Probabilistic Analysis and Randomized Algorithms

Pedro Ribeiro

DCC/FCUP

2018/2019

Worst-Case Running Time

- We are interested on how the running time **scales** with input size
- Normally we are interested in **worst case** running time (worst case of all inputs of a given size)

Definition - Worst-Case Running Time

I - some input
T(**I**) - running time for input *I*
T(**n**) - worst-case running time for input size *n*

$$\mathbf{T(n) = max\{T(I)\}_{inputs\ I\ of\ size\ n}}$$

Average-Case Running Time

- We could be interested in **average-case**
- Measure performance in **"typical"** inputs
  - What is a typical input?
- There are algorithms with **large** gap between "average performance" and worst case.
- Can we **improve** worst-case by adding randomization?

The hiring problem

- Imagine you need to hire a new office assistant
- Suppose you use the following algorithm:

Hiring Algorithm for n candidates

```
$best = 0$ // candidate 0 is a least-qualified dummy candidate
for $i = 1$ to $n$
  interview candidate $i$
  if candidate $i$ is better than candidate $best$
    $best = i$
    hire candidate $i$
```

- We will focus not on the running time, but instead on the **costs incurred by interviewing and hiring**.
  - Interviewing has a low cost (let's call it c_i)
  - Hiring has an higher cost (let's call it c_h)

The hiring problem: cost

- What is the cost of the algorithm?
- Suppose you end up hiring m persons out of the n candidates
- The total cost is $\mathcal{O}(\mathbf{c_i n + c_h m})$:
  - No matter how many people we hire, we always interview n candidates
  - The cost associated with interviewing is always used: $c_i n$
  - We can therefore **focus on the hiring cost** ($c_h m$), which varies according to the candidates and the order in which they are interviewed
- Note how this models a common algorithm paradigm:
  - finding a min or max value in a sequence and how often we update the notion on who's winning

The hiring problem: cost

- Assume we have a **total order** among all the candidates
  - We can compare two candidates and decide which one is better
  - We can rank each candidate with an unique number from 1 to n. We will use $rank(i)$ to denote the rank of candidate i (assume an higher rank means a better candidate)
- The input can now be described as a permutation of $\langle 1, 2, \ldots, n \rangle$
- **Worst-case analysis**
  - What is the worst possible input?
  - If the candidates come in **increasing ranking order**, then we will need to hire all of them: $\mathcal{O}(c_h n)$
  - Good thing this is not always the case... In fact we don't know in which order will they come. What would happen on a typical case? (what would the best-case scenario be?)

Probabilistic Analysis

Probabilistic Analysis

The use of **probability** in the analysis of algorithms:

- We must have some knowledge or make assumptions about the **distribution of the input**
- We can then make an **average-case analysis**, averaging the cost over all possible inputs
- For our problem, we could assume all possible permutations are **equally likely** (the ranks form an **uniform random permutation**).
    - But what if the above is not the case? What if the real distribution is skewed and some permutations are more likely than other?

Hiring Problem: a randomized algorithm

Randomized Hiring Algorithm for n candidates

```
randomly permute the list of candidates // the only "new" line
$best = 0$ // candidate 0 is a least-qualified dummy candidate
for $i = 1$ to $n$
    interview candidate $i$
    if candidate $i$ is better than candidate $best$
        $best = i$
        hire candidate $i$
```

- In order to have **great control over the order** of the candidates, we could explicitly choose **randomly** which candidate to interview next
  - We now are enforcing a **random order**, regardless of the input!

Randomized Algorithms

Randomized algorithms

We call an algorithm **randomized** if its behavior is determined not only by its input but also by values produced by a **random-number generator**

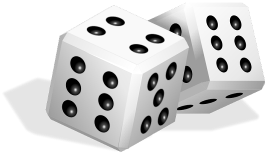
- Most programming environments offer a (deterministic) **pseudorandom-number generator**: it returns numbers that *"look"* statistically random
- We typically refer to the analysis of randomized algorithms by talking about the **expected cost** (ex: the **expected running time**)
- We can use **probabilistic analysis** to analyse randomized algorithms

Basics of Probabilistic Analysis

- Consider rolling **two dice** and observing the results.
- We call this an **experiment**.
- It has **36 possible outcomes**:

  1-1, 1-2, 1-3, 1-4, 1-5, 1-6, 2-1, 2-2, 2-3, ..., 6-4, 6-5, 6-6
- Each of these outcomes has probability $\mathbf{1/36}$ (assuming fair dice)
- What is the probability of the sum of dice being 7?

