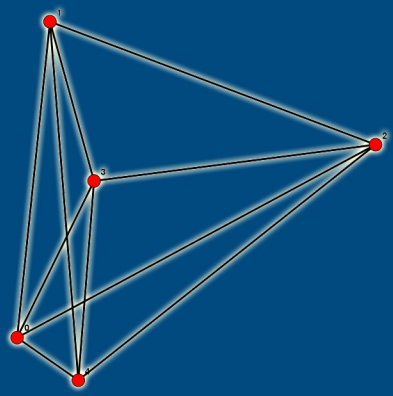
UNIUNEA EUROPEANĂ
GUVERNUL ROMÂNIEI MINISTERUL MUNCII, FAMILIEI, PROTECŢIEI SOCIALE ŞI PERSOANELOR VÂRSTNICE AMPOSDRU
Fondul Social European POSDRU 2007-2013
Instrumente Structurale 2007-2013
MINISTERUL EDUCAŢIEI NAŢIONALE
OIPOSDRU
UNIVERSITATEA POLITEHNICA DIN BUCUREŞTI

Proiect cofinanţat din Fondul Social European prin Programul Operaţional Sectorial Dezvoltarea Resurselor Umane 2007-2013
Investeşte în oameni!

Răzvan ANDONIE Angel CAȚARON Honorius GÂLMEANU
Mihai IVANOVICI Lucian SASU

Algoritmi și Structuri de Date pentru Imagistică și Bioinformatică

NOTE DE CURS



EDITURA UNIVERSITĂȚII TRANSILVANIA DIN BRAŞOV

Răzvan ANDONIE Angel CAȚARON Honorius GÂLMEANU

Mihai IVANOVICI Lucian SASU

Algoritmi și Structuri de Date

pentru

Imagistică și Bioinformatică

EDITURA UNIVERSITĂȚII TRANSILVANIA DIN BRAŞOV

2013

EDITURA UNIVERSITĂȚII *TRANSILVANIA* DIN BRAȘOV

Adresa: 500091 Brașov,
B-dul Iuliu Maniu 41A
Tel:0268 – 476050
Fax: 0268 476051
E-mail : editura@unitbv.ro

Tipărit la:
Tipografia Universității *Transilvania* din Brașov
B-dul Iuliu Maniu 41A
Tel: 0268 – 476050

Editură acreditată de CNCSIS
Adresa nr.1615 din 29 mai 2002

Referenți științifici: **Prof. univ. dr. ing. Gheorghe TOACȘE**
Conf. univ. dr. ing. Mihai CIUC

Descrierea CIP a Bibliotecii Naționale a României
Algoritmi şi structuri de date pentru imagistică şi bioinformatică : note de curs / Răzvan Andonie, Angel Cațaron, Honorius Gâlmeanu, ... – Brașov : Editura Universității "Transilvania", 2013
Bibliogr.
ISBN 978-606-19-0206-4
I. Andonie, Răzvan
II. Cațaron, Angel
III. Gâlmeanu, Honorius
004.421

CUPRINS

1. STRUCTURI ELEMENTARE DE DATE 1
2. ANALIZA EFICIENȚEI ALGORITMILOR 45
3. TEHNICI DE SINTEZĂ A ALGORITMILOR 89
4. STRUCTURI DE DATE AVANSATE 153
5. COMPLEXITATE NP 193
6. COMPLEXITATE PARALELĂ 217
7. TEHNICI DE PRELUCRARE A SECVENȚELOR DE ȘIRURI 227
8. PATTERN-URI STRUCTURALE 257
9. ALGORITMI RANDOMIZAȚI 277
10. ALGORITMI DE APROXIMARE PENTRU PROBLEME NP-COMPLETE . 295

Prefață

Cartea de față este o a doua realizare a colectivului de autori în cadrul proiectului POS-DRU/86/1.2/S/61756 intitulat *Tehnici de Analiză, Modelare și Simulare pentru Imagistică, Bioinformatică și Sisteme Complexe (ITEMS)*. Colectivul ITEMS de la Universitatea Transilvania din Brașov a răspuns de conceperea și predarea cursului de *Algoritmi și Structuri de Date* pentru programul de Masterat de Excelență organizat sub umbrela acestui proiect la Universitatea Politehnica din București. Materialul a fost rafinat în perioada septembrie 2010 - februarie 2013 în mai multe faze și în special în urma interacțiunii cu două serii de studenți masteranzi. Prezentul volum constituie suportul de curs, într-un format sintetic, ce urmărește în mod fidel cele zece prelegeri concepute pentru acest curs. Chiar dacă a fost conceput pentru masteranzii ITEMS, suntem convinși că lucrarea poate fi utilă multor studenți și specialiști din domeniul Calculatoare și Tehnologia Informației.

Echipa de autori este foarte omogenă, grație faptului că ne cunoaștem de ani buni și am realizat multe lucrări didactice și de cercetare împreună. Practic, prin prisma activității colectivului din cadrul fostei filiale de la Brașov a Institutului de Tehnică de Calcul și a actualului Departament de Electronică și Calculatoare din cadrul Universității Transilvania din Brașov, putem vorbi despre o adevărată școală de algoritmi la Brașov. O parte dintre noi suntem cadre didactice universitare, o alta lucrăm în firme de software, fapt ce ne-a permis îmbinarea optimă a elementelor teoretice cu cele practice. Pentru noi, perioada elaborării acestui material a fost extrem de creativă, în special datorită lucrului în echipă. Este motivul pentru care autorii sunt listați alfabetic, fiecare cu contribuții egale, greu de separat.

Brașov, februarie 2013

Autorii

Cursul nr. 1.

STRUCTURI ELEMENTARE DE DATE

Cuprins

Liste

Stive

Cozi

Grafuri

Arbori

Heap-uri

Structuri de mulțimi disjuncte

Arbori binari de căutare și parcurgeri în grafuri

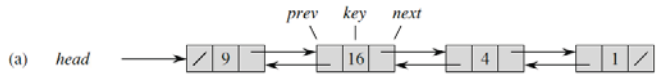
Liste - caracteristici

- **lista:** colecție de elemente aranjate într-o ordine prestabilită
- **lungimea listei:** numărul de elemente
- **scop:** accesarea rapidă a primului/ultimului element, respectiv a elementului predecesor/succesor celui curent
- **listă liniară:** înlănțuire simplă de elemente
- **listă circulară:** fiecare element are în mod necesar atât un predecesor cât și un succesor

Liste - variante de implementare

- **Implementarea secvențială:**
  - stocare în locații succesive
  - acces rapid la predecesor/succesor
  - inserare/ștergere relativ complicată
  - din rațiuni de eficiență, nu se folosește întreg spațiul alocat listei
- **Implementarea înlănțuită:**
  - fiecare nod conține, pe lângă informație (key) și adresele nodurilor succesor (next) respectiv predecesor (prev) (liste simplu- dublu- înlănțuite)
  - fiecare element se poate aloca dinamic (sau se poate păstra o listă prealocată cu elemente libere)
  - accesul la un element necesită parcurgerea predecesorilor
  - inserarea/ștergerea este rapidă

Căutarea într-o listă înlănțuită

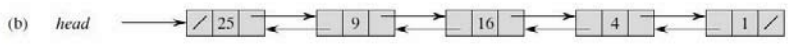


function find-list (L, k)

1: $x \leftarrow L.head$
2: **while** $x \neq NIL$ and $x.key \neq k$ **do**
3: $\quad x \leftarrow x.next$
4: **end while**
5: **return** x

- necesită un timp în $\Theta(n)$ pentru cazul cel mai nefavorabil
- exemplu: (a) căutarea elementului cu cheia 1

Inserarea într-o listă înlănțuită

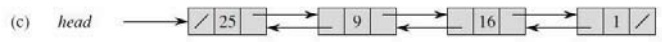


procedure insert-list (L, x)

1: $x.next \leftarrow L.head$
2: $x.prev \leftarrow NIL$
3: **if** $L.head \neq NIL$ **then**
4: $\quad L.head.prev \leftarrow x$
5: **end if**
6: $L.head \leftarrow x$

- ordinul de timp este $O(1)$
- exemplu: (b) inserarea elementului 25

Ștergerea într-o listă înlănțuită



procedure remove-list (L, x)

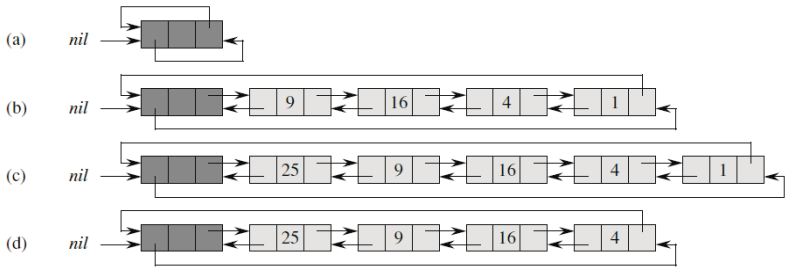
```
1: if x.prev ≠ NIL then
2:     x.prev.next ← x.next
3: else
4:     L.head ← x.next
5: end if
6: if x.next ≠ NIL then
7:     x.next.prev ← x.prev
8: end if
```

- ștergerea se face în timp constant, $O(1)$
- exemplu: (c) ștergerea elementului 4
- legarea legaturilor primului și ultimului element transformă lista într-o listă circulară, navigarea se face fără condiție de capăt

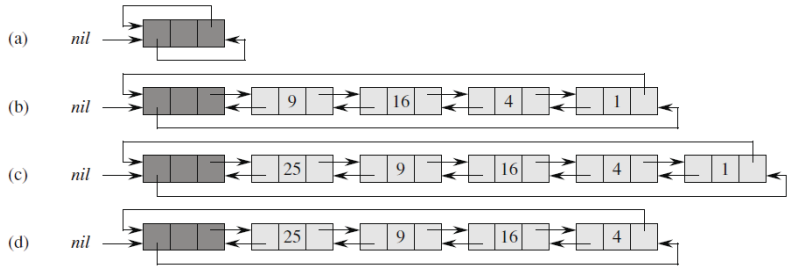
Santinele

Santinela:

- obiect fictiv a cărui folosire face inutilă scrierea condițiilor de capăt
- transformă lista întro listă circulară
- lista vidă conține doar santinela (a)



Căutarea, inserarea și ștergerea folosind santinela I



function find1-list (L, k)

1: $x \leftarrow L.nil.next$
2: **while** $x \neq L.nil$ and $x.key \neq k$ **do**
3: $\quad x \leftarrow x.next$
4: **end while**
5: **return** x

Căutarea, inserarea și ștergerea folosind santinela II

procedure insert1-list (L, x)

1: $x.next \leftarrow L.nil.next$
2: $L.nil.next.prev \leftarrow x$
3: $L.nil.next \leftarrow x$
4: $x.prev \leftarrow L.nil$

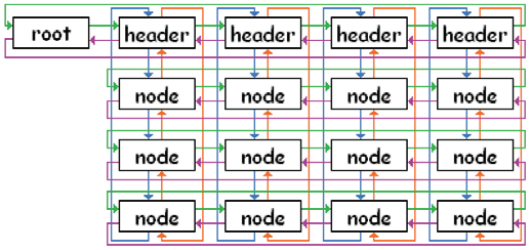
Exemplu: (c) lista după inserarea elementului 25.

procedure remove1-list (L, x)

1: $x.prev.next \leftarrow x.next$
2: $x.next.prev \leftarrow x.prev$

Exemplu: (d) după ștergerea elementului 1.

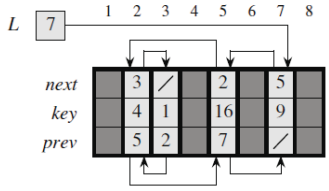
Dancing links



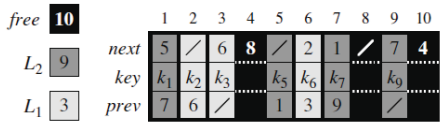
- un element are legături către 4 elemente vecine
- structură folosită în rezolvarea problemei acoperirii exacte, propusă de D. Knuth
- bazată pe observația simplă că ascunderea / revelarea unui nod se face ușor prin modificarea legăturilor
- $node.next.prev \leftarrow node.prev$ $\quad$ $node.next.prev \leftarrow node$
- $node.prev.next \leftarrow node.next$ $\quad$ $node.prev.next \leftarrow node$
- parcurgînd o linie, se poate ușor ascunde / revela acea linie

Tehnica tablourilor paralele

- Implementarea listelor 'simulând' adrese
- Fiecare tablou implementează un câmp (key, next, prev)
- Lista reprezentată este { 9, 16, 4, 1 }
- Elementele nealocate pot fi de asemenea păstrate într-o structură similară

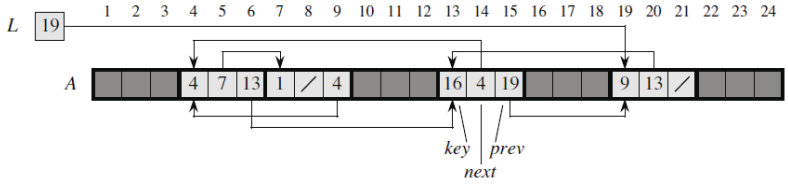


(a) Tablouri paralele



Reprezentarea prin tablou unic

- Obiectul este reprezentat ca o structură complexă
- Componentele sale sunt accesate prin adresarea indexată față de adresa de început a obiectului
- Tabloul presupune toate elementele omogene



- Lista conține elementele { 9, 16, 4, 1 }

Stiva ca tablou I

- **Stiva:** ultimul sosit, primul servit
- Pentru o stivă implementată printr-un tablou $S[1 \ldots n]$, vârful stivei este $S[S.top]$, elementul de la baza stivei este $S[1]$
- **Stivă vidă:** $S.top = 0$, **stivă plină:** numărul de elemente depășește n

function empty-stack (S)

1: **if** $S.top = 0$ **then**
2: **return** true
3: **else**
4: **return** false
5: **end if**

procedure push-stack (S, x)

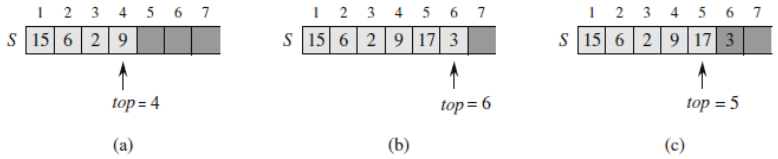
1: $S.top \leftarrow S.top + 1$
2: $S[S.top] \leftarrow x$

Stiva ca tablou II

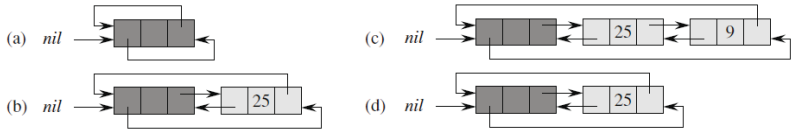
function pop-stack (S)

1: **if** empty-stack(S) **then**
2: **print** stivă vidă
3: **else**
4: $S.top \leftarrow S.top - 1$ **return** $S[S.top + 1]$
5: **end if**

- Ordinul de timp este $O(1)$ pentru fiecare din cele trei operații
- (a) S are 4 elemente (b) stiva după apelurile push-stack(S,17) și push-stack(S,3), (c) S după apelul pop-stack(S) care a întors 3



Stiva ca listă înlănțuită I



procedure push-stack-linkedlist (S, x)

1: $x.prev \leftarrow L.nil.prev$
2: $x.next \leftarrow L.nil$
3: $L.nil.prev.next \leftarrow x$
4: $L.nil.prev \leftarrow x$

- (a) stiva implementată ca listă folosind santinela
- (b), (c) adăugarea la coadă

Stiva ca listă înlănțuită II

function pop-stack-linkedlist (S)

1: $x \leftarrow L.nil.prev$
2: $L.nil.prev \leftarrow L.nil.prev.prev$
3: **return** x

- (c), (d) scoaterea elementului final

Calculul expresiilor aritmetice

Dacă avem de calculat expresia:

$5 * ((9 + 8) * (4 * 6)) + 7)$

Se poate folosi o stivă pentru a memora rezultatele intermediare. Scrierea formei postfixate (poloneze inverse):

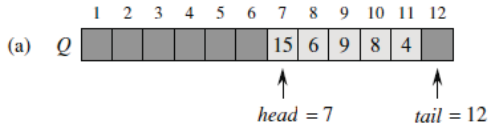
5 9 8 + 4 6 * * 7 + *

Operațiile de evaluare folosind o stivă:

push(5); push(9); push(8); push(pop + pop); push(4); push(6); push(pop * pop); push(pop * pop); push(7); push(pop + pop); push(pop * pop); write (pop);

Coadă implementată ca vector

- **Coada:** primul sosit, primul servit
- Primul element din coadă: $Q.head$
- Prima locație liberă: $Q.tail$
- Indicii se parcurg circular, după locația n urmează imediat locația 1
- Condiția de coadă goală: $Q.head = Q.tail$
- Condiția de coadă plină: $Q.head = Q.tail + 1$
- Coada va avea astfel cel mult $n - 1$ elemente, ca să putem distinge între cele două teste (plină / goală)
- (a) Coadă cu 5 elemente, locațiile $Q[7 \ldots 11]$

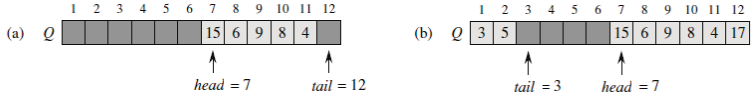


Inserarea unui element

function insert-queue (Q, x)

1: $Q[Q.tail] \leftarrow x$
2: **if** $Q.tail = n$ **then**
3: $\quad Q.tail \leftarrow 1$
4: **else**
5: $\quad Q.tail \leftarrow Q.tail + 1$
6: **end if**
7: **if** $Q.head = Q.tail$ **then**
8: $\quad$ **return** coadă plină
9: **else**
10: $\quad$ **return** succes
11: **end if**

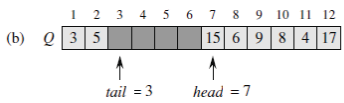
- (b) Se pun în coadă elementele 17, 3 și 5

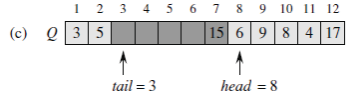


Ștergerea unui element

```
function delete-queue (Q)
1: if Q.head = Q.tail then
2:     return coadă vidă
3: end if
4: x ← Q[Q.head]
5: if Q.head = n then
6:     Q.head ← 1
7: else
8:     Q.head ← Q.head + 1
9: end if
10: returns x
```

- (c) Se scoate din coadă elementul 15
- Cozile pot fi implementate și ca liste înlănțuite

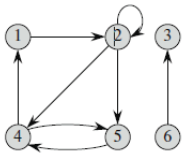




Grafuri

- **Graf:** pereche $G =< V, M >$ unde V - mulțime de vârfuri iar $M \subseteq V \times V$ - mulțime de muchii
- **graf orientat:** muchia (a, b) diferă de muchia (b, a)
- **graf neorientat:** mulțimea muchiilor este formată din perechi de vârfuri neordonate
- **arce incidente** într-un / dintr-un anumit vârf
- **vârfuri adiacente:** există muchia care le unește
- **gradul unui vârf:** numărul muchiilor incidente
- **gradul exterior/interior:** numărul arcelor care pleacă/intră
- **vârf izolat:** grad zero
- **drum de lungime k:** $< v_0, v_1 \ldots v_k >$ unde v_0 vârful inițial, v_k vârful final și $(v_{i-1}, v_i) \in M$
- **drum elementar:** vârfurile sale, în afară de capete, sunt distincte
- ciclul $< 2, 4, 1, 2 >$ este elementar, dar $< 1, 2, 4, 5, 4, 1 >$ nu

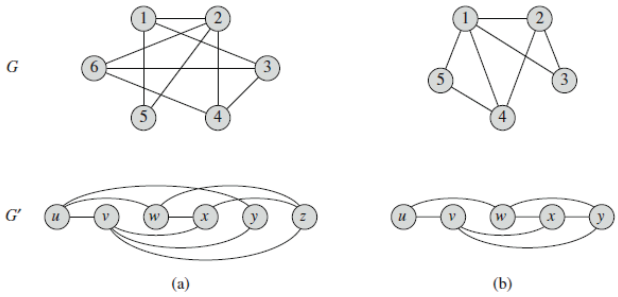
Grafuri



- **graf aciclic:** fără cicluri
- **graf conex:** fiecare pereche de vârfuri este conectată printr-un drum
- **componente conexe:** $\{1, 2, 5\}$, $\{3, 6\}$ și $\{4\}$
- **graf tare conex:** graf orientat în care oricum am alege două vârfuri i și j, există atât un drum de la i la j cât și un drum de la j la i
- **drum hamiltonian:** într-un graf neorientat, un drum elementar care vizitează fiecare vârf al grafului exact o singură dată (câte drumuri hamiltoniene distincte avem?)

Grafuri

- **grafuri izomorfe:** $G =< V, M >$ izomorf cu $G' =< V', M' >$ dacă există o bijecție $f : V \to V'$ a.î. $(u, v) \in M \Leftrightarrow (f(u), f(v)) \in M'$
- bijecția se poate interpreta ca o reetichetare



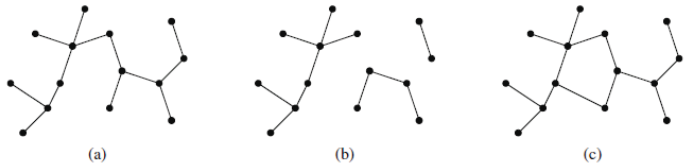
Grafuri

- **subgraf:** $G' =< V', M' >$ este subgraf al lui $G =< V, M >$ dacă $V' \subseteq V$ și $M' \subseteq M$
- **subgraf indus:** acel G' pentru care $M' = \{(u, v) \in M : u, v \in V'\}$
- **graf complet:** graf neorientat în care oricare două vârfuri sunt adiacente
- **graf bipartit:** graf $G =< V, M >$ în care mulțimea de vârfuri V poate fi partiționată în două mulțimi V_1 și V_2 astfel că $(u, v) \in M$ implică fie $u \in V_1$ și $v \in V_2$, fie $u \in V_2$ și $v \in V_1$ (toate muchiile merg de la V_1 la V_2 sau invers)
- **pădure:** graf neorientat aciclic (pot fi mai multe componente conexe)
- **arbore (liber):** graf neorientat aciclic conex

Reprezenarea grafurilor

- **matrice de adiacență**
  - memoria necesară în $O(n^2)$, unde $n = |V|$
  - verificarea adiacenței în $O(1)$
  - obținerea vârfurilor adiacente în $O(n)$
- **liste de adiacență**
  - suma lungimilor listelor de adiacență este $2m$ / m (neorientat / orientat), unde $m = |M|$
  - preferabilă pentru număr mic de muchii
  - verificarea adiacenței a două vârfuri poate necesita inspectarea ambelor liste de adiacență
  - obținerea vârfurilor adiacente se poate realiza cu mai puțin de n operații
- **listă de muchii**
  - reprezentare eficientă când este necesară examinarea tuturor muchiilor grafului

Arbori

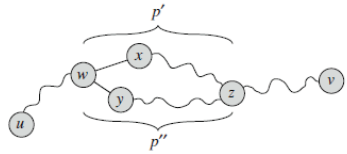


(a) un arbore (liber), (b) o pădure, (c) un graf ce conține un ciclu (nici arbore, nici pădure)

Proprietăți echivalente

1. G este un arbore liber
2. Oricare două vârfuri din G sunt conectate printr-un drum unic
3. G este conex, dar prin eliminarea oricărei muchii din el, nu mai este conex
4. G este conex și $|M| = |V - 1|$
5. G este aciclic și $|M| = |V - 1|$
6. G este aciclic, dar dacă adăugăm o muchie oarecare în M, se formează un ciclu

(1) $\Rightarrow$ (2)



- arborele fiind conex, oricare două vârfuri sunt conectate prin cel puțin un drum elementar
- fie u și v două astfel de vârfuri
- considerăm că există două astfel de drumuri
- pentru a fi diferite, trebuie să aibă vârfuri comune (în afară de capete)
- drumurile p' și p'' nu au vârfuri comune cu excepția capetelor
- vor crea un ciclu, contradicție

(2) ⇒ (3)

- dacă avem un drum între oricare două vârfuri, graful este conex
- fie muchia (u, v); ea este singurul drum de la u la v
- odată eliminată, nu mai există acel drum
- graful nu mai e conex

(3) ⇒ (4)

- se demonstrează prin inducție
- dacă am elimina din graf o muchie la întâmplare, graful nu ar mai fi conex
- cele două componente sunt conexe, verifică relația $|M| = |V| - 1$
- înseamnă că trebuie adunată și muchia ce a fost eliminată

(4) $\Rightarrow$ (5)

- presupunem că graful, deși conex, are cel puțin un ciclu cu k vârfuri
- considerăm numai subgraful G_k determinat de acel ciclu
- numărul de muchii este exact k
- deoarece G e conex, pentru un nou vârf v_{k+1} va exista o muchie ce îl conectează la graf
- noul graf G_{k+1} va avea exact $k+1$ muchii
- se continuă până se înglobează toate cele n vârfuri ale lui G
- graful $G_n = G$ inițial ajunge astfel să aibă $|V|$ muchii, contradicție

(5) $\Rightarrow$ (6)

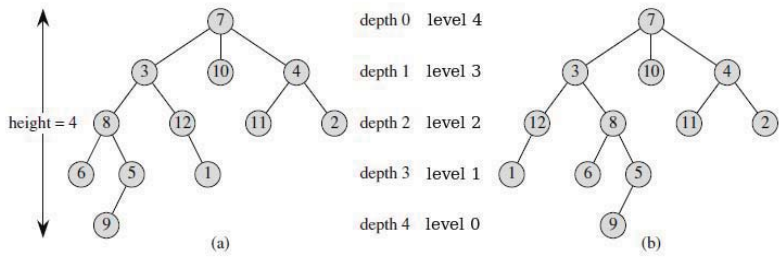
- considerăm k numărul componentelor conexe
- fiecare componentă conexă este un arbore, deci numărul total de muchii este $|V| - k$
- avem deci o singură componentă conexă
- acesta e un arbore, deci oricare noduri sunt deja conectate printr-un drum simplu
- adăugarea unei noi muchii determină un ciclu

(6) $\Rightarrow$ (1)

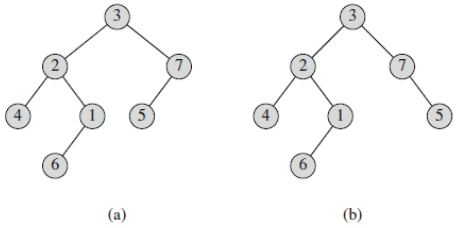
- fie două noduri arbitrar alese, u și v
- dacă adăugăm muchia (u, v), formăm un ciclu
- înseamnă că u și v sunt deja conectate printr-un drum
- deci orice pereche de vârfuri sunt deja conectate printr-un drum

