Probabilistic algorithms

AiC FME, UPC

Fall 2021

What is probability?

What is probability?

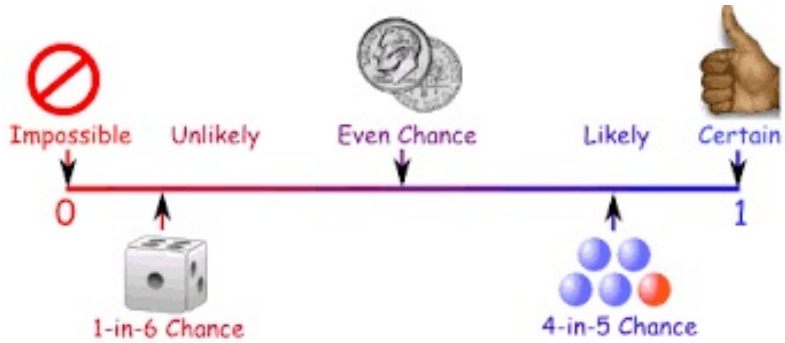
Probability: useful technique to simulate and explain real world.

What is probability?

Probability: useful technique to simulate and explain real world.
Any English speaking person understands the words likely and unlikely.

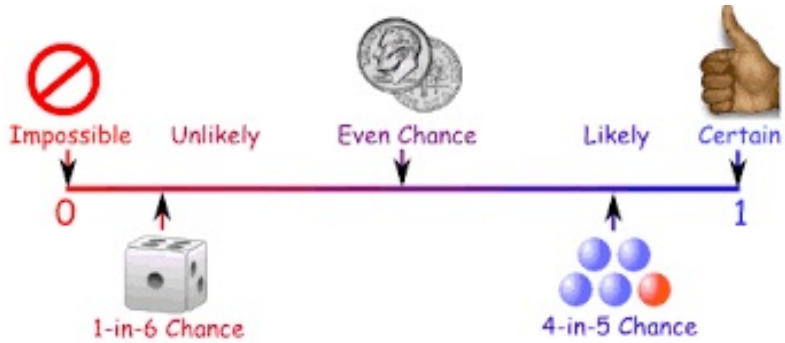
What is probability?

Probability: useful technique to simulate and explain real world.
Any English speaking person understands the words likely and unlikely.



What is probability?

Probability: useful technique to simulate and explain real world.
Any English speaking person understands the words likely and unlikely.



But in everyday life, do we consciously think in terms of probability?

What is probability?

As far as we know, many phenomena in *nature* seem to be generated by random choices, but it is difficult to simulate truly unpredictable random experiments:

What is probability?

As far as we know, many phenomena in *nature* seem to be generated by random choices, but it is difficult to simulate truly unpredictable random experiments:

Flipping a coin or tossing a dice are *deterministic* experiments; Given the initial angle of the coin, the spin, humidity, etc. we can predict the outcome of flipping a coin.

What is probability?

As far as we know, many phenomena in *nature* seem to be generated by random choices, but it is difficult to simulate truly unpredictable random experiments:

Flipping a coin or tossing a dice are *deterministic* experiments; Given the initial angle of the coin, the spin, humidity, etc. we can predict the outcome of flipping a coin.

In the same way, in todays computers, the *random generator* functions are *deterministic* programs, which simulates randomness. What is denoted as pseudorandom generators.

Probability and computers

The most basic method is the linear congruential generator: from a seed integer $x_0 \in \mathbb{Z}^+$, produce a sequence of pseudo-random values

$$x_{n+1} = (a\,x_0 + b) \mod m,$$

for a, b constants and m a large integer.

Probability and computers

The most basic method is the linear congruential generator: from a seed integer $x_0 \in \mathbb{Z}^+$, produce a sequence of pseudo-random values

$$x_{n+1} = (a\,x_0 + b) \mod m,$$

for a, b constants and m a large integer.

In C/C++ rand( ), m is a 32-bit integer, $a = 22695477, b = 1$

Probability and computers

The most basic method is the linear congruential generator: from a seed integer $x_0 \in \mathbb{Z}^+$, produce a sequence of pseudo-random values

$$x_{n+1} = (a\,x_0 + b) \mod m,$$

for a, b constants and m a large integer.

In C/C++ rand( ), m is a 32-bit integer, $a = 22695477, b = 1$

A computer deterministically generates pseudorandom numbers.

Probability and computers

The most basic method is the linear congruential generator: from a seed integer $x_0 \in \mathbb{Z}^+$, produce a sequence of pseudo-random values

$$x_{n+1} = (a\,x_0 + b) \mod m,$$

for a, b constants and m a large integer.

In C/C++ rand( ), m is a 32-bit integer, $a = 22695477, b = 1$

A computer deterministically generates pseudorandom numbers.

How would you generate a vector with a sequence of pseudorandom bits?