  Add the probabilities of all the outcomes satisfying this condition: 1-6, 2-5, 3-4, 4-3, 5-2, 6-1 (probability is $\mathbf{1/6}$)



Basics of Probabilistic Analysis

In the language of probability theory, this setting is characterized by a **sample space** S and a **probability measure** p.

- **Sample Space** is constituted by all possible outcomes, which are called **elementary events**
- In a **discrete probability distribution** (d.p.d.), the probability measure is a function $p(e)$ (or $Pr(e)$) over elementary events e such that:
    - $p(e) \geq 0$ for all $e \in S$
    - $\sum_{e \in S} p(e) = 1$
- An **event** is a subset of the sample space.
- For a d.p.d. the probability of an event is just the **sum** of the probabilities of its elementary events.

Basics of Probabilistic Analysis

- A **random variable** is a function from elementary events to integers or reals:

  Ex: let X_1 be a random variable representing result of first die and X_2 representing the second die.
  $X = X_1 + X_2$ would represent the sum of the two
  We could now ask: what is the probability that $X = 7$?

- One property of a random variable we care is **expectation**:

Expectation

For a discrete random variable X over sample space S, the expected value of X is:

$$\mathbf{E}[X] = \sum_{e \in S} Pr(e)X(e)$$

Basics of Probabilistic Analysis

- In **words**: the expectation of a random variable X is just its average value over S, where each elementary event e is weighted according to its probability.

  Ex: If we roll a single die, the expected value is 3.5
  (all six elementary events have equal probability).

- One possible rewrite of the previous equation, grouping elementary events:

Expectation (possible rewrite)

$$\mathbf{E}[X] = \sum_a Pr(X = a)a$$

Basics of Probabilistic Analysis

- More generally:

Expectation (rewrite using disjoint events)

For any partition of the sample space into disjoint events $A_1, A_2, \ldots$:

$$\mathbf{E}[X] = \sum_i \sum_{e \in A_i} Pr(e)X(e) = \sum_i Pr(A_i)\,\mathbf{E}[X|A_i]$$

- $\mathbf{E}[X|A_i]$ is the expected value of X given A_i, defined to be: $\frac{1}{Pr(A_i)} \sum_{e \in A_i} Pr(e)X(e)$.

Basics of Probabilistic Analysis

An important fact about expected values is **Linearity of Expectation**:

Theorem - Linearity of Expectation

For any two random variables X and Y: $\mathbf{E[X+Y]} = \mathbf{E[X]} + \mathbf{E[Y]}$

Proof for discrete random variables:

$$\mathbf{E}[X+Y] = \sum_{e \in S} Pr(e)(X(e) + Y(e)) =$$
$$= \sum_{e \in S} Pr(e)X(e) + \sum_{e \in S} Pr(e)Y(e) = \mathbf{E}[X] + \mathbf{E}[Y]$$

- It is not necessary that the variables are independent
- This theorem is **very important for the analysis of algorithms**: complicated variables become a sum of simple variables which we can analyse separately.

A first example

Suppose we unwrap a fresh deck of n cards and **shuffle** it until the cards are completely random.

How many cards do we expect to be in the same position as they were at the start?