Arbori cu rădăcină

- arbore liber în care un vârf este denumit rădăcină
- strămoș, descendent
- subarbore determinat de rădăcina $x = 3$
- nod părinte, nod copil, noduri frați
- gradul unui nod
- adâncimea unui nod, înălțimea arborelui

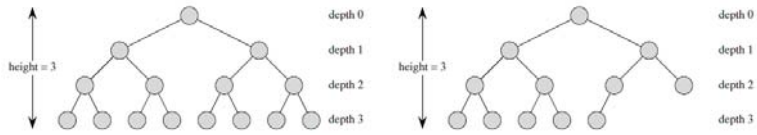


Arbori binari cu rădăcină



- rădăcină, subarbore stâng, subarbore drept
- nodurile terminale sunt denumite frunze
- pentru un nod, de regulă fiul său stâng este parcurs înaintea fiului drept
- (a) un subarbore binar
- (b) arborele ordonat este identic, dar, privit ca arbore binar poziţional, diferă; spre deosebire de (a), fiul lui 7 este fiu drept şi nu fiu stâng

Arbori binari



(g) Arbore binar plin (h) Arbore binar complet

- **arbore binar plin:** fiecare nod este fie frunză, fie are exact doi copii
- **arbore binar complet:** arbore binar plin, dar pe de pe ultimul nivel lipsesc toate frunzele de la un indice încolo
- copiii unui nod se etichetează pornind de la 1
- **arbore k-ar:** arbore poziţional ce are copii etichetaţi doar până la ordinul k
- **arbore k-ar plin:** toate frunzele au aceeaşi adâncime
- numărul nodurilor de pe nivelul i este k^i

Arbori k-ari

- numărul nodurilor unui arbore k-ar plin de înălțime h este:

$$1 + k + k^2 + \cdots + k^{h-1} = \sum_{i=0}^{h-1} k^i = \frac{k^h - 1}{k - 1}$$

- numărul nodurilor unui arbore k-ar complet este:

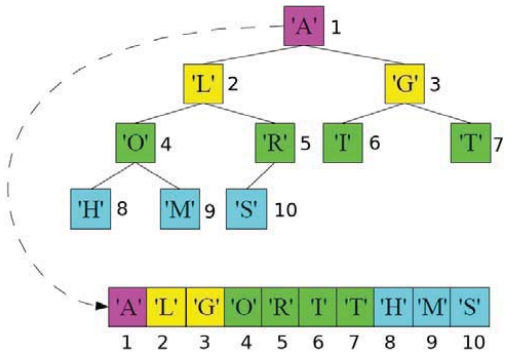
$$\frac{k^{h-1} - 1}{k - 1} < n \leq \frac{k^h - 1}{k - 1}$$

- respectiv, pentru un arbore binar:

$$2^{h-1} \leq n < 2^h$$

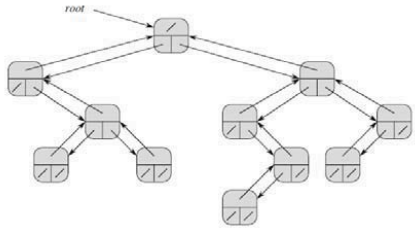
- înălțimea unui arbore binar complet cu n vârfuri este $\lfloor log_2 n \rfloor$
- unde $\lfloor x \rfloor = max\{n | n \leq x, n \in \mathbb{Z}\}$ și $\lceil x \rceil = min\{n | n \geq x, n \in \mathbb{Z}\}$

Reprezentarea sub forma unui tablou



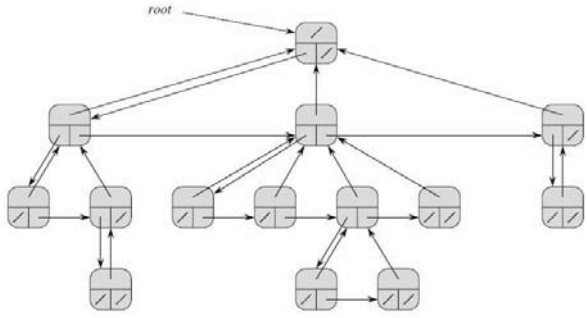
- orice arbore k-ar complet se poate reprezenta sub formă de tablou
- nodul i, fiul stâng: $T[2i]$, fiu drept: $T[2i + 1]$, părinte: $T[i/2]$
- cât de avantajoasă este reprezentarea sub formă de tablou a unui arbore binar care nu este complet?

Reprezentarea prin adrese (links)



- structura unui nod conține adrese pentru a indica părintele, fiul stâng și fiul drept
- dacă un nod este terminal, atunci adresele pentru copii au valoarea NIL
- rădăcina are adresa părintelui = NIL
  - se poate folosi schema pentru arbori k-ari
  - fiecare nod va avea alocată o structură pentru k fii
  - risipă de memorie pentru arbori dezechilibrați, $O(nk)$ spațiu

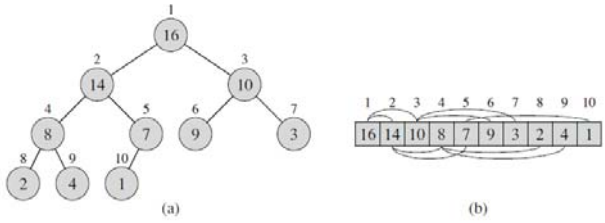
Arbori cu număr nelimitat de ramuri



- reprezentarea:
  - fiul stâng este o adresă ce indică spre cel mai din stânga descendent
  - fiul drept este o adresă către fratele cel mai apropiat spre dreapta
  - fiecare nod are o adresă ce indică spre părinte
- reprezentare ce utilizează $O(n)$ spațiu

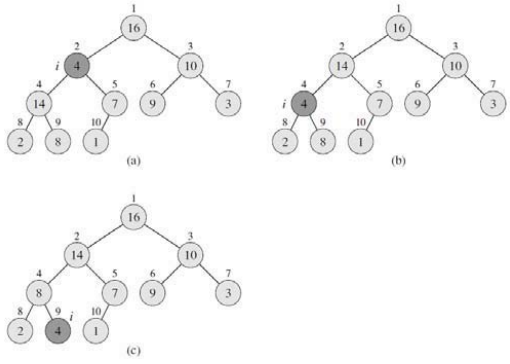
Heap-uri

- **heap:** arbore binar complet (arbore binar plin, din care lipsesc ultimele frunze)
- rădăcina heap-ului este $A[1]$
- se pot determina foarte ușor părintele, fiul stâng respectiv fiul drept al unui nod i
- în setul de instrucțiuni aceste operații se reduc la simple shift-ări
- **proprietatea de heap:** $A[i \textbf{ div } 2] \geq A[i]$
- ce rezultă de aici privitor la rădăcină?
- merită implementat un heap ca listă înlănțuită?



Proprietatea de heap

- **înălțimea** unui nod este numărul muchiilor celui mai lung drum care leagă nodul cu o frunză
- înălțimea arborelui este înălțimea rădăcinii, adică $\Theta(\log n)$
- dacă proprietatea de heap nu este respectată, se realizează 'scufundarea' nodului în heap



Reconstituirea proprietății de heap

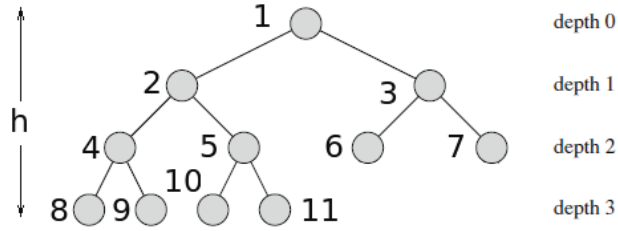
procedure sift-down-heap (A[1 .. n], i)

```
1: l ← i div 2
2: r ← l + 1
3: if l ≤ n and A[l] > A[i] then
4:     max ← l
5: else
6:     max ← i
7: end if
8: if r ≤ n and A[r] > A[max] then
9:     max ← r
10: end if
11: if max ≠ i then
12:     A[i] ↔ A[max]
13:     sift-down-heap(A, max)
14: end if
```

Procedura recursivă sift-down-heap

- la fiecare pas se determină indicele elementului maxim dintre i și cei doi fii ai săi
- dacă elementul i este cel mai mare, procedura se termină
- altfel, elementul maxim este interschimbat cu i
- subarborele asociat nodului fiu tocmai interschimbat este verificat la rândul lui d.p.d.v. al proprietății de heap
- pentru subarborele considerat, ne interesează să vedem cât de dezechilibrat poate fi
- puternic dezechilibrat înseamnă cazul cel mai nefavorabil
- intuitiv, pentru un nod i care se interschimbă cu fiul său stâng, considerăm că pe ramura fiului drept sunt cele mai putine noduri posibile
- situația corespunde unui arbore binar complet

Cazul cel mai nefavorabil I



- un arbore binar plin de înălțime h are $2^h - 1$ vârfuri
- subarborele stâng are $2^{h-1} - 1$ vârfuri,
- subarborele drept are $2^{h-2} - 1$ vârfuri
- în total arborele are

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

Cazul cel mai nefavorabil II

$$\frac{\textit{numar noduri subarbore}}{n} = \frac{2^{h-1} - 1}{3 \cdot 2^{h-2} - 1} \leq \frac{2}{3}$$

- numărul de noduri din subarborele stâng este maxim $2n/3$
- timpul de execuție al procedurii sift-down-heap este $\Theta(1)$ plus timpul de execuție pentru cazul subarborelui stâng:

$$T(n) \leq T(2n/3) + \Theta(1)$$

- folosind teorema master (prezentată ulterior) se obține $T(n) = O(\textit{log } n)$, respectiv $O(h)$

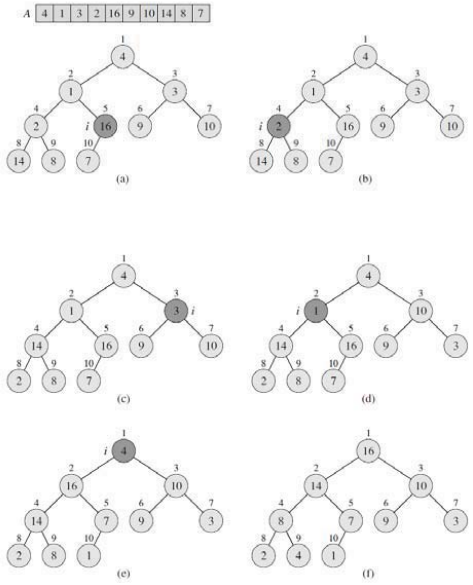
Construirea unui heap

procedure make-heap (A[1 .. n])

1: **for** $i \leftarrow n$ div 2 **downto** 1 **do**
2: sift-down-heap(A, i)
3: **end for**

- elementele A[n div 2 + 1 .. n] sunt frunze
- pot fi considerate deja heap-uri
- procedura restabilește proprietatea de heap începînd cu nodul n div 2
- fiecare apel al procedurii sift-down-heap necesită un timp $O(log\ n)$
- timpul de execuție ar părea că este în $O(n\ log\ n)$
- totuși, ordinul de timp este mai bun

Make-heap. Exemplu



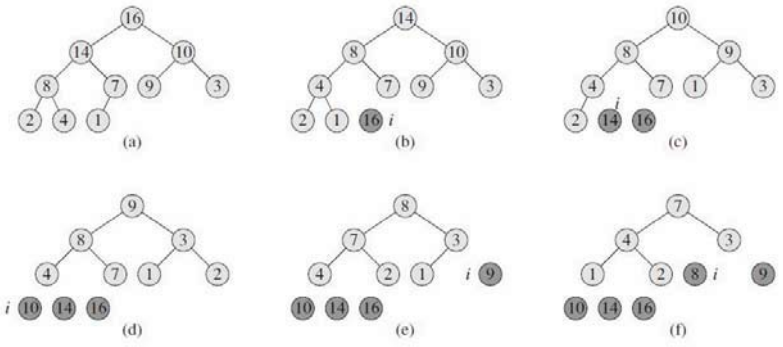
Algoritmul heapsort

procedure sort-heap (A[1 .. n])

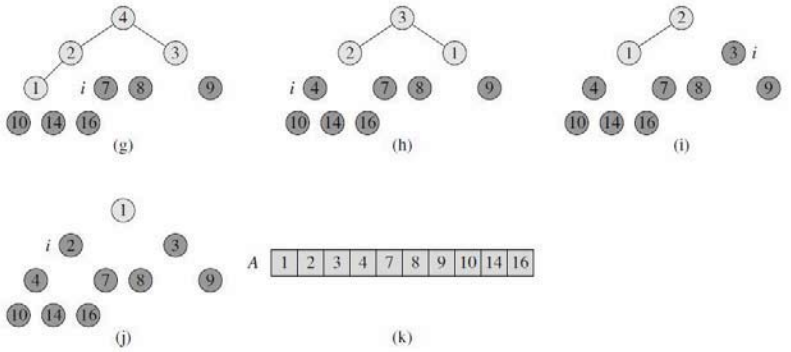
1: make-heap(A)
2: **for** $i \leftarrow n$ **downto** 2 **do**
3: $A[1] \leftrightarrow A[i]$
4: sift-down-heap(A[1 .. i-1], 1)
5: **end for**

- elementul maxim este în $A[1]$
- cel mai mic element se pune apoi în $A[1]$ și se reface proprietatea de heap
- dimensiunea heap-ului descreşte de la $n-1$ până la 2
- timpul procedurii este $O(n \log n)$

Heapsort. Exemplu I



Heapsort. Exemplu II



Cozi de priorități

- **coada de priorități:** structură de date în care fiecare element conținut are asociată o **cheie**
- se pot face următoarele operații:
  - insert-heap(S, x), inserează elementul x în S
  - maxim-heap(S), întoarce maximul din S, timp în $\Theta(1)$
  - extract-max-heap(S), extrage maximul din S, cu ștergere
- aplicații
  - planificarea lucrărilor pe un calculator (cheile fiind prioritățile)
  - simularea unor evenimente controlate (cheia este momentul de timp, min-heap)

Extragerea din heap

function extract-max-heap (A[1 .. n])

1: $max \leftarrow A[1]$
2: $A[1] \leftarrow A[n]$
3: sift-down-heap(A[1 .. n-1], 1)
4: **return** max

- timpul de execuție $O(log\ n)$, datorat înălțimii heap-ului

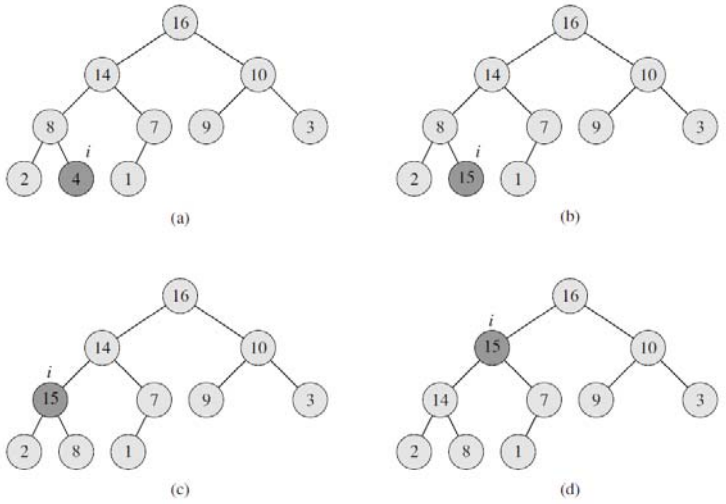
Inserarea în heap

procedure insert-heap (A[1 .. n], x)

1: $i \leftarrow n + 1$
2: **while** $i > 1$ and $A[i\ div\ 2] < x$ **do**
3: $\quad A[i] \leftarrow A[i\ div\ 2]$
4: $\quad i \leftarrow i\ div\ 2$
5: **end while**
6: $A[i] \leftarrow x$

- dimensiunea heap-ului crește de la n elemente la $n + 1$ (evident, dar trebuie prevăzut spațiu unde heap-ul are loc să crească)
- timpul este în $O(log\ n)$
- așadar, pe un heap se poate efectua orice operație într-un timp logaritmic

Insert-heap. Exemplu



Structuri de mulțimi disjuncte

Proprietăți:

- submulțimi care nu au elemente comune
- considerăm că avem N elemente și o partiție S_1, S_2, ..., două câte două disjuncte
- trebuie să se poată realiza următoarele operații:
  - reuniunea a două submulțimi, $S_i \cup S_j$
  - găsirea submulțimii care conține un element dat
- se alege într-un prim pas, ca etichetă a unei mulțimi, elementul său minim
- se creează tabloul $set[1 \ldots N]$, ce păstrează etichetele mulțimilor din care fac parte elementele $1 \ldots n$
- avem proprietatea $set[i] \leq i$, $1 \leq i \leq N$

Varianta nr. 1 I

function find1 (x)

1: **return** $set[x]$

procedure merge1 (a, b)

1: $i \leftarrow a$
2: $j \leftarrow b$
3: **if** $i > j$ **then**
4: $\quad i \leftrightarrow j$
5: **end if**
6: **for** $k \leftarrow j$ to N **do**
7: $\quad$ **if** $set[k] = j$ **then**
8: $\quad\quad set[k] \leftarrow i$
9: $\quad$ **end if**
10: **end for**

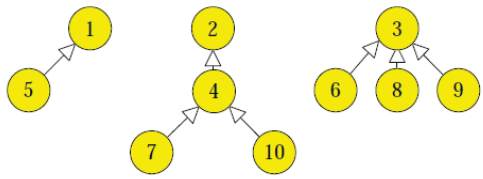
Varianta nr. 1 II

1	2	3	2	1	3	4	3	3	4
set[1]	set[2]	set[3]	set[4]	set[5]	set[6]	set[7]	set[8]	set[9]	set[10]

- avem o serie de n operații merge1() și find1()
- cazul cel mai nefavorabil apare dacă o mulțime de un element crește cu câte un element la fiecare pas
- secvența este $merge1(N, N-1)$, $merge1(N-1, N-2)$, ..., $merge1(N-n+1, N-n)$
- timpul va fi în $O(n^2)$

Varianta nr. 2 I

1	2	3	2	1	3	4	3	3	4
set[1]	set[2]	set[3]	set[4]	set[5]	set[6]	set[7]	set[8]	set[9]	set[10]



- reprezentăm mulțimea ca arbore 'inversat'
- dacă $set[i] = i$, atunci i este eticheta mulțimii, adică rădăcina arborelui
- dacă $set[i] \neq i$, atunci $set[i]$ este părintele lui i în arbore

Varianta nr. 2 II

function find2(x)

1: $i \leftarrow x$
2: **while** $set[i] \neq i$ **do**
3: $\quad i \leftarrow set[i]$
4: **end while**
5: **return** i

procedure merge2(x)

1: **if** $a < b$ **then**
2: $\quad set[b] \leftarrow a$
3: **else**
4: $\quad set[a] \leftarrow b$
5: **end if**

- reuniunea se face mai rapid, dar s-a pierdut din performanța find-ului
- pentru cazul cel mai nefavorabil (care?), pentru n operații find2() și merge2() timpul este tot în $O(n^2)$

Varianta nr. 3 I

- fuzionarea arborilor ar fi bine să se facă astfel încât arborele de înălțime mai mică să devină fiu al celeilalte rădăcini
- la fuzionarea a doi arbori de înălțime h_1 și h_2 înălțimea arborelui rezultat este $max(h_1, h_2)$ dacă $h_1 \neq h_2$ sau $h_1 + 1$ dacă $h_1 = h_2$
- prin această regulă, un arbore cu k vârfuri va avea înălțimea cel mult $\lfloor log\ k \rfloor$
- convenim să stocăm înălțimea subarborilor în $H[1..N]$
- inițial, $H[i] = 0$, $i = 1 \ldots N$
- se modifică algoritmul de fuzionare

Varianta nr. 3 II

procedure merge3(a, b)

```
1:  if H[a] = H[b] then
2:     H[a] ← H[a] + 1
3:     set[b] ← a
4:  else
5:     if H[a] > H[b] then
6:        set[b] ← a
7:     else
8:        set[a] ← b
9:     end if
10: end if
```

- o serie de n operații find2() și merge3() necesită, în cazul cel mai nefavorabil, un timp $O(n\ log\ n)$
- drumul se poate comprima și mai mult dacă, atunci când se face un find(), se 'agață' toate vârfurile parcurse direct de rădăcina căutată

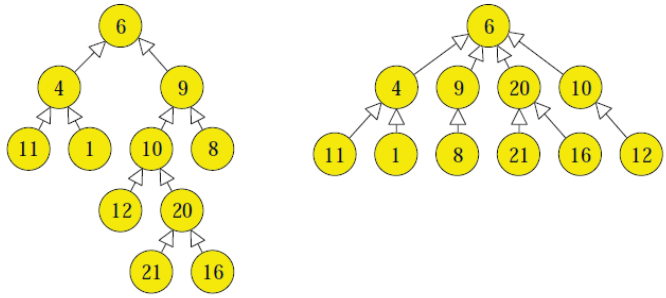
Varianta nr. 3 III

function find3(x)

```
1:  r ← x
2:  while set[r] ≠ r do
3:      r ← set[r]
4:  end while
5:  i ← x
6:  while i ≠ r do
7:      j ← set[i]
8:      set[i] ← r
9:      i ← j
10: end while
11: return r
```

- $H[a]$ va da în acest caz o limită superioară a înălțimii arborelui
- procedura merge3() rămâne valabilă

Varianta nr. 3 IV



- (a) arborele înaintea comprimării drumului
- (b) același arbore după operația find(20)

Ordinul de timp pentru find3() și merge3()

- o serie de n operații find3() și merge3() necesită, în cazul cel mai nefavorabil, un timp în $O(n\ log^*\ n)$
- logaritmul stelat este definit ca:

$$log^*\ N\ = min\{k|\ \underbrace{log\ log\ \ldots\ log}_{de\ k\ ori}\ N \leq 0\}$$

- $log^*\ N$ crește încet, $log^*\ N \leq 5$ pentru orice $N \leq 65536$
- $log^*\ N \leq 6$ pentru orice $N \leq 2^{65536}$
- practic timpul este liniar

Funcția lui Ackermann I

- funcție cu creștere foarte rapidă, este inversa logaritmului stelat

$$A_k(j) = \begin{cases} j+1, & k=0 \\ A_{k-1}^{(j+1)}(j), & k \geq 1 \end{cases}$$

- unde $A^{(i)}(j)$ este aplicarea de i ori a funcției $A(j)$ peste rezultatul aplicării ei anterioare, adică

$$\begin{aligned} A^{(i)}(j) &= A^{(i-1)}(A(j)) = A^{(i-2)}(A(A(j))) \\ &= \cdots = \underbrace{A(A(\ldots A(j)\ldots))}_{de\ i\ ori} \end{aligned}$$

- atunci $A_k^{(1)}(j) = A_k(A_k^{(0)})(j) = A_k(j)$,
- iar $A_0^{(0)}(j) = j$ este funcția identică

Funcția lui Ackermann II

- calculul primilor termeni:

$$\begin{aligned} A_0^{(i)}(j) &= A_0(A_0^{(i-1)}(j)) = A_0^{(i-1)}(j) + 1 = A_0(A_0^{(i-2)}(j)) + 1 \\ &= A_0^{(i-2)}(j) + 2 = \cdots = A_0^{(1)}(j) + i - 1 = i + j \end{aligned}$$

$$\begin{aligned} A_1(j) &= A_0^{(j+1)}(j) = 2j + 1 \\ A_2(j) &= A_1^{(j+1)}(j) = A_1(A_1^{(j)}(j)) = 2 \cdot (A_1^{(j)}(j)) + 1 \\ &= 2 \cdot A_1(A_1^{(j-1)}(j)) + 1 = 2 \cdot (2 \cdot A_1^{(j-1)}(j) + 1) + 1 \\ &= 2^2 \cdot A_1^{(j-1)}(j) + 2 + 1 = \dots \\ &= 2^{j+1} \cdot A_1^{(0)}(j) + 2^j + 2^{j-1} + \cdots + 2 + 1 \\ &= 2^{j+1}(j+1) - 1 \\ A_3(j) &= A_2^{(j+1)}(j) = A_2(A_2^{(j)}(j)) = 2^{A_2^{(j)}(j)+1} \cdot (A_2^{(j)}(j) + 1) - 1 \end{aligned}$$

Funcția lui Ackermann III

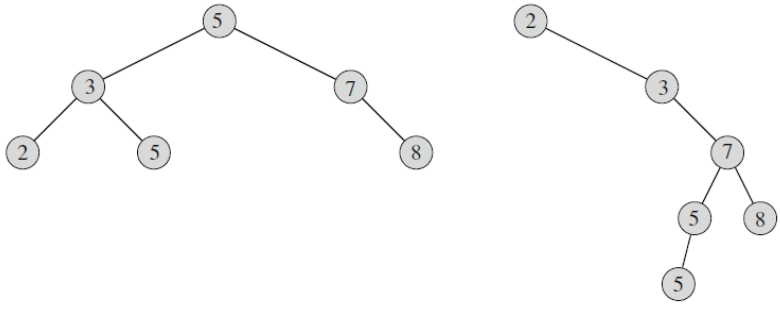
- folosind calculele anterioare:

$$\begin{aligned} A_0(1) &= 1 + 1 = 2 \\ A_1(1) &= 2 \cdot 1 + 1 = 3 \\ A_2(1) &= 2^{1+1} \cdot (1+1) - 1 = 7 \\ A_3(1) &= A_2^{(2)}(1) = A_2(A_2^{(1)}(1)) = A_2(7) = 2^8 \cdot 8 - 1 = 2^{11} - 1 = 2047 \\ A_4(1) &= A_3^{(2)}(1) = A_3(A_3(1)) = A_3(2047) = A_2^{(2048)}(2047) \\ &= A_2(A_2^{(2047)}(2047)) = A_2(A_2(\dots A_2^{(1)}(2047)\dots)) \\ &\gg A_2(2047) = 2^{2048} \cdot 2048 - 1 = 2^{2059} - 1 > 2^{2056} \\ &= (2^4)^{514} = 16^{514} > 10^{514} \gg 10^{80} \end{aligned}$$

- unde 10^{80} este numărul estimat al atomilor din universul observabil

Arbori binari de căutare

- fiecare nod poate regăsi, pe lângă cheia proprie, nodurile părinte, fiu stânga și fiu dreapta
- **proprietatea arborelui binar de căutare:**
  - dacă y este un nod din subarborele stâng al lui x, atunci $y.key \leq x.key$
  - dacă y este un nod din subarborele drept al lui x, atunci $x.key \leq y.key$



(a) (b)

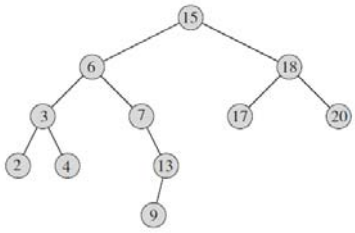
Parcurgerea arborelui

procedure inorder-tree-walk (x)

```
1: if x ≠ NIL then
2:     inorder-tree-walk(x.left)
3:     print x.key
4:     inorder-tree-walk(x.right)
5: end if
```

- algoritmul se lansează cu un apel inorder-tree-walk(root)
- proprietatea arborelui binar de căutare permite afișarea tuturor cheilor:
  - crescător, dacă arborele este parcurs în inordine;
  - descrescător, dacă la parcurgerea în inordine inversăm poziția parcurgerii fiului drept cu cea a parcurgerii fiului stâng
- parcurgrea în preordine tipărește cheia rădăcinii înaintea cheilor fiilor
- similar, parcurgerea în postordine va tipări cheia rădăcinii după cheile din subarbori (exemplu? ordin de timp?)

