



## Table of contents

- 1 Introduction
  - Classification of randomized algorithms
- 2 Las Vegas Algorithms
  - Game tree evaluation
  - RandomizedQuickSort
  - A randomized median algorithm
- 3 Monte Carlo algorithms
  - Verifying matrix multiplication
  - A randomized min-cut algorithm
- 4 Probabilistic Method
  - Satisfiability Problem
- 5 Bibliography

## Introduction

- There are roughly two types of algorithms: *deterministic* and *non-deterministic* ones.
- This classification gave birth to the well known complexity classes: P and NP.
- A simplified definition of an algorithm: a process which transforms an input into an output.
- A *deterministic algorithm* is one that for the same input will give the same result exhibiting the same behavior on any run.
- A *non-deterministic algorithm* is one that can give different results for the same input exhibiting different behaviors (e.g. values for variables are different).
- Essentially the difference in a non-deterministic process is caused by *choices* that it can make, the interaction between running threads etc.

## Introduction

- A deterministic algorithm has to solve a problem correctly (always) and as quickly as possible (usually the number of steps should be polynomial in the size of the input).
- Examples of non-deterministic algorithms: concurrent algorithms and *randomized algorithms*.
- A randomized algorithm in addition to input takes a source of random numbers and makes random choices during execution. Usually the designer tries to show that its behavior is likely to be "good" on every input.

### Definition 1.1

A **randomized algorithm** is an algorithm which during the execution makes some probabilistic choices.

- The above probabilistic choices consist in generating values from a random variable. Afterwards these values are involved in computations the algorithm performs.

# Introduction

- Randomized algorithms are among the most known non-deterministic algorithms and randomization became a standard approach in algorithm design often because of the simplicity and speed.
- These kind of advances of theory of probability in modern computer science are witnessed in the last few decades.
- In areas like communication, cryptography, and discrete optimization randomness and probabilistic methods has become common investigating tools:
  - The protocol of an Ethernet card uses randomness to choose when it next tries to acces the communication medium.
  - Testing primality in criptography uses random numbers.
  - Sometimes NP-hard problems can be solved on most of the inputs using randomized algorithms.

## More detailed applications

- *Data structures*: sorting, order statistics, searching.
- *Algebraic identities*: polynomial and matrix identity verification.
- *(Algorithmic) Graph Theory*: minimum spanning trees, shortest paths, minimum cuts.
- Counting and enumeration: matrix permanent, counting combinatorial structures.
- Parallel and distributed computing: deadlock avoidance, distributed consensus.
- *Probabilistic existence proofs*: prove that a certain combinatorial object arises with non-zero probability among objects drawn from a particular probability space.

## Classification of randomized algorithms

- The study of the random variables associated with a randomized algorithm is used for analyzing the efficiency and the error probability of algorithm.
- The fundamental classification of randomized algorithms: algorithms that (always) correctly solves the problem and algorithms that err.
- **Monte Carlo** algorithms have a positive error probability.
- **Las Vegas** algorithms never err (their error probability is zero).

## Las Vegas Algorithms

- Las Vegas algorithms are randomized algorithms that guarantee that every computed output is correct.
- The literature established a some-how different meaning of the syntagma "correct output" in that it allows the algorithm to answer with "I don't know" or "?".

## Definition 2.1

Let  $A$  be a randomized algorithm that allows the answer "?".  $A$  is called a **Las Vegas algorithm** for computing a function  $F$  if, for any input  $x$  (any argument of  $F$ ),

- (i)  $P(A(x) = F(x)) \geq 1/2$ .
- (ii)  $P(A(x) = "?") = 1 - P(A(x) = F(x)) \leq 1/2$ .

## Las Vegas Algorithms

- For a Las Vegas algorithms we usually investigate the expected complexity: the expected time or expected space complexity.
- This happens because the runs of the algorithm on a given input have different lengths (in CPU time and used memory space).
- We wish to devise Las Vegas algorithms with expected running time that is bounded (e.g., by a polynomial in the size of the input).
- In the above definition the constant  $1/2$  can be changed with any other constant  $\epsilon \in (0, 1)$  as the following result shows.