- X: number of cards that end in the same position as they started
- We are looking for $\mathbf{E}[X]$!
- By linearity of expectation we can write this as a sum of X_i, where $X_i = 1$ if the i-th card ends up in position i, and $X_i = 0$ otherwise: $X = X_1 + X_2 + \ldots + X_n = \sum_{i=1}^{n} X_i$
- $Pr(X_i = 1) = 1/n$ where n is the number of cards!
- $Pr(X_i = 1)$ is also $\mathbf{E}[X_i]$ $\quad (\mathbf{E}[X_i] = 1 \cdot Pr(X_i = 1) + 0 \cdot Pr(X_i = 0))$
- $\mathbf{E}[X] = \mathbf{E}[X_1 + \ldots + X_n] = \mathbf{E}[X_1] + \ldots + \mathbf{E}[X_n] = 1$

Indicator Variables

- In the previous example we used an **indicator random variable**:

Indicator Random Variable

The indicator random variable $\mathbf{I}\{\mathbf{A}\}$ associated with event A is defined as:

$$I\{A\} = \begin{cases} 1 & \text{if } A \text{ occurs} \\ 0 & \text{if } A \text{ does not occur} \end{cases}$$

- Indicator random variables may be very handy in simplifying our analysis, by giving us a **simpler** way to model our desired cost
- Note that if $X_A = I\{A\}$, then $\mathbf{E}[X_A] = Pr(A)$
  $\mathbf{E}[X_A] = 1 \cdot Pr(A) + 0 \cdot Pr(\overline{A})$ where $\overline{A}$ is the complement of A)

Example using indicator variables - Birthday Paradox

- Suppose we have n persons in a room. What is the **expected number of persons having the same birthday**?
- What is the probability that any two persons i and j have the same birthday? (suppose birthdays are independent)
  - Let's assume a year has $y = 365$ days, all equally likely for a birthday
  - Let b_k the birthday of person k
  - Probability of two persons having as birthday the day d is:
    $Pr(b_i = d \text{ and } b_j = d) = Pr(b_i = d) \cdot Pr(b_j = d) = \frac{1}{y^2}$
  - **Probability of two persons having the same birthday is:**
    $$Pr(b_i = b_j) = \sum_{d=1}^{y} Pr(b_i = d \text{ and } b_j = d) = \sum_{d=1}^{y} \frac{1}{y^2} = \mathbf{\frac{1}{y}}$$
  - More intuitively, after we choose the first birthday b_i, the probability that b_j is the same is $1/y$

Example using indicator variables - Birthday Paradox

- Let's use the following random indicator variable:
  $X_{ij} = \begin{cases} 1 & \text{if persons } i \text{ and } j \text{ have the same birthday} \\ 0 & \text{otherwise} \end{cases}$
- $\mathbf{E}[X_{ij}] = Pr(i \text{ and } j \text{ having the same birthday}) = 1/y$
- Let X be the random variable that counts the **number of pairs of people with the same birthday**:
  $X = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}$
- Taking expectation of both sides and applying linearity of expectations:
  $\mathbf{E}[X] = \mathbf{E}[\sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbf{E}[X_{ij}] = \binom{n}{2}\frac{1}{y} = \mathbf{\frac{n(n-1)}{2y}}$

Example using indicator variables - Birthday Paradox

- Expected number of persons having the same birthday: $\mathbf{\frac{n(n-1)}{2y}}$
  (n is the number of persons in the room; y the number of days in an year)
- How many persons so that we can expect that (at least) one pair of persons will have same birthday?
  - When $n(n-1) \geq 2y$, $\mathbf{E}[X] \geq 1$
  - For the case of $y = 365$ the smallest integer is $\mathbf{n} = \mathbf{28}$:
    $28 \times 27 = 756 > 2 \times 365 = 730$
  - For a given y, the answer is in the order of $\boldsymbol{\Theta}(\sqrt{\mathbf{y}})$
    (is this what your intuition told you when you heard the problem?)

Hiring Problem: a randomized algorithm

Back to our example:

Randomized Hiring Algorithm for n candidates

```
randomly permute the list of candidates
$best = 0$ // candidate 0 is a least-qualified dummy candidate
for $i = 1$ to $n$
  interview candidate $i$
  if candidate $i$ is better than candidate $best$
    $best = i$
    hire candidate $i$
```

- What is the **expected hiring cost** of this algorithm?