Probability and computers

The most basic method is the linear congruential generator: from a seed integer $x_0 \in \mathbb{Z}^+$, produce a sequence of pseudo-random values

$$x_{n+1} = (a\,x_0 + b) \mod m,$$

for a, b constants and m a large integer.

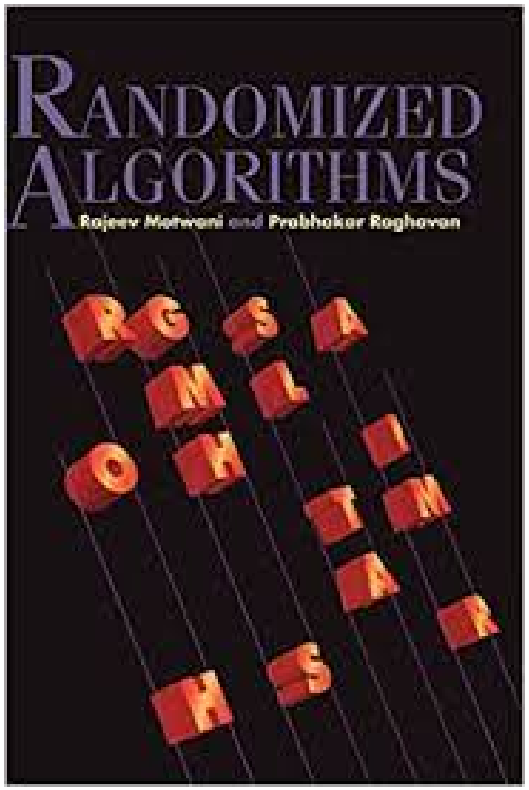
In C/C++ rand( ), m is a 32-bit integer, $a = 22695477, b = 1$

A computer deterministically generates pseudorandom numbers.

How would you generate a vector with a sequence of pseudorandom bits?

```
for (i = 0; i < n; i++) do
    values[i]=rand() % 2;
    printf("%d", values[i]);
end for
```

More references



Randomized algorithms

We can design a randomized algorithms, where the algorithm takes random choices and continues the computation according to the output of the random choices.

Randomized algorithms

We can design a randomized algorithms, where the algorithm takes random choices and continues the computation according to the output of the random choices.
In this case, we may have to perform a probabilistic analysis of the complexity.

Randomized algorithms

We can design a randomized algorithms, where the algorithm takes random choices and continues the computation according to the output of the random choices.
In this case, we may have to perform a probabilistic analysis of the complexity.
There are two main types of probabilistic algorithms:

Randomized algorithms

We can design a randomized algorithms, where the algorithm takes random choices and continues the computation according to the output of the random choices.
In this case, we may have to perform a probabilistic analysis of the complexity.
There are two main types of probabilistic algorithms:

- Monte-Carlo: Always halt in finite time, but may output the wrong answer. If the answer is binary (yes/not) the error can be in one direction, *one-side error*, or the error could be in both answers *two-side error*. In Monte-Carlo algorithms it is important to bound the error probability.

Randomized algorithms

We can design a randomized algorithms, where the algorithm takes random choices and continues the computation according to the output of the random choices.
In this case, we may have to perform a probabilistic analysis of the complexity.
There are two main types of probabilistic algorithms:

- Monte-Carlo: Always halt in finite time, but may output the wrong answer. If the answer is binary (yes/not) the error can be in one direction, *one-side error*, or the error could be in both answers *two-side error*. In Monte-Carlo algorithms it is important to bound the error probability.
- Las Vegas: The output is always correct but the running time may be unbounded.

Randomized algorithms

We can design a randomized algorithms, where the algorithm takes random choices and continues the computation according to the output of the random choices.
In this case, we may have to perform a probabilistic analysis of the complexity.
There are two main types of probabilistic algorithms:

- Monte-Carlo: Always halt in finite time, but may output the wrong answer. If the answer is binary (yes/not) the error can be in one direction, *one-side error*, or the error could be in both answers *two-side error*. In Monte-Carlo algorithms it is important to bound the error probability.
- Las Vegas: The output is always correct but the running time may be unbounded.

It is easy to convert a Las Vegas algorithm into a Monte-Carlo, how?. The contrary is not always true.

Randomized algorithms

We can design a randomized algorithms, where the algorithm takes random choices and continues the computation according to the output of the random choices.
In this case, we may have to perform a probabilistic analysis of the complexity.
There are two main types of probabilistic algorithms:

- Monte-Carlo: Always halt in finite time, but may output the wrong answer. If the answer is binary (yes/not) the error can be in one direction, *one-side error*, or the error could be in both answers *two-side error*. In Monte-Carlo algorithms it is important to bound the error probability.
- Las Vegas: The output is always correct but the running time may be unbounded.

