



# Cuprins

- 1 **Introducere**
  - Clasificarea algoritmilor aleatori
- 2 **Algoritmi Las Vegas**
  - Evaluarea unui game-tree
  - RandomizedQuickSort
  - Un algoritm aleator pentru determinarea medianei
- 3 **Algoritmi Monte Carlo**
  - Verificarea înmulțirii matricilor
  - Un algoritm aleator pentru min-cut
- 4 **Metoda probabilistică**
  - Aplicații la problema satisfiabilității
- 5 **Bibliography**

## Introducere

- Algoritmii pot fi clasificați drept algoritmi *determiniști* și *nedeterminiști*.
- Această clasificare a dat naștere și la binecunoscutele clase de complexitate: P și NP.
- O definiție simplificată a unui algoritm este următoarea: o intrare (input) care este procesată și transformată într-o ieșire (output).
- Un *algoritm determinist* este un algoritm pentru care rezultatul (ieșirea) este același, având același comportament în orice execuție pentru o aceeași intrare.
- Un *algoritm nedeterminist* este un algoritm care poate da rezultate diferite pentru același input, având eventual, comportamente diferite la execuții diferite (e.g. valorile variabilelor pot fi diferite în timpul execuției).
- Diferența esențială constă în aceea că un proces nedeterminist este influențat de *alegerile* care pot fi făcute sau de interacțiunile dintre firele de execuție etc.

## Introducere

- Un algoritm determinist trebuie să rezolve (întotdeauna) corect o problemă și cât de repede posibil (de obicei se cere ca numărul de pași/operații să fie polinomial în dimensiunea inputului).
- Exemple de algoritmi nedeterminiști: algoritmi concurenți și *algoritmi aleatori*.
- Un algoritm aleator primește la intrare și o sursă de numere aleatoare care îi permite să facă alegeri aleatoare în timpul execuției. Cel care îl proiectează încearcă să arate că se comportă "corect" pentru fiecare input.

### Definition 1.1

*Un algoritm aleator este un algoritm care în cursul execuției face anumite alegeri probabilistice.*

- Aceste alegeri probabilistice constau în generarea de valori ale unei variabile aleatoare. Aceste valori sunt implicate în calculele pe care algoritmul se presupune că le face.

## Introducere

- Algoritmii aleatori sunt printre cei mai cunoscuți algoritmi de tip nedeterminist, iar randomizarea a devenit o abordare standard în proiectarea algoritmilor datorită simplității și vitezei sporite.
- Acest tip de aplicații ale teoriei probabilităților în informatică a devenit frecvent în ultimele decade.
- În domenii precum comunicațiile, criptografia și optimizarea discretă randomizarea și metodele probabilistice au devenit instrumente uzuale de investigație:
  - Protocolul Ethernet folosește alegeri aleatoare când accesează mediul de comunicație.
  - Testarea primalității (în criptografie) folosește alegeri probabilistice.
  - Unele probleme NP-dificile pot fi rezolvate pentru majoritatea intrărilor de către algoritmi aleatori.

## Aplicații detaliate

- *Structuri de date*: sortare, statistici ordonate, căutare.
- *Identități algebrice*: verificarea identității polinoamelor și matricilor.
- *Teoria (algoritmică a) grafurilor*: arbori parțiali de cost minim, drumuri de cost minim, tăieturi de pondere minimă.
- Numărare și enumerare: permanentul unei matrici, numărarea structurilor combinatoriale.
- Calcul paralel și distribuit: evitarea blocajului (deadlock), consens distribuit.
- *Demonstrații probabilistice existențiale*: dovezi că un anumit obiect combinatorial există cu probabilitate nenulă printre obiectele unui spațiu probabilistic.

## Clasificarea algoritmilor aleatori

- Studiul variabilelor aleatoare asociate unui algoritm aleator este utilizat pentru a analiza eficiența și probabilitatea de a greși ale algoritmului.
- Clasificarea fundamentală a algoritmilor aleatori: algoritmi care rezolvă corect (întotdeauna) problema (asociată) și algoritmi care greșesc.
- Algoritmii **Monte Carlo** au o probabilitate pozitivă de a greși.
- Algoritmii **Las Vegas** nu greșesc niciodată (probabilitatea lor de a greși este nulă).