Hiring Problem: probabilistic analysis

- Let's use the following random indicator variable:
  Consider the event the *i*-th iteration of the loop in the algorithm:
  $$X_i = \begin{cases} 1 & \text{if candidate } i \text{ is hired} \\ 0 & \text{otherwise} \end{cases}$$
- $\mathbf{E}[X_i] = Pr(\text{candidate } i \text{ is hired})$
- Because the order is random:
  - Cand. i has prob. $1/i$ of being better than cand. 1 through $i-1$ (can you see why?)
  - $\mathbf{E}[X_i] = 1/i$
- Let X be the **number of candidates we hire**. Then:
  $$X = X_1 + X_2 + \ldots + X_n$$
  $$\mathbf{E}[X] = \mathbf{E}[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} \mathbf{E}[X_i] = \sum_{i=1}^{n} 1/i = \ln(n) + \mathcal{O}(1)$$
  [the sum $H_n = 1 + \frac{1}{2} + \cdots + \frac{1}{n}$ is known as the **harmonic series** and $H_n = \ln(n) + \mathcal{O}(1)$]
- We interview n people, but we only hire approximately $\ln(n)$ of them!
- The **expected hiring cost** is $\mathcal{O}(c_h \log n)$

Quicksort

Quicksort Algorithm

Given array with n elements

1. Pick an element p of the array as the **pivot** (or halt if the array has size 0 or 1).
2. **Split** the array into sub-arrays LESS, EQUAL, and GREATER by comparing each element to the pivot
   - **LESS** has all elements less than p
   - **EQUAL** has all elements equal to p
   - **GREATER** has all elements greater than p.
3. **recursively** sort LESS and GREATER.

The algorithm is not fully specified: how to pick the **pivot?**

Naive Quicksort

For a first version let's do the following:

Naive Quicksort

Run Quicksort as given before, each time choosing the **leftmost** element as the pivot

You can see an animation in the VisuAlgo website.

Naive Quicksort

Worst Case

What is the **worst-case running time**?

Image the array is **already sorted**:

- The pivot will be the smallest element
- In step 2, all other elements will go to GREATER
- Since the GREATER array will be sorted, the process will continue, each time with one less element
- This will result in $\mathbf{\Omega(n^2)}$ time.
- Since step 1 is executed at most n times, and step 2 takes at most n steps, time will be $\mathcal{O}(\mathbf{n^2})$
- Thus, the worst-case running time is $\mathbf{\Theta(n^2)}$

(what about the best case)?

Naive Quicksort

Average Case

- It turns out that the **average-case** running time is $\mathcal{O}(\mathbf{n} \log \mathbf{n})$ (averaged over all different orderings of n elements)
- Small consolation if the inputs we have are the bad ones... (ex: almost sorted arrays)
- Can we get around this problem?

Randomized Quicksort

Let's now do the following:

Randomized Quicksort

Run Quicksort as given before, each time choosing **random** element as the pivot

You can see an animation in the VisuAlgo website.

Randomized Quicksort

What is now the **worst-case running time**?

- We will prove that for **any** given input array I, the expected time of this algorithm, $\mathbf{E}[\mathbf{T}(\mathbf{I})]$, is $\mathcal{O}(\mathbf{n} \log \mathbf{n})$.
- This is the **Worst-Case Expected Time** bound.
- Better than the average-case bound: we are **no longer assuming anything** from the input!
  - Ex: if the input is almost sorted, it will not affect this.
- Peculiar, as before: as making the algorithm **probabilistic** gives us **more control** over the running time.

Analysing Randomized Quicksort

The running time of quicksort is **dominated by the number of direct comparisons between elements**, made when we are comparing with the pivot and splitting the array into LESS, EQUAL and GREATER.

(all the other parts take $\mathcal{O}(n)$, as selecting the pivot takes $\mathcal{O}(1)$ and there will be $< 2n$ calls to the recursive function)

Theorem - Comparisons in Randomized Quicksort

The expected number of comparisons made by randomized quicksort on an array of size n is $\mathcal{O}(n \log n)$.

Let us prove this theorem using... indicator variables!

Analysing Randomized Quicksort

Let X_{ij} be a random indicator variable with value:

- 1 if the algorithm does compare the i-th smallest and j-th smallest elements in the course of sorting
- 0 if it does not

Let X denote the total number of comparisons made by the algorithm. Since we never compare the same pair of elements twice, we have:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

And therefore

$$\mathbf{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbf{E}[X_{ij}]$$

Analysing Randomized Quicksort

Consider one X_{ij}, for $i < j$.