It is easy to convert a Las Vegas algorithm into a Monte-Carlo, how?. The contrary is not always true.

In this course we will be working mainly with Monte-Carlo algorithms.

A randomized sorting algorithm

What do you know about QuickSort?

A randomized sorting algorithm

What do you know about QuickSort?

- General deterministic sorting algorithm

A randomized sorting algorithm

What do you know about QuickSort?

- General deterministic sorting algorithm
- Runs in time

A randomized sorting algorithm

What do you know about QuickSort?

- General deterministic sorting algorithm
- Runs in time $O(n^2)$

A randomized sorting algorithm

What do you know about QuickSort?

- General deterministic sorting algorithm
- Runs in time $O(n^2)$
- Average time

A randomized sorting algorithm

What do you know about QuickSort?

- General deterministic sorting algorithm
- Runs in time $O(n^2)$
- Average time $O(n \log n)$

A randomized sorting algorithm

What do you know about QuickSort?

- General deterministic sorting algorithm
- Runs in time $O(n^2)$
- Average time $O(n \log n)$ when the input follows the uniform distribution.

We want to keep the input deterministic and devise a randomized algorithm that sorts in expected $O(n \log n)$ time.

A randomized sorting algorithm

A randomized sorting algorithm

Input a vector $A[n]$
Compute a uniform random permutation of $[n]$ in B
Rearrange A according to B
Run Quicksort on A

A randomized sorting algorithm

Input a vector $A[n]$
Compute a uniform random permutation of $[n]$ in B
Rearrange A according to B
Run Quicksort on A

The algorithm reaches our goal, if we can compute a random permutation within the right time.

Generating a permutation uniformly at random

A permutation Π over $[n]$ defines a re-ordering of the elements, formally a bijective function $\pi : [n] \to n$.

The number of different permutations is $n!$.

Considering the experiment of generating a uniformly random permutation, we get the probability space $\Omega = \{\pi_1, \pi_2, \ldots, \pi_{n!}\}$, i.e. $|\Omega| = n!$.

Generating a permutation uniformly at random (u.a.r) means, for each n, generate a particular permutation π with probability

$$\frac{1}{|\Omega|} = \frac{1}{n!}.$$

Randomized algorithm to generate u.a.r. a permutation

Fisher-Yates Algorithm (also known as Knuth's algorithm)

Random-Perm (n)
for $i = 0$ to $n-1$ **do**
  $\pi[i] = i$
end for
for $i = n-1$ to 1 **do**
  choose $j = \text{Rand}(i+1)$
  Interchange $\pi[j]$ and $\pi[i]$
end for

Rand(i) provides a random number in $[0, i)$.

Fisher-Yates algorithm

- The algorithm considers the items in the array one at a time from the end and swaps each element with an element in the array from that point to the beginning. This has cost $O(n)$
- Notice that each element has an equal probability, of $1/n$, of being chosen as the last element in the array (including the element that starts out in that position).
- Applying this analysis recursively, we see that the output permutation has probability

$$\frac{1}{n}\frac{1}{n-1}\cdots\frac{1}{2} = \frac{1}{n!}$$

- That is, each permutation is equally likely.

Fisher-Yates algorithm

- The algorithm considers the items in the array one at a time from the end and swaps each element with an element in the array from that point to the beginning. This has cost $O(n)$
- Notice that each element has an equal probability, of $1/n$, of being chosen as the last element in the array (including the element that starts out in that position).
- Applying this analysis recursively, we see that the output permutation has probability

$$\frac{1}{n}\frac{1}{n-1}\cdots\frac{1}{2} = \frac{1}{n!}$$

- That is, each permutation is equally likely.

Lemma Random-Perm (n) produces a u.a.r. permutation of $[n]$ in $\Theta(n)$ steps.

Principle of deferred decisions

Not to assume that the entire set of random choices is made in advance. Rather, at each step of the process concentrate only on the random choices that are relevant to the algorithm outcome

When applicable it provides a simplified probability space to perform the probabilistic analysis.

Analyzing the Clock Solitaire game

From MR 3.5

The Clock Solitaire game: randomly shuffle a standard pack of 52 cards. Then, split the cards into 13 piles of 4 cards each; label piles as A, 2, . . . , 10, J, Q, K; take the first card from the "K" pile; take the next card from the pile "X", where X is the value of the previous card taken; repeat until:

- either all cards removed ("win")
- or you get stuck ("lose")

We want to evaluate the probability of "win".

Game termination?

The last card we take before the game ends (either winning or loosing) is a "K".

Let us assume that at iteration j we draw card X but the pile X is empty (thus the game terminates).

Let $X \neq K$ (i.e. we lose). Because pile X is empty and $X \neq K$, we must have already drawn (prior to draw j) 4 X cards. But then, we can not draw an X card at the j-th iteration, a contradiction.