## Algoritmi Las Vegas

- Algoritmii Las Vegas sunt algoritmi aleatori care garantează că fiecare output este corect (este soluție a problemei pe care algoritmul încearcă să o rezolve).
- Literatura de specialitate a stabilit o definiție puțin diferită a sintagmei "output corect" permițând algoritmului să dea și răspunsul "Nu știu."

## Definition 2.1

Fie  $A$  un algoritm aleator căruia îi este permis răspunsul "?". A este numit **algoritm Las Vegas** pentru calculul funcției  $F$  dacă, pentru orice intrare  $x$  (orice argument al lui  $F$ ),

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

## Algoritmi Las Vegas

- Pentru algoritmi Las Vegas ne interesează de obicei complexitatea medie: timpul mediu de execuție și spațiul mediu de memorie folosit.
- Aceasta este posibil pentru că diferite execuții ale algoritmului (pentru o aceeași intrare) au diferite caracteristici (în termeni de timp CPU și spațiu de memorie).
- Căutăm să descriem algoritmi Las Vegas cu timp de execuție mediu mărginit (de obicei de un polinom în dimensiunea inputului).
- În definiția de mai sus constanta  $1/2$  poate fi schimbată cu orice altă constantă  $\epsilon \in (0, 1)$ , după cum o arată următorul rezultat.

## Algoritmi Las Vegas

Probabilități și Statistică  
 Probabilități și Statistică  
 Probabilități și Statistică  
 Probabilități și Statistică

Probabilități și Statistică  
 Probabilități și Statistică  
 Probabilități și Statistică  
 Probabilități și Statistică

Probabilități și Statistică  
 Probabilități și Statistică  
 Probabilități și Statistică  
 Probabilități și Statistică

## Proposition 2.1

*Fie  $\epsilon \in (0, 1)$  și  $A$  un algoritm aleator care calculează funcția  $F$  așa încât*

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

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

*Fie, pentru  $k \in \mathbb{N}^*$ ,  $A_k$ , un algoritm aleator care pentru orice intrare  $x$  execută în mod independent algoritmul  $A$  de  $k$  ori asupra lui  $x$ . Există un  $k$  astfel ca  $P(A_k(x) = F(x)) \geq 1/2$ .*

Probabilități și Statistică  
 Probabilități și Statistică  
 Probabilități și Statistică  
 Probabilități și Statistică

Probabilități și Statistică  
 Probabilități și Statistică  
 Probabilități și Statistică  
 Probabilități și Statistică

Probabilități și Statistică  
 Probabilități și Statistică  
 Probabilități și Statistică  
 Probabilități și Statistică

## Evaluarea unui game-tree

- Un *game tree* este un arbore cu rădăcină în care nodurile interne aflate la distanță pară față de rădăcină sunt etichetate cu MIN (rădăcină are o etichetă MIN), iar restul nodurilor interne sunt etichetate cu MAX. Fiecărei frunze îi este asociată o valoare - un număr real.
- Evaluarea arborelui se face astfel: fiecare frunză returnează valoarea asociată ei, fiecare nod MAX returnează cea mai mare valoare a unuia dintre fi, iar fiecare nod MIN returnează cea mai mică valoare a unuia dintre fi.
- Dat un game-tree, *problem evaluării* constă în a determina valoarea returnată de rădăcină.
- Acest tip de evaluare are un rol important în IA (în probleme game-playing).