## Las Vegas Algorithms

## Proposition 2.1

Let  $\epsilon \in (0, 1)$  and  $A$  be a randomized algorithm that computes a function  $F$  such that

$$P(A(x) = F(x)) \geq \epsilon \text{ and}$$

$$P(A(x) = "?") = 1 - P(A(x) = F(x))$$

Let, for  $k \in \mathbb{N}^*$ ,  $A_k$  be the randomized algorithm that for any input  $x$  executes  $k$  independent runs of  $A$  on  $x$ . Estimate the smallest  $k$  such that  $P(A_k(x) = F(x)) \geq 1/2$ .

## Game tree evaluation

- A *game tree* is a rooted tree in which internal nodes at even distances from the root are labeled MIN (thus, the root has a MIN label), and the remaining internal vertices are labeled MAX. Associated with each leaf is a real number, which is its value.
- The evaluation of the tree is the following process: each leaf returns the value associated with it, each MAX node returns the largest value returned by its children, and each MIN node returns the smallest value returned by its children.
- Given a game tree, the *evaluation problem* is to determine the value returned by the root.
- This kind of evaluation plays an important role in AI (in game-playing problems).

## Game tree evaluation

- We limit our discussion to the special case in which the values at the leaves are bits (thus, each MIN node can be thought as a logical AND operation and each MAX node as logical OR operation).
- We consider a complete binary tree having depth  $2h$ , with  $N = 4^h$  leaves.
- One can easily see that every deterministic algorithm can be forced to read all  $N$  leaves: in an AND node the first evaluated child can have value 1, and in an OR node the value 0 - in both cases we are forced to evaluate the second child.
- The randomized algorithm will evaluate recursively at random a child of the current node. If the current value does not determine the value of the current node, evaluate the other child.

## Game tree evaluation

```

RandomEval(x) {
  if(x.operation == AND){
    choose uniformly at random a child u;
    if(RandomEval(u) == 1) {
      let v the other child;
      return RandomEval(v);
    }
    return 0;
  }
  :
}

```

## Game tree evaluation

Probabilități și Statistică

Probabilități și Statistică

Probabilități și Statistică

## Theorem 2.1

*For a game tree with depth  $2h$ , the expected cost of evaluation is at most  $3^h$ .*

Probabilități și Statistică

Probabilități și Statistică

Probabilități și Statistică

**Proof.** First consider an OR node that returns 1 with two leaves as children (the worst case for a deterministic algorithm would be a 0 value leaf and an 1 value leaf); with probability  $1/2$  the algorithm chooses first the leaf with 0 value, therefore the expected number of steps is  $1/2 \cdot 1 + 1/2 \cdot 2 = 3/2$ ; in a similar manner for an AND node that returns 0 with two leaves as children, the expected number of steps is  $3/2$ .

Obviously, the number of steps is 2 for an AND node evaluating at 1 and an OR node evaluating at 0 - there is no saving. But the benefit of the randomized algorithm is that at an internal AND node in a tree returning 1 both OR children must return 1 which is the "good" case for OR.

## Game tree evaluation

We proceed by induction. We evaluate first the expected number of steps<sup>1</sup> for  $h = 1$ , that is, for a root labeled with AND with two OR children each having two leaves as children.

If the root returns 0, then the average number of evaluations is at most  $1/2 \cdot 2 + 1/2 \cdot 3/2 = 7/4$  (1 for one of its children and 2 for both leaves of it).

If the root returns 1, then the number of evaluations is  $3/2 + 3/2 = 3$  (both children are in the "good" case for OR nodes).

