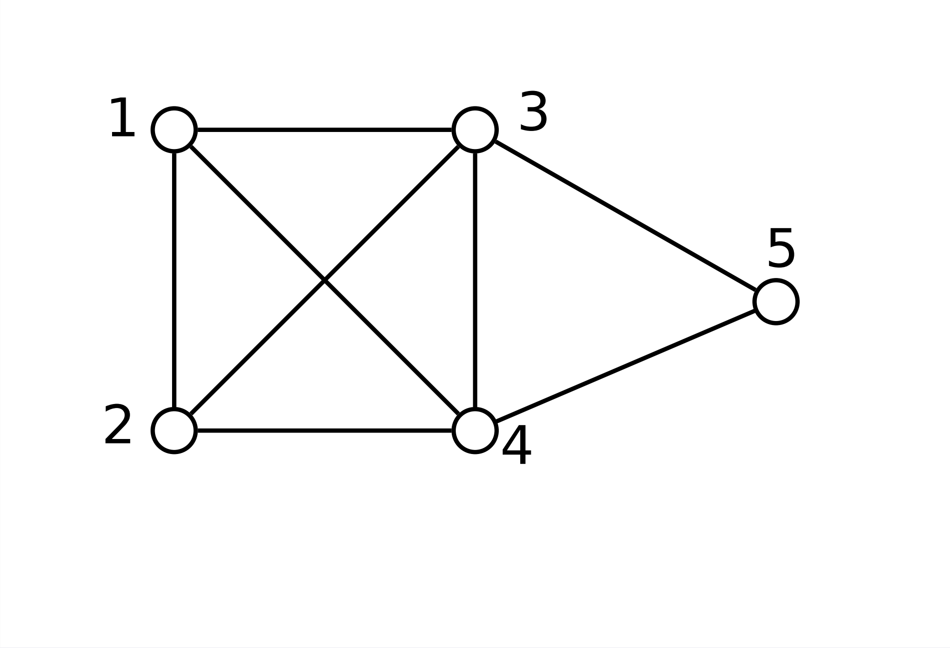
Algorithm for Min-Cut on multi-graphs (allow parallel edges).

1. If $n = 2$ output the trivial cut.
2. Otherwise, pick a random edge $(u, v) \in E$.
3. Contract (u, v) (i.e. merge u, v).
4. Go back to step 1.

Min-Cut Algorithm

Algorithm for Min-Cut on multi-graphs (allow parallel edges).

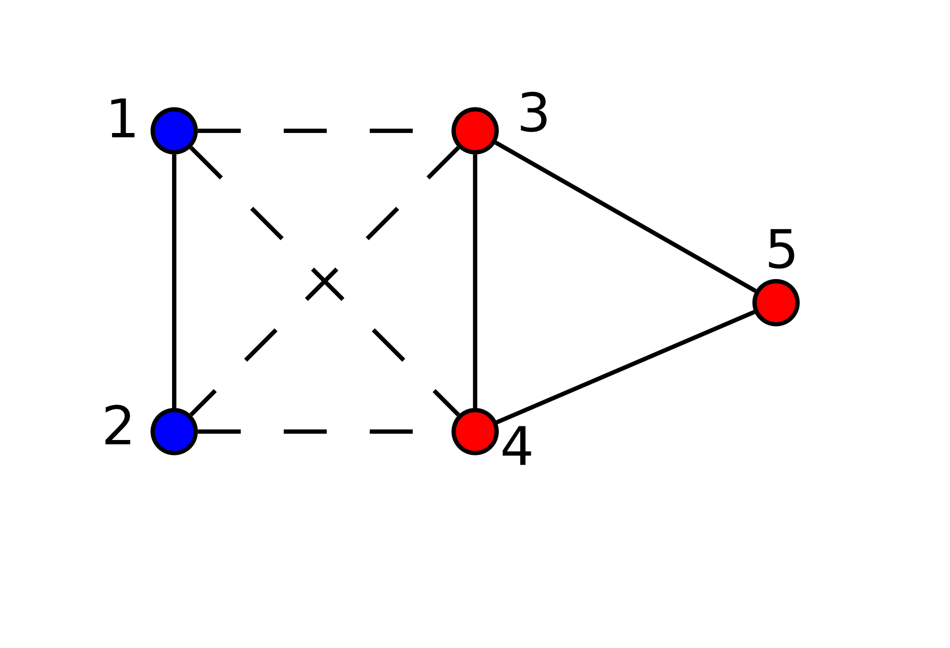
1. If $n = 2$ output the trivial cut.
2. Otherwise, pick a random edge $(u, v) \in E$.
3. Contract (u, v) (i.e. merge u, v).
4. Go back to step 1.