## Evaluarea unui game-tree

- Vom limita analiza noastră la cazul special când în frunze valorile sunt biți (astfel un nod MIN este o operație logică AND, iar un nod MAX are o operație logică OR).
- Considerăm un arbore binar complet cu adâncimea  $2h$  și  $N = 4^h$  frunze.
- Este ușor de văzut că un algoritm determinist va fi forțat să citească toate frunzele: într-un nod AND primul copil evaluat poate returna valoarea 1, iar într-un nod OR valoarea 0 - în amândouă cazurile algoritmul este forțat să evalueze ambii descendenți.
- Algoritmul aleator evaluează recursiv un descendent ales la întâmplare al nodului curent. Dacă valoarea rezultată nu determină valoarea nodului se evaluează și celălalt descendent.

## Evaluarea unui game-tree

```

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;
  }
  :
}

```

## Evaluarea unui game-tree

Probabilități și Statistică

Probabilități și Statistică

Probabilități și Statistică

### Theorem 2.1

*Pentru un game-tree cu adâncimea  $2h$ , costul mediu al evaluării este cel mult  $3^h$ .*

**Proof.** Considerăm mai întâi un nod OR care returnează 1 cu două frunze drept copii (cazul cel mai nefavorabil pentru un algoritm determinist constă într-o frunză 0 și o frunză 1); dacă fii au valori 0 și 1, atunci cu probabilitate  $1/2$  algoritmul alege mai întâi frunza cu valoare 0, numărul mediu de pași este  $1/2 \cdot 1 + 1/2 \cdot 2 = 3/2$ ; în mod similar pentru un nod AND care returnează 0 cu două frunze drept copii, numărul mediu de pași este cel mult  $3/2$ .

Evident numărul de pași este 2 pentru un nod AND evaluat la 1 și pentru un nod OR evaluat la 0 - nu există vreo economie. Câștigul este că, spre exemplu, într-un nod intern AND care returnează 1 amândoi copii OR trebuie să returneze 1 ceea ce este cazul "bun" pentru OR.

## Evaluarea unui game-tree

Folosim metoda inducției. Evaluăm mai întâi numărul de pași<sup>1</sup>; în cazul  $h = 1$ , i. e., un arbore cu rădăcina AND care are doi copii OR, fiecare având câte două frunze drept copii.

Dacă rădăcina returnează 0, atunci numărul mediu de evaluări în cazul cel mai nefavorabil este  $1/2 \cdot 2 + 1/2 \cdot 3/2 = 7/4$ .

Dacă rădăcina returnează 1, numărul de evaluări este  $3/2 + 3/2 = 3$  (amândoi copiii sunt în cazul "bun" pentru noduri OR).

Considerăm acum un nod OR ai cărui copii AND sunt rădăcinile câte unui arbore cu adâncimea  $2(h - 1)$ . Dacă rădăcina OR returnează 1, atunci cel puțin unul dintre copii returnează 1. Cu probabilitate  $1/2$  acest copil este ales mai întâi și, cu probabilitate  $1/2$ , amândoi subarborii sunt evaluați. Costul mediu este cel mult

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

<sup>1</sup>Un pas este o evaluare a unei frunze.

## Evaluarea unui game-tree

Dacă rădăcina OR returnează 0, amândoi copiii vor trebui evaluați ceea ce impune un cost de cel puțin  $2 \cdot 3^{h-1}$ .

Considerăm acum un nod AND rădăcină a unui arbore cu adâncimea  $2h$ . Dacă rădăcina returnează 0, atunci cel puțin unul dintre copiii OR ai rădăcinii returnează 0; cu probabilitate  $1/2$  acest copil este ales mai întâi și costul mediu este cel mult

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

Dacă rădăcina returnează 1, amândoi copiii trebuie evaluați impunând un cost de cel mult

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

- Dacă  $N$  este numărul de frunze al unui astfel de arbore, atunci numărul mediu de pași (i. e., numărul mediu de noduri evaluate) este cel mult  $N^{\log_4 3} = N^c$  ( $c < 0.8$ ).