Now consider an OR node those AND children are roots of subtrees with  $2(h - 1)$  depth each. If the root were to evaluate to 1, at least one of its children returns 1. With probability  $1/2$  this child is chosen first, and, with probability  $1/2$  both subtrees are evaluated. The expected cost is at most

$$\frac{1}{2} \cdot 3^{h-1} + \frac{1}{2} \cdot 2 \cdot 3^{h-1} = \frac{3}{2} 3^{h-1}.$$

<sup>1</sup>A step is a leaf evaluation.

## Game tree evaluation

If the OR root were to evaluate to 0, both children must be evaluated, incurring a cost of at most  $2 \cdot 3^{h-1}$ .

Consider now an AND node as the root of a  $2h$  depth tree. If the root were to evaluate to 0, then at least one of its OR children returns 0; with probability  $1/2$  this child is chosen first, and the expected cost is at most

$$2 \cdot 3^{h-1} + \frac{1}{2} \cdot \frac{3}{2} \cdot 3^{h-1} \leq 3^h.$$

If the AND root were to evaluate to 1, both children must be evaluated, incurring a cost of at most

$$2 \cdot \frac{3}{2} \cdot 3^{h-1} = 3^h. \blacksquare$$

- If we denote by  $N$  the number of leaves of such a tree, then the expected number of steps (node evaluations) is at most  $N^{\log_4 3} = N^c$  ( $c < 0.8$ ).

## QuickSort and RandomizedQuickSort

- We consider the problem of sorting the elements of a given set  $S$  (having a linear order on all elements) by comparisons of pairs of elements only.
- If we could find a member  $x$  of  $S$  such that half of the members of  $S$  are smaller than  $x$ , then we could use the following procedure:
  - partition  $S \setminus \{x\}$  into two subsets  $S_1$  and  $S_2$ , where  $S_1$  consists of those elements that are smaller than  $x$ , and  $S_2$  the rest of the elements.
  - recursively sort  $S_1$  and  $S_2$ , then return the elements of  $S_1$  in ascending order, followed by  $x$ , and then the elements of  $S_2$  in ascending order.

## QuickSort and RandomizedQuickSort

- If we could find  $x$  in  $cn$  steps (for some constant  $c$ ), then we could partition  $S \setminus \{x\}$  in  $(n - 1)$  steps by comparing each element with  $x$ .
- Denote by  $T(n)$  the number of steps for such a procedure in the worst-case scenario with an input of length  $n = |S|$ .
- $T(n)$  would be given by the following recurrence
 
$$T(n) \leq 2T(n/2) + (c + 1)n.$$
- The solution of this recurrence is  $T(n) \leq \tilde{c}n \log n$  (for some constant  $\tilde{c}$ ) which give a running time of  $\mathcal{O}(n \log n)$ .
- The time complexity remains the same even for an  $x$  which ensures only a roughly even split.

## RandomizedQuickSort

- The question is how to quickly find such an  $x$ ?
- One simple answer is to choose an element of  $S$  at random; the result is RandomizedQuickSort - an example of Las Vegas randomized algorithm.

RandQuickSort( $S$ ) {

  choose  $x \in S$  *independently and uniformly* at random;

//  $x$  is called the pivot.

$S_1 \leftarrow \{y \in S : y < x\}$ ;

$S_2 \leftarrow \{y \in S : y > x\}$ ;

  return [RandQuickSort( $S_1$ ),  $x$ , RandQuickSort( $S_2$ )]; }

- We are interested in counting the number of comparisons - this is the dominant cost of the algorithm.

## RandomizedQuickSort

- Suppose that true linear order of elements in  $S$  is  $x_1 < x_2 < \dots < x_n$ .
- In an worst-case scenario the time complexity of the algorithm is  $\mathcal{O}(n^2)$  (for example if the input is  $x_1 = n, x_2 = n - 1, \dots, x_n = 1$ , and the pivot is always in the first position).
- As we already pointed out we are interested in computing the expected number of comparisons.
- The total number of comparisons,  $X$ , is obviously a random variable. If we denote by  $X_{ij}$  the following Bernoulli variable

$$X_{ij} = \begin{cases} 1, & x_i \text{ and } x_j \text{ are compared during the run} \\ 0, & \text{otherwise} \end{cases}$$

$$X \text{ can be described as } X = \sum_{i=1}^n \sum_{j>i} X_{ij}.$$

## RandomizedQuickSort

Probabilități și Statistică

Probabilități și Statistică

Probabilități și Statistică

## Theorem 2.2

For any input the expected number of comparisons made by *RandQuickSort* is  $2n \ln n + O(n)$ .