Consider e_k to be the element on the k-th position, and imagine the elements in sorted order.

- If the pivot is between e_i and e_j, then they will go to separate buckets and we never compare them
- If the pivot is e_i or e_j, then we do compare them
- If the pivot is smaller than e_i or greater than e_j, then they will go to the same bucket and we need to choose another pivot

Analysing Randomized Quicksort

$Pr(X_{ij} = 1) = Pr(e_i$ or e_j is selected as pivot in the interval $[i, j])$

This interval has size $j - i + 1$, and only 2 pivots out of these positions (precisely i and j) would give origin to a comparison between e_i and e_j

Therefore, overall, the probability that $X_{ij} = 1$ is $2/(j - i + 1)$.
(In words, e_i is compared to e_{i+1} with probability 1, e_i is compared to e_{i+2} with probability $2/3$, e_i is compared to e_{i+2} with probability $2/4$, and so on)

$$\mathbf{E}[x] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbf{E}[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k}$$

Analysing Randomized Quicksort

$\mathbf{E}[X] < \sum_{i=1}^{n-1} \sum_{k=i}^{n} \frac{2}{k} = \sum_{i=1}^{n-1} 2(H_n - 1)$

Remember that $1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{n}$ is denoted as H_n, and is called the n-th **Harmonic Number** (as we saw before), and that $H_n = \mathcal{O}(\log n)$

Therefore:

$\mathbf{E}[x] < 2n(H_n - 1) = \mathcal{O}(n \log n)$ (We have proved what we wanted!)

In terms of the number of comparisons it makes, **Randomized Quicksort is equivalent to randomly shuffling the input and then handing it off to Naive Quicksort**.

So, we have also proven that Naive Quicksort has $\mathcal{O}(n \log n)$ average-case running time.

Types of Randomized Algorithms

- QuickSort always returns a correct result (a sorted array) but its **runtime is a random variable** (with $\mathcal{O}(n \log n)$ in expectation)
- We are now going to see a different type of randomized algorithm: the runtime is fixed, but **it may give incorrect answers** some of the time.

Las Vegas Algorithms

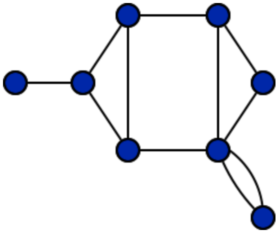
Randomized algorithms which always output the correct answer, and whose runtimes are random variables, are called **Las Vegas algorithms**.

Monte Carlo Algorithms

Randomized algorithms which always terminate in given time bound, but output the correct answer with at least some (high) probability (say with $3/4$ prob.) are called **Monte Carlo algorithms**.

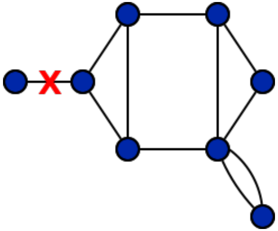
The Minimum Cut Problem

- We are now going to present a simple but elegant Monte Carlo randomized algorithm for the **Minimum Cut Problem**
- A **multigraph** is a graph where there might be more than one edge between the same pair of nodes. In this lecture, when we say graph, we mean a multigraph.
- Consider a graph $G = (V, E)$, with number of vertices $|V| = n$ and number of edges $|E| = m$. The figure below shows an example of a graph with $n = 8$ nodes and $m = 11$ edges.



The Minimum Cut Problem

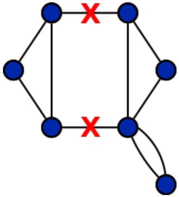
- Given a graph G, a **cut** is a set of edges whose removal **splits the graph into at least two connected components**. The edges of a cut are know as ***crossing edges***.
- A **minimum cut** is a cut of minimum size, that is, with the minimum possible number of edges in it.
- For the example graph given above, the minimum cut has size 1:



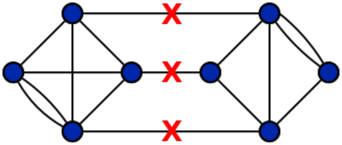
The Minimum Cut Problem

- For other graphs the minimum cut might be bigger:

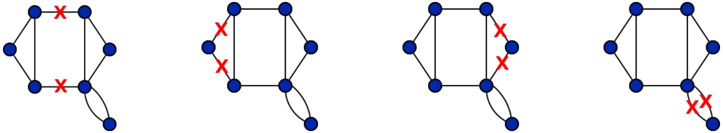
min cut of size 2



min cut of size 3



- There might be more than one minimum cut in the same graph:



- The minimum cut it at most equal to the minimum degree of any node in G. Can you see why?