## QuickSort și RandomizedQuickSort

- Considerăm problema sortării elementelor unei mulțimi  $S$  (pe care există o ordine totală) doar prin compararea perechilor de elemente.
- Dacă am putea găsi un element  $x$  al lui  $S$  așa încât jumătate dintre elementele lui  $S$  să fie mai mici ca  $x$ , atunci am putea utiliza următoarea procedură:
  - partiționează  $S \setminus \{x\}$  în două submulțimi  $S_1$  și  $S_2$ , unde  $S_1$  constă din acele elemente care sunt mai mici decât  $x$ , iar  $S_2$  din restul elementelor.
  - sortează recursiv  $S_1$  și  $S_2$ , apoi returnează elementele lui  $S_1$  în ordine crescătoare, urmate de  $x$  și apoi de elementele lui  $S_2$  în ordine crescătoare.

## QuickSort și RandomizedQuickSort

- Dacă am putea determina  $x$  în  $cn$  pași (pentru o constantă  $c$ ), atunci am putea partiționa  $S \setminus \{x\}$  în  $(n - 1)$  pași comparând fiecare element cu  $x$ .
- Notăm cu  $T(n)$  numărul de pași pentru a asemenea procedură (în cazul cel mai nefavorabil) pentru  $n = |S|$ .

- $T(n)$  este dat de următoarea recurență

$$T(n) \leq 2T(n/2) + (c + 1)n.$$

- Soluția acestei recurențe este  $T(n) \leq \tilde{c}n \log n$  (pentru o constantă  $\tilde{c}$ ) ceea ce dă un timp de execuție de  $\mathcal{O}(n \log n)$ .
- Complexitatea timp rămâne aceeași chiar dacă  $x$  nu împarte exact (ci doar aproximativ) în două mulțimea  $S$ .

## RandomizedQuickSort

- Întrebarea este cât de repede poate fi găsit un astfel de  $x$ ?
- Un răspuns simplu este să alegem la întâmplare un astfel de  $x$  din  $S$ ; ceea ce rezultă este algoritmul RandomizedQuickSort - un exemplu de algoritm aleator de tip Las Vegas.

```

RandQuickSort( $S$ ) {
  choose  $x \in S$  independently and uniformly at random;
  //  $x$  se numește pivot.
   $S_1 \leftarrow \{y \in S : y < x\}$ ;
   $S_2 \leftarrow \{y \in S : y > x\}$ ;
  return [RandQuickSort( $S_1$ ),  $x$ , RandQuickSort( $S_2$ )]; }

```

- Suntem interesați să numărăm comparațiile - acesta este costul dominant al algoritmului.

## RandomizedQuickSort

- Să presupunem că ordinea totală a elementelor lui  $S$  este  $x_1 < x_2 < \dots < x_n$ .
- În cazul cel mai nefavorabil complexitatea timp a algoritmului este  $\mathcal{O}(n^2)$  (când, de exemplu pentru intrarea  $x_1 = n, x_2 = n-1, \dots, x_n = 1$ , când pivotul este întotdeauna în prima poziție).
- Suntem interesați în calcul numărului mediu de comparații.
- Numărul total de comparații,  $X$ , este evident o variabilă aleatoare. Notăm cu  $X_{ij}$  următoarea variabilă Bernoulli

$$X_{ij} = \begin{cases} 1, & x_i \text{ and } x_j \text{ sunt comparate în timpul execuției} \\ 0, & \text{altfel} \end{cases}$$

$$X \text{ poate fi scrisă ca } X = \sum_{i=1}^n \sum_{j>i} X_{ij}.$$

## RandomizedQuickSort

Probabilități și Statistică

Probabilități și Statistică

Probabilități și Statistică

## Theorem 2.2

Pentru orice intrare numărul mediu de comparații făcute de *RandQuickSort* este  $2n \ln n + \mathcal{O}(n)$ .

**Proof.** Dacă  $p_{ij}$  este probabilitatea ca  $x_i$  și  $x_j$  să fie comparate în timpul execuției algoritmului,

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

$x_i$  și  $x_j$  sunt comparate dacă și numai dacă  $x_i$  sau  $x_j$  este primul pivot din mulțimea  $S^{ij} = \{x_i, x_{i+1}, \dots, x_j\}$ .