A possible input

Algorithm for Min-Cut on multi-graphs (allow parallel edges).

1. If $n = 2$ output the trivial cut.
2. Otherwise, pick a random edge $(u, v) \in E$.
3. Contract (u, v) (i.e. merge u, v).
4. Go back to step 1.

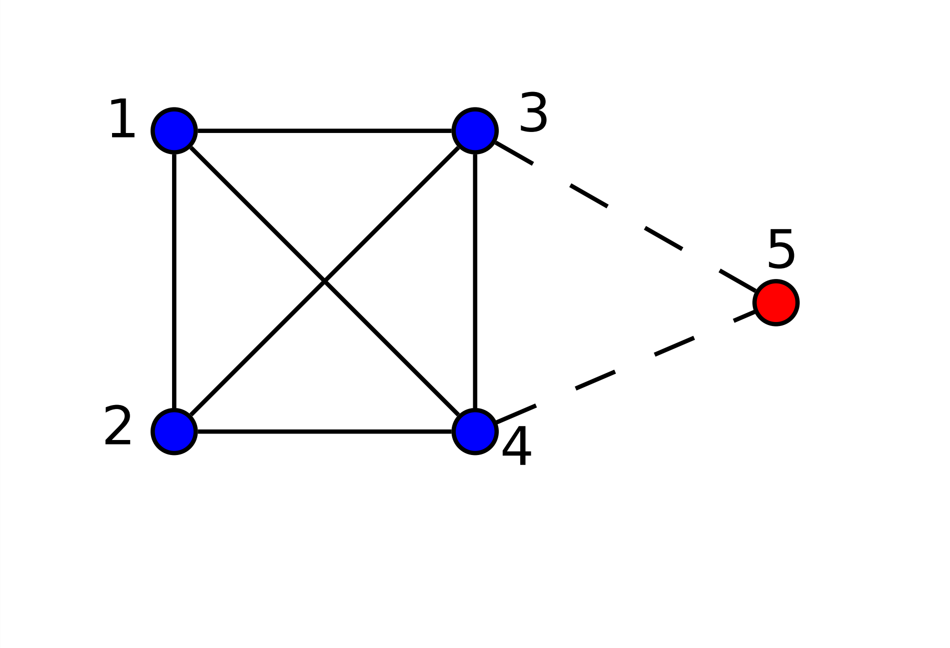


A bad solution

Min-Cut Algorithm

Algorithm for Min-Cut on multi-graphs (allow parallel edges).

1. If $n = 2$ output the trivial cut.
2. Otherwise, pick a random edge $(u, v) \in E$.
3. Contract (u, v) (i.e. merge u, v).
4. Go back to step 1.