Interogarea într-un arbore binar de căutare



- tipuri de interogări:
  - căutarea unui nod anume
  - minim
  - maxim
  - predecesor
  - succesor
- fiecare se poate realiza în $O(h)$ pe un arbore de înălțime h
- în cazul cel mai nefavorabil, arborele fiind nebalansat, cât poate fi înălțimea lui?

Căutarea

function tree-search (x, k)
1: **if** $x = NIL$ or $x.key = k$ **then**
2: **return** x
3: **end if**
4: **if** $k < x.key$ **then**
5: **return** tree-search(x.left, k)
6: **else**
7: **return** tree-search(x.right, k)
8: **end if**

function iterative-tree-search(x, k)
1: **while** $x \neq NIL$ and $k \neq x.key$ **do**
2: **if** $k < x.key$ **then**
3: $x \leftarrow x.left$
4: **else**
5: $x \leftarrow x.right$
6: **end if**
7: **end while**
8: **return** x

Minimul și maximul

function tree-minimum (x)

1: **while** $x.left \neq NIL$ **do**
2: $x \leftarrow x.left$
3: **end while**
4: **return** x

- care este drumul parcurs în fiecare din cele două cazuri?

function tree-maximum (x)

1: **while** $x.right \neq NIL$ **do**
2: $x \leftarrow x.right$
3: **end while**
4: **return** x

Succesorul și predecesorul unui nod

function tree-successor (x)

1: **if** $x.right \neq NIL$ **then**
2: **return** tree-minimum($x.right$)
3: **end if**
4: $y \leftarrow x.parent$
5: **while** $y \neq NIL$ and $x = y.right$ **do**
6: $x \leftarrow y$
7: $y \leftarrow y.parent$
8: **end while**
9: **returns** y

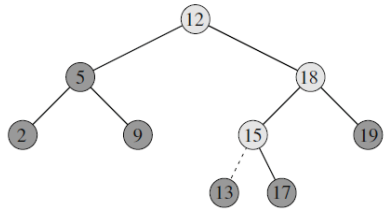
- succesorul unui nod x este nodul avînd cea mai mică cheie mai mare decât cheia lui x
- dacă subarborele drept al nodului x este vid, atunci succesorul lui x va fi cel mai de jos nod strămoș al său al cărui fiu stâng e de asemenea strămoș pentru x (exemplu, nodul 13)

Inserarea și ștergerea I

procedure tree-insert(T, z)

```
 1: y ← NIL
 2: x ← T.root
 3: while x ≠ NIL do
 4:     y ← x
 5:     if z.key < x.key then
 6:         x ← x.left
 7:     else
 8:         x ← x.right
 9:     end if
10: end while
11: z.parent ← y
12: if y = NIL then
13:     T.root ← z
14: else if z.key < y.key then
15:     y.left ← z
16: else
17:     y.right ← z
18: end if
```

Inserarea și ștergerea II



- plecînd de la rădăcină, se caută un nod în jos
- ultimul nod nenul va fi păstrat în y
- când nodul x devine nul, el indică poziția unde trebuie plasat noul nod
- exemplu, nodul 13
- timp în ordinul înălțimii arborelui

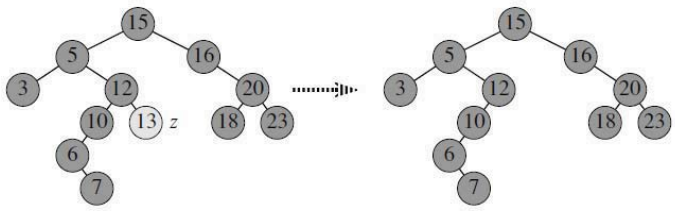
procedure tree-remove(T, z)

```
1: if z.left = NIL or z.right = NIL then
2:     y ← z
3: else
```

Inserarea și ștergerea III

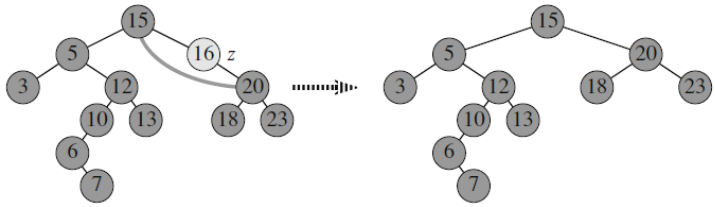
```
4:     y ← tree − successor(z)
5:  end if
6:  if y.left ≠ NIL then
7:     x ← y.left
8:  else
9:     x ← y.right
10: end if
11: if x ≠ NIL then
12:    x.parent ← y.parent
13: end if
14: if y.parent = NIL then
15:    T.root ← x
16: else if y = y.parent.left then
17:    y.parent.left ← x
18: else
19:    y.parent.right ← x
20: end if
21: if y ≠ z then
22:    z.key ← y.key
23: end if
24: return y
```

- se determină mai întâi nodul y care se șterge; fie nodul z fie succesorul lui z, dacă acesta are doi fii
- în x se determină fiul care se deplasează, fie NIL dacă z nu are fii
- se elimină din arbore nodul y, prin modificarea legăturilor
- se realizează mutarea datelor în cazul eliminării succesorului
- se întoarce ca rezultat nodul șters



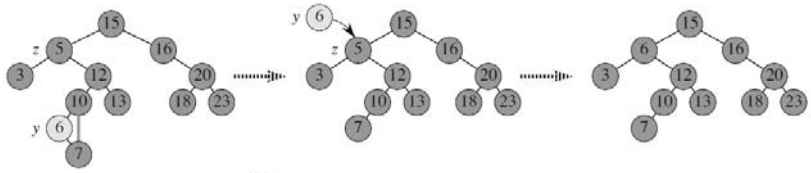
(a)

- dacă z nu are fii, se va modifica părintele său pentru a-i înlocui fiul cu NIL



(b)

- dacă z are un singur fiu, atunci se elimină prin modificarea părintelui său, astfel încât acea legătură nu mai indică spre z ci spre fiul lui z



(c)

- dacă z are doi fii, se caută succesorul y al lui z
- acest succesor y nu are fiu stâng, pentru că, dacă ar avea, acesta ar avea cheia mai mică sau egală cu cheia lui y
- asta ar însemna că acela de fapt ar fi succesorul lui z
- se înlocuiește informația lui z cu informația lui y astfel găsit
- y se șterge din vechiul său loc, prin refacerea legăturii părintelui său

Parcurgerea arborilor în adâncime I

procedure ad (v)

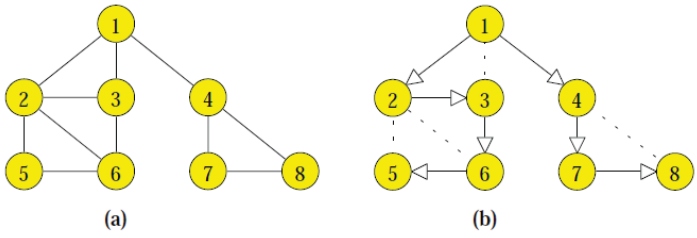
```
1: marca[v] ← vizitat
2: for fiecare vârf w adiacent lui v do
3:     if marca[w] = nevizitat then
4:         ad(v)
5:     end if
6: end for
```

procedure parcurge(G)

```
1: for fiecare v ∈ V do
2:     marca[v] ← nevizitat
3: end for
4: for fiecare v ∈ V do
5:     if marca[v] = nevizitat then
6:         ad(v)
7:     end if
8: end for
```

Parcurgerea arborilor în adâncime II

- se alege un vârf ca punct de plecare, se marchează
- dacă unul din vecinii săi nu e marcat, inițiem apeluri pornind de la acesta, recursiv
- doar la revenirea din recursie se parcurg restul vârfurilor vecine
- se încearcă astfel inițierea a cât mai multe apeluri înainte de a se reveni din apel
- tehnica explorării unui labirint



Parcurgerea arborilor în adâncime III

- presupunem un graf cu m muchii și n vârfuri
- fiecare vârf e vizitat o dată, deci n apeluri ale procedurii ad() cu el ca argument
- din vârf se testează toți vecinii
- numărul de verificări este m pentru un graf orientat sau $2m$ pentru unul neorientat
- timpul de execuție este în $\Theta(max(m, n)) = \Theta(m + n)$
- dacă graful e reprezentat prin listă de adiacență, inspectarea vecinilor unui nod se face în $\Theta(n)$, deci timpul total va fi $\Theta(n^2)$
- parcurgerea arborelui generează un arbore / pădure de arbori

Parcurgerea grafurilor în lățime (level order) I

procedure lat(v)

```
1:  C ← coadă vidă
2:  marca[v] ← vizitat
3:  insert-queue(C, v)
4:  while C nu este vidă do
5:     u ← delete-queue(C)
6:     for fiecare vârf w adiacent lui u do
7:        if marca[w] = nevizitat then
8:           marca[w] ← vizitat
9:           insert-queue(C, w)
10:       end if
11:    end for
12: end while
```

- mai întâi se parcurg vârfurile adiacente de adîncime 0 (vecinii)
- se folosește o coadă pentru a parcurge mai târziu vârfurile adiacente de adâncime 1

Parcurgerea grafurilor în lățime (level order) II

- procedura lat() se va chema din aceeași procedură parcurge()
- ordinea de parcurgere coincide chiar cu numerotarea nodurilor
- la fel, rezultatul parcurgerii este un arbore / pădure de arbori
- același ordin de timp, $\Theta(m+n)$ sau $\Theta(n^2)$
- se folosește pentru explorarea radială, din aproape în aproape
- determinarea celui mai scurt drum între două vârfuri

Cursul nr. 2

ANALIZA EFICIENȚEI ALGORITMILOR

Cuprins

Notația asimptotică

Tehnici de analiză

Analiza algoritmilor recursivi

Teorema master

Analiza amortizată

Anexa 1: aplicație pentru analiza amortizată

Anexa 2: formule utile

Notația asimptotică: scop

- Permite caracterizarea comportamentului algoritmilor, din punct de vedere al vitezei de calcul și al memoriei ocupate
- Se folosește pentru determinarea comportamentului *asimptotic* al unui algoritm, adică pentru situații în care dimensiunea datelor este suficient de mare
- Se referă într–un mod concis la ordinul de creștere a funcțiilor
- Utilitate:
  - permite compararea a doi algoritmi ce rezolvă o aceeași problemă
  - permite determinarea eficienței și scalabilității algoritmului, a punctelor în care performanța ar trebui crescută
  - poate determina căutarea de alternative (euristici, algoritmi de aproximare etc)

Notația asimptotică - Θ

- Notația asimptotică se adresează funcțiilor care sunt definite pe $\mathbb{N} = \{0, 1, \dots\}$
- Uneori este convenabil să se extindă notația la argumente numere reale pozitive
- Notația Θ: pentru o funcție $g : \mathbb{N} \to \mathbb{R}$

$$\Theta(g) = \{f : \mathbb{N} \to \mathbb{R} | \ \exists c_1, c_2 > 0, \exists n_0 \in \mathbb{N} : \\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \ \forall n \geq n_0\} \quad (1)$$

- Notații: $f \in \Theta(g)$, $f = \Theta(g)$, $f(n) \in \Theta(g(n))$, $f(n) = \Theta(g(n))$
- Notația cere ca funcțiile f și g să fie asimptotic nenegative, adică de la un prag în sus valorile lor să fie ≥ 0

Notația asimptotică - Θ

- $\Theta(g)$ se mai numește și *ordinul exact* al lui g
- Exemplu: $5n^3 - 2n^2 \in \Theta(n^3)$, asta presupunând a găsi constantele reale $c_1, c_2 > 0$ și pragul n_0 pentru care definiția (1) este satisfăcută (la tablă)
- Se poate arăta că $5n^3 - 2n^2 \notin \Theta(n^2)$ și $5n^3 - 2n^2 \notin \Theta(n^4)$
- Mai general, pentru o funcție polinomială $f(n) = \sum_{i=0}^{k} c_i n^i$ cu $c_k > 0$ se poate arăta că $f(n) = \Theta(n^k)$
- Pentru $k = 0$: un termen constant este în $\Theta(n^0) = \Theta(1)$ – notație simbolică folosită pentru o funcție care este constantă în raport cu variabila n

Notația asimptotică - O

- Remarcă: notația Θ dă atât minorare, cât și majorare de funcție *folosind o singură funcție, g*
- Pentru cazul în care se studiază doar majorarea funcției se folosește notația O
- Pentru $g : \mathbb{N} \to \mathbb{R}$

$$O(g) = \{f : \mathbb{N} \to \mathbb{R} | \ \exists c > 0, \exists n_0 \in \mathbb{N} : \\ 0 \leq f(n) \leq cg(n) \ \forall n \geq n_0\} \qquad (2)$$

- $c \cdot g$ este un majorant începând de la un prag
- Evident, dacă $f \in \Theta(g)$ atunci $f \in O(g)$ și deci $\Theta(g) \subset O(g)$
- Notația O definește un majorant pentru o funcție, dar fără a exista obligația ca acest majorant să fie foarte apropiat de funcția considerată
- $5n^3 - 2n^2 \in O(n^3)$ dar avem și că $5n^3 - 2n^2 \in O(n^4)$

Notația asimptotică - Ω

- Pentru cazul în care se dorește doar studierea minorării unei funcții se folosește notația Ω
- Pentru $g : \mathbb{N} \to \mathbb{R}$

$$\Omega(g) = \{f : \mathbb{N} \to \mathbb{R} | \ \exists c > 0, \exists n_0 \in \mathbb{N} : \\ 0 \leq cg(n) \leq f(n) \ \forall n \geq n_0\} \tag{3}$$

- Evident, dacă $f \in \Theta(g)$ atunci $f \in \Omega(g)$ deci $\Theta(g) \subset \Omega(g)$
- $5n^3 - 2n^2 \in \Omega(n^3)$ dar avem și că $5n^3 - 2n^2 \in \Omega(n^2)$
  - ca si in cazul lui O, notația Ω nu obligă la o minorare foarte strânsă

Proprietăți ale notațiilor asimptotice

Propoziții

Pentru orice două funcții f și g avem că $f \in \Theta(g)$ dacă și numai dacă $f \in O(g)$ și $f \in \Omega(g)$.
$\Theta(g) = O(g) \cap \Omega(g)$
Reflexivitate: $f \in X(f)$, unde X este Θ, O sau Ω
Tranzitivitate: dacă $f \in X(g)$ și $g \in X(h)$ atunci $f \in X(h)$, X ca mai sus
Simetrie: $f \in \Theta(g)$ dacă și numai dacă $g \in \Theta(f)$
Simetrie transpusă: $f \in O(g)$ dacă și numai dacă $g \in \Omega(f)$.

- Putem defini ierarhii de funcții:

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n\sqrt{n}) \subset O(n^2) \subset \\ \subset O(n^3) \subset O(2^n) \subset O(n!) \subset \dots$$

Notația asimptotică

- Se poate ca timpul de execuție al unui algoritm să depindă de mai mulți parametri
- În acest caz, notațiile asimptotice se extind natural, *e.g.* pentru $g : \mathbb{N} \times \mathbb{N} \to \mathbb{R}$:

$$O(g) = \{f : \mathbb{N} \times \mathbb{N} \to \mathbb{R} | \ \exists c > 0, \exists m_0, n_0 \in \mathbb{N} : \\ 0 \leq f(m,n) \leq c \cdot g(m,n) \ \forall m \geq m_0, n \geq n_0\} \quad (4)$$

și analog pentru Ω, Θ.

Notația asimptotică condiționată

- Uneori se pot considera cazuri particulare de funcții, ce satisfac anumite condiții convenabil alese (predicate)
- Se poate folosi *notația asimptotică condiționată*:
  - Considerăm $f : \mathbb{N} \to \mathbb{R}_+$ o funcție arbitrară și $P : \mathbb{N} \to \{false, true\}$ un predicat[1]
  - Definim notația asimptotică condiționată $O(\cdot|\cdot)$:

$$O(f|P) = \{t : \mathbb{N} \to \mathbb{R}_+ | \exists c > 0 \ \exists n_0 \in \mathbb{N} \ \forall n \geq n_0 \\ [P(n) \Rightarrow t(n) \leq cf(n)]\} \quad (5)$$

- Similar se definesc notațiile pentru Ω, Θ

[1] $R_+ = [0, \infty]$

Notația asimptotică condiționată

- O funcție $f : \mathbb{N} \to \mathbb{R}_+$ se numește *asimptotic nedescrescătoare* dacă
$$(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)\ f(n) \leq f(n+1)$$
- Prin inducție se poate arăta că o astfel de funcție are proprietatea:
$$f(n) \leq f(m)\ \forall n \leq m, n, m \geq n_0$$

Definiție

Pentru o funcție asimptotic nedescrescătoare, pentru un întreg $b \geq 2$ spunem că funcția este b-netedă dacă $f(bn) \in O(f(n))$

- Este n^3 2–netedă? este 2^n o funcție 3-netedă? este $n!$ 4-netedă?
- Se poate arăta că orice funcție care este b-netedă pentru un $b \geq 2$ este de asemenea b-netedă pentru orice întreg $b \geq 2$; vom numi o funcție "b netedă" mai simplu: "netedă"

Notația asimptotică condiționată

Propoziție

Fie $b \geq 2$ un întreg oarecare, $f : \mathbb{N} \to \mathbb{R}_+$ o funcție netedă și $t : \mathbb{N} \to \mathbb{R}_+$ o funcție asimptotic nedescrescătoare, astfel încât

$$t(n) \in X(f(n) | n \text{ este o putere a lui } b) \tag{6}$$

unde X poate fi O, Ω, Θ. Atunci $t \in X(f)$. Suplimentar, dacă $t \in \Theta(f)$, atunci și funcția t este netedă.

Propoziție

Fie $T : \mathbb{N} \to \mathbb{R}_+$ o funcție asimptotic nedescrescătoare cu $T(n) = aT(n/b) + cn^k$ unde $n > n_0 \geq 1$, $b \geq 2$, $k \geq 0$ întregi, a și c numere reale pozitive iar n/n_0 este o putere a lui b. Atunci avem:

$$T(n) \in \begin{cases} \Theta(n^k) & \text{pentru } a < b^k \\ \Theta(n^k \log n) & \text{pentru } a = b^k \\ \Theta(n^{\log_b a}) & \text{pentru } a > b^k \end{cases} \tag{7}$$

Scopul analizei

- Analiza se face pentru fiecare algoritm în mod specific
- Scopul analizei: obținerea complexității în timp a unui algoritm
- Analiza poate fi făcută și pentru spațiul de memorie necesar rulării algoritmului
- Rezultatul analizei: suport decizional pentru alegerea unui algoritm dintr-o mulțime de algoritmi care rezolvă o aceeași problemă
- Analiza se face pentru cazuri asimptotice = pentru dimensiune a datelor de intrare dincolo de un anumit prag

Tehnici de analiză - sortarea prin selecție

```
Sortare-prin-selectie(T[1...n])
1  for i=1 to n − 1 do
2      minj ← i; minx ← T[i]
3      for j ← i + 1 to n do
4          if T[j] < minx then
5              minj ← j; minx ← T[j]
6      T[minj] ← T[i]; T[i] ← minx
```

Analiza:

- Testul de la linia 4 și eventualele atribuiri de la linia 5 au un cost majorat de o constantă a, independentă de datele de intrare
- Pentru ciclul de la linia 3, costul este dat de: numărul de iterații executate $(n - i)$ înmulțit cu (costul liniilor 4 și 5 + cost datorat inițializării ciclului, b); în total, pașii 3-5 au cost $b + (n - i)a'$, unde $a' = a +$ timpul de incrementare a contorului $j +$ timpul necesar pentru efectuarea testului de terminare a ciclului
- Linia 2 are un cost constant c, independent de datele de intrare
- Liniile 1-5 au costul $\sum_{i=1}^{n}(c' + b + (n - i)a')$ unde $c' = c+$ costul atribuirilor din linia 6 + timpul de incrementare a contorului $i +$ timpul necesar pentru efectuarea testului de terminare a ciclului

Tehnici de analiză - sortarea prin selecție

- Inițializarea ciclului din linia 1 are un cost d; în total, costul este
$$T(n) = d + \sum_{i=1}^{n}(c' + b + (n-i)a') \qquad (8)$$
- Ecuația (8) se reduce la $T(n) = \frac{a'}{2} \cdot n^2 + (b + c' - \frac{a}{2})n + (d - c' - b)$ deci complexitatea algoritmului este $\Theta(n^2)$ (vezi discuția despre complexitatea funcțiilor polinomiale, pagina 5)
- Ordinea elementelor nu influențează complexitatea algoritmului; nu există un caz cel mai (ne)favorabil
- Se admite ca analiza să nu fie atât de detaliată: pentru cazul de mai sus, instrucțiunea de test de la linia 4 poate fi considerată drept "barometru"
- Instrucțiunea barometru este executată de $n(n-1)/2$ ori pentru orice set de date de n elemente, deci complexitatea este $\Theta(n^2)$.

Tehnici de analiză - sortarea prin inserție

```
Sortare-prin-insertie(T[1...n])
1  for i=2 to n do
2      x ← T[i]; j ← i − 1
3      while j > 0 and x < T[j]
4          T[j + 1] ← T[j]
5          j ← j − 1
6      T[j + 1] ← x
```

- Element folosit ca barometru: comparația $x < T[j]$
- Pentru un i fixat:
  - cazul cel mai nefavorabil este când $x < T[j]$ pentru $j = i-1, i-2, \ldots 1$
  - număr de comparații efectuate în acest caz: $i - 1$
  - cazul cel mai nefavorabil corespunde șirului inițial ordonat descrescător
- Numărul total de comparații pentru cazul cel mai nevaforabil:
$$\sum_{i=1}^{n}(i-1) = \frac{n(n-1)}{2} \in \Theta(n^2)$$

Tehnici de analiză - sortarea prin inserţie

Estimarea timpului mediu:

- Presupunem că elementele lui T sunt distincte şi că toate permutările sunt egal probabile
- Probabilitatea ca $x \geq T[i-1]$ este $1/i$, deci cu probabilitatea $1/i$ se executa exact o singura data comparatia "$x < T[j]$" din `while`;
- Probabilitatea ca $T[i-2] \leq x < T[i-1]$ este $1/i$, deci cu probabilitatea $1/i$ se executa exact de doua ori comparatia "$x < T[j]$" din `while`, etc;
- Probabilitatea ca să se execute de exact $i-1$ ori ciclul `while` este $2/i$:
  - o dată pentru cazul $T[1] \leq x < T[2]$
  - încă o data pentru cazul $x < T[1]$

Tehnici de analiză - sortarea prin inserţie

Estimarea timpului mediu (continuare):

- Numărul *mediu* de execuţii al ciclului `while` pentru un $i \geq 2$ oarecare este:

$$c_i = 1 \cdot \frac{1}{i} + 2 \cdot \frac{1}{i} + \cdots + (i-2) \cdot \frac{1}{i} + (i-1) \cdot \frac{2}{i} = \frac{i+1}{2} - \frac{1}{i}$$

- Numărul total *mediu* de execuţii este:

$$\sum_{i=2}^{n} c_i = \sum_{i=2}^{n} \left(\frac{i+1}{2} - \frac{1}{i} \right) = \frac{n^2+3n}{4} - H_n$$

  unde H_n este suma parţială de ordinul n a seriei armonice:

$$H_n = \sum_{k=1}^{n} \frac{1}{k} \in \Theta(\log n) \text{ — vezi pagina 84, ecuaţia (32)}$$

- Complexitatea *medie* este $\Theta(n^2)$

Tehnici de analiză - heapsort

```
Heapsort(T[1...n])
1   Make-heap(T)
2   for i ← n downto 1
3       T[1] ↔ T[i]
4       Sift-down(T[1...i-1], 1)
```

```
Make-heap(T[1...n])
1   {Formează din T un heap}
2   for i ← ⌊n/2⌋ downto 1
3       Sift-down(T, i)
```

```
Sift-down(T[1...n], i)
1   k ← i
2   repeat
3       j ← k
4       if 2j ≤ n and T[2j] > T[k]
5         then k ← 2j
6       if 2j + 1 ≤ n  and T[2j + 1] > T[k]
7         then k ← 2j + 1
8       T[i] ↔ T[k]
9   until j = k
```

- Barometru: instrucțiunile din interiorul ciclului **repeat**
- Fie m numărul maxim de repetări al buclei **repeat**, pentru apelul `Sift-down(T[1...n], i)`.
- Fie j_t valoarea care se atribuie lui j în linia 3, la a t-a repetare a buclei
  - evident, $j_1 = i$
  - pentru $1 < t \leq m$, la sfarșitul celei de a $(t-1)$-a bucle avem că $j \neq k$ și $k \geq 2j = 2j_{t-1}$
  - cum $j_t = k$ avem că $j_t \geq 2j_{t-1}$ pentru $1 < t \leq m$

Tehnici de analiză - heapsort

- Rezultă:

$$n \geq j_m \geq 2j_{m-1} \geq 4j_{m-2} \geq \cdots \geq 2^{m-1}j_1 = 2^{m-1}i \Rightarrow m \leq 1+\log_2 \frac{n}{i} \tag{9}$$

- Numărul total de execuții al ciclului **repeat** datorate apelurilor din procedura `Make-heap` are proprietatea

$$T(n) \leq \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \left(1 + \log_2 \frac{n}{i}\right) = \left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{n}{2} \right\rfloor \log_2 n - \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \log_2 i \tag{10}$$

- Conform Anexei, relația (35), avem:

$$\sum_{i=2}^{\lfloor \frac{n}{2} \rfloor} \log_2 i \geq \int_1^{\lfloor \frac{n}{2} \rfloor} \log_2(x)\mathrm{d}x = \frac{1}{\ln 2}\left(x \ln x - x\right)\Big|_1^{\lfloor \frac{n}{2} \rfloor} =$$

$$= \frac{\lfloor \frac{n}{2} \rfloor \ln \lfloor \frac{n}{2} \rfloor - \lfloor \frac{n}{2} \rfloor + 1}{\ln 2} = \left\lfloor \frac{n}{2} \right\rfloor \log_2 \left\lfloor \frac{n}{2} \right\rfloor - \frac{\lfloor \frac{n}{2} \rfloor - 1}{\ln 2} \tag{11}$$

Tehnici de analiză - heapsort

- Deducem că:

$$T(n) \leq 2\left\lfloor \frac{n}{2} \right\rfloor + \frac{\left\lfloor \frac{n}{2} \right\rfloor - 1}{\ln 2} \in O(n)$$

  deci procedura `Make-heap` necesită timp $T(n) \in O(n)$.
- Este evident că construirea unui heap necesită parcurgerea fiecărui element, deci $T(n) \in \Omega(n)$
- În concluzie, timpul necesar construirii heap-ului prin `Make-heap` este $\Theta(n)$
- Din inecuatia (9) și ținând cont că $i \geq 1$ rezultă că tipul necesar lui `Sift-down` este $O(\log n)$
- Timpul procedurii `Heapsort` este deci $\Theta(n) + O(n \log n) = O(n \log n)$
- Folosind un rezultat care spune că orice algoritm de sortare bazat pe comparații are complexitatea în cazul cel mai nefavorabil de $\Omega(n \log n)$, rezultă că Heapsort are complexitatea $\Theta(n \log n)$.