There is no contradiction if the last card is a "K" and all other cards have been already removed (in that case the game terminates with win).

Game win?

We win if the fourth "K" card is drawn at the 52 iteration.

Whenever we draw for the 1st, 2nd or 3rd time a "K" card, the game does not terminate because the K pile is not empty so we can continue.

When the fourth K is drawn at the 52nd iteration then all cards are removed and the game's result is "win"

The probability of win?

According to the previous observations

$$\begin{aligned} Pr\{win\} &= Pr\{\text{4th ``K'' at the 52nd iteration}\} \\ &= \frac{\#\text{game evolutions: 52nd card} = \text{4th ``K''}}{\#\text{all game evolutions}} \end{aligned}$$

Considering all possible game evolutions is a rather naive approach since we have to count all ways to partition the 52 cards into 13 distinct piles, with an ordering on the 4 cards in each pile. This complicates the probability evaluation because of the dependence introduced by each random draw of a card.

We define another probability space that better captures the random dynamics of the game evolution.

The principle of deferred decisions

Basic idea: rather than fix (and enumerate) the entire set of potential random choices in advance, instead let the random choices unfold with the progress of the random experiment.

In this particular game at each draw any card not drawn yet is equally likely to be drawn.

A winning game corresponds to a dynamics where the first 51 random draws include 3 "K" cards exactly.

This is equivalent to draw the 4th "K" at the 52nd iteration.
So we forget how the first 51 draws came out and focus on the 52nd draw, which must be a "K".

The probability of win

We actually have 13x4=52 distinct positions (13 piles, 4 positions each) where 52 distinct cards are placed. This gives a total of 52! different placements.

Each game evolution actually corresponds to an ordered permutation of the 52 cards.

The winning permutations are those where the 52nd card is a "K" (4 ways) and the 51 preceding cards are arbitrarily chosen (51!). Thus:

$$Pr\{win\} = \frac{4\,51!}{52!} = \frac{4}{52} = \frac{1}{13}.$$

The probability of win

We actually have 13x4=52 distinct positions (13 piles, 4 positions each) where 52 distinct cards are placed. This gives a total of 52! different placements.

Each game evolution actually corresponds to an ordered permutation of the 52 cards.

The winning permutations are those where the 52nd card is a "K" (4 ways) and the 51 preceding cards are arbitrarily chosen (51!). Thus:

$$Pr\{win\} = \frac{4\,51!}{52!} = \frac{4}{52} = \frac{1}{13}.$$

A simpler way to get the same: The probability is $\frac{1}{13}$ because of symmetry (e.g. the type of the 52nd card is random uniform among all 13 types).

The idea was to defer, i.e. first consider the last choice and then conditionally the previous ones!

Checking matrix multiplication

Problem: Given 3 square matrices ($n \times n$), A, B and C, we want to see if $A \times B = C$.
Easy solution: compute $A \times B$ and compare with C.

$n \times n$ matrix multiplication:

1. Naive algorithm: $O(n^3)$
2. Strassen (1969): $O(n^{2.81})$
3. Coppersmith-Winograd (1987): $O(n^{2.376})$
4. Vassilevska (2015): $O(n^{2.373})$

Can we (randomly) check in $O(n^2)$ if $A \times B = C$?

Freivald's algorithm for checking if $A \times B = C$ (1977)

From MU 1.3, MR 3.5

Given $n \times n$ matrices A, B, C

Freivald(A, B, C)
choose u.a.r. $r \in \{0,1\}^n$
if $A(Br) = Cr$ **then**
  output true
else
  output false
end if

Choosing u.a.r. r can be done choosing independently with probability $1/2$ each of its n bits. This makes the probability of any given r $1/2^n$, and the cost of generate the vector $O(n)$.

Freivald's algorithm for checking if $A \times B = C$ (1977)

From MU 1.3, MR 3.5

Given $n \times n$ matrices A, B, C

Freivald(A, B, C)
choose u.a.r. $r \in \{0,1\}^n$
if $A(Br) = Cr$ **then**
  output true
else
  output false
end if

The time complexity of Freivald's is $\Theta(n^2)$.

Notice: if $AB = C$ the algorithm yields always the correct answer.

It could be that $AB \neq C$ and the algorithms may yield the wrong answer

Error probability

Theorem

If $AB \neq C$ *then* $\mathbf{Pr}\left[A(B(r)) = Cr\right] \leq \frac{1}{2}$.

Proof.