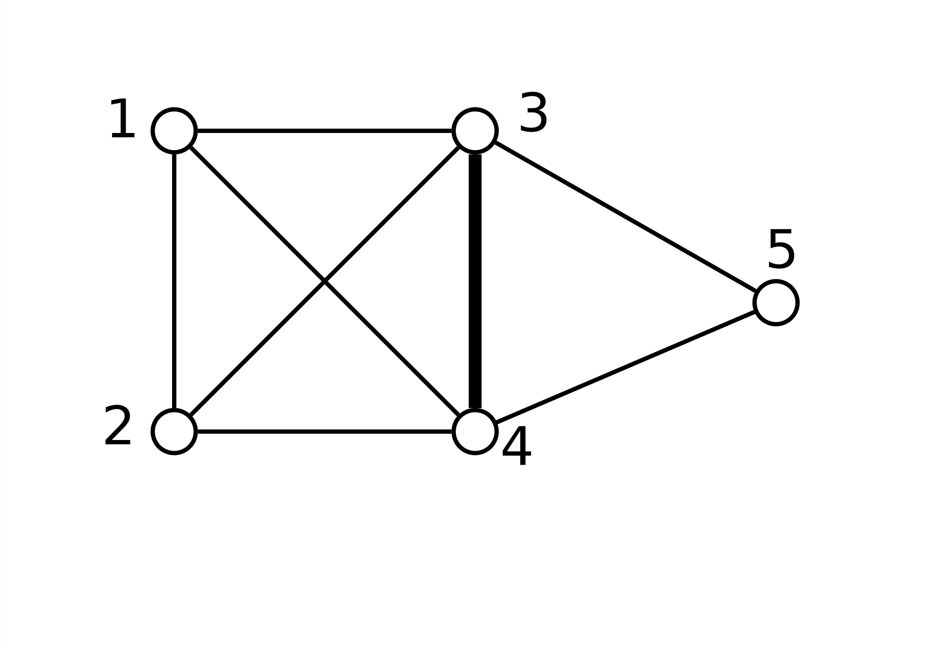


A good solution

Min-Cut Algorithm

Algorithm for Min-Cut on multi-graphs (allow parallel edges).

1. If $n = 2$ output the trivial cut.
2. Otherwise, pick a random edge $(u, v) \in E$.
3. Contract (u, v) (i.e. merge u, v).
4. Go back to step 1.

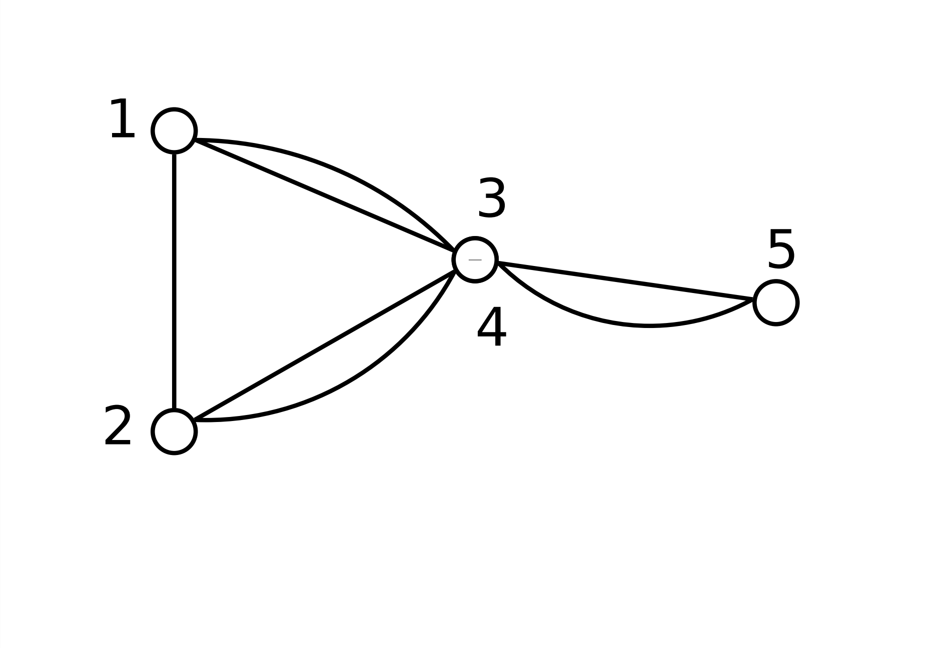


A "good" run of the algorithm:
We never contract an edge of the min cut.
Solution size $= 2$

Min-Cut Algorithm

Algorithm for Min-Cut on multi-graphs (allow parallel edges).

1. If $n = 2$ output the trivial cut.
2. Otherwise, pick a random edge $(u, v) \in E$.
3. Contract (u, v) (i.e. merge u, v).
4. Go back to step 1.