Pivotul fiind ales independent și uniform aleator din fiecare submulțime,  $p_{ij} = 2/(j - i + 1)$ , pentru că orice element din submulțime are aceeași șansă să fie ales.

## RandomizedQuickSort

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

## Un algoritm aleator pentru determinarea medianeii

Probabilități și Statistică

Probabilități și Statistică

Probabilități și Statistică

### Definition 2.2

*Dată o mulțime total ordonată cu  $n$  elemente,  $S$ , **mediana lui  $S$**  este un element  $m \in S$  astfel încât cel puțin  $\lfloor n/2 \rfloor$  elemente din  $S$  sunt mai mici sau egale cu  $m$  și cel puțin  $\lfloor n/2 \rfloor + 1$  elemente din  $S$  sunt mai mari sau egale cu  $m$ .*

Probabilități și Statistică

Probabilități și Statistică

Probabilități și Statistică

- Mediana poate fi găsită, determinist, în  $\mathcal{O}(n \log n)$  pași prin sortare și alegerea elementului adecvat din mulțimea sortată; există, de asemenea un algoritm determinist mai complex, liniar, pentru determinarea medianeii ([Blum73]).
- Vom prezenta un algoritm simplu, aleator, de complexitate liniară a cărui idee de bază este de a găsi două elemente apropiate în ordonarea crescătoare a lui  $S$  și care conțin mediana între ele.
- Acest algoritm este tot de tip Las Vegas.

Probabilități și Statistică

Probabilități și Statistică

Probabilități și Statistică

Probabilități și Statistică

Probabilități și Statistică

Probabilități și Statistică

Probabilități și Statistică

Probabilități și Statistică

Probabilități și Statistică

Probabilități și Statistică

Probabilități și Statistică

Probabilități și Statistică

Probabilități și Statistică

Probabilități și Statistică

Probabilități și Statistică

Probabilități și Statistică

Probabilități și Statistică

Probabilități și Statistică

Probabilități și Statistică

Probabilități și Statistică

Probabilități și Statistică

## Un algoritm aleator pentru determinarea medianeii

```

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  $R$ ; // se utilizează un algoritm optim determinist în  $\mathcal{O}(|R| \log |R|)$  pași
  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  $C$ ; }

```

## Un algoritm aleator pentru determinarea medianei

- Alegerea cardinalului mulțimii  $R$  și a lui  $d$  și  $u$  sunt făcute pentru a garanta că
  - (i)  $C$  include median lui  $S$  cu probabilitate mare și că
  - (ii)  $C$  este suficient de mică încât să poată fi sortată în timp subliniar cu probabilitate mare.

### Theorem 2.3

*Algoritmul aleator pentru determinarea medianei are o complexitate liniară și oferă un răspuns corect.*

**Proof.** Algoritmul ar oferi un răspuns negativ dacă și numai dacă mediana nu aparține mulțimii  $C$  când  $l_d \leq n/2$ ,  $l_u \leq n/2$  și  $|C| \leq 4 \lceil \sqrt[4]{n^3} \rceil$ ; aceasta înseamnă că  $l_d > n/2$  sau  $l_u > n/2$  sau  $|C| > 4 \lceil \sqrt[4]{n^3} \rceil$ . În ceea ce privește complexitatea timp, sortarea elementelor lui  $C$  se poate face în  $O(\sqrt[4]{n^3} \log \sqrt[4]{n^3}) = O(n)$ . ■

## Un algoritm aleator pentru determinarea medianei

- Următorul rezultat, dat aici fără demonstrație (puteți consulta [Mitzenmacher] încheie analiza algoritmului.

### Theorem 2.4

*Probabilitatea ca algoritmul aleator pentru determinarea medianei să nu găsească mediana este cel mult  $1/\sqrt[4]{n}$ .*

## Algoritmi Monte Carlo (cu eroare unilateral mărginită)

- Algoritmii Monte Carlo sunt algoritmi aleatori care nu pot garanta un răspuns corect (nu oferă întotdeauna o soluție a problemei).