Exemplu de problemă recursivă

- Problemă: să se calculeze al n-lea element al șirului lui Fibonacci:

$$f(n) = \begin{cases} n & \text{pentru } n \in \{0, 1\} \\ f(n-1) + f(n-2) & \text{pentru } n \geq 2 \end{cases}$$

- Se pune întrebarea: dacă se face implementare directă a formulei, prin funcție recursivă a formulei de mai sus, este ea o implementare eficientă?

Metoda iterației

- Strategie: se execută primii pași ai iterației, intuindu–se formula; se demonstreaza apoi prin inducție matematică validitatea ei
- Strategia funcționează doar pe unele cazuri.
- Exemplu: plecăm de la o recurență mai simplă:

$$t(n) = \begin{cases} 1 & \text{pentru } n = 1 \\ 2t(n-1)+1 & \text{pentru } n > 1 \end{cases}$$

- Pentru un $n > 1$ avem:

$$t(n) = 2t(n-1)+1 = 2^2 t(n-2)+2+1 = \cdots = 2^{n-1}t(1)+\sum_{i=0}^{n-2} 2^i$$

- Prin inducție matematică se confirmă formula termenului general $t(n) = 2^n - 1$

Inducția constructivă

- Strategie: se folosește inducția matematică **chiar pentru o formulă parțial specificată**
- Rezultat: se completează părțile necunoscute ale formulei și se demonstrează și corectitudinea ei
- Exemplu:

$$f(n) = \begin{cases} 0 & \text{pentru } n = 0 \\ f(n-1)+n & \text{pentru } n > 0 \end{cases} \tag{12}$$

- Avem: $f(k) - f(k-1) = k$; însumând pentru $k = n, n-1, \ldots, 2$ obținem $f(n) = \sum_{i=1}^{n} i \leq \sum_{i=1}^{n} n = n^2$

Inducția constructivă (cont.)

- Formulăm ipoteza inducție parțial specificate $IISP(n)$: $f(n) = an^2 + bn + c$
- Presupunem $IISP$ adevărată pentru $n-1$ și din (12) obținem: $f(n) = (a(n-1)^2 + b(n-1) + c) + n = an^2 + (1 + b - 2a)n + (a - c + b)$
- Facem egalarea coeficienților cu expresia corespunzătoare lui $IISP(n)$ și rezultă: $a = b = 1/2$, c oarecare. Valoarea lui c se obține din condiția inițială $f(0) = 0$.
- Am arătat că dacă $IISP(n-1)$ este adevărată, atunci și $IISP(n)$ este adevărată și am dedus totodată forma exactă a lui $f(n)$
- Denumire alternativă a metodei: metoda substituției ("Introducere în algoritmi", Cormen *et al.*, secțiunea 4.1)

Recurențe liniare omogene

- Considerăm ecuații recurente liniare omogene:

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = 0 \tag{13}$$

  unde t_i sunt valorile căutate, a_j sunt coeficienți constanți
- Se caută soluții de forma: $t_n = x^n$, cu x constantă (deocamdata necunoscută)
- Substituind forma în (13), obținem:

$$a_0 x^n + a_1 x^{n-1} + \cdots + a_k x^{n-k} = 0 \tag{14}$$

- Excludem soluția trivială $x = 0$ și obținem **ecuația caracteristică** a recurenței (13):

$$a_0 x^k + a_1 x^{k-1} + \cdots + a_k = 0 \tag{15}$$

Recurențe liniare omogene (cont)

- Pentru cazul în care cele k rădăcini $r_1, \ldots, r_k$ ale ecuației (15) sunt distincte, termenul general corespunzător recurenței omogene inițiale (13) are forma:

$$t_n = \sum_{i=1}^{k} c_i r_i^n \tag{16}$$

- Coeficienții (constantele) $c_i, 1 \leq i \leq k$ se determină din condițiile inițiale date ale recurenței

Recurențe liniare omogene (exemplu)

- Pentru șirul lui Fibonacci: $t_n = t_{n-1} + t_{n-2},\ t_0 = 0,\ t_1 = 1$, ecuația caracteristică este:

$$x^2 - x - 1 = 0$$

- Rădăcinile sunt distincte: $r_{1,2} = \frac{1 \pm \sqrt{5}}{2}$
- Soluția generală are forma $t_n = c_1 r_1^n + c_2 r_2^n$;
- Se aplică condițiile inițiale:

$$\begin{cases} f(0) = 0,\ n = 0: & c_1 + c_2 = 0 \\ f(1) = 1,\ n = 1: & r_1 c_1 + r_2 c_2 = 1 \end{cases}$$

- Se obțin coeficienții $c_{1,2} = \pm 1/\sqrt{5}$
- Concluzie: implementarea recursivă pentru calcularea lui $f(n)$ (vezi pagina 22) are complexitate exponențială

Recurențe liniare omogene (cont)

- Cazul în care ecuația caracteristică (15) nu are rădăcini distincte se tratează astfel:
  - dacă r este o rădăcină cu grad de multiplicitate m a ecuației caracteristice (15), atunci se poate arăta că:

$$t_n = r^n, t_n = nr^n, t_n = n^2 r^n, \ldots, t_n = n^{m-1} r^n$$

    sunt soluții ale ecuație caracteristice.
  - soluția generală este o combinație liniară a termenilor de această formă și a celorlalte rădăcini distincte ale ecuației caracteristice
- Exemplu: considerăm recurența $t_n = 5t_{n-1} - 8t_{n-2} + 4t_{n-3}$, $n \geq 3$ cu condițiile inițiale: $t_0 = 0$, $t_1 = 1$, $t_2 = 2$.
  - ecuația caracteristică are rădăcinile 1 (de multiplicitate 1) și 2 (de multiplicitate 2).
  - soluția generală este: $t_n = c_1 1^n + c_2 2^n + c_3 n 2^n$
  - din condițiile inițiale obținem $c_1 = -2$, $c_2 = 2$, $c_3 = -1/2$

Recurențe liniare neomogene

- Considerăm ecuațiile recurente de forma:

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = b^n p(n) \qquad (17)$$

  unde b este o constantă iar p este polinom în n de gradul d
- Strategie de lucru: prin manipulări se aduce (17) la o recurență omogenă
- Se ajunge la **ecuația caracteristică**:

$$(a_0 x^k + a_1 x^{k-1} + \cdots + a_k)(x - b)^{d+1} = 0 \qquad (18)$$

  care se tratează ca în cazul omogen

Recurențe liniare neomogene (cont)

Exemplificare / motivație pentru ecuația (18):

$$t_n - 2t_{n-1} = 3^n \tag{19}$$

- $b = 3$, $p(n) = 1$, $d = 0$
- înmulțim (19) cu 3:

$$3t_n - 6t_{n-1} = 3^{n+1} \tag{20}$$

- rescriem (19) pentru $n \to n + 1$

$$t_{n+1} - 2t_n = 3^{n+1} \tag{21}$$

- scădem ultimele două ecuații și obținem: $t_{n+1} - 5t_n + 6t_{n-1} = 0$

Schimbare de variabilă

- Uneori, folosind o schimbare de variabilă se poate ajunge la o expresie mai simplă
- Exemplu: pentru $T(n) = 4T(n/2) + n,\ n > 1$
- Considerăm valorile lui n care sunt puteri ale lui 2: $n = 2^k$, notăm $t_k = T(2^k) = T(n)$
- Se obține: $t_k = 4t_{k-1} + 2^k$ având ecuația caracteristică $(x - 4)(x - 2) = 0$
- Prin tehnicile explicate anterior se ajunge la forma: $t_k = c_1 4^k + c_2 2^k$
- Făcând schimbarea de variabilă $k = \log_2 n$ obținem $T(n) = c_1 n^2 + c_2 n$, deci:

$$T(n) \in O(n^2 | n \text{ este o putere a lui } 2)$$

Adăugând condiția ca $T(n)$ să fie asimptotic nedescrescătoare și ținând cont că funcția n^2 este netedă, pe baza propoziției de la pagina 12 se obține că $T(n) \in O(n^2)$.

- Alternativă: se folosește teorema master

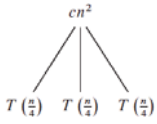
Arborele de recurență

- Metodă utilă pentru abordarea recurențelor
- Servește ca punct de plecare pentru inducția constructivă sau se poate folosi pentru funcții netede nedescrescătoare
- Exemplu: care este complexitatea funcției T dată de

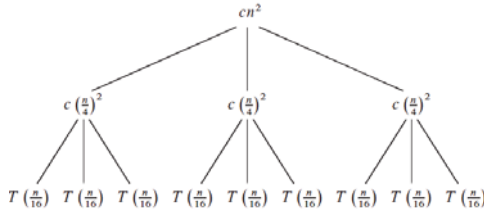
$$T(n) = 3T(\lfloor n/4 \rfloor) + cn^2, c > 0$$

- Pentru început se poate presupune că n este o putere a lui 4
- Strategie: se creează un arbore de recurență ce permite aflarea costului computațional
- Arborele este construit pe nivele succesive până când costul nodurilor la care se ajunge este cunoscut

Arborele de recurență

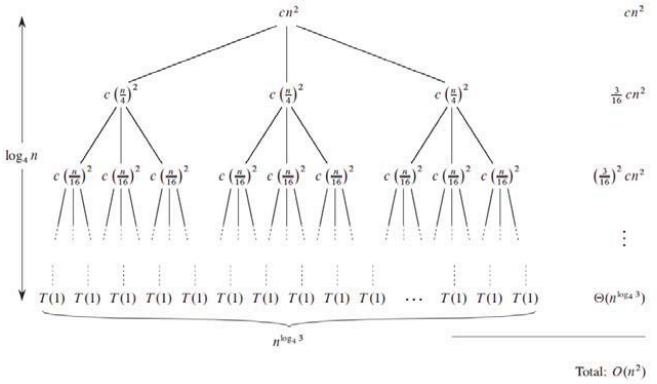


Figură: Costul lui $T(n)$ este dat de costul celor trei noduri copil corespunzătoare lui $T(n/4)$ plus termenul cn^2



Figură: Fiecare din copiii din arborele anterior este expandat și are costul dat de copiii lui plus $c \cdot \left(\frac{n}{4}\right)^2$

Arborele de recurență



Figură: Se continuă cu expandarea fiecărui nod până când se ajunge la noduri pentru care valoarea este cunoscută. În dreptul fiecărui nivel este costul care se adaugă la cel dat de nodurile copil din nivelul imediat inferior.

Arborele de recurență

- Generarea de nivele în arbore continuă până când costul nodurilor se cunoaște
- În exemplul considerat: adâncimea unui astfel de nod este acel k pentru care $n/4^k = 1 \Leftrightarrow k = \log_4 n$
- $T(n)$ se compune din termenii scriși pe marginea arborelui de recurență plus costul tuturor frunzelor din arborele de recurență

$$\begin{aligned} T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \\ &+ \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}) \\ &= cn^2 \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i + \Theta(n^{\log_4 3}) \\ &< cn^2 \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i + \Theta(n^{\log_4 3}) = \end{aligned} \tag{22}$$

Arborele de recurență

$$\begin{aligned} T(n) &= \frac{16}{13}cn^2 + \Theta(n^{\log_4 3}) = \\ &= O(n^2) \end{aligned} \tag{23}$$

- $T(n) = O(n^2 | n \text{ putere a lui } 4)$
- Se poate continua în două moduri:
  - Se folosește rezultatul de mai sus ca punct de plecare pentru ipoteza $T(n) \leq a \cdot n^2$ și se demonstrează prin inducție validitatea ei;
  - Adăugând la problema inițială condiția ca $T(n)$ să fie asimptotic nedescrescătoare[2] și ținând cont că funcția n^2 este netedă, pe baza ecuației (12) se obține că $T(n) \in O(n^2)$

[2]Ipoteza este rezonabilă: dacă dimensiunea datelor crește este de așteptat ca efortul computațional să fie mai mare.

Metoda master

- Utilitate: este o "rețetă" ce se poate aplica pentru recurențe de forma:
  $$T(n) = aT(n/b) + f(n)$$
  cu constantele $a \geq 1$, $b > 1$ și f asimptotic pozitivă
- Rezultat: 3 cazuri ce se aplică în funcție de legăturile dintre a, b, f
- Deși n/b s-ar putea să nu fie număr întreg, acceptăm notația, interpretând–o fie ca $\lfloor n/b \rfloor$, fie ca $\lceil n/b \rceil$
  - intepretarea nu schimbă comportamentul asimptotic

Metoda master

Teoremă

Fie $a \geq 1$ și $b > 1$ constante, fie $f(\cdot)$ o funcție și $T(\cdot)$ definită pe întregii nenegativi prin recurența:

$$T(n) = aT(n/b) + f(n) \tag{24}$$

unde interpretăm n/b fie ca $\lfloor n/b \rfloor$, fie ca $\lceil n/b \rceil$. Atunci T poate fi delimitată asimptotic după cum urmează:

1. *Dacă $f(n) \in O(n^{\log_b a - \varepsilon})$ pentru o anumită constantă $\varepsilon > 0$ atunci $T(n) \in \Theta(n^{\log_b a})$*
2. *Dacă $f(n) \in \Theta(n^{\log_b a})$, atunci $T(n) \in \Theta(n^{\log_b a} \log n)$*
3. *Dacă $f(n) \in \Omega(n^{\log_b a + \varepsilon})$ pentru constanta $\varepsilon > 0$ și dacă $af(n/b) < cf(n)$ pentru o anumită constantă $c < 1$ și toți n dincolo de un prag, atunci $T(n) \in \Theta(f(n))$.*

Metoda master

- În fiecare caz se compară $f(n)$ cu $n^{\log_b a}$
- Cea mai mare din cele două influențează rezultatul final
- În primul caz: $f(n)$ trebuie să fie *polinomial* mai mică decât $n^{\log_b a}$: abstracție făcând de o constantă multiplicativă, $f(n)$ trebuie să fie de n^{ε} ori mai mic decât $n^{\log_b a}$, pentru un ε constant; o discutie asemanatoare este pentru al treilea caz
- Există un "gol" între cazurile 1 și 2, respectiv 2 și 3
  - Putem aplica teorema master pentru a determina complexitatea funcției: $T(n) = 2T(n/2) + n \log_2 n$?

Metoda master - schiță a demonstrației

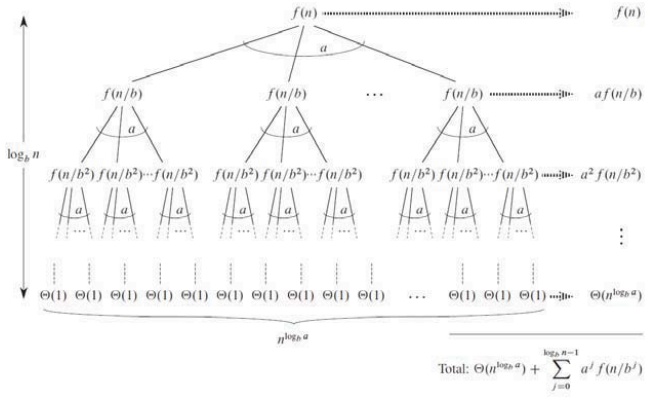
- Se demonstrează pentru valori ale lui n care sunt puteri exacte ale lui b, apoi pentru valori n intermediare
- Pentru $n = b^i$, $i = 0, 1, \ldots,$ pentru T definit ca:

$$T(n) = \begin{cases} \Theta(1) & \text{dacă } n = 1 \\ aT(n/b) + f(n) & \text{dacă } n = b^i, i \geq 1 \end{cases} \tag{25}$$

are forma generală:

$$T(n) = \Theta\left(n^{\log_b a}\right) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) \tag{26}$$

Metoda master - schiță a demonstrației (cont)



Figură: Arborele de recurență pentru ecuația (25). Costul nodurilor de la un nivel este scris în dreapta arborelui.

Metoda master - schiță a demonstrației (cont)

- Rădăcina are costul $f(n)$ și a copii, fiecare de cost $f(n/b)$
- Sunt a^j noduri la distanță j de rădăcină, fiecare cu costul $f(n/b^j)$
- Frunzele au costul $\Theta(1)$ corespunzător lui $f(n/b^{\log_b n})$; adâncimea arborelui e $\log_b n$; sunt $a^{\log_b n} = n^{\log_b a}$ frunze; totalul este cel din ec. (26)

Metoda master - schiță a demonstrației (cont)

- Se pune problema determinării complexității funcției $g(n) = \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)$ din (26)
- Se poate arăta că:

$$g(n) = \begin{cases} O(n^{\log_b a}) \text{ pt. } f(n) = O(n^{\log_b a - \varepsilon}) \text{ pentru } \varepsilon > 0, \text{ const.} \\ \Theta(n^{\log_b a} \log n) \text{ pt. } f(n) = \Theta(n^{\log_b a}) \\ \Theta(f(n)) \text{ pt. } af(n/b) \leq cf(n) \text{ cu } c \text{ const } < 1 \text{ şi } n \geq b \end{cases} \quad (27)$$

Metoda master - schiță a demonstrației (cont)

Demonstrație pentru primul caz: $f(n) = O(n^{\log_b a-\varepsilon}) \Rightarrow$
$f(n/b^j) = O((n/b^j)^{\log_b a-\varepsilon}) \Rightarrow$
$g(n) = O\left(\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a-\varepsilon}\right)$
$\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a-\varepsilon} = n^{\log_b a-\varepsilon} \sum_{j=0}^{\log_b n-1} \left(\frac{ab^\varepsilon}{b^{\log_b a}}\right)^j$
$= n^{\log_b a-\varepsilon} \sum_{j=0}^{\log_b n-1} (b^\varepsilon)^j = n^{\log_b a-\varepsilon} \left(\frac{n^\varepsilon-1}{b^\varepsilon-1}\right) = n^{\log_b a-\varepsilon} O(n^\varepsilon)$
$= O(n^{\log_b a})$.
Similar se demonstrează și celelalte două cazuri. Unind rezultatul din ecuația (27) cu expresia lui $T(n)$ din ec. (26) se obține teorema master pentru valorile lui n puteri ale lui b.
Pentru valorile lui n între două puteri succesive ale lui b se obțin mărginiri ale lui $T(n)$ folosindu-ne de $\lfloor n/b \rfloor \leq n/b \leq \lceil n/b \rceil$

Analiza amortizată: scop

- Scop: determinarea timpului necesar pentru execuția unei *secvențe* de operații
- Rezultat: se determină media costului pentru secvență și de aici costul mediu al unei operații
- A nu se confunda cu analiza cazului mediu — unde se ia în considerare probabilitatea (frecvența) operațiilor sau distribuția datelor
- Analiza amortizată dă costul mediu pe operație pentru situația (secvența) *cea mai nefavorabilă*;

Analiza amortizată: metode

1. **Metoda de agregare:** se determină marginea superioară $T(n)$ a unei secvențe de n operații; costul amortizat pentru o operație este $T(n)/n$;
2. **Metoda de cotare:** se asignează câte un cost (resursă) fiecărei operații, de așa manieră încât resursele totale alocate să nu fie depășite; măsurarea resurselor cheltuite dă costul total al secvenței;
3. **Metoda de potențial:** similar cu metoda de cotare, dar costul este asociat cu întreaga structură de date.

Analiza amortizată: enunț de problemă

Problemă: contor binar pentru incrementare

- Se operează pe un șir de k biți $A[0 \ldots k-1]$ reprezentând un contor binar
- Cifra cea mai nesemnificativă este în $A[0]$; numărul reprezentat de contor este $x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$
- Inițial contorul este setat la 0
- Se aplică procedură de incrementare

```
Incrementeaza(A)
1  i ← 0
2  while i < k AND A[i] = 1
3      A[i] = 0
4      i ← i + 1
5  if i < k
6      A[i] ← 1
```

Analiza amortizată: metoda de agregare (1)

- Ideea de bază: pentru orice secvență de n operații se calculează timpul $T(n)$
- Ca atare, timpul amortizat este $T(n)/n$ pentru fiecare operație, chiar dacă secvența conține tipuri diferite de operații
- Diferență față de celelalte două metode, pentru care costul poate să difere de la o operație la alta

Analiza amortizată: metoda de agregare (2)

Valoarea contorului	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Costul total
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15

Tabel: Evoluția contorului binar pe 8 biți, cu valori de la 0 la 8. Biții evidențiați sunt cei care vor fi afectați de următoarea aplicare a metodei `Incrementeaza`. Ultima coloană reprezinta numărul cumulat de biți care se modifică (costul real cumulat).

Analiza amortizată: metoda de agregare (3)

- Costul metodei `Incrementeaza` este liniar în numărul de biți care se modifică
- Analiză brută a performanței:
  - O execuție a metodei `Incrementeaza` are complexitatea $O(k)$, corespunzătoare cazului pentru care toți biții sunt pe 1;
  - O secvență de n operații `Incrementeaza` are complexitatea pentru cazul cel mai nefavorabil $O(nk)$
- Rezultatul este corect, dar nu suficient de exact.

Analiza amortizată: metoda de agregare (4)

- Analiza amortizată prin agregare:
  - Observăm că la un apel nu toți biții comută, deci marginea $O(k)$ este prea mare în majoritatea cazurilor
  - Conform tabelului anterior: bitul $A[0]$ comută de fiecare dată, deci pentru o secvență de n apeluri, bitul $A[0]$ comută de n ori; bitul $A[1]$ comută la fiecare două incrementări, deci pentru o secvență de n apeluri, bitul $A[1]$ comută de $\lfloor n/2 \rfloor$; etc
  - Regula generală: pentru o secvență de n apeluri ale metodei `Incrementeaza`, bitul $A[i]$ se schimbă de $\lfloor n/2^i \rfloor$ ori
  - Numărul de comutări pentru n apeluri succesive este deci numarul total de schimbari pentru cei k biți, adică

$$\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor \leq \sum_{i=0}^{k-1} \frac{n}{2^i} < n \cdot \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

Analiza amortizată: metoda de agregare (5)

- Rezultă că pentru o secvență de n operatori pentru un contor binar cu valoarea inițială 0 timpul în cazul cel mai nefavorabil este $O(n)$
- Costul amortizat al fiecărei operații din secvență este $O(n)/n = O(1)$
- **Raportarea la costul pe secvență are sens, deoarece de regulă se execută o secvență de operații și nu doar un singur pas**

Analiza amortizată: metoda de cotare (1)

- Ideea de bază: pentru fiecare din operații asignăm costuri convenabil alese
- Unele operații vor fi subcotate, altele supracotate
- Primele beneficiază de surplusul dat de supracotare
- Valoarea de cotare a unei operații = *costul amortizat*
- Apare diferența față de agregare, unde toate operațiile din secvență au același cost (calculat la final)
- La o operație, creditul supracotării este fie depozitat (daca este o operatie supractotata), fie folosit (daca este operatie subcotata)

Analiza amortizată: metoda de cotare (2)

- Cum alegem costurile care se acordă diferitelor operații?
  - costul total amortizat al unei secvențe trebuie să fie o margine superioară pentru costul real total al secvenței
  - oricare ar fi secvența de operații, nu trebuie să ajungi la valoare negativă
  - altfel zis: pentru cel mai nefavorabil caz, creditul de care mai dispui să fie cel puțin 0

Analiza amortizată: metoda de cotare (3)

Problema incrementării contorului binar

- Pentru setarea unui bit pe 1 se folosesc 2$: unul pentru setarea propriu-zisă și celălalt ca și credit
- La setarea unui bit pe 0 (de la 1) se folosește creditul de 1$ pe acel bit
- La fiecare apel al metodei `Incrementeaza`:
  - ciclul `while` in care se setează biții pe 0 beneficiază de creditul existent pentru fiecare valoare 1 care se transformă
  - instrucțiunea `if` se execută cel mult o dată, deci cheltuim cel mult 2$
- Rezultă că costul apelului metodei `Incrementeaza` este de cel mult 2$
- Pentru n apeluri cheltuim cel mult $2n$$
- Concluzia: costul amortizat pentru orice secvență de n operații este $O(n)$

Analiza amortizată: metoda de potențial (1)

- Ideea de bază: efortul de lucru (potențialul) este asignat întregii structuri de date, nu obiectelor în sine
- Potențialul este folosit pentru a plăti operațiile
- Asupra structurii de date se aplică o succesiune de operații:

$$D_0 \xrightarrow{op.\ 1} D_1 \xrightarrow{op.\ 2} D_2 \cdots \xrightarrow{op.\ n} D_n$$

- O funcție de potențial convenabil aleasă dă potențialul $\Phi(D_i)$ al fiecărei stări D_i a structurii de date
- Pentru operația i de cost real c_i, costul amortizat $\hat{c}_i$ este:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

Analiza amortizată: metoda de potențial (2)

- Costul amortizat total:

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

- Dacă funcția Φ are proprietatea că $\Phi(D_n) \geq \Phi(D_0)$ atunci costul total amortizat $\sum_{i=1}^{n} \hat{c}_i$ este un majorant pentru costul real
- Valoarea lui n este apriori necunoscută, așa că e mai sigur să cerem ca $\Phi(D_i) \geq \Phi(D_0)$, $\forall i = 1, 2, \ldots$
- Fără restrângerea generalității, se poate impune condiția $\Phi(D_0) = 0$

Analiza amortizată: metoda de potențial (3)