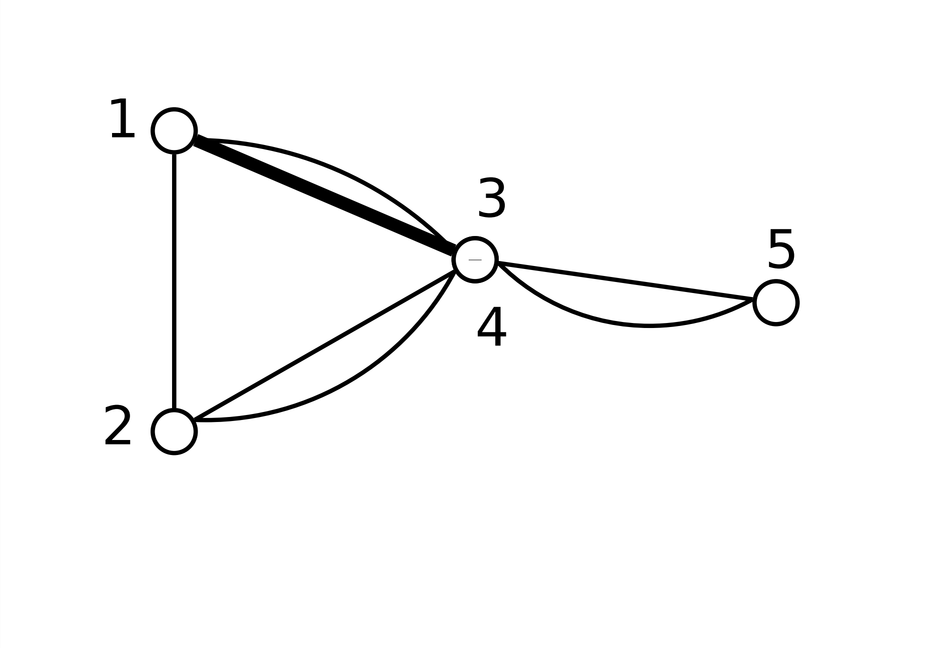


A "good" run of the algorithm:
We never contract an edge of the min cut.
Solution size $= 2$

Min-Cut Algorithm

Algorithm for Min-Cut on multi-graphs (allow parallel edges).

1. If $n = 2$ output the trivial cut.
2. Otherwise, pick a random edge $(u, v) \in E$.
3. Contract (u, v) (i.e. merge u, v).
4. Go back to step 1.

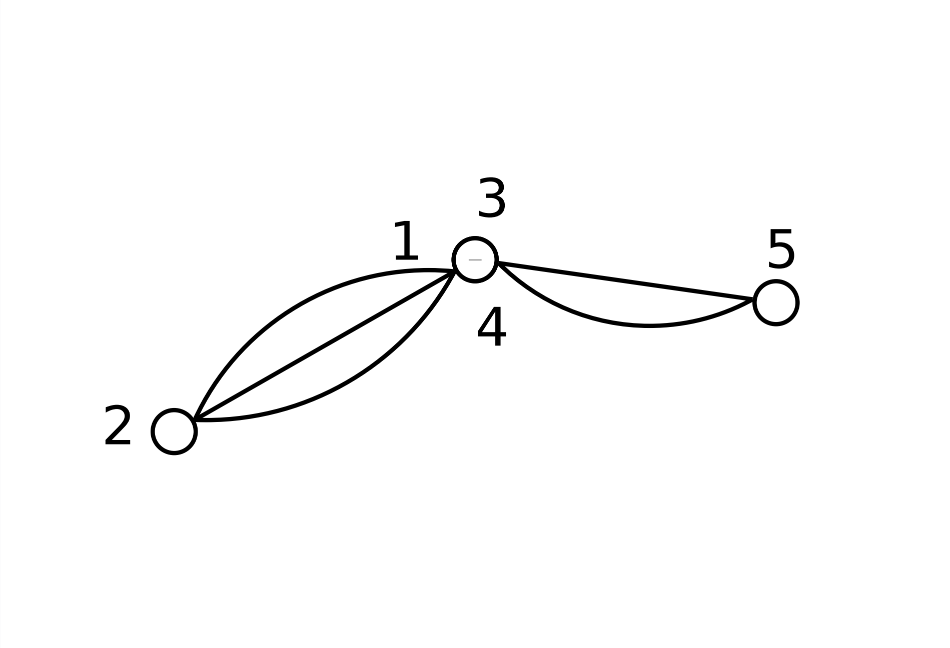


A “good” run of the algorithm:
We never contract an edge of the min cut.
Solution size $= 2$

Min-Cut Algorithm

Algorithm for Min-Cut on multi-graphs (allow parallel edges).