## Definition 3.1

Un algoritm aleator  $A$  este numit **algoritm Monte Carlo** pentru calculul funcției  $F$  dacă, pentru orice intrare  $x$  (orice argument al lui  $F$ ),

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

- Astfel de algoritmi apar adeseori în probleme de optimizare. Foarte relevant pentru un algoritm Monte Carlo este că probabilitatea de a greși este majorată.

## Amplificarea unui algoritm Monte Carlo

- Ca și în definiția algoritmilor Las Vegas constanta  $1/2$  poate fi înlocuită prin orice altă constantă  $\epsilon \in (0, 1)$ .
- *Amplificarea* este operația de repetare independentă a execuției algoritmului până când probabilitatea erorii scade suficient de mult. Este ca și cum am încerca să transformăm un algoritm Monte Carlo într-unul Las Vegas.
- Să presupunem că probabilitatea erorii unui algoritm  $A$  este cel mult  $\epsilon$  și că repetăm în mod independent algoritmul de  $k$  ori; probabilitatea de a obține numai răspunsuri eronate este cel mult  $\epsilon^k$ ,
- Cum  $\lim_{k \rightarrow \infty} \epsilon^k = 0$ , pentru valori mari ale lui  $k$ , putem face această probabilitate oricât de mică.
- **Atenție:** valori mari ale lui  $k$  pot mări foarte mult complexitatea timp.

## Verificarea înmulțirii matricilor

- Să presupunem că avem trei matrici pătratice de ordin  $n$ ,  $A$ ,  $B$  și  $C$ ; pentru simplificare vom presupune că operațiile sunt modulo 2.
- Dorim să verificăm dacă  $A \cdot B = C$ . Aceasta este o problemă de decizie (se poate răspunde doar cu "da" sau "nu").
- O metodă constă în a înmulți  $A$  cu  $B$  după care se compară rezultatul cu  $C$ ; înmulțirea matricilor necesită  $\mathcal{O}(n^3)$  timp sau, utilizând algoritmi mai sofisticăți,  $\mathcal{O}(n^{2.37})$ .
- Descriem un algoritm aleator care permite verificarea mai rapidă cu riscul de a primi un răspuns greșit, dar cu probabilitate scăzută.

## Verificarea înmulțirii matricilor

```

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

```

- Când algoritmul returnează "nu" suntem siguri că  $A \cdot B \neq C$ . Singura eroare pe care algoritmul o poate face este aceea de a returna "da" dar răspunsul să fie, de fapt, "nu".
- Probabilitatea de a greși poate fi majorată:

### Theorem 3.1

Pentru  $r \in \{0, 1\}^n$ , ales ca în algoritmul anterior,  $P(ABr = Cr) \leq \frac{1}{2}$ , știind că  $A \cdot B \neq C$ . ( $P(ABr = Cr | A \cdot B \neq C) \leq \frac{1}{2}$ .)

## Verificarea înmulțirii matricilor

**Proof.** Cum  $D = AB - C \neq 0$  - putem presupune că  $d_{1n} \neq 0$ ; dacă  $Dr = 0$ , atunci

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

După ce am ales  $r_1, \dots, r_{n-1}$  există două posibile valori pentru  $r_n$ , deci probabilitatea ca egalitatea de mai sus să aibă loc este cel mult  $1/2$ .

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

## Verificarea înmulțirii matricilor

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

- Algoritm constă în generarea a  $n$  numere aleatoare uniforme și în trei înmulțiri matrice  $\times$  vector ( $\mathcal{O}(n^2)$ ). Complexitatea timp a algoritmului este deci  $\mathcal{O}(n^2)$ .
- Dacă executăm în mod independent de  $k$  ori algoritmul obținem o probabilitate de a greși de cel mult  $2^{-k}$ .