The Minimum Cut Problem
Motivation

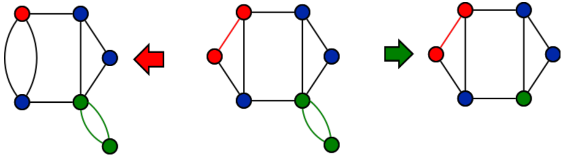
- This problem has a long history and is applicable in many areas:
  - Imagine you want to know how ***"robust"*** is a network in the sense of the minimum number of links whose failure disconnects the network
  - Imagine you want to ***partition a graph*** into two groups of nodes with the least amount of connections between them
  - ...
- The classical way to solve this problem is to use **maximum flow algorithms**, whose most efficient form can be very complex
- Today we will describe a simple **randomized algorithm** for this task
  (The algorithm was discovered by David Karger when he was a PhD student)

The Minimum Cut Problem

Contracting an Edge

- Karger's algorithm uses a primitive graph operation called **contracting** or **collapsing** an edge.
- To contract an edge (u, v), we create a new node uv, keeping all the edges between all u and v to outside nodes, and removing the self-loops in the new node.

  The following figure gives two examples of edge contractions:



- We will use the notation $\mathbf{G}/(\mathbf{u}, \mathbf{v})$ to denote the resulting graph after contracting the edge (u, v)

The Minimum Cut Problem

Contracting an Edge

- What happens to the minimum cut when we contract an edge?
  - For any cut in $G/(u,v)$, there is a cut in G with the exact same number of crossing edges
    (note that the converse is not necessarily true)
  - This implies that an edge contraction **cannot decrease** the minimum cut size
  - Moreover, it only increases the minimum cut size if the contracted edge is part of all possible minimum cuts!
- These observations are at the heart of Karger's Algorithm.

The Minimum Cut Problem

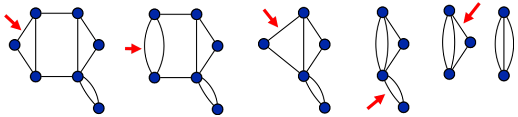
Karger's Algorithm - a first version

GuessMinCut(G) - version #1

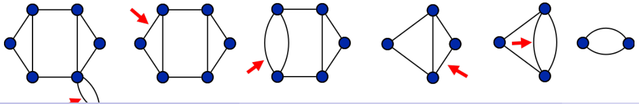
1. Repeat while there are still more than 2 nodes in the graph
   - Pick an edge (u, v) uniformly at random in current graph G and **contract** it
2. Return the only existing cut in the graph G

- Let's see examples of the algorithm running:

A run which returns a non-optimal minimum cut (with 3 edges):



A run which actually returns an optimal cut (with 2 edges):



The Minimum Cut Problem

Probability of being correct

- An easy and simple algorithm. But how well does it do?

Theorem

GuessMinCut(G) version #1 returns a minimum cut with probability at least:

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

- It seems like a tiny probability of being correct... (and is!)

  (for a 1000 node graph, we are claiming a 1-in-million chance of being correct!)

- On the other hand, out of the 2^m possible subsets of edges, this simple algorithm zooms in a minimum cut with probability $1/n^2$... pretty awesome, really!

  (we will soon see how to make the probability much better)

The Minimum Cut Problem

Probability of being correct

Let's prove the theorem of the previous slide:

- There might be several minimum cuts in G. Let's fix some minimum cut C with k edges.
- When we are at a graph with $n - i$ nodes ($i \geq 0$):
  - We will have at least $k(n-i)/2$ edges: every node must have at least degree k and every edge is incident to two nodes.
  - If we choose edges uniformly at random, the probability we choose an edge of C is therefore at most k out of $k(n-i)/2$ which is $2/(n-i)$.
  - In other words, the probability that we don't "screw up" by choosing an edge of the minimum cost on this iteration is at least $1 - 2/(n-i)$.