**Proof.** If  $p_{ij}$  is the probability that  $x_i$  and  $x_j$  are compared during the execution of the algorithm, then

$$\mathbb{E}[X] = \sum_{i=1}^n \sum_{j>i} \mathbb{E}[X_{ij}] = \sum_{i=1}^n \sum_{j>i} p_{ij},$$

$x_i$  and  $x_j$  are compared if and only if  $x_i$  or  $x_j$  is the first pivot chosen from the subset  $S^{ij} = \{x_i, x_{i+1}, \dots, x_j\}$ .

The pivot being chosen independently and uniformly at random from each subset, the pivot it is likely to be any element of this subset, that is,  $p_{ij} = 2/(j - i + 1)$ .

## RandomizedQuickSort

$$\begin{aligned}
 \mathbb{E}[X] &= \sum_{i=1}^n \sum_{j>i} \frac{2}{j-i+1} = 2 \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{1}{j-i+1} \\
 &= 2 \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{1}{k} = 2 \sum_{k=1}^n \sum_{i=1}^{n-k+1} \frac{1}{k} = 2 \sum_{k=1}^n \frac{n-k+1}{k} \\
 &= (2n+2) \sum_{k=1}^n \frac{1}{k} - 2n = 2n \ln n + \mathcal{O}(n). \blacksquare
 \end{aligned}$$

## A randomized median algorithm

Probabilități și Statistică

Probabilități și Statistică

Probabilități și Statistică

### Definition 2.2

*Given a set  $S$  of  $n$  totally ordered elements, the **median of  $S$**  is an element  $m$  of  $S$  such that at least  $\lfloor n/2 \rfloor$  elements in  $S$  are less or equal to  $m$  and at least  $\lfloor n/2 \rfloor + 1$  elements in  $S$  are greater or equal to  $m$ .*

Probabilități și Statistică

Probabilități și Statistică

Probabilități și Statistică

- The median can be easily found deterministically in  $\mathcal{O}(n \log n)$  steps by sorting and choosing the appropriate element of  $S$  in ascending order; there is also a rather complex linear time complexity deterministic algorithm for computing median ([Blum73]).
- We present a simpler randomized linear time algorithm whose basic idea is to find two elements that are close together in the sorted order of  $S$  and that have the median lie between them.
- This algorithm is also of Las Vegas type.

## A randomized median algorithm

```

RandMedian( $S$ ) {
  let  $R$  be the multi-set of  $\lceil \sqrt[4]{n^3} \rceil$  chosen elements of  $S$  independently
  and uniformly at random with replacement;
  sort  $R$ ; // use an optimal deterministic algorithm in  $\mathcal{O}(|R| \log |R|)$  steps
  let  $d \leftarrow$  the  $\lfloor \sqrt[4]{n^3}/2 - \sqrt{n} \rfloor$ th smallest element in the sorted order
  of  $R$ ;
  let  $u \leftarrow$  the  $\lceil \sqrt[4]{n^3}/2 + \sqrt{n} \rceil$ th smallest element in the sorted order
  of  $R$ ;
   $C \leftarrow \{x \in S : d \leq x \leq u\}$ ;
   $l_d = |\{x \in S : x < d\}|$ ;  $l_u = |\{x \in S : x > u\}|$ ;
  if ( $l_d > n/2$  or  $l_u > n/2$  or  $|C| > 4 \lceil \sqrt[4]{n^3} \rceil$ )
    return "no median found";
  sort  $C$ ;
  return the  $(\lfloor n/2 \rfloor - l_d + 1)$ th smallest element in the sorted order
  of  $C$ ; }

```

## A randomized median algorithm

- The choice of the size of  $R$  and the choices for  $d$  and  $u$  are tailored to guarantee both that
  - $C$  include the median of  $S$  with high probability and
  - $C$  is sufficiently small that it can be sorted in sublinear time with high probability.