## Algoritmul lui Karger

- Fie  $G = (V, E, w)$  un multigraf, unde  $w : E \rightarrow \mathbb{N}^*$  determină multiplicitatea muchiilor (altfel spus avem un graf cu ponderi întregi pe muchii).
- O *tăietură* (*cut*) în  $G$  este generată de o bipartiție a lui  $V$ ,  $(A, B)$ , iar costul ei este  $w(A, B) = \sum_{e \in E(A, B)} w(e)$ , unde  $E(A, B) = \{uv \in E : u \in A, v \in B\}$  sunt muchiile tăieturii.
- Problema *tăieturii de cost minim* (*min-cut problem*) cere să se determine o tăietură de cost  $w$  minim în  $G$ . Pentru rezolvarea acestei probleme există un algoritm determinist<sup>2</sup> de complexitate timp  $\mathcal{O}(nm + n^2 \log n)$  (prin contrast problema *max-cut* este NP-hard).
- Analizăm în această secțiune un algoritm aleator pentru o tăietură minimă bazat pe contractia muchiilor unui graf.

<sup>2</sup>Algoritmul Stoer-Wagner.

## Algoritmul lui Karger

- Operația de bază, *contraction*( $u, v$ ), înlocuiește în graful curent nodurile  $u$  și  $v$  cu un nod nou  $z$  asignează mulțimea muchiilor incidente lui  $z$  ca fiind muchiile incidente cu  $u$  și  $v$  (altele decât cele dintre  $u$  și  $v$ ).
- în timpul contracției muchiile incidente cu  $u$  și  $v$  cu un capăt comun sunt reținute ca muchii multiple.

```

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

```

## Algoritmul lui Karger

- După fiecare pas numărul de noduri scade cu unul și orice tăietură în noul graf corespunde unei tăieturi anterioare.
- După fiecare pas anumite tăieturi din graful original pot dispărea, dar algoritmul se termină cu o mulțime de muchii care reprezintă o tăietură din graful original.
- O strategie utilă este aceea de a termina algoritmul cu  $k \geq 2$  noduri în loc de 2 și după aceea se poate aplica un algoritm determinist pentru a determina o tăietură de cost minim în graful rămas.

```

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  $E(G)$ ; }

```

## Algoritmul lui Karger

Probabilități și Statistică

Probabilități și Statistică

Probabilități și Statistică

## Theorem 3.2

*Algoritmul aleator RandMinCut( $G, w, k$ ) are o complexitate timp polinomială și determină o tăietură minimă cu probabilitate de cel puțin  $\frac{k(k-1)}{n(n-1)}$ .*

**Proof.** Fiecare contracție poate fi executată în  $\mathcal{O}(n)$  pași (de ce?); astfel, complexitatea algoritmului RandMinCut( $G, w, k$ ) este  $\mathcal{O}((n-k)n)$ . Dacă într-un multigraf cu  $n$  noduri avem o tăietură de cost minim  $h$ , atunci gradul minim al unui nod este cel puțin  $h$ , deci numărul de muchii din  $G$  este cel puțin  $hn/2$ ; probabilitatea ca o muchie care se află într-o tăietură de cost minim să fie contractată va fi cel mult  $\frac{h}{hn/2} = \frac{2}{n}$ . Fie  $C$  o tăietură de cost minim. Estimăm probabilitatea ca algoritmul să nu returneze  $C$ .

## Algoritmul lui Karger

Dacă  $C$  este returnată, atunci nici una dintre muchiile lui  $C$  nu a fost contractată.

Fie  $A_i$  evenimentul ca o muchie a lui  $C$  a fost contractată în pasul  $i$  și  $B_i$  evenimentul ca nicio muchie a lui  $C$  nu a fost contractată în primele  $(i - 1)$  iterații. Numărul de noduri după  $i$  iterații este  $(n - i)$ , deci

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

Pe de altă parte,

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

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

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

## Algoritmul lui Karger

- Pentru  $k = 2$  algoritmul  $\text{RandMinCut}(G, w)$  garantează o probabilitate a erorii de cel mult  $[1 - 2/n(n-1)] \geq (1 - 2/n^2)$ ;
- Putem reduce această probabilitate repetând execuția algoritmului: dacă îl executăm de  $n^2/2$  ori (mărind complexitate timp la  $\mathcal{O}(n^4)$ ) probabilitatea erorii este cel mult

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

- Dacă executăm de  $n(n-1) \ln n$  ori algoritmul (mărind complexitate timp la  $\mathcal{O}(n^4 \log n)$ ) probabilitatea erorii este cel mult

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

- Marele avantaj al acestui algoritm este simplitatea lui.