- Neat trick: As $AB \neq C$ taking $D = AB - C$, then $D \neq (0)$.
  $\Rightarrow \exists d_{ij} \in D$ s.t. $d_{ij} \neq 0$. W.l.o.g. assume $d_{11} \neq 0$.
  If $\exists r$ s.t. $A(Br)) = Cr$ then $Dr = 0$.
  $Dr = 0 \Rightarrow \sum_{j=1}^{n} d_{1j} r_j = 0$, but as $d_{11} \neq 0$ then $r_1 = \frac{-\sum_{j=2}^{n} d_{1j} r_j}{d_{11}}$.
- Second trick: Choose $r = (r_1, \ldots, r_n)$ from r_n to r_1 and stop at r_2, just before choosing r_1, which could be only 0 or 1.
  Then the equality $r_1 = \frac{-\sum_{j=2}^{n} d_{1j} r_j}{d_{11}}$ holds with prob. $= 1/2$

□

Notice that by considering $r_n, \ldots, r_2$ to be fixed, we reduce the sample space to $r_1 \in \{0, 1\}$

Randomized algorithms and amplification

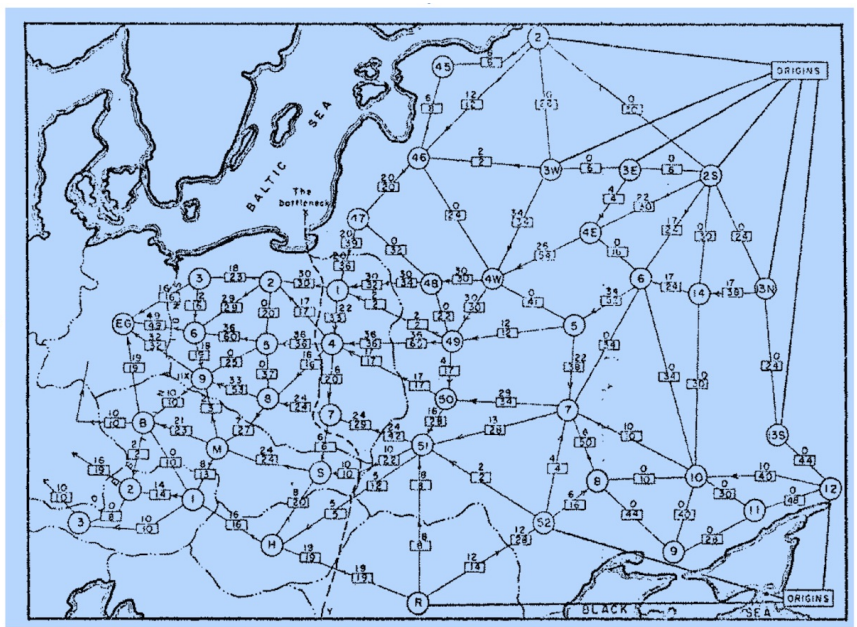
Notice Freivald's algorithm finish always in finite time ($\Theta(n^2)$) but may output the wrong answer. That type of randomized algorithms are called Monte-Carlo algorithms.
Moreover Freivald's also is a one side error, if $AB = C$ we always get the correct answer, but if $AB \neq C$ we may get the wrong answer with a "small" probability.

One-side Monte-Carlo algorithms have the nice characteristic that can be amplified: Each run of the algorithm can be considered as an independent "experiment", so they can be repeated, at each run we generate a new random choice, and by independence, each run decreases the probability of error.

If we repeat k times Freivald's algo. and each time we generate a new v, the answer keep being true, the probability of error is $\leq 1/2^k$.

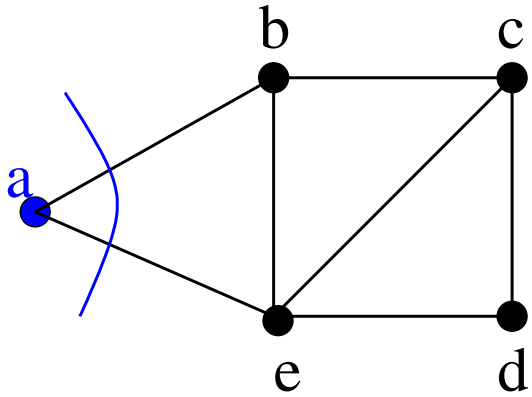
The Minimum Cut problem

In the mid 50's Harris and Ross studied the railway links between cities in the URSS and easter Europe and determined the easiest way to break the network by removing edges. The minimum cut of the graph.



The Minimum Cut problem

Given an undirected graph $G = (V, E)$ a cut is a partition of V in S and $\bar{S}$. The capacity of the cut is the number of edges with an end in S and the other in $\bar{S}$. The min cut is the cut with minimum capacity.