(continues on next slide)

The Minimum Cut Problem

Probability of being correct

- The probability that the cut C survives the first iteration is at least $1 - \frac{2}{n} = \frac{n-2}{n}$
- More generally, the probability that the cut C survives the $(i+i)$-th iteration is at least $1 - \frac{2}{n-i} = \frac{n-i-2}{n-i}$
- So, the probability that the cut survives all the $(n-2)$ iterations until we are left with a graph of two nodes is:

$$
\begin{aligned}
&\prod_{i=0}^{n-3} Pr[C \text{ survives round } \#(i+1)] \\
\geq &\prod_{i=0}^{n-3} \frac{n-i-2}{n-i} \\
= &\frac{n-2}{n} \times \frac{n-3}{n-1} \times \frac{n-4}{n-2} \times \ldots \frac{1}{3} \\
= &\frac{2}{n(n-1)} \quad \square \text{(and our proof is finished)}
\end{aligned}
$$

The Minimum Cut Problem

Running time

- What about the **running time**?
- Each iteration implies one contraction and costs $\mathcal{O}(n)$ time (can you see how to implement it?)
- The entire algorithm takes therefore $\mathcal{O}(n^2)$
  (we could also use a minimum spanning tree to obtain an equivalent formulation for a $\mathcal{O}(m \log m)$ running time, but we will not discuss it here)
- **We therefore now have a simple Monte Carlo algorithm with time $\mathcal{O}(\mathbf{n^2})$ and with probability of being correct at least $\frac{1}{\mathbf{n^2}}$**

  We want something better, and we will now see how to improve (a lot) this probability.

The Minimum Cut Problem

Improving the Success Probability

- One way to improve the success probability is just to... ***try, try again***!

GuessMinCut(G, M) - (improved) version #2

1. Run the previous GuessMinCut(G) algorithm M times, independently
2. Return the smallest cut found in these M runs

- The **runtime** of the algorithm is clearly $\mathcal{O}(Mn^2)$
- What about the **probability**?
  - We are happy if at least one of the runs finds a minimum cut C
  - So, if we fail, *all runs must have failed to find* C! What are the odds?

$$\begin{aligned}
&Pr(\text{all } M \text{ runs fail})\\
&= Pr(\text{1st run fails}) \times Pr(\text{2nd run fails}) \times \ldots \times Pr(M\text{-th run fails})\\
&\leq (1-\frac{1}{n^2}) \times (1-\frac{1}{n^2}) \times \ldots \times (1-\frac{1}{n^2})\\
&= (1-\frac{1}{n^2})^M
\end{aligned}$$

The Minimum Cut Problem

Improving the Success Probability

- The failure probability is at most $(1 - \frac{1}{n^2})^M$

One of most useful inequalities ever

$(1 + x) < e^x$ for all $x \in \mathbb{R}$

- Using this inequality, the failure probability is at most e^{-M/n^2}
- If we repeat the algorithm $M = n^2 \ln n$ times, the failure probability is at most $e^{-\ln n} = 1/n$
- In fact, if we repeat $M = cn^2 \ln n$, the failure probability is at most $e^{-c \ln n} = 1/n^c$
- We went from *small probability of success* $(1/n^2)$ to a *small probability of failure* $(1/n^c)$!
- **We have now a simple Monte Carlo algorithm with time $\mathcal{O}(\mathbf{cn^4} \log \mathbf{n})$ and with probability of being incorrect at most $\frac{\mathbf{1}}{\mathbf{n^c}}$**

Monte Carlo Algorithms

Success Probability

- Recall this important *"trick"* with Monte Carlo algorithms:

Amplification

If you have have a Monte Carlo algorithm with a small probability of success, you can usually repeat the algorithm independently many times and reduce the failure probability.