1. If $n = 2$ output the trivial cut.
2. Otherwise, pick a random edge $(u, v) \in E$.
3. Contract (u, v) (i.e. merge u, v).
4. Go back to step 1.

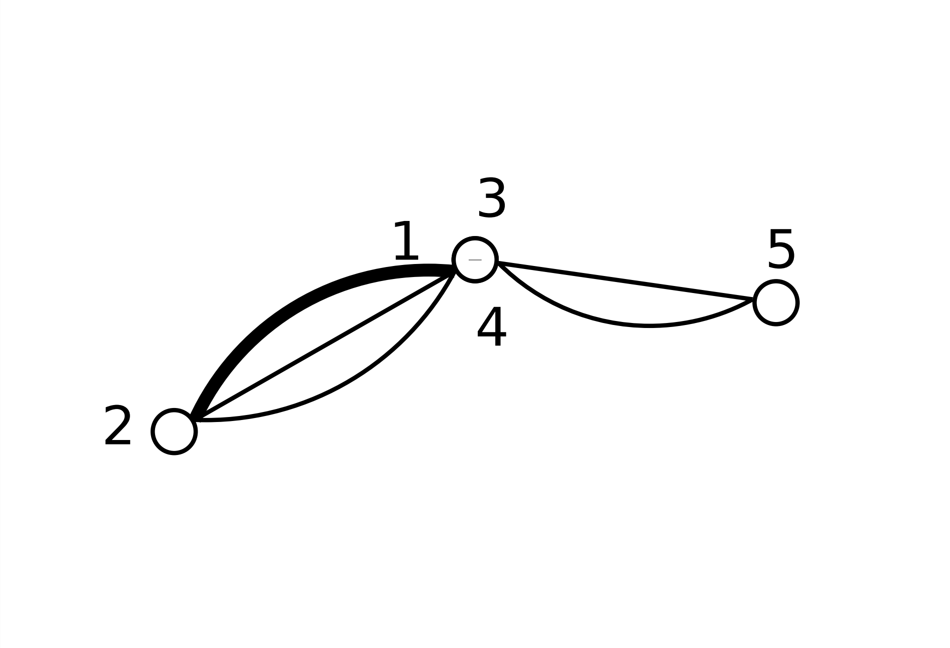


A "good" run of the algorithm:
We never contract an edge of the min cut.
Solution size $= 2$

Min-Cut Algorithm

Algorithm for Min-Cut on multi-graphs (allow parallel edges).

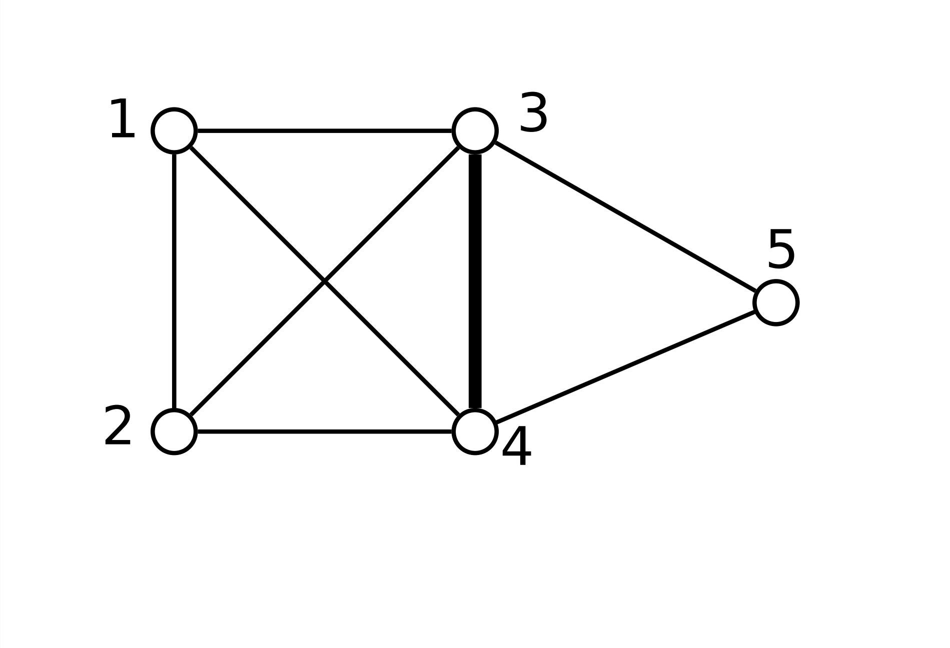
1. If $n = 2$ output the trivial cut.
2. Otherwise, pick a random edge $(u, v) \in E$.
3. Contract (u, v) (i.e. merge u, v).
4. Go back to step 1.