Incrementarea contorului binar

- $\Phi(D_i) = b_i =$ numărul de biți cu valoarea 1 după operația i
- Fie t_i numărul de biți setați la 0 după operația i
- Costul c_i real al operației este cel mult $t_i + 1$, deoarece cel mult încă un bit este setat la 1
- Se poate arata usor ca: $b_i \leq b_{i-1} - t_i + 1$
- $\Phi(D_i) - \Phi(D_{i-1}) = b_i - b_{i-1} \leq (b_{i-1} - t_i + 1) - b_{i-1} = 1 - t_i$
- Costul amortizat al operației i:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \leq (t_i + 1) + (1 - t_i) = 2$$

- Contorul pornește de la zero, deci $\Phi(D_0) = 0$; $\Phi(D_i) \geq 0 = \Phi(D_0)$
- $\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} \hat{c}_i - \Phi(D_n) + \Phi(D_0) \leq \sum_{i=1}^{n} 2 - b_n + b_0 \leq 2n + b_0 \leq 2n + k = O(n)$ pentru $n = \Omega(k)$

Analiza amortizată: aplicație

Problemă: operatori de stivă

- Se pleacă de la tipul de date stivă care are două metode asociate:
- `Pune-in-stiva(`S`,` x`)` – adaugă în stiva S obiectul x
- `Scoate-din-stiva(`S`)` – extrage un obiect din stiva S și îl returnează ca rezultat

Analiza amortizată: metoda de agregare (1)

- Costul acestor operații este constant față de dimensiunea stivei, deci $O(1)$
- O secvență oarecare de n operații `Pune-in-stiva(`S`,` x`)` și `Scoate-din-stiva(`S`)` are costul n iar timpul necesar e $\Theta(n)$
- Considerăm operația: `Scoatere-multipla-din-stiva(`S`,` k`)` care extrage succesiv k obiecte din stiva S
- Dacă S are mai puțin de k obiecte, apelul se consideră valid și duce la golirea stivei

```
Scoatere-multipla-din-stiva(S, k)
1  while not Stiva-Vida(S) and k ≠ 0
2      Scoate-din-stiva(S)
3      k ← k − 1
```

Analiza amortizată: metoda de agregare (2)

- Dacă stiva conține s elemente, atunci numărul de iterații este $\min\{s, k\}$
- Timpul de execuție este o funcție liniară de numărul de iterații
- Problemă: care e costul unei secvențe de n operații `Pune-in-stiva`, `Scoate-din-stiva`, `Scoatere-multipla-din-stiva` pentru o stivă inițial vidă?
- Prima analiză: costul unei operații `Scoatere-multipla-din-stiva` este $O(n)$, deoarece stiva va ajunge să aibe maxim n elemente
- Costul unei secvențe e deci $O(n^2)$, deoarece în cazul cel mai nefavorabil putem avea n apeluri `Scoatere-multipla-din-stiva`
- Rezultatul este corect, dar costul este prea grosier
- Analiza amortizată oferă un cost mai realist pentru secvență

Analiza amortizată: metoda de agregare (3)

- O analiză mai atentă: un obiect depus în stivă poate fi extras cel mult o dată
- Numărul de apeluri pentru `Scoate-din-stiva` – inclusiv apelurile din cadrul `Scoatere-multipla-din-stiva` – este cel mult numărul de elemente introduse în stivă
- Rezultat: pentru orice n, orice secvență de operații `Pune-in-stiva`, `Scoate-din-stiva` și `Scoatere-multipla-din-stiva` va avea costul total $O(n)$
- Costul mediu (*i.e.* amortizat) pentru o operație este deci $O(n)/n = O(1)$
- De remarcat că acest cost este pentru orice secvență de operații

Analiza amortizată: metoda de cotare (1)

- Costurile *reale* ale operațiilor:

`Pune-in-stiva(`S`, `x`)`	1
`Scoate-din-stiva(`S`)`	1
`Scoatere-multipla-din-stiva(`S`, `k`)`	$\min(k, \lvert S \rvert)$

- Costurile *amortizate* atașate operațiilor:

`Pune-in-stiva(`S`, `x`)`	2 (\$)
`Scoate-din-stiva(`S`)`	0 (\$)
`Scoatere-multipla-din-stiva(`S`, `k`)`	0 (\$)

- Fiecare operație este fie plătită (prima), fie va folosi credit existent (ultimele două)

Analiza amortizată: metoda de cotare (2)

Interpretarea costurilor amortizate:

- Cand se introduce un obiect în stivă, 1\$ se plătește pentru această acțiune iar restul de 1\$ rămâne drept credit pentru obiectul introdus
- Când obiectul este scos din stivă se utilizează creditul de 1\$
- Creditul asigurat la operația de introducere permite ca la extragere să nu se ajungă la un credit negativ
- Chiar și extragerea repetată, via metoda `Scoatere-multipla-din-stiva` va profita de rezerva asigurată acordată fiecărui obiect la introducerea în stivă

Concluzia: pentru orice secvență de n operații vom cheltui maxim $2n$\$, deci chiar în cazul cel mai nefavorabil costul este liniar.

Analiza amortizată: metoda de potențial (1)

- Funcția de potențial a stivei este numărul de obiecte din stivă
- Numărul de elemente din stivă este nenegativ, deci $\Phi(D_i) \geq 0 = \Phi(D_0)$, unde $D_0 =$ starea de stivă vidă
- Deci: costul amortizat este un majorant al costului real
- Costul amortizat pentru operații este:
  - $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + (s+1) - s = 2$ dacă se aplică operația `Pune-in-stiva(`S`,` x`)` cu $\Phi(D_{i-1}) = s - 1$
  - $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + (s-1) - s = 0$ dacă se aplică operația `Scoate-din-stiva(`S`,` x`)` cu $\Phi(D_{i-1}) = s$
  - 0 pentru `Scoatere-multipla-din-stiva(`S`,` k`)`, deoarece se aplică repetat `Scoate-din-stiva(`S`,` x`)`
- Costurile amortizate pentru fiecare din operații sunt în $O(1)$, deci pentru o succesiune oarecare de n operații avem cost amortizat total $O(n)$
- Costul total este majorat de costul amortizat total, deci costul total este $O(n)$

Analiza amortizată: aplicație

- Tabele dinamice: colecții de date pentru care spațiul ce trebuie alocat nu se cunoaște apriori
  - dacă numărul de elemente prealocate e mai mic decât necesarul, se va aloca un tabel mai mare și se copiază elementele în acesta
  - dacă se șterg date din tabel și el devine "prea gol", el se poate contracta
- Prin analiza amortizată se va demonstra că costul amortizat este $O(1)$
- Trăsătură suplimentară: numărul de elemente nefolosite din tabelul dinamic să nu depășească o anumită fracție din lungimea tabelului

Analiza amortizată: aplicație

- Pe post de tabel dinamic: tablou, stivă, coadă, heap, tabelă de dispersie
- Operații:
  - `Tabel-Insereaza` — inserează un element în tabelă, la sfârșit; posibil să ducă la expandare și copiere
  - `Tabel-Sterge` — șterge un element; posibil să ducă la copiere și colapsare
- Notații:
  - $dim[T]$ = lungimea tabelei T
  - $num[T]$ = numarul de poziții ocupate în tabelă
- Factorul de încărcare al unui tablou T: $\alpha(T) = num[T]/dim[T]$

Tabele dinamice: expandări repetate

- Se execută o succesiune de inserări
- Când se încearcă adăugarea de date într-un tablou plin, lungimea lui se dublează și elementele vechi sunt mutate în noul tablou
  - Care este factorul de încărcare minim/maxim?
- Notații:
  - T — tabloul
  - $ref[T]$ — adresa tabloului (e.g. referință către primul element)
- inițial: $num[T] = dim[T] = 0$

Tabele dinamice: expandări repetate

Tabel-Insereaza(T[1...n])

```
1  if dim[T] = 0    then
2          alocă pentru ref[T] o locație de memorie
3          dim[T] ← 1
4  if num[T] = dim[T]      then
5          alocă pentru tablou − nou 2 · dim[T] locații
6          copiază articolele din T în tablou − nou
7          eliberează ref[T]
8          ref[T] ← tablou − nou
9          dim[T] ← 2 · dim[T]
10 adaugă x în T
11 num[T] ← num[T] + 1
```

- Acceptăm: (de)alocarea de memorie costă mai puțin decât copierea articolelor; copierea are complexitate liniară în numărul de elemente copiate.

Tabele dinamice: analiza complexității

- Secvență de n inserări asupra tabloului
- Costul operației i: c_i
- $c_i = ?$
- $$c_i = \begin{cases} 1, \text{dacă } T \text{ nu trebuie extins} \\ i = (i-1) \text{ mutări} + 1 \text{ adăugare, altfel} \end{cases}$$
- Pentru n operații, costul în cazul cel mai nefavorabil este $O(n)$
- Total: $O(n^2)$
- Complexitatea este bună, dar nu suficient de exactă
  - De ce a ieșit costul prea mare?

Tabele dinamice: analiză prin metoda de agregare

- Expandarea tabloului este o operație mai puțin frecventă decât se crede
- $$c_i = \begin{cases} i & \text{dacă } i-1 \text{ este o putere a lui 2} \\ 1 & \text{altfel} \end{cases}$$
- $$\sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{\lfloor \log_2 n \rfloor} 2^j < n + 2n = 3n$$
- Concluzia: costul a n operații este $3n$ și costul amortizat al unei operații este $3 = O(1)$

Tabele dinamice: analiză prin metoda de cotare

- Alocăm 3$ pentru fiecare element adăugat:
  1. 1$ pentru adăugarea sa
  2. 1$ pentru când va fi copiat în urma unei expandări; la o expandare, acest dolar este cheltuit
  3. 1$ pentru un element cu credit 0 care a ajuns în tabloul curent datorită unei copieri anterioare – vezi punctul de mai sus
- După o secvență de inseraări:
  - e posibil să avem măcar un element fără niciun dolar asignat?
  - e posibil ca niciun element să nu aibă dolar asignat?

Tabele dinamice: analiză prin metoda de potențial

- Căutăm o funcție de potențial cu proprietățile:
  - la început sau imediat după o expandare să aibă valoarea 0
  - să aibă valoare maximă chiar înainte de expandare
- Propunem:
$$\Phi(T) = 2 \cdot num[T] - dim[T] \tag{28}$$
- După operația i:
  - num_i: numărul de elemente inserate
  - dim_i: lungimea lui T
- Inițial: $num_0 = 0$, $dim_0 = 0$, $\Phi_0 = 0$

Tabele dinamice: analiză prin metoda de potențial

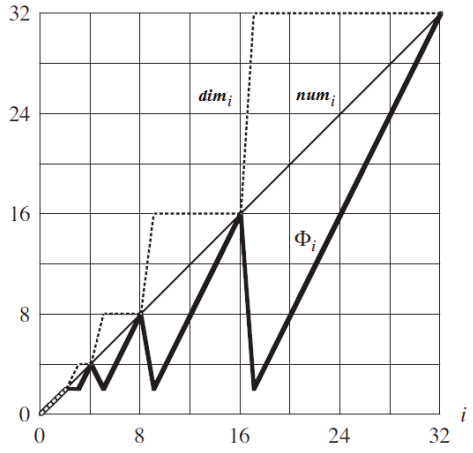
- Dacă operația i de inserare nu duce la expansiune:
  - $dim_i = dim_{i-1}$, $num_i = num_{i-1} + 1$
  - costul amortizat al operației este:

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2 \cdot num_i - dim_i) - (2 \cdot num_{i-1} - dim_{i-1}) \\ &= 1 + 2 \cdot num_{i-1} + 2 - dim_{i-1} - 2 \cdot num_{i-1} + dim_{i-1} \\ &= 3\end{aligned}$$

- Dacă operația i de inserare duce la expansiune:
  - $dim_i = 2 \cdot dim_{i-1}$, $dim_{i-1} = num_{i-1} = num_i - 1$

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= num_i + (2 \cdot num_i - dim_i) - (2 \cdot num_{i-1} - dim_{i-1}) \\ &= 3\end{aligned}$$

Tabele dinamice: analiză prin metoda de potențial



Figură: Evoluția lui dim_i, num_i, Φ_i pentru o secvență de operații de inserare.

Tabele dinamice: analiză prin metoda de potențial

- Expandarea și contractarea tabelului:
  - contractare: dacă rămân prea multe locații neocupate în tablou
  - contractarea produce copierea datelor în tabloul mai mic
- Ce urmărim?
  - costul amortizat al unei operații să fie mărginit superior de o constantă
  - factorul de încărcare $\alpha(T)$ să fie minorat de o constantă

Tabele dinamice: analiză prin metoda de potențial

- Strategia 1:
  - se dublează dimensiunea tabloului când se umple
  - se înjumătățește dimensiunea tabloului după o ștergere, dacă numărul de elemente ajunge mai puțin de jumătate din dimensiunea lui
  - $\alpha(T) \in [1/2, 1]$
- Scenariu nefavorabil: efectuăm $n = 2^k$ operații:
  - primele $n/2$ sunt inserări, cu cost $\Theta(n)$
  - următoarele $n/2$ operații sunt: inserare, ștegere, ștegere, inserare, inserare, ștegere, ștegere, inserare, inserare, ...
  - prima inserare: se expandează tabloul; următoarele ștergeri coboară $num[T]$ la mai puțin de jumătate din $dim[T]$; următoarele 2 inserări duc la expandarea tabloului etc; în total avem $\Theta(n)$ operații de expandare și compactare, fiecare declanșând copierea de $\Theta(n)$ elemente, deci complexitate $\Theta(n^2)$
  - concluzie: strategia de expandare/colapsare propusă duce la cost amortizat pe operație $\Theta(n)$ în cel mai nefavorabil caz

Tabele dinamice: analiză prin metoda de potențial

- Strategia 2: permitem ca $\alpha(T) < 1/2$
- Se va înjumătăți tabloul atunci când $num[T] < 1/4 \cdot dim[T]$
- $\alpha(T) \in [1/4, 1]$
- Pseudocodul procedurii `Tabel-Sterge` este similar cu cel de inserare
- Funcția Φ de potențial:
  - maximă înainte de o operație de copiere între tablouri
  - 0 imediat după ce are loc o expandare sau colapsare
  - crescătoare pe măsură ce $\alpha(T)$ tinde spre 1 sau $1/4$

$$\Phi_i = \begin{cases} 2 \cdot num_i - dim_i = 2 \cdot dim_i \cdot (\alpha_i - 1/2) & \text{pentru } 1 \geq \alpha_i \geq 1/2 \\ dim_i/2 - num_i = dim_i \cdot (1/2 - \alpha_i) & \text{pentru } 1/4 \leq \alpha_i < 1/2 \end{cases}$$

Tabele dinamice: analiză

- Scopul analizei: o operație are cost amortizat majorată de o constantă, chiar pentru cazul cel mai nefavorabil?
- Dacă operația este de tip inserare:
  - pentru $\alpha_{i-1} \geq 1/2$, analiza lui Φ_i e ca mai sus
  - pentru $\alpha_{i-1} < 1/2$:
    - tabloul nu se poate expanda la inserarea i
    - dacă și $\alpha_i < 1/2$, costul amortizat al operației i este:

$$\begin{aligned} \hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} = 1 + (dim_i/2 - num_i) - (dim_{i-1}/2 - num_{i-1} \\ &= 1 + (dim_i/2 - num_i) - (dim_i/2 - (num_i - 1)) = 0 \end{aligned}$$

    - dacă $\alpha_i \geq 1/2$:

$$\begin{aligned} \hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} = 1 + (dim_i/2 - num_i) - (dim_{i-1}/2 - num_{i-1} \\ &= 1 + (2(num_{i-1} + 1) - dim_{i-1}) - (dim_{i-1}/2 - num_{i-1}) \\ &= 3 \cdot num_{i-1} - \frac{3}{2} dim_{i-1} + 3 = 3\alpha_{i-1} dim_{i-1} - \frac{3}{2} dim_{i-1} + 3 \\ &< \frac{3}{2} dim_{i-1} - \frac{3}{2} dim_{i-1} + 3 = 3 \end{aligned}$$

Tabele dinamice: analiză

- Dacă operația este de tip ștergere:
  - $num_i = num_{i-1} + 1$
  - dacă $\alpha_{i-1} < 1/2$ e posibil să se contracte tabloul
    - dacă tabloul nu trebuie contractat, atunci $dim_i = dim_{i-1}$ și

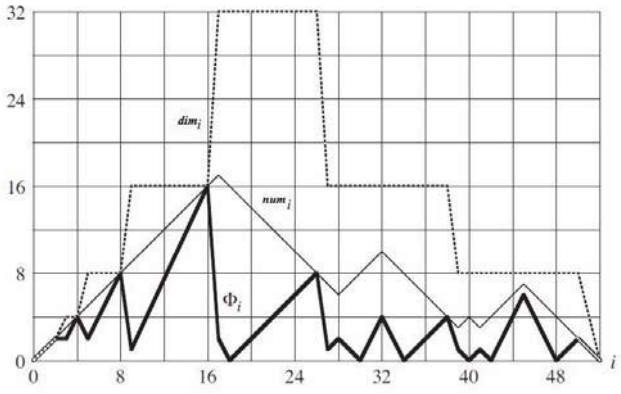
$$\begin{aligned} \hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} = 1 + (dim_i/2 - num_i) - (dim_{i-1}/2 - num_{i-1}) \\ &= 2 \end{aligned}$$

    - dacă tabloul se contractă la ștergerea i, atunci $c_i = num_i + 1$ (copiere), $dim_i/2 = dim_{i-1}/4 = num_{i-1} = num_i + 1$

$$\begin{aligned} \hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= (num_i + 1) + (dim_i/2 - num_i) - (dim_{i-1}/2 - num_{i-1}) \\ &= (num_i + 1) + ((num_i + 1) - num_i) - ((2 \cdot num_i + 2) - \\ &- (num_i + 1)) - ((2 \cdot num_i + 2) - (num_i + 1)) = 1 \end{aligned}$$

  - dacă $\alpha_i \geq 1/2$: $\hat{c}_i$ este de asemenea majorat de o constantă (temă)

Tabele dinamice: analiză



Figură: Efectul unei secvențe de n operații de insere și ștergere asupra lui num_i, dim_i și Φ_i.

Formule utile pentru sume

- Serie aritmetică:
  $a_1, a_2 = a_1 + q, \ldots, a_n = a_{n-1} + q = a_1 + (n-1)q$

$$\sum_{k=1}^{n} a_k = \frac{(a_1 + a_n)n}{2} = \frac{2a_1 + (n-1)q}{2} \tag{29}$$

- Serie geometrică: $a_1 = x, a_2 = xq, \ldots, a_n = a_{n-1}q = xq^{n-1}$

$$\sum_{k=1}^{n} a_k = x \sum_{k=0}^{n-1} q^k = x\frac{q^n - 1}{q - 1} \tag{30}$$

  Pentru $|q| < 1$:

$$\lim_{n\to\infty} \sum_{k=0}^{n} q^k = \frac{1}{1-q} \tag{31}$$

Formule utile pentru sume (2)

- Serie armonică: al n-lea număr armonic este

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \sum_{k=1}^{n} \frac{1}{k} = \ln n + O(1) \tag{32}$$

- Integrarea și diferențierea seriilor: derivând seria geometrică (30), se obține:

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2} \tag{33}$$

- Serii telescopante: pentru șirul $a_0, a_1, \ldots, a_n$

$$\sum_{k=1}^{n} (a_k - a_{k-1}) = a_n - a_0 \tag{34}$$

Formule utile pentru sume (3)

- Aproximarea prin integrale, pentru sumă ce poate fi exprimată sub forma $\sum_{k=m}^{n} f(k)$
  - dacă f este monoton crescătoare:

$$\int_{m-1}^{n} f(x)\mathrm{d}x \leq \sum_{k=m}^{n} f(k) \leq \int_{m}^{n+1} f(x)\mathrm{d}x \qquad (35)$$

  - dacă f este monoton descrescătoare:

$$\int_{m}^{n+1} f(x)\mathrm{d}x \leq \sum_{k=m}^{n} f(k) \leq \int_{m-1}^{n} f(x)\mathrm{d}x \qquad (36)$$

Bibliografie

- *Algoritmi fundamentali. O perspectiva C++*, Răzvan Andonie și Ilie Gârbacea, Editura Libris, 1995, disponibila la: http://www.cwu.edu/ andonie/Cartea%20de%20algoritmi/cartea%20de%20algoritmi.pdf
- *Introduction to Algorithms*, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, The MIT Press, 3rd edition, 2009

Cursul nr. 3

TEHNICI DE SINTEZĂ A ALGORITMILOR

Cuprins

Algoritmi de tip greedy

Algoritmi de tip divide et impera

Algoritmi de programare dinamică

Backtracking și branch-and-bound

Principiile algoritmilor greedy

- **greedy**(lacom), algoritmi simpli și folosiți la probleme de optimizare (cea mai bună ordine de execuție a unor task-uri pe calculator, cel mai scurt drum în graf)
- mulțime de candidați
- verificare dacă mulțimea de candidați este fezabilă
- verificarea mulțimii de candidați ca soluție posibilă
- funcția de selecție a celui mai promițător candidat
- funcția obiectiv ce dă valoarea soluției
- selecția celui mai promițător candidat, pas cu pas, mulțime fezabilă, soluție optimă
- exemplu: problema numărului minim de monezi, valori 1, 5, 10, 25, suma 44
- alt exemplu: valori 1, 12, 25, suma 36

Minimizarea timpului mediu de așteptare

- o stație de servire satisface cererile a n clienți
- timpul de servire a clientului i este t_i
- timpul mediu de așteptare minim corespunde timpului total de așteptare minim:

$$T = \sum_{i=1}^{n} (\text{timpul de așteptare al clientului i})$$

- timpul de așteptare al unui client înglobează de timpii de așteptare a clienților dinaintea lui
- algoritmul greedy selectează la fiecare pas clientul cu cel mai mic timp de servire

Minimizarea timpului mediu de așteptare

- o permutare a clienților: $I = (i_1, i_2, \ldots, i_n)$ are timpul total de așteptare:

$$T(I) = nt_{i_1} + (n-1)t_{i_2} + \cdots + t_{i_n} = \sum_{k=1}^{n}(n-k+1)t_{i_k}$$

- dacă pentru doi indici a și b, $a < b$, avem $t_{i_a} > t_{i_b}$, prin interschimbare, timpul total obținut este mai mic:

$$T(J) = (n-a+1)t_{i_b} + (n-b+1)t_{i_a} + \sum_{k=1,k\neq a,b}^{n}(n-k+1)t_{i_k}$$

$$\begin{aligned} T(I) - T(J) &= (n-a+1)(t_{i_a} - t_{i_b}) + (n-b-1)(t_{i_b} - t_{i_a}) \\ &= (b-a)(t_{i_a} - t_{i_b}) > 0 \end{aligned}$$

- privit ca matroid, ce înseamnă o mulțime independentă?