- When the failure probability is a polynomial fraction (like $1/n^c$), we say that the algorithm is correct with ***high probability***

The Minimum Cut Problem

Improving the runtime

- $\mathcal{O}(n^4 \log n)$ seems still a little high cost for our minimum cut randomized algorithm. Can we **improve the runtime**?
- Notice that the *initial iterations are less riskier the final ones*:
  - In the first iteration our chance of "screwing up" is at most $2/n$
  - On the last iteration this goes up to $2/3$!
- We are now going to talk about a **speed-up idea** due to D. Karger and C. Stein (the "S" in "CLRS"):
- First, we should **run the normal algorithm in a *"safe"* phase** until we have t nodes remaining:
  - What would a good value of t be? Let's try to repeat this while we still have a cumulative chance of success higher than $1/2$
  - Doing the calculations like before (see slide 44), the probability that we still have a minimum cut when we have t nodes left is: $\frac{t(t-1)}{n(n-1)}$
  - When we have about $t = \frac{n}{\sqrt{(2)}}$, then $\frac{(n/\sqrt{2})\cdot(n/\sqrt{(2)}-1)}{n(n-1)} \approx \frac{1}{2}$

The Minimum Cut Problem

Improving the runtime

- The full improved algorithm is the following:

BetterGuess(G) - (improved) version #3

```
If $n \leq 8$ then     % ($n = |V(G)|$)
    find the min-cut by brute force
Else
    $H \leftarrow$ graph $G$ contracted until we reach $n/\sqrt{2}$ nodes
    $X_1 \leftarrow$ BetterGuess($H$)
    $X_2 \leftarrow$ BetterGuess($H$)
    return $min(X_1, X_2)$
```

- The dual recursive call is there to improve the probability of success
- What is now the **running time**?
  - This recursion corresponds to the recurrence:
    $T(n) = 2T(\frac{n}{\sqrt{2}}) + \mathcal{O}(n^2) + = \mathcal{O}(\mathbf{n^2} \log \mathbf{n})$ (using master theorem)

The Minimum Cut Problem

Improving the runtime

- Ok, so our algorithm has **runtime $\mathcal{O}(\mathbf{n^2 \log n})$** (much better!)
- But what about the **probability of success**? Let P_n be the probability that a min-cut C survives in a graph with n nodes.

$$\begin{aligned} P_n &= Pr(C \text{ survives contractions}) \times Pr(C \text{ survives one of the recursive calls}) \\ &\geq \frac{1}{2} \cdot Pr(C \text{ survives 1 of the recursive calls}) \\ &= \frac{1}{2} \cdot (1 - Pr(C \text{ is killed in both the recursive calls})) \\ &= \frac{1}{2} \cdot (1 - (1 - P_{n/\sqrt{2}})^2) \end{aligned}$$

This recursion can be solved to show that $P_n = \Omega(1/\log n)$

(a proof can be seen in the auxiliary material available at the course website; note also we were a little bit lose on the limit of t which should more formally be $\lceil n/\sqrt{2} + 1 \rceil$ - this "constant" factor, however, does not affect our result)

- This means our **probability of success is at least $\mathbf{1/\log n}$**

Monte Carlo Algorithms

Improving the runtime

- So our improved algorithm has $\Omega(1/\log n)$ success probability [much better than the initial version, which was $\Omega(1/n^2)$]
- At the same time, the runtime is $\mathcal{O}(n^2 \log n)$ [not that much worse than the $\mathcal{O}(n^2)$ of the initial version]
- Using **amplification**, we can also improve this success probability to $1/n$ (it suffices to repeat it $M = \log^2 n$ times) (using the "most useful inequality ever")
- Because the runtime is $\mathcal{O}(Mn^2 \log n)$, **we have now a Monte Carlo algorithm with time $\mathcal{O}(\mathbf{cn^2 \log^3 n})$ and with probability of being incorrect at most $\frac{\mathbf{1}}{\mathbf{n^c}}$** (and that is more than enough for today...)

Monte Carlo vs Las Vegas

- Can we **transform a Las Vegas algorithm into a Monte Carlo one**?
- Can we **transform a Monte Carlo algorithm into a Las Vegas one**?
- That will be one of the subjects of the exercises in **homework #2** :)