### Theorem 2.3

*The randomized median algorithm has a linear time complexity and gives a correct output.*

**Proof.** The algorithm could give an incorrect answer if and only if the median doesn't belong to set  $C$  when  $l_d \leq n/2$ ,  $l_u \leq n/2$ , and  $|C| \leq 4\lceil \sqrt[4]{n^3} \rceil$ ; but this means that  $l_d > n/2$  or  $l_u > n/2$  or  $|C| > 4\lceil \sqrt[4]{n^3} \rceil$ . In what concerns the time complexity, the sorting of  $C$  could be done in  $\mathcal{O}(\sqrt[4]{n^3} \log \sqrt[4]{n^3}) = \mathcal{O}(n)$ . ■

## A randomized median algorithm

- The following result, given here without a proof (you can consult [Mitzenmacher65]), concludes the analysis of the randomized median algorithm.

### Theorem 2.4

*The probability that the randomized median algorithm finds no median is at most  $1/\sqrt[4]{n}$ .*

## (One sided bounded-error) Monte Carlo algorithms

- Monte Carlo algorithms are randomized algorithms that can't guarantee that every computed output is correct (sometime they cannot give a solution to the problem).

## Definition 3.1

A randomized algorithm  $A$  is called a **Monte Carlo algorithm** for computing a function  $F$  if, for any input  $x$  (any argument of  $F$ ),

$$P(A(x) = F(x)) \geq 1/2.$$

- Many such algorithms appear in optimization problems. Very relevant for a Monte Carlo algorithm is that the error probability is bounded from above.

## Amplification with Monte Carlo algorithms

- Like in the Las Vegas algorithm definition the constant  $1/2$  from the former definition can be replaced with any other constant  $\epsilon \in (0, 1)$ .
- The *amplification* is the operation of independently repeating of the algorithm until the error probability decreases as desired. It is like we try to transform a Monte Carlo algorithm in a Las Vegas one.
- Suppose that the error probability of an algorithm  $A$  is at most  $\epsilon$  and we independently repeat the algorithm  $k$  times; the probability of getting only erroneous answers is at most  $\epsilon^k$ ,
- Since  $\lim_{k \rightarrow \infty} \epsilon^k = 0$ , for large enough  $k$ , one can make this probability as small as desired.
- **Warning:** large values of  $k$  may dramatically increase the overall time complexity.

## Verifying matrix multiplication

- Suppose we are given three  $n \times n$  matrices  $A$ ,  $B$ , and  $C$ ; assume also that we are working over integers modulo 2.
- We want to verify whether  $A \cdot B = C$ . This is a decision problem (it accepts only a "yes" or a "no" as answer).
- One way is to multiply  $A$  and  $B$  and compare the result with  $C$ ; the matrix multiplication takes  $\mathcal{O}(n^3)$  or using more sophisticated algorithms  $\mathcal{O}(n^{2.37})$ .
- We devise a randomized algorithm that allows faster verification at the expense of possibly returning a wrong answer with small probability.

## Verifying matrix multiplication

```

RandomVerify( $A, B, C$ ) {
  generate  $r_1, r_2, \dots, r_n \in \{0, 1\}$  independently and uniformly at random;
   $r \leftarrow (r_1, r_2, \dots, r_n)$ ;
  if( $ABr = Cr$ )
    return "yes";
  return "no";
}

```

- When the algorithm returns "no" we are sure that  $A \cdot B \neq C$ . The only error the algorithm can make is to return "yes" when the true answer is "no".
- We will bound this error probability using the following result.