Bazele teoretice ale metodei greedy. Liniar independența

- **matroizi:** structuri combinatorice ce generalizează conceptul de liniar independență întâlnit la spațiile vectoriale

$$v_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \quad v_2 = \begin{bmatrix} 0 \\ 2 \\ -2 \end{bmatrix} \quad v_3 = \begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix} \quad v_4 = \begin{bmatrix} 4 \\ 2 \\ 3 \end{bmatrix}$$

- vectorii v_1, v_2 și v_3 sunt liniar independenți, adică dacă pentru a_1, a_2, $a_3 \in \mathbb{R}$ avem $a_1 \cdot v_1 + a_2 \cdot v_2 + a_3 \cdot v_3 = 0$, atunci avem obligatoriu $a_1 = a_2 = a_3 = 0$
- oricare trei vectori din cei patru sunt liniar independenți
- v_1, v_2, v_3 și v_4 nu sunt liniar independenți
- mulțimile $\{v_1\}$, $\{v_1, v_2\}$ și $\{v_1, v_2, v_3\}$ sunt seturi de vectori liniar independenți

Matroizi

- **matroid:** pereche ordonată $M = (S, I)$ unde:
  1. S = mulțime finită nevidă
  2. I **este ereditară**: I = familie nevidă de submulțimi ale lui S, cu $|I| \leq 2^{|S|}$, denumite submulțimi **independente**, astfel încât dacă $B \in I$ și $A \subseteq B$, atunci $A \in I$
  3. **proprietatea de schimb:** dacă $A \in I$, $B \in I$ și $|A| < |B|$, atunci există un $x \in B - A$ pentru care $A \cup \{x\} \in I$
- mulțimile $\{v_1\}$, $\{v_1, v_2\}$ și $\{v_1, v_2, v_3\}$ sunt independente (matroizi matriceali)
- mulțimea vidă $\emptyset$ este automat un element al lui I
- pentru **matroizii matriceali**, elementele din S sunt coloane într-o matrice iar submulțimile independente se formează din grupuri de coloane liniar independente

Matroizi

- **matroidul grafic** $M_G = (S_G, I_G)$ este definit în termenii grafului neorientat $G = (V, M)$ astfel:
  1. mulțimea S_G este mulțimea M a muchiilor;
  2. dacă $A \subseteq M$ este un subset de muchii, atunci $A \in I_G$ dacă și numai dacă A nu are cicluri, adică mulțimea independentă A formează o pădure
- cu ajutorul matroidului grafic se studiază algoritmii de calculare a arborelui parțial de cost minim

Matroizi

- dacă G este un graf neorientat, atunci $M_G = (S_G, I_G)$ este un **matroid**
  1. mulțimea de muchii $S_G = M$ este nevidă
  2. I_G este ereditară, pentru că o submulțime a unei păduri este tot o pădure; prin înlăturarea de muchii nu se pot creea cicluri
  3. presupunem că A și B sunt două păduri și că $|B| > |A|$. Pentru o pădure V cu k muchii, ea conține exact $|V| - k$ arbori (ușor de observat că, plecînd de la $|V|$ arbori, zero muchii, adăugînd o muchie obținem $|V| - 1$ arbori). Cum pădurea A are mai puțini arbori, atunci există în ea un arbore care unește vârfuri în B ce fac parte din arbori diferiți în B. Arborele va trebui să conțină o muchie (u, v) ce unește două vârfuri, în arbori diferiți din B. Atunci această muchie nu introduce un ciclu, ci numărul arborilor din B scade cu 1, ceea ce satisface proprietatea de schimb

Matroizi

- pentru o mulțime $A \in I_G$, un element $x \notin A$ se numește **extensie** a lui A dacă x poate fi adăugat ls A cu păstrarea independenței (nu formează un ciclu)
- mulțimea $A \in I_G$ este **maximală** dacă nu are extensii; cu alte cuvinte, ea nu este submulțime a niciunei alte mulțimi independente din setul I_G
- toate submulțimile independente maximale dintr-un matroid au aceeași dimensiune
  1. presupunem că ar exista A și B, cu $|A| < |B|$, maximale. Din proprietatea de schimb, ar exista un element $x \in B - A$ pentru care A ar fi extensibilă, deci A nu ar mai fi maximală
- pentru un graf G, mulțimile maximale sunt arbori cu exact $|V| - 1$ muchii, ce leagă toate vârfurile din G - arbori de acoperire

Algoritmi greedy pe un matroid ponderat

- matroidul se numește **ponderat** dacă fiecărei muchii $x \in S_G = M$ i se asociază o pondere $w(x)$; ponderea totală a unui arbore devine $w(A) = \sum_{x \in A} w(x)$
- algoritmii greedy determină soluția optimă ca o submulțime maximală $A \in I$ într-un matroid ponderat $M = (S, I)$, pentru care ponderea $w(A)$ să fie maximă
- pentru arborele de acoperire minim, fie $w_0 = l(x_0)$ lungimea celei mai mari (lungi) muchii
- ponderea unei muchii x, $w(x) = w_0 - l(x)$, este cu atât mai mare cu cât muchia este mai scurtă (are asociat un cost mai mic)
- găsirea mulțimii indepentente maximale cu cea mai mare pondere în matroidul $M = (S, I)$ dă algoritmul greedy generic

Algoritmi greedy pe un matroid ponderat

procedure greedy (M, w)

1: $A \leftarrow \emptyset$
2: sortează S descrescător după ponderile $w(x)$, $x \in S$
3: **for** fiecare $x \in S$ **do**
4: **if** $A \cup \{x\} \in I$ **then**
5: $A \leftarrow A \cup \{x\}$
6: **end if**
7: **end for**
8: **return** A

- impicit, mulțimea vidă $\emptyset$ este independentă
- elementul x poate fi adăugat la A dacă menține independența
- algoritmul returnează o submulțime independentă A din I

Algoritmi greedy pe un matroid ponderat

- pentru un matroid $M = (S, I)$ ponderat și mulțimea S ordonată descrescător, se determină x primul element din S astfel ca $\{x\}$ să fie independentă. Dacă x există, atunci există o submulțime optimă $A \in I$ care îl conține pe x
  1. dacă x nu există, atunci singura mulțime independentă este mulțimea vidă $\emptyset$
  2. dacă x există, considerăm o mulțime B, optimă, cu $x \notin B$
  3. deoarece x este prima alegere, orice element $y \in B$ are ponderea $w(y) < w(x)$
  4. la început, $A = \{x\}$
  5. folosind proprietatea de schimb în mod repetat, deoarece $A, B \in I$, se adaugă câte un element din B la A, până când $|A| = |B|$
  6. deoarece B este optimă, atunci și A este optimă (are aceeași dimensiune)

$$A = B - \{y\} \cup \{x\}$$
$$w(A) = w(b) - w(y) + w(x) \geq w(B)$$

Algoritmi greedy pe un matroid ponderat

- pentru un matroid, dacă x nu este o extensie a $\emptyset$, atunci nu este extensie a nici unei submulțimi independende A a lui S.
  1. presupunem că x este o extensie a unei mulțimi A, dar nu și a $\emptyset$
  2. deoarece $A \cup \{x\}$ este independentă, orice submulțime a sa, deci și $\{x\}$, este independentă, ceea ce contrazice ipoteza făcută
- un element care nu poate fi folosit la un moment dat, nu mai poate fi folosit ulterior niciodată

Algoritmi greedy pe un matroid ponderat

- fie $\{x\}$ primul element ales de algoritmul greedy pentru matroidul $M = (S, I)$. Problema de găsire a submulțimii independente de pondere maximă care să îl conțină pe x se reduce la găsirea submulțimii independente de pondere maximă pentru matroidul $M' = (S', I')$ unde:

$$S' = \{y \in S : \{x, y\} \in I\}$$
$$I' = \{B \subseteq S - \{x\} : B \cup \{x\} \in I\}$$

- principiul greedy - pe baza optimului local se formulează optimul global
- la fiecare pas, algoritmul greedy alege cel mai bun candidat x pentru care mulțimea $\{x\}$ este independentă
- algoritmul greedy nu poate greși, pentru că există o submulțime optimă care îl conține pe $\{x\}$

O problemă de planificare a task-urilor

- optimizarea planificării task-urilor care durează un singur tact, pe un procesor, unde fiecare task are asociate un deadline și o penalizare, dacă task-ul se termină mai târziu
- setul de task-uri $S = \{a_1, a_2, \ldots, a_n\}$ de durată 1
- setul de deadline-uri $d_1, d_2, \ldots, d_n$ unde $1 \leq d_i \leq n$, task-ul a_i trebuie să se termine înainte sau la timpul d_i
- un set de penalizări $w_1, w_2, \ldots, w_n$ penalizarea w_i fiind aplicată numai dacă task-ul a_i se termină după timpul d_i
- o planificare constă într-o permutare a lui S, care dă ordinea în care se execută task-urile
- un task este **întârziat** dacă se termină după deadline, sau **prematur** în caz contrar

O problemă de planificare a task-urilor

- o planificare poate fi adusă în **forma prematură**, în care task-urile premature le preced pe cele întârziate; astfel, dacă a_i care este întârziat îl precede pe a_j, care este prematur:

$$\ldots, a_i, \ldots, a_j, \ldots$$

- pozițiile lor pot fi interschimbate în planificare, iar a_i va rămâne tot întârziat, iar a_j tot prematur
- **forma canonică** presupune că task-urile premature se află înaintea celor întârziate iar cele premature sunt planificate în ordinea crescătoare a deadline-urilor lor
- putem pune task-urile în forma prematură; apoi, pentru două task-uri premature consecutive a_i și a_j, ce se termină la timpii k și $k+1$, dacă $d_i > d_j$, avem $k+1 \leq d_j$, deci $k+1 < d_i$, deci a_i rămâne prematur după interschimbarea lui a_i cu a_j

O problemă de planificare a task-urilor

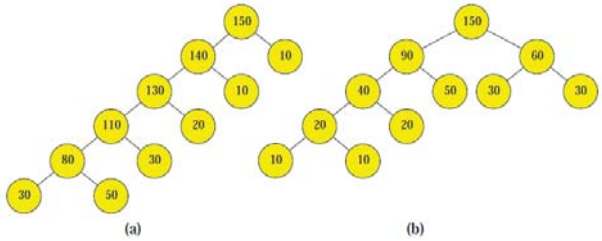
- căutarea planificării optime se reduce la găsirea unui set A de task-uri care sunt premature; apoi se ordonează task-urile crescător după deadline, iar task-urile întârziate în orice ordine
- un set A de task-uri e independent dacă există o planificare astfel ca nici un task să nu fie întârziate
- fie $N_t(A)$ numărul de task-uri din A a căror deadline este cel mult t. $N_0(A) = 0$ pentru orice set A. Următoarele sunt echivalente:
  1. setul A este independent
  2. pentru $t = 0, 1, \ldots, n$, avem $N_t(A) \leq t$
  3. dacă task-urile din A sunt sortate crescător în ordinea deadline-urilor, nu avem nici un task întârziat
- dacă am avea $N_t(A) > t$, ar fi imposibil de planificat, ar fi prea multe
- setul A fiind independent, am arătat anterior că se pot ordona task-urile, fără a deveni întârziate

O problemă de planificare a task-urilor

- problema minimizării penalităților task-urilo întârziate este aceeași cu maximizarea penalității task-urilor premature
- fiind dat un set S de task-uri și I toate seturile independente de task-uri, (S, I) este matroid
  1. S este o mulțime finită, nevidă, de task-uri
  2. fiecare subset al unui set independent de task-uri, este și el independent
  3. fie două seturi A și B, independente, cu $|B| > |A|$. Luăm k valoarea maximă a lui t pentru care $N_t(B) \leq N_t(A)$. Deoarece $N_n(A) = |A| < |B| = N_n(B)$, avem $k < n$, și $N_j(B) > N_j(A)$ pentru $k + 1 \leq j \leq n$. De aceea, B conține mai multe task-uri cu deadline $k + 1$ decât A. Fie $a_i \in B - A$ cu deadline $k + 1$. Atunci $A' = A \cup \{a_i\}$. Până la $k + 1$, $N_t(A') \leq t$, chiar task-urile din A. Pentru $k + 1$, $N_{k+1}(A') \leq N_{k+1}(B) \leq k + 1$, așadar și mulțimea A' este independentă

Interclasarea optimă a șirurilor ordonate

- interclasarea a două șiruri S_1 și S_2 ordonate crescător, presupune $|S_1 + S_2|$ operații de copiere pentru obținerea șirului ordonat crescător $S = S_1 \cup S_2$
- interclasarea șirurilor ordonate crescător $S_1, S_2, \ldots, S_n$ se face în așa fel încât numărul de operații de copiere să fie cât mai mic
- $|S_1| = 30$, $|S_2| = 10$, $|S_3| = 20$, $|S_4| = 30$, $|S_5| = 50$, $|S_6| = 10$
- două strategii de interclasare (care este costul fiecăreia?):



Interclasarea optimă a șirurilor ordonate

- la interclasarea șirurilor ordonate $S_1, S_2, \ldots, S_n$ de lungimi $q_1, q_2, \ldots, q_n$ se obține pentru fiecare strategie un arbore binar cu n frunze și $n-1$ vârfuri neterminale
- numărul total de interclasări este suma valorilor nodurilor neterminale, sau **lungimea externă ponderată minimă**:

$$L(A) = \sum_{i=1}^{n} a_i \cdot q_i$$

unde a_i este adâncimea în arbore a vârfului i.

- soluția cu lungimea externă ponderată minimă este arborele corespunzător strategiei greedy - la adâncimi mari, șiruri scurte - acestea se interclasează primele

Interclasarea optimă a șirurilor ordonate

- se poate aplica același raționament ca la minimizarea timpului mediu de așteptare
  1. fie A arborele corespunzător strategiei de interclasare greedy
  2. presupunem că există nodurile q_i la adâncimea a_i și q_j la adâncimea a_j, pentru care $q_i > q_j$ și $a_i > a_j$
  3. nodul j e situat 'mai jos' în arbore
  4. obținem arborele B prin interschimbarea celor două noduri

     $$L(A) - L(B) = q_i a_i + q_j a_j - q_i a_j - q_j a_i = (a_i - a_j)(q_i - q_j) > 0$$

  5. noul arbore are lungimea mai mică
  6. presupunerea inițială nu e viabilă, nu există un șir mai lung situat 'mai jos' în arbore

Interclasarea optimă a șirurilor ordonate

- pentru sortarea nodurilor se folosește un min-heap
- fiecare element este o pereche (q, i) unde q este lungimea șirului i
- fiecare nod are asociată lungimea șirului $LU[i]$, fiul stâng $ST[i]$ și fiul drept $DR[i]$ (heap de arbori, pădure)
- care este ordinul de timp al algoritmului ce construiește arborele de interclasare?

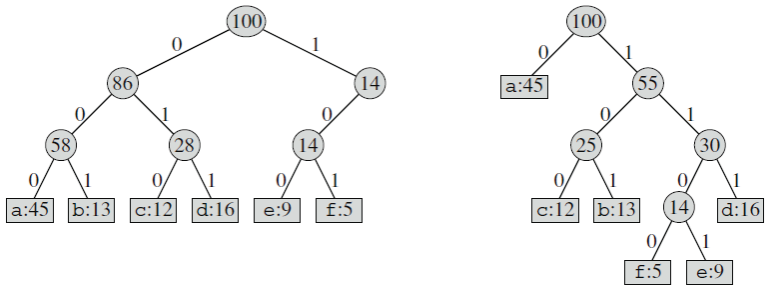
Interclasarea optimă a șirurilor ordonate

procedure interopt(Q[1..n])

1: {construiește arborele strategiei greedy de interclasare}
2: {al șirurilor de lungimi $Q[i] = q_i$, $1 \leq i \leq n$}
3: $H \leftarrow$ min-heap vid
4: **for** $i \leftarrow 1$ to n **do**
5: $\quad (Q[i], i) \Rightarrow H$ {inserează în min-heap}
6: $\quad LU[i] \leftarrow Q[i]$; $ST[i] \leftarrow 0$; $DR[i] \leftarrow 0$
7: **end for**
8: **for** $i \leftarrow n + 1$ to $2n - 1$ **do**
9: $\quad (s, j) \Leftarrow H$ {extrage rădăcina lui H}
10: $\quad (r, k) \Leftarrow H$ {extrage rădăcina lui H}
11: $\quad ST[i] \leftarrow j$; $DR[i] \leftarrow k$; $LU[i] \leftarrow s + r$
12: $\quad (LU[i], i) \Rightarrow H$ {inserează în min-heap}
13: **end for**

Coduri Huffman

- metodă eficientă pentru compactarea datelor (20% - 90%)
- fiecare caracter are asociată o frecvență de apariție
- problema: reprezentarea fiecărui caracter cu un cod binar unic
- codificarea cu lungime variabilă alocă caracterelor cu frecvențe mai mari, coduri mai scurte



Coduri prefix

- **codificări prefix:** nici un cuvânt de cod nu este prefix al unui alt cuvânt de cod
- codificarea unui caracter este drumul parcurs de la rădăcină până la acel nod (frunză)
- 0 - fiu stâng, 1 - fiu drept
- pentru un caracter, $f(c)$ - frecvența de apariție a caracterului, iar $d_T(c)$ - adâncimea frunzei în arbore
- fiecare operație de concatenare elimină două noduri și creează exact unul
- pentru un alfabet C, câte frunze și câte noduri intermediare are arborele codificării?
- numărul de biți necesar pentru codificarea unui text este lungimea externă ponderată minimă:

$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

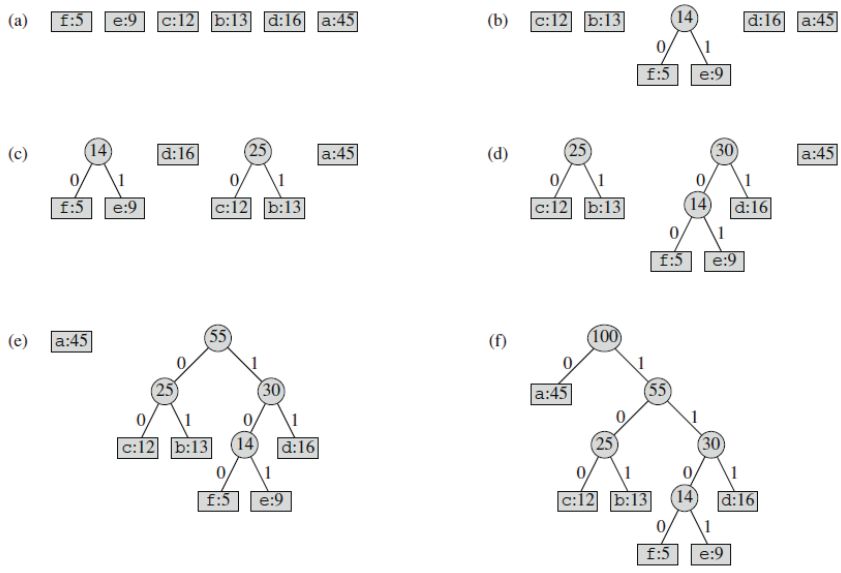
Construcția codului Huffman

function Huffman(C)

1: $n \leftarrow |C|$
2: $Q \leftarrow C$
3: **for** $i \leftarrow 1$ to $n - 1$ **do**
4: $z \leftarrow$ create-node()
5: $x \leftarrow z.left \leftarrow$ extract-min(Q)
6: $y \leftarrow z.right \leftarrow$ extract-min(Q)
7: $z.frequency \leftarrow x.frequency + y.frequency$
8: insert(Q, z)
9: **end for**
10: **return** extract-min(Q)

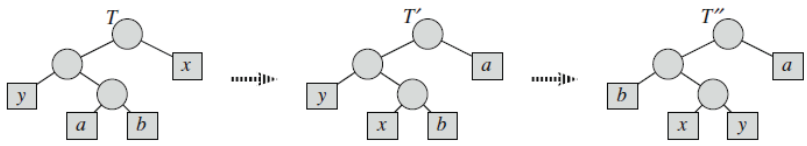
- Q este o coadă de priorități (heap)
- care este ordinul timpului de execuție?

Pașii algoritmului Huffman



Corectitudinea algoritmului Huffman

- Avem un alfabet C în care fiecare caracter are frecvența $f[c]$. Fie x și y caracterele cu cele mai mici frecvențe. Atunci există o codificare prefix optimă în care cuvintele de cod pentru x și y au aceeași lungime și diferă doar prin ultimul bit
  1. considerăm două frunze a și b, frați, situate la cel mai adânc nivel în arbore
  2. putem presupune că $f[a] \leq f[b]$ și că $f[x] \leq f[y]$
  3. fiind frecvențele cele mai scăzute, avem că $f[x] \leq f[a]$ și $f[y] \leq f[b]$
  4. vom interschimba a cu x, obținînd T', apoi b cu y, obținînd T''



Corectitudinea algoritmului Huffman

$$\begin{aligned} B(T) - B(T') &= \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\ &= f[x] d_T(x) + f[a] d_T(a) - f[x] d_{T'}(x) - f[a] d_{T'}(a) \\ &= f[x] d_T(x) + f[a] d_T(a) - f[x] d_T(a) - f[a] d_T(x) \\ &= (f[a] - f[x])(d_T(a) - d_T(x)) \geq 0 \end{aligned}$$

5. analog, interschimbarea lui b cu y nu mărește costul, diferența $B(T') - B(T'') \geq 0$, nenegativă
6. $B(T'') \leq B(T)$, dar deoarece T este optimal, $B(T) \leq B(T'')$, de aceea $B(T) = B(T'')$

Corectitudinea algoritmului Huffman

- Avem arborele de codificare prefix optimă reprezentată sub forma arborelui T, cu alfabetul C. Considerăm două noduri frați x și y, terminale, și fie z tatăl lor. Atunci, considerînd $f[z] = f[x] + f[y]$, arborele $T' = T - \{x, y\}$ reprezintă o codificare prefix optimă pentru alfabetul $C' = C - \{x, y\} \cup \{z\}$
  1. pentru fiecare $c \in C - \{x, y\}$, adâncimea este aceeași, $d_T(c) = d_{T'}(c)$
  2. deoarece $d_T(x) = d_T(y) = d_{T'}(z) + 1$, avem:

$$\begin{aligned} f[x]d_T(x) + f[y]d_T(y) &= (f[x] + f[y])(d_{T'}(z) + 1) \\ &= f[z]d_{T'}(z) + (f[x] + f[y]) \end{aligned}$$

  3. astfel, $B(T) = B(T') + f[x] + f[y]$ adică lungimea externă ponderată crește cu suma fiilor acelui nod

Corectitudinea algoritmului Huffman

- (continuare)
  4. dacă T' nu ar reprezenta codificarea optimă, atunci înseamnă că există un T'' pentru care $B(T'') < B(T')$
  5. dar în acel arbore T'' nodului z i se pot adăuga fiii săi, x și y, rezultînd un arbore T'''. Atunci:

$$B(T''') = B(T'') + f[x] + f[y] < B(T') + f[x] + f[y] = B(T)$$

     ceea ce contrazice faptul că T reprezintă o codificare optimă.
- așadar, procedura Huffman realizează o codificare prefix optimă

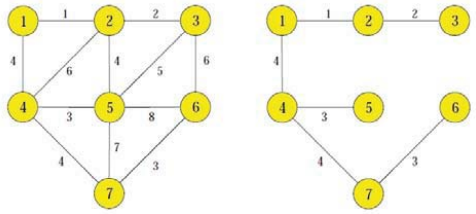
Arbori parțiali de cost minim

- pe un graf $G = (V, M)$ determinăm o submulțime $A \in M$ astfel încât toate vârfurile V să rămână conectate, iar suma lungimii muchiilor din A să fie cea mai mică posibilă
- se mai numește și problema conectării orașelor cu cost minim
- ne referim la setul mulțimilor de muchii ce formează păduri
- toate pădurile sunt independente, în sensul matroizilor (adică nu au cicluri)
- o pădure este denumită aici mulțime fezabilă
- o mulțime fezabilă este denumită promițătoare dacă poate fi completată pentru a forma soluția optimă

Algoritmul lui Kruskal

- este chiar ilustrarea matroizilor grafici
- se alege întâi muchia de cost minim, apoi se adaugă repetat muchia de cost imediat superior, care nu formează cu precedentele un ciclu
- fiecare alegere a unei muchii leagă două componente conexe, în fapt doi arbori parțiali de cost minim pentru vârfurile pe care le conectează
- muchia respectivă este cea mai bună alegere pentru conectarea lor

Algoritmul lui Kruskal



Pasul	Muchia considerata	Componentele conexe ale subgrafului $<V, A>$
initializare	—	{1}, {2}, {3}, {4}, {5}, {6}, {7}
1	{1, 2}	{1, 2}, {3}, {4}, {5}, {6}, {7}
2	{2, 3}	{1, 2, 3}, {4}, {5}, {6}, {7}
3	{4, 5}	{1, 2, 3}, {4, 5}, {6}, {7}
4	{6, 7}	{1, 2, 3}, {4, 5}, {6, 7}
5	{1, 4}	{1, 2, 3, 4, 5}, {6, 7}
6	{2, 5}	respinsa (formeaza ciclu)
7	{4, 7}	{1, 2, 3, 4, 5, 6, 7}

Algoritmul lui Kruskal

function Kruskal(G=(V, M))

1: sortează crescător M în funcție de cost
2: $n \leftarrow |V|$
3: $A \leftarrow \emptyset$
4: inițializează n mulțimi disjuncte fiecare având câte un element din V
5: **repeat**
6: $\{u, v\} \leftarrow$ muchia de cost minim ce nu a fost deja considerată
7: $ucomp \leftarrow find(u)$
8: $vcomp \leftarrow find(v)$
9: **if** $ucomp \neq vcomp$ **then**
10: merge(ucomp, vcomp)
11: $A \leftarrow A \cup \{\{u, v\}\}$
12: **end if**
13: **until** $|A| = n - 1$
14: **return** A

Algoritmul lui Kruskal I

- se folosește o structură de mulțimi disjuncte pentru un graf de m muchii și n vârfuri
- ordinul de timp pentru sortarea muchiilor?

- se folosește o structură de mulțimi disjuncte pentru un graf de m muchii și n vârfuri
- sortarea muchiilor: $O(m \log m) \subseteq O(m \log n)$ (de ce?)

- se folosește o structură de mulțimi disjuncte pentru un graf de m muchii și n vârfuri
- sortarea muchiilor: $O(m \log m) = O(m \log n)$
- inițializarea celor n mulțimi disjuncte?

- se folosește o structură de mulțimi disjuncte pentru un graf de m muchii și n vârfuri
- sortarea muchiilor: $O(m \log m) = O(m \log n)$

Algoritmul lui Kruskal II

- inițializarea celor n mulțimi disjuncte: $O(n)$

- se folosește o structură de mulțimi disjuncte pentru un graf de m muchii și n vârfuri
- sortarea muchiilor: $O(m \log m) = O(m \log n)$
- inițializarea celor n mulțimi disjuncte: $O(n)$
- cel mult $2m$ operații find() și n operații merge()?

- se folosește o structură de mulțimi disjuncte pentru un graf de m muchii și n vârfuri
- sortarea muchiilor: $O(m \log m) = O(m \log n)$
- inițializarea celor n mulțimi disjuncte: $O(n)$
- cel mult $2m$ operații find() și n operații merge(): $O(m \log n)$

- ordinul total: $O(m \log n)$
- dacă arborele este găsit rapid, putem folosi un min-heap
- avantajul este că nu va trebui să sortăm și muchiile ce nu vor fi folosite

Algoritmul lui Prim

- la fiecare pas, mulțimea de muchii selectată nu mai formează o pădure, ci un arbore parțial de cost minim pentru vârfurile ce le conectează
- arborele crește în mod natural cu câte o ramură
- fiecare ramură nouă leagă câte un vârf exterior la arborele existent
- se numerotează vârfurile din V de la $1 \ldots n = |V|$
- matricea $C[i, j]$ dă costul muchiei ce unește i de j, $C[i, j] = +\infty$ dacă acea muchie nu există
- $vecin[i]$ va da vârful din afara arborelui care se conectează la arbore printr-o muchie de cost minim
- $mincost[i]$ exprimă acest cost
- se pornește cu construcția arborelui pornind de la vârful 1

Algoritmul lui Prim

```
function Prim(C[1..n, 1..n])
 1: A ← ∅
 2: for i ← 2 to n do
 3:     vecin[i] ← 1
 4:     mincost[i] ← C[i, 1]
 5: end for
 6: for n − 1 times do
 7:     min ← +∞
 8:     for j ← 2 to n do
 9:         if 0 < mincost[j] < min then
10:             min ← mincost[j]; k ← j
11:         end if
12:     end for
13:     A ← A ∪ {{k, vecin[k]}}
14:     mincost[k] ← −1
15:     for j ← 2 to n do
16:         if C[k, j] < mincost[j] then
17:             mincost[j] ← C[k, j]; vecin[j] ← k
18:         end if
19:     end for
20: end for
21: return A
```

Algoritmul lui Prim

- ordinul de timp este în $O(n^2)$ (de ce?)
- pentru un graf dens, m se apropie de $n(n-1)/2$, algoritmul lui Kruskal necesită un timp în $O(n^2 \log n)$, Prim este mai eficient
- pentru un graf rar, m se apropie de n, Kruskal este mai eficient

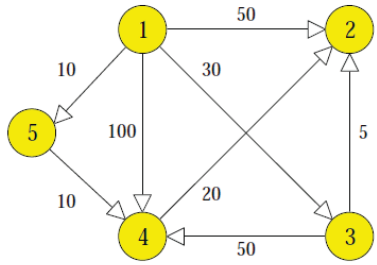
Cele mai scurte drumuri care pleacă din același punct

- pentru un graf orientat $G = (V, M)$, un vârf este desemnat ca **sursă**
- problema: determinarea celor mai scurte drumuri de la sursă către fiecare vârf
- cu C se notează mulțimea vârfurilor ce mai sunt disponibile (candidații)
- vârfurile sunt numerotate, $V = \{1, 2, \ldots, n\}$, vârful 1 este sursa
- matricea $L[i, j]$ dă lungimea fiecărei muchii, $L[i, j] = +\infty$ dacă muchia nu există
- la fiecare pas, se selectează un vârf și se introduce în mulțimea S (la început, S conține doar vârful sursă)
- un drum ce are toate vârfurile în S se numește **drum special**
- rezultatul se construiește în tabloul $D[2 \ldots n]$, care dă lungimea celui mai scurt drum de la sursă la vârful considerat

Algoritmul lui Dijkstra

- la fiecare pas al algoritmului, tabloul D conține lungimea celui mai scurt drum de la sursă la fiecare vârf
- se selectează vârful, neselectat anterior, care e caracterizat de cel mai scurt drum de la sursă până la el
- se adaugă vârful la S
- folosind acest vârf ca intermediar, se încearcă găsirea unei rute mai ieftine
- la fiecare pas, D va conține valorile drumurilor speciale de la sursă la fiecare vârf (drumuri construite numai cu vârfuri din S)
- la terminarea algoritmului, toate vârfurile, cu o singură excepție, sunt în S

Algoritmul lui Dijkstra



Pasul	v	C	D
initializare	—	{2, 3, 4, 5}	[50, 30, 100, 10]
1	5	{2, 3, 4}	[50, 30, 20, 10]
2	4	{2, 3}	[40, 30, 20, 10]
3	3	{2}	[35, 30, 20, 10]

Algoritmul lui Dijkstra

function Dijkstra(L[1..n, 1..n])

1: $C \leftarrow \{2, 3, \ldots, n\}$
2: **for** $i \leftarrow 2$ to n **do**
3: $\quad D[i] \leftarrow L[i, 1]$; $P[i] \leftarrow 1$
4: **end for**
5: **for** $n - 2$ times **do**
6: $\quad v \leftarrow$ vârful din C care minimizează $D[v]$
7: $\quad C \leftarrow C - \{v\}$
8: $\quad$ **for** fiecare $w \in C$ **do**
9: $\quad\quad$ **if** $D[w] > D[v] + L[v, w]$ **then**
10: $\quad\quad\quad D[w] \leftarrow D[v] + L[v, w]$; $P[w] \leftarrow v$
11: $\quad\quad$ **end if**
12: $\quad$ **end for**
13: **end for**
14: **return** D

Algoritmul lui Dijkstra

- $P[i]$ va conține vârful care se află înaintea ajungerii în i pe drumul de la sursă la i
- ordinul de timp al algoritmului?
- se poate îmbunătăți ordinul de timp dacă păstrăm vârfurile într-un heap $(v, D[v])$, iar graful este memorat sub formă de liste de adiacență (Dijkstra-modificat)
  1. extragerea din min-heap a unui vârf se face în $O(\log n)$
  2. se vor inspecta doar muchiile adiacente vârfului respectiv, deci în total se vor inspecta m muchii
  3. în caz că găsim un cost mai mic, va trebui inserat în min-heap
  4. Dijkstra-modificat necesită un timp în $O(m \log n)$
  5. pentru un graf rar / dens, care algoritm este mai rapid, Dijkstra sau Dijkstra-modificat?