A "good" run of the algorithm:
We never contract an edge of the min cut.
Solution size $= 2$

Algorithm for Min-Cut on multi-graphs (allow parallel edges).

1. If $n = 2$ output the trivial cut.
2. Otherwise, pick a random edge $(u, v) \in E$.
3. Contract (u, v) (i.e. merge u, v).
4. Go back to step 1.

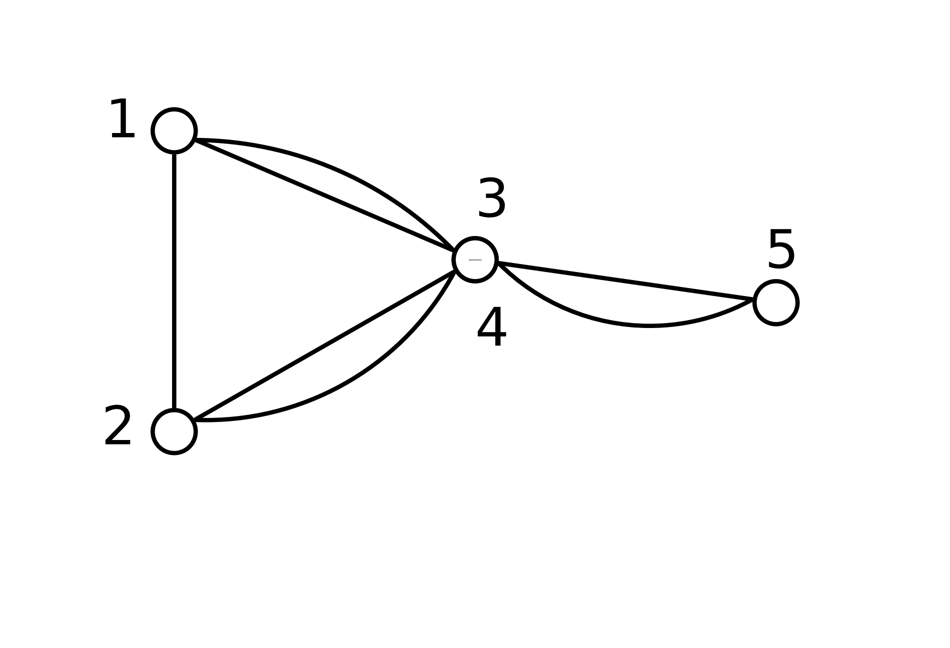


A “good” run of the algorithm:
We never contract an edge of the min cut.
Solution size $= 2$
A “bad” run of the algorithm:
We mistakenly contract an edge of the min cut.
Solution size ≥ 3

Min-Cut Algorithm

Algorithm for Min-Cut on multi-graphs (allow parallel edges).