### Theorem 3.1

For  $r \in \{0, 1\}^n$ , chosen like in the algorithm,  $P(ABr = Cr) \leq \frac{1}{2}$ , provided that  $A \cdot B \neq C$ . ( $P(ABr = Cr | A \cdot B \neq C) \leq \frac{1}{2}$ .)

## Verifying matrix multiplication

**Proof.** Since  $D = AB - C \neq 0$  - we can suppose that  $d_{1n} \neq 0$ ; if  $Dr = 0$ , then

$$\sum_{i=1}^n d_{1i} r_i = 0 \Rightarrow r_n = -\frac{\sum_{i=1}^{n-1} d_{1i} r_i}{d_{1n}}.$$

After  $r_1, \dots, r_{n-1}$  were chosen there are two possible values for  $r_n$ , therefore the probability that the above equality holds is at most  $1/2$ .

$$\begin{aligned} & P(ABr = Cr) = \\ &= \sum_{(x_1, \dots, x_{n-1}) \in \{0,1\}^{n-1}} P[(ABr = Cr) \cap \{r_i = x_i : i = 1, n-1\}] \\ &\leq \sum_{(x_1, \dots, x_{n-1}) \in \{0,1\}^{n-1}} P \left[ \left( r_n = -\frac{\sum_{i=1}^{n-1} d_{1i} r_i}{d_{1n}} \right) \cap \{r_i = x_i : i = 1, n-1\} \right] \end{aligned}$$

## Verifying matrix multiplication

$$\begin{aligned}
 &= \sum_{(x_1, \dots, x_{n-1}) \in \{0,1\}^{n-1}} P \left[ \left( r_n = \frac{\sum_{i=1}^{n-1} d_{1i} r_i}{d_{1n}} \right) \right] P(\cap \{r_i = x_i : i = \overline{1, n-1}\}) \\
 &\leq \sum_{(x_1, \dots, x_{n-1}) \in \{0,1\}^{n-1}} \frac{1}{2} P(\cap \{r_i = x_i : i = \overline{1, n-1}\}) = \frac{1}{2}. \blacksquare
 \end{aligned}$$

- The algorithm consists in generating  $n$  uniform random numbers and three matrix  $\times$  vector multiplications ( $\mathcal{O}(n^2)$ ). Hence, the time complexity of the algorithm is  $\mathcal{O}(n^2)$ .
- If we independently run the algorithm  $k$  times we will get an error probability smaller than  $2^{-k}$ .

## Karger's algorithm

- We consider a multi-graph  $G = (V, E, w)$ , where  $w : E \rightarrow \mathbb{N}^*$  determines its edges multiplicities (that is, we have a graph with positive integer weights on its edges).
- A *cut* in  $G$  is generated by a bipartition of  $V$ ,  $(A, B)$ , and its cost is  $w(A, B) = \sum_{e \in E(A, B)} w(e)$ , where  $E(A, B) = \{uv \in E : u \in A, v \in B\}$  are the edges of the cut.
- The *min-cut problem* asks to find a cut of minimum cost in  $G$ . For solving this problem there exists an  $\mathcal{O}(nm + n^2 \log n)$  time complexity deterministic algorithm<sup>2</sup> (in contrast to the *max-cut problem* which is NP-hard).
- We study in this section a randomized min-cut algorithm which is based on the contraction of edges in a graph.

<sup>2</sup>The Stoer-Wagner algorithm.