Tehnica divide et impera I

- tehnica 'divide și stăpânește'
  1. descompune problema în subcazuri de mărime mai mică
  2. rezolvă independent fiecare subcaz
  3. recompune subsoluțiile pentru a construi soluția cazului inițial
- **exemplu:** un algoritm A pătratic, cu $t_A(n) \leq cn^2$
- prima variantă, algoritmul B împarte cazul inițial în 3 subcazuri de mărime $\lceil n/2 \rceil$, pe care le rezolvă cu algoritmul A, apoi recompune soluția în timpul dn

$$\begin{aligned} t_B(n) &= 3t_A(\lceil n/2 \rceil) + t(n) \leq 3c((n+1)/2)^2 + dn \\ &= 3/4cn^2 + (3/2 + d)n + 3/4c \end{aligned}$$

- algoritmul este cu 25% mai rapid

Tehnica divide et impera II

- a doua variantă, algoritmul C care continuă recursiv descompunerea subcazurilor

$$t_C(n) = \begin{cases} t_A(n) & \text{pentru } n \leq n_0 \\ 3t_C(\lceil n/2 \rceil) + t(n) & \text{pentru } n > n_0 \end{cases}$$

- conform notației asimptotice condiționate, timpul $t_C(\cdot)$ este în $O(n^{log\ 3}) \simeq O(n^{1.59})$
- descompunerea cazurilor suficient de mici nu mai aduce nici un câștig de performanță, din contra; sub un n_0 se apelează algoritmul A
- recursivitatea se poate uneori elimina printr-un ciclu iterativ

Căutarea binară I

- algoritmul după care se caută un cuvânt în dicționar
- problema: șirul $T[1 \ldots n]$, găsirea poziției lui x în T sau a poziției unde poate fi inserat
- în cazul cel mai nefaforabil, $\Theta(n)$ **function** sequential(T[1..n], x)

```
1: for i ← 1 to n do
2:     if T[i] > x then
3:         return i − 1
4:     end if
5: end for
6: return n
```

- bucla for se execută de $(n^2 + 3n - 2)/2n$ ori - de ce? (indicație: ne gândim probabilistic)

Căutarea binară II

- numărul de repetări ale buclei, în medie:

$$\frac{1}{n} \cdot \frac{2 + 3 + \cdots + n + n}{n}$$

function binrec(T[i..j], x)

```
1: if i = j then
2:     return i
3: end if
4: k ← (i + j + 1) div 2
5: if x < T[k] then
6:     return binrec(T[i ... k − 1], x)
7: else
8:     return binrec(T[k ... j], x)
9: end if
```

- ordinul de timp al algoritmului recursiv este $\Theta(log\ n)$ (de ce ?)

Căutarea binară III

- algoritmul iterativ:
  function iterbin(T[1..n], x)

```
1: if n = 0 or x < T[1] then
2:     return 0
3: end if
4: i ← 1; j ← n
5: while i < j do
6:     {T[i] ≤ x < T[j + 1]}
7:     k ← (i + j + 1) div 2
8:     if x < T[k] then
9:         j ← k − 1
10:    else
11:        i ← k
12:    end if
13: end while
14: return i
```

- algoritmul poate fi modificat așa încât ordinul de timp în cazul cel mai favorabil să fie $\Theta(1)$ (cum?)

Mergesort I

- dorim să sortăm crescător tabloul $T[1 \ldots n]$
- înjumătățim tabloul, sortăm recursiv jumătățile, apoi interclasăm rezultatele
  procedure mergesort(T[1..n])

```
1: if n este mic then
2:     sortează prin inserție tabloul T
3: else
4:     arrays U[1 .. n div 2], V[1..(n+1) div 2]
5:     U ← T[1 .. n div 2]
6:     V ← T[1 + n div 2 .. n]
7:     mergesort(U); mergesort(V)
8:     merge(T, U, V)
9: end if
```

- mărimea stivei de apel este în $O(\log n)$

Mergesort II

- pentru a sorta $n = 2^k$ elemente, spațiul de memorie este

$$2 \cdot (2^{k-1} + 2^{k-2} + \cdots + 2 + 1) = 2 \cdot 2^k = 2n$$

- ordinul timpului de execuție este
  $t(n) \in t(\lfloor n/2 \rfloor + \lceil n/2 \rceil) + \Theta(n) = \Theta(n \log n)$
- dacă în algoritmul mergesort, mărimea cazurilor este puternic dezechilibrată, U cu $n-1$ elemente și V cu un element, timpul devine $\Theta(n^2)$ (de ce?)

Quicksort I

- partea nerecursivă a algoritmului este dedicată construirii subcazurilor și nu recombinării soluțiilor
  procedure quicksort(T[i..j])

```
1: if j − i este mic then
2:     sortează T prin inserție
3: else
4:     pivot(T[i..j])
5:     {după pivotare, avem:}
6:     {i ≤ k < l ⇒ T[k] ≤ T[l]}
7:     {l < k ≤ j ⇒ T[k] > T[l]}
8:     quicksort(T[i .. l-1]); quicksort(T[l+1 .. j])
9: end if
```

- tabloul T se partiționează în două subtablouri, prin pivotare

Quicksort II

- numai elementul pivot se poziționează pe poziția lui finală în șir
- pentru echilibrarea subtablourilor, ideal ar fi să luăm elementul care ar fi pe a $\lceil n/2 \rceil$-a poziție dacă șirul ar fi sortat - **mediana**
- algoritmul de pivotare trebuie să meargă în timp liniar
- parcurge tabloul, pornind de la ambele capete

procedure pivot(T[i..j], l)
1: {la final, elementele T[i .. l-1] sunt $\leq p$}
2: {T[l] = p iar elementele T[l+1 .. j] sunt $> p$}
3: $p \leftarrow T[i]$
4: $k \leftarrow i$; $l \leftarrow j + 1$
5: **repeat**
6: $k \leftarrow k + 1$
7: **until** $T[k] > p$ or $k \geq j$
8: **repeat**
9: $l \leftarrow l - 1$
10: **until** $T[l] \leq p$
11: **while** $k < l$ **do**
12: interschimbă T[k] și T[l]
13: **repeat**

Quicksort III

14: $k \leftarrow k + 1$
15: **until** $T[k] > p$
16: **repeat**
17: $l \leftarrow l - 1$
18: **until** $T[l] \leq p$
19: **end while**
20: {pivotul este mutat pe poziția lui finală}
21: interschimbă T[i] și T[l]

- algoritmul quicksort este ineficient dacă cele două subcazuri $T[i \ldots l-1]$ și $T[l+1 \ldots j]$ sunt puternic dezechilibrate (similar cu mergesort, reiese un timp $O(n^2)$)
- pentru cazul mediu, avem de sortat n elemente, iar fiecare permutare a lor este echiprobabilă
- apelarea procedurii pivot poate poziționa primul element pe oricare din cele n poziții cu probabilitatea $1/n$; timpul mediu este:

$$t(n) \in \Theta(n) + 1/n \sum_{l=1}^{n} (t(l-1) + t(n-l)) < dn + 2/n \sum_{i=0}^{n-1} t(i)$$

- vom considera, folosind metoda inducției constructive, că ordinul de timp căutat pentru cazuri $i < n$ este $t(i) < cn \log n$

Quicksort IV

$$\begin{aligned} dn + 2/n \sum_{i=0}^{n-1} t(i) &= dn + 2/n \sum_{i=0}^{n-1} (i \ log \ i) \\ &< dn + 2/n \sum_{i=0}^{n-1} (i \ log \ n) \\ &= dn + \frac{2}{n} \cdot \frac{n(n-1) \ log \ n}{2} \\ &= dn + (n-1) \ log \ n \in O(n \ log \ n) \end{aligned}$$

- constanta multipicativă este mai mică pentru quicksort față de heapsort sau mergesort

Selecția unui element dintr-un tablou I

- ne interesează un algoritm eficient pentru determinarea medianei, pe care să o folosim drept pivot
- mediana unui tablou $T[1 \ldots n]$ verifică relațiile:

$$\#\{i \in \{1, \ldots, n\} | \, T[i] < m\} < \lceil n/2 \rceil$$
$$\#\{i \in \{1, \ldots, n\} | \, T[i] < m\} \geq \lceil n/2 \rceil$$

sau, prima condiție poate fi scrisă ca

$$\#\{i \in \{1, \ldots, n\} | \, T[i] \geq m\} \geq \lfloor n/2 \rfloor$$

- dacă am sorta(!) tabloul și am extrage mediana, aceasta ar necesita un timp în $\Theta(n \ log \ n)$
- vom considera o problemă mai generală

Selecția unui element dintr-un tablou II

- problema găsirii celui de-al k-lea element al unui șir dacă acesta ar fi sortat crescător este găsirea elementului m:

$$\#\{i \in \{1, \ldots, n\} | T[i] < m\} < k$$
$$\#\{i \in \{1, \ldots, n\} | T[i] \leq m\} \geq k$$

- de exemplu, mediana este cel de-al $\lceil n/2 \rceil$-lea cel mai mic element al lui T

function selection(T[1..n], k) {găsește al k-lea cel mai mic element}

```
1: if n este mic then
2:     sortează T; return T[k]
3: end if
4: p ← un element pivot din T[1..n]
5: u ← #{i ∈ 1,...,n}|T[i] < p}
6: v ← #{i ∈ 1,...,n}|T[i] ≤ p}
7: if u ≥ k then
8:     array U[1..u]
9:     U ← elementele T mai mici decât p
10:    {cel de-al k-lea cel mai mic element al lui T}
11:    {este și cel de-al k-lea cel mai mic element al lui U}
```

Selecția unui element dintr-un tablou III

```
12:    return selection(U,k)
13: else if v ≥ k then
14:    return p
15: end if
16: {situația când u < k și v < k}
17: array V[1 .. n-v]
18: v ← elementele din T mai mari ca p
19: {cel de-al k-lea cel mai mic element al lui T este}
20: {și cel de-al (k-v)-lea cel mai mic element al lui V}
21: return selection(V, k-v)
```

- interesant este că avem nevoie de un pivot, iar pentru a fi eficient, acesta ar trebui să fie chiar mediana (cerc vicios)
- dacă găsirea pivotului ar fi o operație elementară, timpul pentru problema selecției ar fi liniar
- vom încerca găsirea unei aproximări bune pentru mediană (pivot)
- presupunem $n \geq 5$ și că pseudomed() dă mediana a exact cinci elemente

function pseudomed(T[1..n])

```
1: {găsește o aproximare a medianei lui T}
2: s ← n div 5
3: array S[1..s]
4: for i ← 1 to s do
```

Selecția unui element dintr-un tablou IV

5: $S[i] \leftarrow$ adhocmed5(T[5i-4 .. 5i]
6: **end for**
7: **return** selection(S, (s+1) div 2)

- considerînd m mediana tabloului S

$$\#\{i \in \{1, \ldots, s\} | S[i] \leq m\} \geq \lceil s/2 \rceil$$

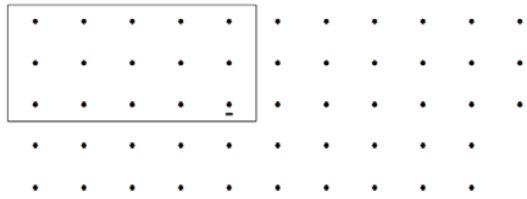
- cum fiecare element din S e o pseudomediană, și e mai mare sau egal decât 3 din cele 5 elemente, mediana pseudomedianelor va fi mai mare sau egală cu jumătate din aceste elemente

$$\#\{i \in \{1, \ldots, n\} | T[i] \leq m\} \geq 3\lceil \lfloor n/5 \rfloor /2 \rceil (3n - 12)/10$$

- similar, pentru numărul de elemente mai mari sau egale cu media, avem în final relațiile:

$$\#\{i \in \{1, \ldots, s\} | T[i] \leq m\} \geq (3n - 12)/10$$
$$\#\{i \in \{1, \ldots, s\} | T[i] < m\} < (7n + 27)/10$$

Selecția unui element dintr-un tablou V



- pentru n suficient de mare, U și V au cel mult $3m/4$ elemente fiecare, deci

$$t(n) \in O(n) + t(\lfloor n/5 \rfloor) + max\{t(i) | t \leq 3n/4\}$$

- prin inducție constructivă se arată că $t(n) \in O(n)$

O problemă de criptologie I

- problema constă în păstrarea secretului comunicației între A și B, chiar dacă E va intercepta toată conversația
- A și B convin asupra unui întreg p cu sute de cifre și asupra unui alt întreg g între 2 și $p-1$
- fiecare alege pentru el câte un întreg A, respectiv B
- A calculează $a = g^A \ mod \ p$ și anunță
- B calculează $b = g^B \ mod \ p$ și anunță
- A va calcula $x = b^A \ mod \ p$
- B va calcula $y = a^B \ mod \ p$
- vor ajunge la același rezultat, $x = y = g^{AB} \ mod \ p$, pe care E nu o cunoaște
- E cunoaște doar a, b, p, g
- pentru a afla x, ea va trebui să găsească un A' astfel ca $g^{A'} \ mod \ p = a$, după care poate calcula $x = x' = b^{A'} \ mod \ p$

O problemă de criptologie II

- calculul lui A' din a, p și g se numește **problema logaritmului discret**

function dlog(g, a, p)

1: $A \leftarrow 0$; $k \leftarrow 1$
2: **repeat**
3: $\quad A \leftarrow A + 1$
4: $\quad k \leftarrow kg$
5: **until** $(a = k \ mod \ p)$ or $(A = p)$
6: **return** A

- pentru un număr p cu 24 de cifre și presupunînd că o iterație se desfășoară într-o microsecundă, timpul mediu, pentru $p/2$ iterații, este mai mare decât 10^{17} secunde $>$ 100 milioane ani!
- algoritmul ce calculează exponențierea $g^A \ mod \ p$ în $O(A)$ nu este cu nimic mai eficient

O problemă de criptologie III

- putem profita de descompunerea exponentului în baza 2, astfel:

$$x^{25} = x^{11001_{(2)}} = (((x^1)^2 x^1)^2 x^0)^2 x^0)^2 x^1$$

- putem realiza un algoritm ce calculează exponențierea pornind de la ultima cifră binară a exponentului:

function dexpo(g, A, p)
1: **if** $A = 0$ **then**
2: **return** 1
3: **end if**
4: **if** A este impar **then**
5: $a \leftarrow$ dexpo(g, A - 1, p)
6: **return** (ag mod p)
7: **else**
8: $a \leftarrow$ dexpo(g, A div 2, p)
9: **return** (aa mod p)
10: **end if**

O problemă de criptologie IV

- dacă $M(p)$ este limita superioară a timpului necesar înmulțirii modulo p a două numere naturale mai mici decât p, dexpo(g, A, p) necesită un timp în $O(M(p)\ log\ A)$
- mai interesant este un algoritm iterativ, ce parcurge reprezentarea binară de la dreapta la stânga:

function dexpoiter(g, A, p)
1: $n \leftarrow A$; $y \leftarrow g$; $a \leftarrow 1$
2: **while** $n > 0$ **do**
3: **if** n este impar **then**
4: $a \leftarrow$ ay mod p
5: **end if**
6: $y \leftarrow$ yy mod p
7: $n \leftarrow$ n div 2
8: **end while**
9: **return** a

Algoritmul lui Strassen pentru înmulțirea matricilor I

- pentru matricile A și B de n elemente, matricea produs necesită n^3 înmulțiri și $(n-1)n^2$ adunări scalare, timp total $\Theta(n^3)$
- presupunem n putere a lui 2; se pot partiționa A și B în patru submatrici de $n/2 \times n/2$ elemente
- matricea produs:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

- respectiv

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} & C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} & C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

- pentru a obține matricea C este nevoie de 8 înmulțiri și 4 adunări de matrici de $n/2 \times n/2$ elemente

Algoritmul lui Strassen pentru înmulțirea matricilor II

- timpul rezultat este $t(n) \in 8t(n/2) + \Theta(n^2)$
- conform teoremei master avem că $t(n) \in \Theta(n^3)$ (adică exact metoda clasică)
- este de dorit să avem mai puține înmulțiri, în dauna numărului de adunări

$$\begin{aligned} P &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ Q &= (A_{21} + A_{22})B_{11} \\ R &= A_{11}(B_{12} - B_{22}) \\ S &= A_{22}(B_{21} - B_{11}) \\ T &= (A_{11} + A_{12})B_{22} \\ U &= (A_{21} - A_{11})(B_{11} + B_{22}) \\ V &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

Algoritmul lui Strassen pentru înmulțirea matricilor III

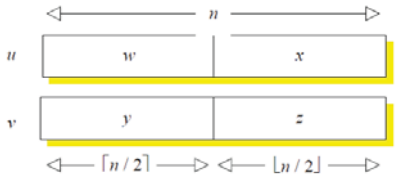
- există exact 36 de moduri diferite de calcul
- unde matricea rezultat se calculează cu:

$$C_{11} = P + S - T + V \qquad C_{12} = R + T$$
$$C_{21} = Q + S \qquad C_{22} = P + R - Q + U$$

- timpul total este $t(n) \in 7t(n/2) + \Theta(n^2)$
- similar, avem că $t(n) \in \Theta(n^{\log 7}) \in O(n^{2.81})$

Înmulțirea întregilor mari I

- numerele întregi mari se folosesc în diverse aplicații: criptologie, generarea numerelor Fibonacci, a zecimalelor lui π
- operații ce nu pot fi considerate elementare
- repezentarea în virgulă mobilă nu e o soluție viabilă
- **problema:** întregi u și v de sute de cifre (n)



- dacă $s = \lfloor n/2 \rfloor$, $u = 10^s w + x$, $v = 10^s y + z$, unde $0 \leq x < 10^s$, $0 \leq z < 10^s$
- astfel, $uv = 10^{2s} wy + 10^s(wz + xy) + xz$
- obținem algoritmul:

Înmulțirea întregilor mari II

function înmulțire(u, v)

1: $n \leftarrow$ numărul de cifre maxim dintre u și v
2: **if** n este mic **then**
3: **return** produsul uv calculat în mod clasic
4: **end if**
5: $s \leftarrow$ n div 2
6: $w \leftarrow u \; div \; 10^s$; $x \leftarrow u \; mod \; 10^s$
7: $y \leftarrow v \; div \; 10^s$; $z \leftarrow v \; mod \; 10^s$
8: **return** înmulțire(w, y)$\times 10^{2s}$
9: $+$ (înmulțire(w, z) $+$ înmulțire(x, y))$\times 10^s$
10: $+$ înmulțire(x, z))

- ordinul de timp este $t(n) \in 3t(\lceil n/2 \rceil) + t(\lfloor n/2 \rfloor) + \Theta(n)$
- pentru n putere a lui 2, $t(n) \in 4t(n/2) + \Theta(n) \in \Theta(n^2)$
- cât este ordinul de timp al înmulțirii clasice?

Înmulțirea întregilor mari III

- folosim ideea algoritmului lui Strassen:
  function înmulțire1(u, v)

1: $n \leftarrow$ numărul de cifre maxim dintre u și v
2: **if** n este mic **then**
3: **return** produsul uv calculat în mod clasic
4: **end if**
5: $s \leftarrow$ n div 2
6: $w \leftarrow u \; div \; 10^s$; $x \leftarrow u \; mod \; 10^s$
7: $y \leftarrow v \; div \; 10^s$; $z \leftarrow v \; mod \; 10^s$
8: $r \leftarrow$ înmulțire(w $+$ x, y $+$ z)
9: $p \leftarrow$ înmulțire(w, y); $q \leftarrow$ înmulțire(x, z)
10: **return** $10^{2s}p + 10^s(r - p - q) + q$

Înmulțirea întregilor mari IV

- ordinul de timp se reduce:

$$\begin{aligned} t(n) &\in t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + t(\lceil n/2 \rceil + 1) + O(n) \\ &\in 3t(1 + \lceil n/2 \rceil) + O(n) \end{aligned}$$

- pentru n putere a lui 2, $T(n) \in 3T(n/2) + O(n)$, de unde, cu ajutorul notației asimptotice condiționate:

$$t(n) \in O(n^{log\ 3}) \in O(n^{1.59})$$

Programarea dinamică

- asemenea 'divide and conquer', rezolvă problema combinînd soluțiile unor subprobleme
- în cazul programării dinamice, rezolvarea presupune subsoluții identice ce trebuie rezolvate în mod repetat
- optimizarea vine să elimine calcularea în mod repetat a acelorași subprobleme
- principii
  1. caracterizarea unei soluții optime
  2. definirea recursivă a valorii acesteia
  3. calculul valorii unei soluții optime în manieră 'bottom-up'
  4. construirea unei soluții din informația calculată
- un exemplu este triunghiul lui Pascal (ce calculează, cum)

Înmulțirea unui șir de matrice I

- problema constă în calculul produsului matricial $M = M_1 M_2 M_3 \dots M_n$
- înmulțirea se poate face grupînd matricile în mai multe moduri
- produsul AB, unde A are $p \times q$ iar B are $q \times r$ elemente necesită pqr înmulțiri scalare
- exemplu: A de 13×5, B de 5×89, C de 89×3 și D de 3×34

$(((AB)C)D)$	10582	înmulțiri
$((AB)(CD))$	54201	înmulțiri
$((A(BC))D)$	2856	înmulțiri
$(A((BC)D))$	4055	înmulțiri
$(A(B(CD)))$	26418	înmulțiri

- cea mai eficientă metodă este de aproape 19 ori mai rapidă decât cea mai ineficientă

Înmulțirea unui șir de matrice II

- fie $T(n)$ numărul de moduri în care se poate paranteza complet produsul de n matrici
- dacă facem o tăietură între a i-a și a $(i+1)$-a matrice

$$M = (M_1 M_2 M_3 \dots M_i)(M_{i+1} M_{i+2} \dots M_n)$$

- vor fi $T(i)$ moduri de parantezare a termenului stâng și $T(n-i)$ moduri de parantezare a celui drept; numărul de parantezări se poate exprima ca:

$$T(n) = \sum_{i=1}^{n-1} T(i)\,T(n-i)$$

Înmulțirea unui șir de matrice III

- șirul $T(1) = 1,\ T(2) = 1,\ T(3) = 2,\ T(4) = 5,\ T(5) = 14,$ $T(10) = 4862,\ T(15) = 2674440$ formează numerele catalane, unde:

$$T(n) = \frac{1}{n}\left(\begin{array}{c} 2n-2 \\ n-1 \end{array} \right) \in \Omega(4^n/n^2)$$

- pentru îmbunătățirea metodei se aplică principiul optimalității, adică dacă cea mai bună parantezare a produsului $M_1 \ldots M_n$ este între i și $i+1$, adunci subprodusele $M_1 \ldots M_i$ și $M_{i+1} \ldots M_n$ se parantezează și ele în mod optim
- dimensiunile matricilor se păstrează în tabloul $d[0 \ldots n]$, unde $d[i-1]$ și $d[i]$ sunt numărul de linii, respectiv coloane ale lui M_i
- tabloul $m[1 \ldots n, 1 \ldots m]$ va da numărul minim de înmulțiri scalare

Înmulțirea unui șir de matrice IV

- $m[i,j]$ numărul minim de înmulțiri pentru construirea produsului $M_i \ldots M_j$
- construim treptat produse de s-1=1, 2, .. n matrici

$$\begin{aligned}
&s = 0 : m[i,i] = 0, i = 1 \ldots n \\
&s = 1 : m[i,i+1] = d[i-1]d[i]d[i+1], i = 1 \ldots n-1 \\
1 < s < n : &m[i,i+s] = \\
&\min_{i \le k < i+s} (m[i,k] + m[k+1,i+s] + d[i-1]d[k]d[i+s]), \\
&i = 1 \ldots n-s
\end{aligned}$$