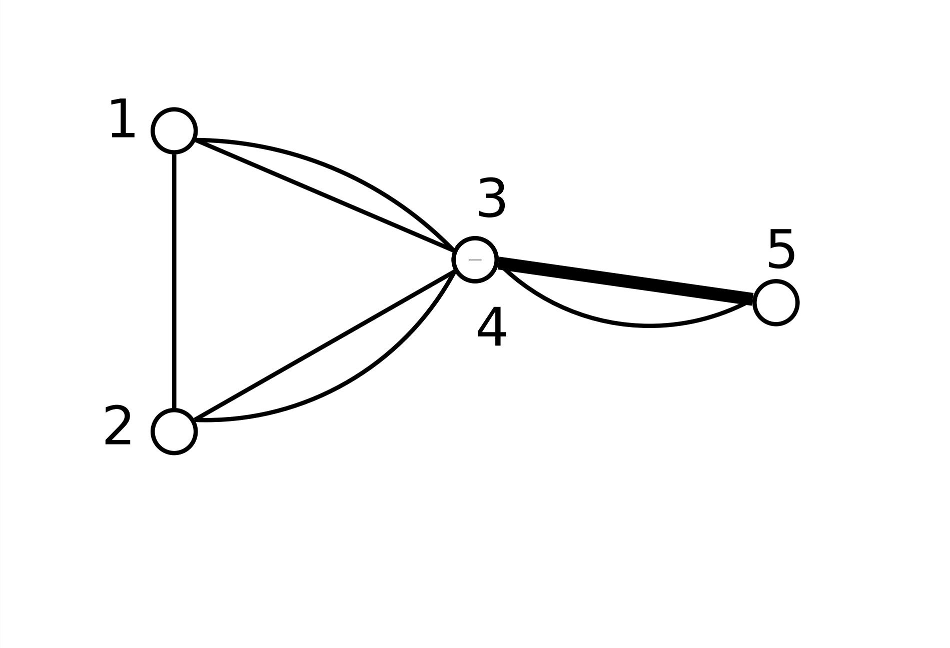
1. If $n = 2$ output the trivial cut.
2. Otherwise, pick a random edge $(u, v) \in E$.
3. Contract (u, v) (i.e. merge u, v).
4. Go back to step 1.



A “good” run of the algorithm:
We never contract an edge of the min cut.
Solution size $= 2$
A “bad” run of the algorithm:
We mistakenly contract an edge of the min cut.
Solution size ≥ 3

Algorithm for Min-Cut on multi-graphs (allow parallel edges).

1. If $n = 2$ output the trivial cut.
2. Otherwise, pick a random edge $(u, v) \in E$.
3. Contract (u, v) (i.e. merge u, v).
4. Go back to step 1.

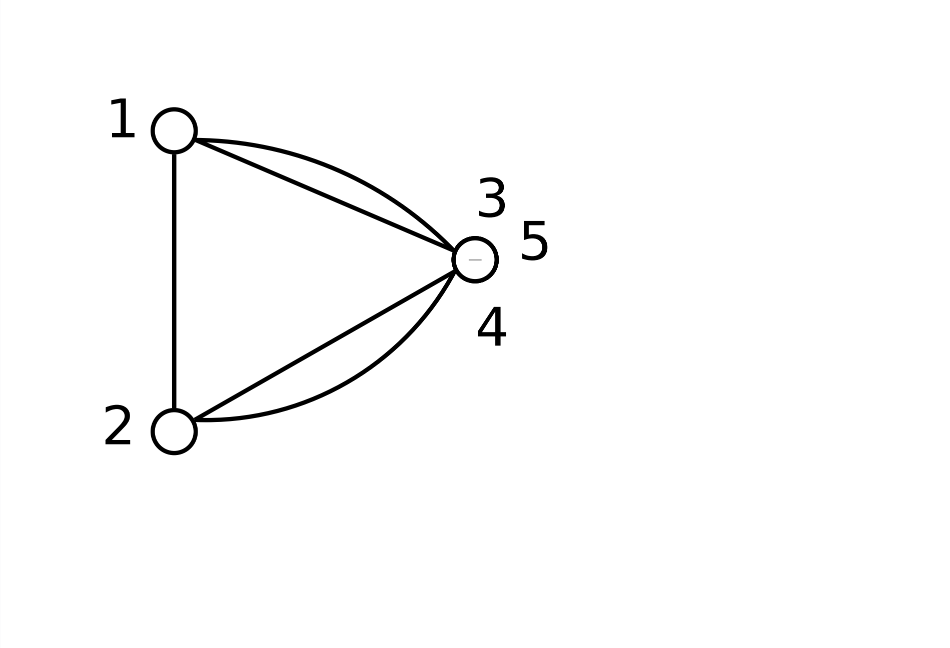


A "good" run of the algorithm:
We never contract an edge of the min cut.
Solution size $= 2$
A "bad" run of the algorithm:
We mistakenly contract an edge of the min cut.
Solution size ≥ 3

Min-Cut Algorithm

Algorithm for Min-Cut on multi-graphs (allow parallel edges).