Complexity for deterministic algorithms

- Using Ford-Fulkerson: Max Flow-Min-Cut $O(n^2 m)$ or $O(nm)$ using J.Orlin's algorithms from 2013.
- Stoer-Wagner's algorithm (1994) $O(nm + n^2 \lg n)$ (non-flow, weighted graphs)

Monte-Carlo algorithm for the Min-Cut problem

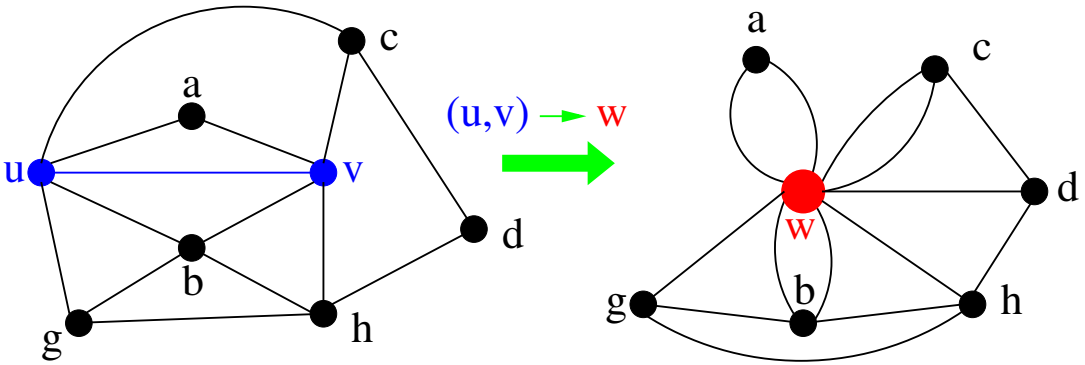
D. Karger, 1993.

Contracting an edge in G

Given a connected undirected graph $G = (V, E)$, we want to contract edges this operation will produce a graph with multiple edges but without self-loops:

Contract $(e = (u, v))$

- replace u and v by a super-node w,
- preserve edges, update endpoints of u and v to w,
- avoid self-loops but keep parallel edges.



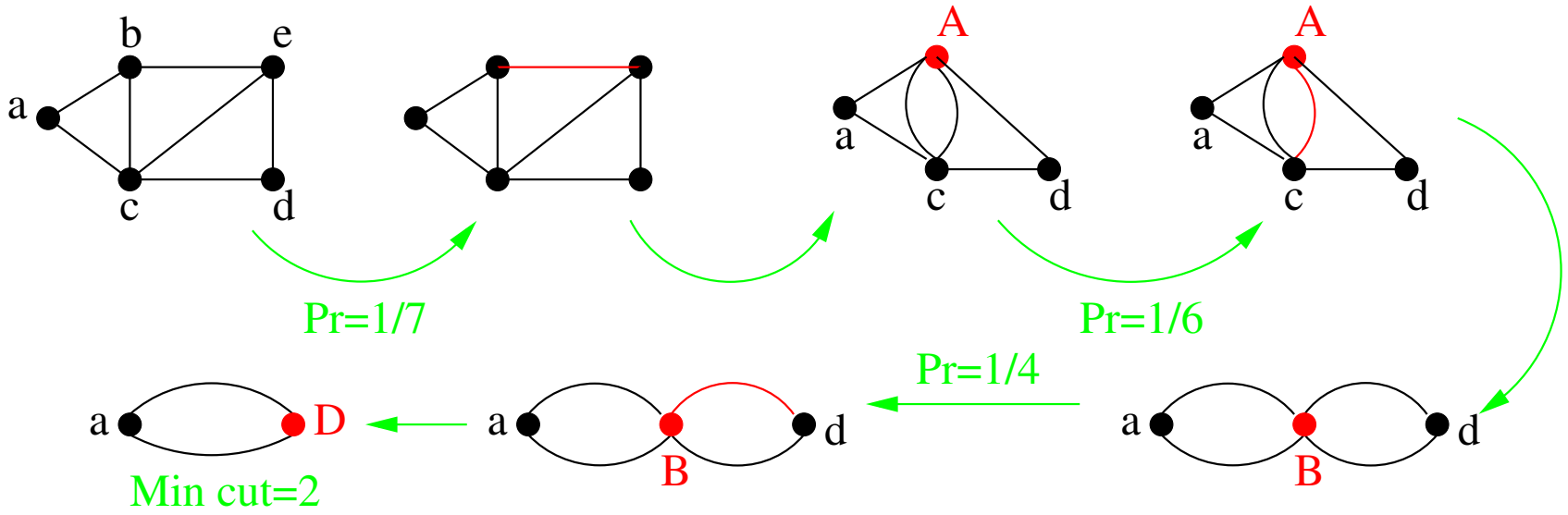
Given G, which DS would you use to implement **Contract**(e)?