## Karger's algorithm

- The basic operation,  $\text{contraction}(u, v)$ , replaces in the current graph vertices  $u$  and  $v$  by a new vertex  $z$  and assigns the set of edges incident with  $z$  by the union of the edges incident on  $u$  and  $v$ .
- During the contraction edges from  $u$  and  $v$  with the same end-point are retained as multiple edges.

```

RandMinCut( $G, w$ ) {
  while( $|V(G)| > 2$ ) {
    choose  $e = uv \in E(G)$  independently and uniformly at random;
     $(G, w) \leftarrow \text{contraction}(u, v)$ ;
  }
  return  $E(G)$ ; }

```

## Karger's algorithm

- After each step the number of vertices decreases by one and any cut in the new graph is a cut in the original graph.
- After each iteration some of the cuts in the original graph may vanish, but the algorithm ends with a set of edges that represents the edges of a cut in the original graph.
- An useful strategy is to terminate the algorithm with  $k \geq 2$  vertices instead of 2, and, after that, to apply a deterministic algorithm to find a minimum cut in the remaining graph.

```

RandMinCut( $G, w, k$ ) {
  while( $|V(G)| > k$ ) {
    choose  $e = uv \in E(G)$  independently and uniformly at random;
     $(G, w) \leftarrow \text{contraction}(u, v)$ ;
  }
  return  $E(G)$ ; }

```

## Karger's algorithm

Probabilități și Statistică

Probabilități și Statistică

Probabilități și Statistică

## Theorem 3.2

*The randomized min-cut algorithm  $\text{RandMinCut}(G, w, k)$  has a polynomial time complexity and computes a min-cut with a probability at least  $\frac{k(k-1)}{n(n-1)}$ .*

**Proof.** Each contraction can be executed in  $\mathcal{O}(n)$  steps (**why?**); hence, the complexity of  $\text{RandMinCut}(G, w, k)$  is  $\mathcal{O}((n-k)n)$ .

If in a  $n$ -vertex (multi-)graph we have a min-cut of cost  $h$ , then the minimum degree of a vertex is at least  $h$  (**why?**), therefore the number edges is at least  $hn/2$ ; the probability that an edge belonging to a min-cut is contracted will be at most  $\frac{h}{hn/2} = \frac{2}{n}$ .

Let us consider a given min-cut  $C$ . We estimate the probability that  $C$  is not returned by the algorithm.

## Karger's algorithm

If  $C$  is returned, then it means that none of the edges of  $C$  has ever been contracted.

Let  $A_i$  be the event that an edge of  $C$  is contracted in the  $i$ th step and  $B_i$  be the event that no edge of  $C$  is contracted in the first  $(i - 1)$  iterations. The number of vertices after  $i$  iterations is  $(n - i)$ , therefore

$$P(\overline{A}_1) \geq 1 - \frac{2}{n}, P(\overline{A}_i | B_{i-1}) \geq 1 - \frac{2}{n - i + 1}.$$

On the other hand,

$$P(B_i) = P(\overline{A}_i \cap B_{i-1}) = P(\overline{A}_i | B_{i-1})P(B_{i-1}), \forall i \leq k.$$

$$P(B_k) = P(\overline{A}_1) \cdot P(\overline{A}_2 | B_1) \cdot \dots \cdot P(\overline{A}_k | B_{k-1}) \geq$$

$$\geq \prod_{i=1}^{n-k} \left(1 - \frac{2}{n - i + 1}\right) \geq \frac{k(k-1)}{n(n-1)}. \blacksquare$$

## Karger's algorithm

- For  $k = 2$  the algorithm  $\text{RandMinCut}(G, c)$  guarantees an error probability of at most  $[1 - 2/n(n-1)] \geq (1 - 2/n^2)$ ;
- We can reduce the error probability by repeating the algorithm: if we execute  $n^2/2$  times the algorithm (increasing the overall time complexity to  $\mathcal{O}(n^4)$ ) the error probability is at most

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

- If we execute  $n(n-1) \ln n$  times the algorithm (increasing the overall time complexity to  $\mathcal{O}(n^4 \log n)$ ) the error probability is at most

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

- The great advantage of this algorithm is its simplicity.

## Probabilistic Method

- The *probabilistic method* is a technique used for proving the existence of some mathematical (combinatorial) objects having certain properties.
- The background idea is to prove that a certain object has a positive probability to exist, hence it must exist.
- In order to use this technique we define a probability space over the set of involved objects and compute, or just estimate, some probabilities.
- This method often uses a randomized algorithm whose analysis gives as a byproduct the proof of existence for some combinatorial objects.