1. If $n = 2$ output the trivial cut.
2. Otherwise, pick a random edge $(u, v) \in E$.
3. Contract (u, v) (i.e. merge u, v).
4. Go back to step 1.



A “good” run of the algorithm:
We never contract an edge of the min cut.
Solution size $= 2$
A “bad” run of the algorithm:
We mistakenly contract an edge of the min cut.
Solution size ≥ 3

Min-Cut Algorithm

Theorem: Algorithm of previous slide finds min cut with probability at least $\frac{1}{n^2}$.

- Suppose min cut size is k. Consider a specific min cut C.
- $\Rightarrow$ min degree is $\geq k$. Therefore, $|E| \geq kn/2$.
- Probability that algorithm avoids cut at first iteration: $\geq 1 - \frac{k}{kn/2} = 1 - \frac{2}{n} = \frac{n-2}{n}$.

Min-Cut Algorithm

Theorem: Algorithm of previous slide finds min cut with probability at least $\frac{1}{n^2}$.

- Suppose min cut size is k. Consider a specific min cut C.
- $\Rightarrow$ min degree is $\geq k$. Therefore, $|E| \geq kn/2$.
- Probability that algorithm avoids cut at first iteration: $\geq 1 - \frac{k}{kn/2} = 1 - \frac{2}{n} = \frac{n-2}{n}$.
- Probability that algorithm avoids cut at first iteration, assuming first iteration OK: $\geq 1 - \frac{k}{k(n-1)/2} = 1 - \frac{2}{n} = \frac{n-3}{n-1}$.

Min-Cut Algorithm

Theorem: Algorithm of previous slide finds min cut with probability at least $\frac{1}{n^2}$.

- Suppose min cut size is k. Consider a specific min cut C.
- $\Rightarrow$ min degree is $\geq k$. Therefore, $|E| \geq kn/2$.
- Probability that algorithm avoids cut at first iteration: $\geq 1 - \frac{k}{kn/2} = 1 - \frac{2}{n} = \frac{n-2}{n}$.
- Probability that algorithm avoids cut at first iteration, assuming first iteration OK: $\geq 1 - \frac{k}{k(n-1)/2} = 1 - \frac{2}{n} = \frac{n-3}{n-1}$.
- Probability that all iterations good:

$$\frac{n-2}{n}\frac{n-3}{n-1}\frac{n-4}{n-2}\cdots\frac{2}{4}\frac{1}{3} = \frac{2}{n(n-1)}$$

Min-Cut Algorithm

Theorem: Algorithm of previous slide finds min cut with probability at least $\frac{1}{n^2}$.

- Suppose min cut size is k. Consider a specific min cut C.
- $\Rightarrow$ min degree is $\geq k$. Therefore, $|E| \geq kn/2$.
- Probability that algorithm avoids cut at first iteration: $\geq 1 - \frac{k}{kn/2} = 1 - \frac{2}{n} = \frac{n-2}{n}$.
- Probability that algorithm avoids cut at first iteration, assuming first iteration OK: $\geq 1 - \frac{k}{k(n-1)/2} = 1 - \frac{2}{n} = \frac{n-3}{n-1}$.
- Probability that all iterations good:

$$\frac{n-2}{n}\frac{n-3}{n-1}\frac{n-4}{n-2}\cdots\frac{2}{4}\frac{1}{3} = \frac{2}{n(n-1)}$$

- Can repeat many (n^2) times to get better ($\Omega(1)$) probability.
- Better idea to run the algorithm until graph small, then use some other algorithm (notice probability of success keeps falling).