Karger's algorithm

Karger $(G = (V, E))$
while $|V| > 2$ **do**
  Chose u.a.r. $e_i = (u, v) \in E$
  $G =$**Contract**(e_i)
end while
return the edges between the 2 remaining vertices

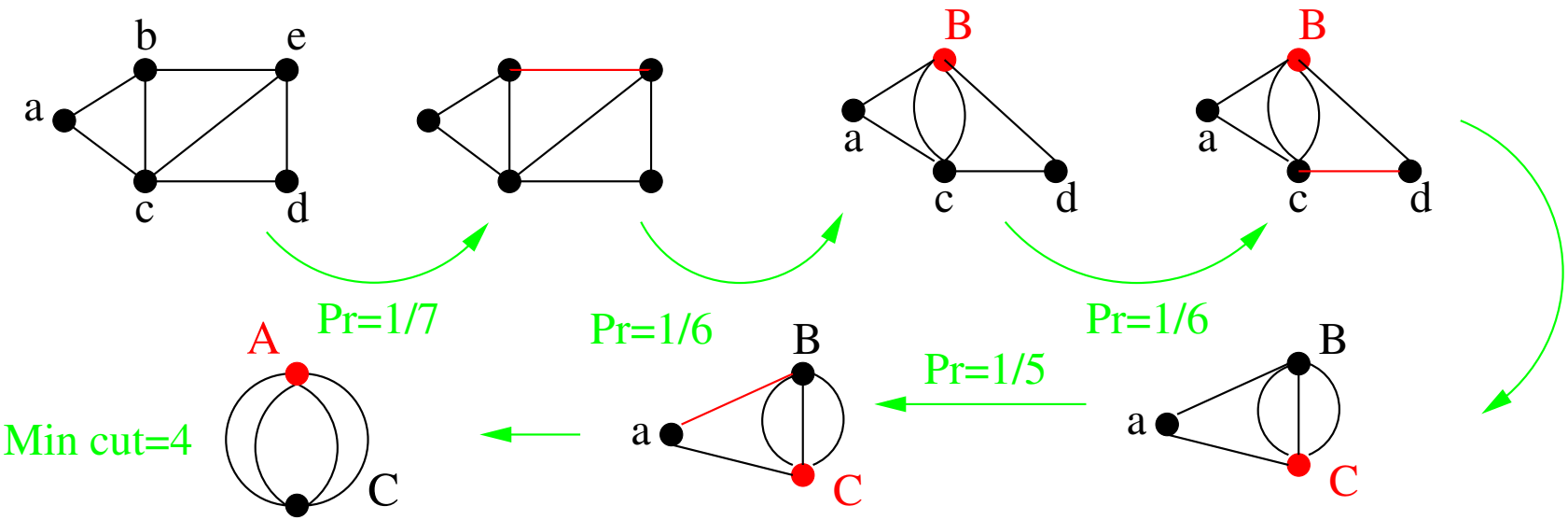
Karger's algorithm

Karger $(G = (V, E))$
while $|V| > 2$ **do**
  Chose u.a.r. $e_i = (u, v) \in E$
  $G =$**Contract**(e_i)
end while
return the edges between the 2 remaining vertices



Karger's algorithm

Karger $(G = (V, E))$
while $|V| > 2$ **do**
  Chose u.a.r. $e_i = (u, v) \in E$
  $G =$**Contract**(e_i)
end while
return the edges between the 2 remaining vertices



Analysis of the algorithm

The running time of the algorithm is $\Theta(n^2)$.

Assume G, with $|V| = n$ has a min-cut set $C \subseteq E$ of size k.

Notice:

- Any cut in a contracted graph is a cut in the initial graph,
- Karger's returns a cut,
- A contraction eliminates all the set of edges among the identified vertices.
- Karger's might provide a cut that is not of minimum size.

Theorem

Karger's algorithm returns a min-cut with probability $\geq 2/n^2$.

Proof of the Theorem

- Let C be a min-cut of G, assume that $|C| = k$
- Let G_i be the graph after i contractions, G_i has $n - i$ nodes.
- If no $e \in C$ has been contracted then C is still a min-cut of G_i.
- Let $\mathcal{E}_i$ be the event none of the edge(s) contracted at the i-iteration is in C and let $\mathcal{O}_i = \cap_{j=1}^{i} \mathcal{E}_i$ i.e. no edge in C is contracted in the first i iterations.

Proof of the Theorem

- We want to compute $\mathbf{Pr}[\mathcal{O}_{n-2}]$ probability of success.
- Notice $\mathbf{Pr}[\mathcal{E}_1] = \mathbf{Pr}[\mathcal{O}_1] \geq 1 - \frac{2k}{nk}$
  As $|C| = k$ all vertices in G must have degree $\geq k$, $|E(G)| \geq nk/2$.
  So 1st contracted edge chosen u.a.r. among the $\geq nk/2$ edges and $|C| = k$)
- The $\mathbf{Pr}[\mathcal{E}_2|\mathcal{O}_1] \geq 1 - \frac{k}{k(n-1)/2} \geq 1 - 2/(n-1)$
  If 1st. contraction did not eliminate an edge in C (i.e conditioning on $\mathcal{O}_1$), we are left with $|V(G_1)| = n - 1$ and $|E(G_1)| \geq k(n-1)/2$, again $\deg(v) \geq k$
- Working iteratively, $\mathbf{Pr}[\mathcal{E}_i|\mathcal{O}_{i-1}] \geq 1 - \frac{2}{(n-i+1)}$.