## Metoda probabilistică

- Metoda probabilistică este o tehnică folosită pentru demonstrarea existenței unor obiecte matematice (combinatorii) cu anumite proprietăți.
- Ideea de bază a acestei metode constă în a demonstra că probabilitatea existenței unui obiect cu proprietățile cerute este strict pozitivă, ceea ce înseamnă că un asemenea obiect există.
- Pentru aceasta, construim un spațiu de probabilitate peste mulțimea obiectelor implicate și arătăm ca probabilitatea obiectului respectiv este nenulă.

## Metoda probabilistică

## Theorem 4.1

(Principiul mediei) Dacă  $X$  este o variabilă aleatoare discretă cu  $\mathbb{E}[X] \geq \alpha$ , atunci  $P\{X \geq \alpha\} > 0$ .

Primele exemple ale aplicării metodei probabiliste sunt legate de problema satisfiabilității.

## Proposition 4.1

Fie  $\mathcal{F} = \{C_1, C_2, \dots, C_m\}$  o familie de  $m$  clauze. Există o asignare a valorilor de adevăr a variabilelor booleene implicate, astfel ca numărul de clauze satisfăcute să fie cel puțin

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

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

## Problema satisfiabilității

proof: Imaginăm următorul experiment aleator abstract: fiecărei variabile booleene  $x$  îi asignăm independent valoarea 1 (*adevărat*) sau 0 (*fals*) cu aceeași probabilitate 0.5. Definim variabilele aleatoare

$$X_i = \begin{cases} 1, & \text{dacă } C_i \text{ este adevărată} \\ 0, & \text{dacă } C_i \text{ este falsă} \end{cases}$$

Probabilitatea ca să fie adevărată clauza  $C_i$  este egală cu probabilitatea ca măcar unul dintre cei  $|C_i|$  literalii ai ei să fie adevărat, adică  $(1 - 2^{-|C_i|})$ . Numărul de clauze satisfăcute este egal cu  $X = \sum_{i=1}^n X_i$  și,

atunci

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

Concluzia urmează conform Teoremei 4.1. ■

## Problema satisfiabilității

Probabilități și Statistică

Probabilități și Statistică

Probabilități și Statistică

Probabilități și Statistică

Probabilități și Statistică

Probabilități și Statistică

Probabilități și Statistică

Probabilități și Statistică

Probabilități și Statistică

### Corollary 4.1

*Orice instanță a problemei  $k$ -SAT cu un număr de clauze mai mic strict decât  $2^k$  este satisfiabilă. (O instanță a problemei  $k$ -SAT are în fiecare clauză exact  $k$  literalii.)*

**Dem.** Reluând argumentul de mai sus, cu  $|C_i| = k, \forall i = \overline{1, m}$ :

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

obținem de aici că  $P\{X \geq m\} = P\{X > m - 1\} > 0$  - deci există o asignare a valorilor de adevăr care să satisfacă toate cele  $m$  clauze. ■

## Problema satisfiabilității

- Minorantul indicat în acest rezultat este cel mai bun posibil deoarece putem defini o instanță a problemei  $k$ -SAT cu  $2^k$  clauze care să fie nesatisfiabilă: de exemplu familia tuturor clauzelor având  $k$  literali definiți peste o mulțime de  $k$  variabile booleene.
- În mod similar se poate demonstra






### Proposition 4.2

O familie de clauze  $\mathcal{F} = \{C_1, \dots, C_m\}$  este satisfiabilă dacă

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



## 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.