## Probabilistic Method

### Theorem 4.1

(**Expectation principle**) *If  $X$  is a discrete random variable with  $M[X] \geq \alpha$ , then  $P\{X \geq \alpha\} > 0$ .*

First applications of this principle come from logic, namely the well known satisfiability problem.

### Proposition 4.1

*Let  $\mathcal{F} = \{C_1, C_2, \dots, C_m\}$  be a family of  $m$  clauses. Then there exists an assignment of truth values over the boolean variables such that the number of true clauses is at least*

$$\sum_{i=1}^m (1 - 2^{-|C_i|}) \geq m (1 - 2^{-h}),$$

where  $h = \min_{1 \leq i \leq m} |C_i|$ .

## Satisfiability Problem

proof: Let's imagine the following abstract random experiment: to each boolean variable  $x$  assign value 1 (*true*) or 0 (*false*) independently and with probability 0.5. Define the (Bernoulli) random variables

$$X_i = \begin{cases} 1, & \text{if } C_i \text{ is true} \\ 0, & \text{if } C_i \text{ is false} \end{cases}$$

The probability that  $C_i$  is true equals the probability that at least one of its  $|C_i|$  literals is true, or  $(1 - 2^{-|C_i|})$ . The number of true clauses is

$$X = \sum_{i=1}^n X_i, \text{ therefore}$$

$$\mathbb{E}[X] = \sum_{i=1}^m \mathbb{E}[X_i] = \sum_{i=1}^m (1 - 2^{-|C_i|}) \geq m (1 - 2^{-h}).$$

The conclusion follows from Theorem 4.1. ■

## Satisfiability Problem

### Corollary 4.1

*Any instance of a  $k$ -SAT problem having less than  $2^k$  clauses is satisfiable. (An instance of  $k$ -SAT is a SAT instance having in each clause exactly  $k$  literals.)*

proof: We reuse the above argument, with  $|C_i| = k, \forall i = \overline{1, m}$ :

$$\mathbb{E}[X] = \sum_{i=1}^m \mathbb{E}[X_i] = \sum_{i=1}^m (1 - 2^{-|C_i|}) = m (1 - 2^{-k}) > m - 1,$$

we get that  $P\{X \geq m\} = P\{X > m - 1\} > 0$  - therefore it must exist a truth assignment which satisfies all the clauses. ■

## Satisfiability problem






- The above bound is best possible as, we can define an unsatisfiable instance of  $k$ -SAT with  $2^k$  clauses: the family of all clauses having each  $k$  literals over  $k$  boolean variables.
- In a similar manner we can prove

### Proposition 4.2

*A family of clauses  $\mathcal{F} = \{C_1, \dots, C_m\}$  is satisfiable if*

$$\sum_{i=1}^m 2^{-|C_i|} < 1.$$

## Bibliography I

-  Alon, N., J. H. Spencer, *The probabilistic method*, Wiley, 2008.
-  Bertsekas, D. P., J. N. Tsitsiklis, *Introduction to Probability*, Athena Scietific, 2002.
-  Blum, M., R. W. Floyd, V. Pratt, R. L. Rivest, R. E. Tarjan, *Time bounds for selection*, J. of Comp. and Sys. Sci. 7, pp. 448-461, 1973.
-  Karger, D., *Global min-cuts in RNC and other ramifications of a simple min-cut algorithm*, ACM-SIAM Symp. on Discr. Alg. 4, pp 21-30, 1993.
-  Mitzenmacher, M., E. Upfal, *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*, Cambridge University Press, 1995.

## Bibliography II



Motwani, R., P. Raghavan, *Randomized Algorithms*, Cambridge University Press, 2005.



Spencer, J. H., *Ten lectures on the probabilistic method*, SIAM, 1994.