Proof of the Theorem

From $\mathbf{Pr}[A \cap B] = \mathbf{Pr}[A|B]\,\mathbf{Pr}[B]$:

$$
\begin{aligned}
\mathbf{Pr}[\mathcal{O}_{n-2}] &= \mathbf{Pr}[\mathcal{E}_{n-2} \cap \mathcal{O}_{n-3}]) \\
&= \mathbf{Pr}[\mathcal{E}_{n-2} \mid \mathcal{O}_{n-3}]\,\mathbf{Pr}[\mathcal{O}_{n-3}] \\
&= \mathbf{Pr}[\mathcal{E}_{n-2}|\mathcal{O}_{n-3}]\,\mathbf{Pr}[\mathcal{E}_{n-3}|\mathcal{O}_{n-4}] \ldots \mathbf{Pr}[\mathcal{E}_2|\mathcal{O}_1]\,\mathbf{Pr}[\mathcal{O}_1] \\
&\geq \prod_{i=1}^{n-2}\left(1 - \frac{2}{n-i+1}\right) = \prod_{i=1}^{n-2}\left(\frac{n-i-1}{n-i+1}\right) \\
&= (\frac{n-2}{n})(\frac{n-3}{n-1})(\frac{n-4}{n-2})\cdots(\frac{3}{5})(\frac{2}{4})(\frac{1}{3}) \\
&= \frac{2}{n(n-1)}
\end{aligned}
$$

□

Amplification

To increase the probability of success, run Karger's algorithm several times.

Theorem

If we run Karger's min-cut algorithm $n(n-1)\lg n$ times and output the smallest cut found in all the runs the probability of failure (it is not the global min-cut) is $\leq$

$$\left(1 - \frac{2}{n(n-1)}\right)^{n(n-1)\lg n} \leq e^{-2\lg n} = \frac{1}{n^2}.$$

The proof is straightforward using the definition of e^{-1}

A randomized algorithm for MAX 3-SAT

A 3-SAT formula is a Boolean formula in CNF such that each clause has exactly 3 literals and each literal corresponds to a different variable.

$$(x_2 \vee \overline{x}_3 \vee \overline{x}_4) \wedge (x_2 \vee x_3 \vee \overline{x}_4) \wedge (\overline{x}_1 \vee x_2 \vee x_4) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee x_3) \wedge (x_1 \vee \overline{x}_2 \vee \overline{x}_4)$$

MAXIMUM 3-SAT. Given a 3-SAT formula, find a truth assignment that satisfies as many clauses as possible.

The problem is NP-hard. We can try to design a randomized algorithm that produces a good assignment, even if it is not optimal.

A randomized algorithm for MAX 3-SAT

Algorithm

For each variable, flip a fair coin, and assign to the variable True (1) if it is heads, False (0), otherwise.

Note that a variable gets 1 with probability $\frac{1}{2}$, and this assignment is made independently of the other variables.

What is the expected number of satisfied clauses?

Assume that the 3-SAT formula has n variables and m clauses.

- Let $Z =$ number of clauses satisfied by the random assignment
- For $1 \leq j \leq m$, define the random variables
  $Z_j = 1$ if clause j is satisfied, 0 otherwise.
- By definition, $Z = \sum_{j=1}^{m} Z_j$.
- $Pr[Z_j = 1] = 1 - (1/2)^3 = 7/8$, so $E[Z_j] = 7/8$. Therefore ,

$$E[Z] = \sum_{j=1}^{m} E[Z_j] = \frac{7}{8}m$$

A randomized algorithm for MAX 3-SAT

How good is the solution computed by the random algorithm?

- For a 3-CNF formula let $opt(F)$ be the maximum number of clauses than can be satisfied by an assignment.
- As for any assignment x the number of satisfied clauses is always $\leq opt(F)$, we have that $E[Z] \leq opt(F)$.
- Of course $opt(F) \leq m$, that is $\frac{7}{8} opt(F) \leq \frac{7}{8} m = E[Z]$, then

$$\frac{opt(F)}{E[Z]} \leq \frac{8}{7}$$

- We get $\frac{7}{8} opt(F) \leq E[Z] \leq opt$, a randomized approximation algorithm.

The probabilistic method

Claim

For any instance of 3-SAT, there exists a truth assignment that satisfies at least a 7/8 fraction of all clauses.

Proof. Random variable must have one event on which the measured value is at least its expectation.

Probabilistic method [Paul Erdös].
Prove the existence of a non-obvious property by showing that a random construction produces it with positive probability