- practic, pentru produsul $M_i \ldots M_{i+s}$, se calculează toate cele $s-1$ parantezări posibile:

$$(M_i M_{i+1} \ldots M_k)(M_{k+1} \ldots M_{i+s})$$

Înmulțirea unui șir de matrice V

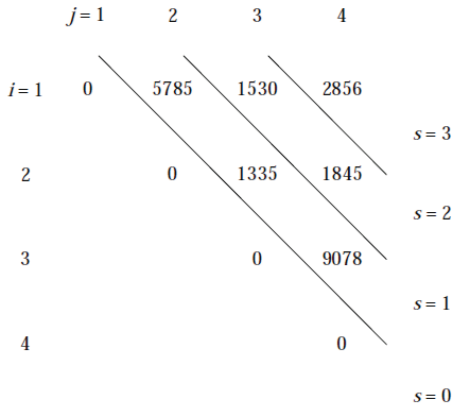
- ca exemplu, $d = (13, 5, 89, 3, 34)$
- pentru $s = 1$, $m[1,2] = 13 \times 5 \times 89 = 5785$, $m[2,3] = 1335$, $m[3,4] = 9078$
- pentru $s = 2$,

$$\begin{aligned} m[1,3] &= min(m[1,1] + m[2,3] + d[0]d[1]d[3], m[1,2] + m[3,3] + d[0]d[2]d[3]) \\ &= min(1530, 9256) = 1530 \\ m[2,4] &= min(m[2,2] + m[3,4] + d[1]d[2]d[4], m[2,3] + m[4,4] + d[1]d[3]d[4]) \\ &= min(24208, 1845) = 1845 \end{aligned}$$

- pentru $s = 3$,

$$\begin{aligned} m[1,4] = min(&m[1,1] + m[2,4] + d[0]d[1]d[4] \\ &+ m[1,2] + m[3,4] + d[0]d[2]d[4] \\ &+ m[1,3] + m[4,4] + d[0]d[3]d[4]) = \\ &= min(4055, 54201, 2856) = 2856 \end{aligned}$$

Înmulțirea unui șir de matrice VI



- timpul exact de execuție însumează elementele de pe toate diagonalele

Înmulțirea unui șir de matrice VII

$$t(n) = \sum_{s=1}^{n-1}(n-s)s = n\sum_{s=1}^{n-1} s - \sum_{s=1}^{n-1} s^2$$
$$= (n_3 - n)/6 \in \Theta(n^3)$$

- s-au propus algoritmi care rezolvă problema în $O(n \log n)$
- valoarea tăieturii k pentru fiecare pereche (i, j) se păstrează în $r[i, j]$
- se vor construi pas cu pas matricile m și r

procedure minscal(d[0..n])
1: **for** $i \leftarrow 1$ to n **do**
2: $m[i, i] \leftarrow 0$
3: **end for**
4: **for** $s \leftarrow 1$ to $n - 1$ **do**
5: **for** $i \leftarrow 1$ to $n - s$ **do**
6: $m[i, i + s] \leftarrow +\infty$
7: **for** $k \leftarrow i$ to $i + s$ **do**
8: $q \leftarrow m[i, k] + m[k + 1, i + s] + d[i - 1]d[k]d[i + s]$
9: **if** $q < m[i, i + s]$ **then**
10: $m[i, i + s] \leftarrow q$
11: $r[i, i + s] \leftarrow k$

Înmulțirea unui șir de matrice VIII

12: **end if**
13: **end for**
14: **end for**
15: **end for**

procedure minmat(i, j)
1: {returnează produsul matriceal $M_i M_{i+1} \dots M_j$}
2: {calculat prin $m[i, j]$ înmulțiri scalare}
3: {se presupune că $i \leq r[i, j] \leq j$}
4: **if** $i = j$ **then**
5: **return** M_i
6: **end if**
7: **arrays** U, V
8: $U \leftarrow minmat(i, r[i, j])$
9: $V \leftarrow minmat(r[i, j] + 1, j)$
10: **return** $produs(U, V)$

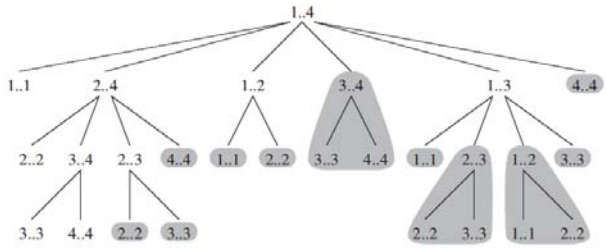
Elemente de programare dinamică I

- pentru a putea aplica programarea dinamică, problema trebuie să prezinte
  1. **substructură optimă:** o soluție a problemei include soluții optime ale subproblemelor
  2. **suprapunerea problemelor:** subcazurile se suprapun, se rezolvă aceleași subprobleme, nu se generează subprobleme noi
- de regulă, numărul de subprobleme distincte e dependent polinomial de mărimea cazului de intrare
- prin contrast, algoritmii divide et impera generează la fiecare etapă, probleme noi
- la programarea dinamică, reapariția problemei găsește rezultatul deja stocat
- formulată ca algoritm recursiv, problema găsirii numărului minim de înmulțiri recalculează subcazuri de mai multe ori: **procedure** matrice-recursiv(d, i, j)

Elemente de programare dinamică II

```
1: if i = j then
2:     return 0
3: end if
4: m[i, j] ← ∞
5: for k ← i to j − 1 do
6:     q ← matrice-recursiv(d, i, k) + matrice-recursiv(d,
       k+1, j) + d[i-1]d[k]d[j]
7:     if q < m[i, j] then
8:         m[i, j] ← q
9:     end if
10: end for
```

Elemente de programare dinamică III



- de câte ori se calculează subcazul $m[3,4]$?
- timpul de execuție:

$$T(1) \geq 1$$
$$T(n) \geq 1 + \sum_{k=1}^{n-1}(T(k) + T(n-k) + 1), \text{ pentru } n > 1$$

Elemente de programare dinamică IV

- prin inducție matematică, vrem să ajungem la $T(n) = \Omega(2^n)$, mai exact presupunem că $T(n) \geq 2^{n-1}$ pentru $n \geq 1$

$$\begin{aligned} T(n) &\geq 2\sum_{k=1}^{n-1} T(k) + n \geq 2\sum_{k=1}^{n-1} 2^{k-1} + n \\ &= 2\sum_{k=0}^{n-2} 2^k + n = 2(2^{n-1} - 1 + n) = (2^n - 2) + n \\ &\geq 2^{n-1} \end{aligned}$$

- apelul top-down recursiv-matrice(d, 1, n) este cel puțin exponențial în n, față de algoritmul bottom-up care are un ordin de timp $\Theta(n^3)$
- **memoizarea:** prevenirea calculării repetate a subproblemelor care au mai apărut ulterior, prin stocarea acestora

Elemente de programare dinamică V

procedure matrice-memoizat(d)

```
1: for i ← 1 to n do
2:     for i ← 1 to n do
3:         m[i, j] ← ∞
4:     end for
5: end for
6: return matrice-echivalent(d, 1, n)
```

procedure matrice-echivalent(d, 1, n)

```
1: if m[i, j] < ∞ then
2:     return m[i, j]
3: end if
4: if i = j then
5:     m[i, j] ← 0
6: else
7:     for k ← i to j − 1 do
```

Elemente de programare dinamică VI

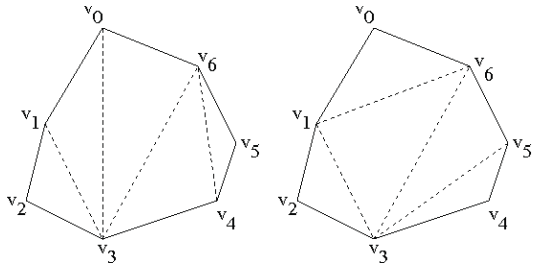
```
8:         q ← matrice-echivalent(d, i, k) + matrice-echivalent(d, k+1, j)
           + d[i-1]d[k]d[j]
9:         if q < m[i, j] then
10:            m[i, j] ← q
11:        end if
12:    end for
13: end if
14: return m[i, j]
```

- algoritmul memoizat economisește timp față de cel recursiv, arborele strategiei apare retezat; memoizarea transformă algoritmul parantezării produselor de matrici din $\Omega(2^n)$ în $O(n^3)$
- fiecare din cele $\Theta(n^2)$ elemente ale tabloului m este inițializat o singură dată; fiecare apel presupune calcularea unui minim în $O(n)$
- soluția iterativă bottom-up îmbunătățește cu un factor constant timpul algoritmului memoizat;
- totuși, dacă prin natura problemei, nu este necesară rezolvarea tuturor subcazurilor, soluția recursivă cu memoizare este de preferat

Triangularea optimă a poligoanelor I

- **poligon:** curbă închisă, plană, formată dintr-o secvență de segmente de dreaptă (laturi)
- presupunem că poligonul este simplu, laturile nu se intersectează
- **poligon convex:** oricare ar fi două puncte x, y de pe frontieră sau din interiorul său, toate punctele segmentului $[x, y]$ sunt în interior sau pe frontieră
- $\overline{v_0v_1}, \overline{v_1v_2}, \ldots$ - laturi
- v_i ,v_j vârfuri neconsecutive, $\overline{v_iv_j}$ - coardă
- o coardă $\overline{v_iv_j}$ împarte poligonul în două poligoane, $< v_i, v_{i+1}, \ldots, v_j >$ și $< v_j, v_{j+1}, \ldots, v_i >$
- **triangulare:** mulțime T de coarde ce îl împart în triunghiuri disjuncte; coardele nu se intersectează

Triangularea optimă a poligoanelor II



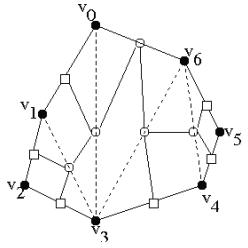
- mulțimea T de coarde este maximală, adică orice coardă care nu aparține mulțimii T va intersecta o coardă din T
- fiecare triangulare a unui poligon cu n vârfuri are $n-3$ coarde și împarte poligonul în $n-2$ triunghiuri

Triangularea optimă a poligoanelor III

- pe fiecare triunghi se poate defini o funcție de ponderare:

$$w(\triangle v_i v_j v_k) = |v_i v_j| + |v_j v_k| + |v_k v_i|$$

- **problema triangulării optime:** găsirea triangulării care minimizează suma ponderilor triunghiurilor



- o triangulare poate fi reprezentată sub forma unui arbore binar
- latura $\overline{v_0 v_6}$ este rădăcina

Triangularea optimă a poligoanelor IV

- $\triangle v_0 v_3 v_6$ determină fiii rădăcinii - $\overline{v_0 v_3}$ și $\overline{v_3 v_6}$
- poligoanele rezultate se împart recursiv
- arborele construit asociat triangulării corespunde exact parantezării produsului a $n-1$ matrici $A_1 A_2 \ldots A_{n-1}$
- cum se poate modifica algoritmul minimizării numărului de înmulțiri așa încât el să calculeze triangularea optimă?

Comparație: programarea dinamică vs. tehnica greedy I

- privesc soluția ca rezultatul unei secvențe de decizii
- ambele se pot conforma principiului optimalității
- **problema rucsacului:** un hoț găsește n obiecte, fiecare cu valoarea v_i și greutatea g_i; greutatea maximă din rucsac este G
- maximizarea funcției obiectiv:

$$V(x) = \sum_{i=1}^{n} x_i v_i, \quad \text{unde} \sum_{i=1}^{n} x_i g_i \leq G,\ x = (x_1, x_2, \ldots, x_n)$$

- problema continuă a rucsacului: $0 \leq x_i \leq 1$, cantitate fracționară
- problema discretă $(0/1)$ a rucsacului: $x_i \in \{0, 1\}$

- **problema continuă a rucsacului**
- maximizarea funcției obiectiv printr-o succesiune de decizii

Comparație: programarea dinamică vs. tehnica greedy II

- decizie asupra lui x_1, apoi x_2, ...
- o soluție greedy se obține, realizînd ordonarea, fără restrângerea generalității:

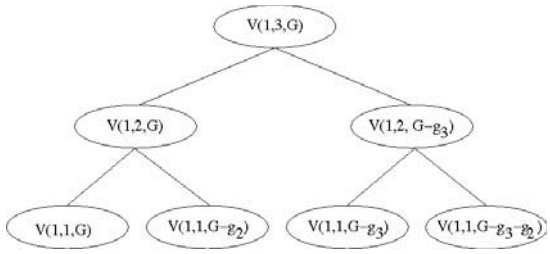
$$v_1/g_1 \geq v_2/g_2 \geq \ldots v_n/g_n$$

- soluția optimă este $x^* = (1, \ldots, 1, x_k^*, 0, \ldots, 0)$, unde $1 \leq k \leq n$, iar $0 \leq x_k \leq 1$
- nici o decizie nu este eronată
- timpul este în $O(n)$
- secvența de decizii de tip optime locale duce la optimul global - se respectă **principiul optimalității**

- **problema discretă a rucsacului**
- exemplu: $g = (1, 2, 3)$, $v = (6, 10, 12)$, $G = 5$
- greedy furnizează soluția $V(1, 1, 0) = 16$

Comparație: programarea dinamică vs. tehnica greedy III

- soluția optimă este $V(0,1,1) = 22$
- nu întotdeauna o succesiune de decizii optime duce la optimul global
- metoda greedy exporează o singură secvență de decizii
- programarea dinamică va trebui să exploreze toate subsecvențele de decizii posibile
- notăm cu $V(1,n,G)$ valoarea maximă ce se poate încărca în rucsac inițial; exprimăm apoi soluția ca alegere între alternativa cu și fără g_n



Comparație: programarea dinamică vs. tehnica greedy IV

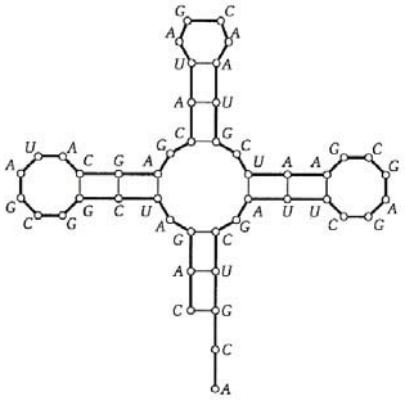
$$V(1,n,G) = max\{V(1,n-1,G), V(1,n-1,G-g_n) + v_n\}$$

- explorarea completă a arborelui de decizie dă un algoritm exponențial în $O(2^n)$
- de cele mai multe ori algoritmii de programare dinamică ce exploatează principiul optimalității sunt polinomiali

Programarea dinamică pe intervale I

- problema discretă a rucsacului presupune rezolvarea cazului $\{1, 2, \ldots, j\}$ pe baza subcazurilor $\{1, 2, \ldots, j-1\}$
- pentru alte probleme de programare dinamică nu putem găsi o recurență pentru subsetul de probleme generat de $\{1, 2, \ldots, j\}$
- inspectînd setul mai mare de subprobleme $\{i, i+1, \ldots, j\}$ pentru toate valorile $\imath \leq j$, putem găsi o relație naturală de recurență între aceste subprobleme
- noua variabilă introdusă i ne va determina să considerăm subprobleme pentru fiecare interval din $\{1, 2, \ldots, n\}$
- **problema:** predicția structurii secundare a moleculei ARN, problemă importantă în biologia computațională
- structura dublu-helix a ADN dată de legăturile de tip pereche între baze complementare, $\{A, C, G, T\}$[1]

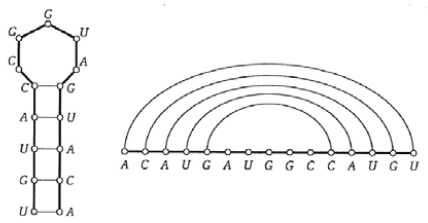
Programarea dinamică pe intervale II



- perechile sunt A-T și C-G
- lanțul simplu al moleculei ARN este componenta cheie în multe procese ce se petrec în cadrul celulei
- molecula ARN tinde să se 'încolăcească' în jurul ei înseși

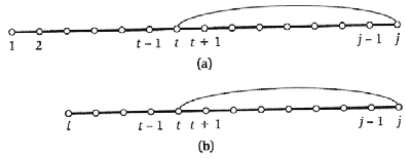
Programarea dinamică pe intervale III

- **structura secundară** formată de molecula ARN determină comportamente specifice în procesele celulare
- lanțul simplu de baze $b_i \in \{A, C, G, U\}^2$ poate fi repezentat ca $B = \{b_1, b_2, \ldots, b_n\}$
- perechile sunt A-U și C-G
- o bază poate face pereche doar cu o singură altă bază din lanț
- nu avem 'noduri', legăturile nu se pot întretăia
- regulile Watson-Crick ale formării perechilor



Programarea dinamică pe intervale IV

- o **structură secundară** a lanțului B este setul de perechi $S = \{(i, j)\}$ cu $j \in \{1, 2, \ldots, n\}$ ce satisface condițiile:
  1. (nu există întoarceri rapide) capetele fiecărei perechi din secvență sunt separate de cel puțin alte patru baze, adică pentru $(i, j) \in S$ atunci $i < j - 4$
  2. perechile nu pot fi decât $\{A, U\}$ sau $\{C, G\}$ (în orice ordine)
  3. nici o bază nu apare decât în maxim o pereche
  4. (condiția de ne-întretăiere) dacă (i, j) și (k, l) sunt două perechi în S, atunci nu putem avea $i < k < j < l$



- structura secundară ce se va forma va fi una cu energie minimă, adică una ce maximizează numărul de perechi

Programarea dinamică pe intervale V

- problema de programare dinamică va formula soluția optimă $OPT(n)$ = numărul maxim de perechi ce se pot forma, unde $OPT(j) = 0$ pentru $j \leq 5$
- structura optimală $b_1 b_2 \dots b_j$ este determinată fie:
  - j nu face parte dintr-o pereche, vom inspecta $OPT(j-1)$
  - j este pereche cu un anume t, $t < j-4$
- al doilea caz, din cauza condițiilor de ne-întretăiere, știm că nu putem avea perechi cu un capăt între $1 \dots t-1$ iar celălalt între $t+1 \dots j-1$
- se separă subproblemele $\{b_1 \dots b_{t-1}\}$ și $\{b_{t+1} \dots b_{j-1}\}$
- vom considera problema $\{b_i, b_{i+1}, \dots, b_j\}$, pentru $i \leq j$
- calculăm numărul maxim de perechi $OPT(i,j)$
- condiția de ne-întretăiere: $OPT(i,j) = 0$ pentru $i \geq j-4$
- dacă j nu face pereche, $OPT(i,j) = OPT(i,j-1)$

Programarea dinamică pe intervale VI

- dacă i face pereche cu $t < j-4$, vom avea de rezolvat $OPT(i,t-1)$ și $OPT(t+1,j-1)$

$$OPT(i,j) = max(OPT(i,j-1), \\ max(1 + OPT(i,t-1) + OPT(t+1,j-1)))$$

unde b_t face pereche cu b_j

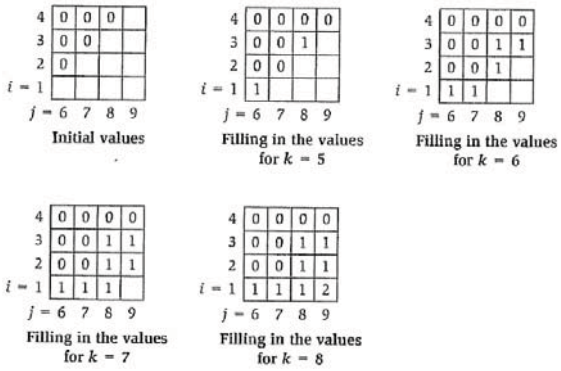
function RNA-secondary-structure(1, n)

1: $OPT(i,j) \leftarrow 0$ for all $i \geq j-4$
2: {k dă lungimea secvenței, pornind de la cea mai scurtă posibilă (5) }
3: **for** $k \leftarrow 5$ to $n-1$ **do**
4: **for** $i \leftarrow 1$ to $n-k$ **do**
5: $j \leftarrow i+k$
6: $m \leftarrow -\infty$
7: **for** all $t \leftarrow i$ to $j-4$ where b_t pairs with b_j **do**
8: $o \leftarrow 1 + OPT(i,t-1) + OPT(t+1,j-1)$

Programarea dinamică pe intervale VII

```
 9:          if m < o then
10:            m ← o
11:          end if
12:        end for
13:        OPT(i, j) ← max(OPT(i, j − 1), m)
14:      end for
15: end for
16: returns OPT(1, n)
```

Programarea dinamică pe intervale VIII



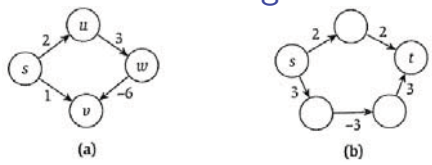
- considerăm secvența ACCGGUAGU; reprezentăm tabelar secvențe de lungime $k = 5, 6, \ldots$
- sunt $O(n^2)$ subprobleme de rezolvat, iar fiecare se rezolvă în $O(n)$; ordinul total este în $O(n^3)$
- pentru un subșir $\{i, j\}$ se poate memora valoarea optimă t găsită

[1] adeine, cytosine, guanine, thymine
[2] baza U a înlocuit pe T

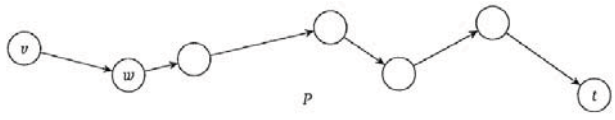
Cele mai scurte drumuri într-un graf I

- **problema:** pentru un graf orientat $G = (V, M)$, fiecare muchie $(i, j) \in M$ are costul c_{ij}
- algoritmul Dijkstra consideră că graful poate avea doar muchii pozitive; aici consierăm și posibilitatea costurilor negative
- algoritm flexibil, descentralizat
- numim **ciclu negativ** un ciclu C pentru care $\sum_{(i,j)\in C} c_{ij} < 0$
- ce se întâmplă dacă într-un graf cu cicluri negative căutăm un drum de cost minim?
- pentru un graf fără cicluri negative, căutăm să aflăm un drum P de la un nod sursă s la un nod destinație t astfel ca lungimea sa $\sum_{(i,j)\in P} c_{ij}$ să fie minimă

Cele mai scurte drumuri într-un graf II



- (a) ideea greedy de la Dijkstra, de a îngloba cea mai scurtă muchie pornind de la sursă, s-ar putea să nu funcționeze
- (b) o idee ar putea fi să folosim totuși Dijkstra pe un graf modificat, în care fiecare muchie să aibă un nou cost $c'_{ij} = c_{ij} + T$ unde T ales astfel încât $c_{ij} + T > 0$ pentru toate $(i, j) \in M$, nu dă rezultate; costurile drumurilor se modifică în mod diferit (exemplu: cresc cu 2T respectiv 3T)



Cele mai scurte drumuri într-un graf III

- **observație:** într-un graf fără cicluri negative, cel mai scurt drum de la s la t va fi un drum simplu (fără noduri care să se repete), și de aceea va avea cel mult $n-1$ muchii
- $OPT(i, v)$ - costul minim al drumului de la vârful v la vârful t folosind cel mult i muchii
- problema inițială: $O(n-1, s)$
- dacă drumul P folosește cel mult $i-1$ muchii, $OPT(i, v) = OPT(i-1, v)$
- dacă drumul P folosește i muchii, și prima muchie este (v, w), atunci $OPT(i, v) = c_{vw} + OPT(i-1, w)$

function Bellman-Ford-shortest-path(G, s, t)

1: $n \leftarrow |V|$
2: **array** $B[0 \ldots n-1]$
3: $B[0, t] \leftarrow 0$; $B[0, v] \leftarrow \infty$ for all other $v \in V$
4: {i dă lungimea drumului, și crește progresiv}

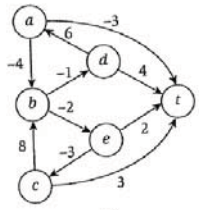
Cele mai scurte drumuri într-un graf IV

5: **for** $i \leftarrow 1$ to $n-1$ **do**
6: **for** $v \in V$ in any order **do**
7: $B[i, v] \leftarrow B[i-1, v]$
8: **for** $w \in V$ **do**
9: $o \leftarrow B[i-1, w] + c_{vw}$
10: **if** $o < B[i, v]$ **then**
11: $B[i, v] \leftarrow o$
12: **end if**
13: **end for**
14: **end for**
15: **end for**
16: **return** $B[n-1, s]$

- pentru drumuri cu lungimi din ce în ce mai mari, se calculează dacă drumul cu lungimea imediat următoare, folosind orice muchie nou inserată până la începutul unui drum vechi, este mai scurt
- procesul presupune calcularea drumurilor, în acest mod, pentru toate vârfurile

Cele mai scurte drumuri într-un graf V

- timpul este în $O(n^3)$
- pentru obținerea drumului, tabela cu vârful w ales la fiecare pas se poate calcula în paralel



(a)

	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4	-6	-6
b	∞	∞	0	-2	-2	-2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

(b)

- o linie din tabelă corespunde celui mai scurt drum de la nodul respectiv la t, pe măsură ce numărul de muchii permis crește

Cele mai scurte drumuri într-un graf VI

- cel mai scurt drum de la d la t este modificat de patru ori, se schimbă de la $d - t$ la $d - a - t$, la $d - a - b - e - t$, și în final la $d - a - b - e - c - t$

- o variantă a algoritmului se obține dacă, pentru un anumit vârf v, nu mai inspectăm legăturile sale cu toate celelalte vârfuri, ci ne limităm la vecinii săi
- astfel, se vor parcurge toate muchiile grafului
- ordinul de timp va fi în $O(nm)$
- strategia este avantajoasă pentru un graf rar
- se poate observa că, la fiecare pas, valoarea $B[i, v]$ se calculează folosind doar valorile calculate la pasul anterior, $B[i-1, w]$, pentru $w \in V$

Cele mai scurte drumuri într-un graf VII

- mărimea lui B se poate reduce de la $O(n^2)$ la $O(n)$, iar B devine vector:

$$B[v] = min(B[v], \min_{w \in V}(c_{vw} + B[w]))$$

- în algoritm, $B[v]$ este lungimea celui mai scurt drum de la v la t, iar după i actualizări, valoarea $B[v]$ nu este mai mare decât lungimea drumului folosind cel mult i muchii
- pentru regăsirea drumului de la s la t, vom adăuga un vector $first[v]$ ce va da nodul ce urmează după nodul v pe acest drum
- când se găsește $\min_{w \in V}(c_{vw} + B[w])$, $first[v] \leftarrow w$
- considerăm succesiunea de vârfuri v, $first[v]$, $first[first[v]]$, ... (drumul în sine)
- dovedim că pe acest drum, dacă are cicluri, atunci ele sunt negative
- fie $v_1, v_2, \ldots, v_k$ nodurile de-a lungul ciclului pe acest drum, și fie (v_k, v_1) ultima muchie adăugată

Cele mai scurte drumuri într-un graf VIII

- înainte de această ultimă adăugare, avem $B[v_i] \geq c_{v_i v_{i+1}} + B[v_{i+1}]$ pentru toți $i = 1, \ldots, k-1$ și de asemenea $B[v_k] > c_{v_k v_1} + B[v_1]$ deoarece am găsit o muchie care ne leagă de t cu un cost mai mic și urmează să modificăm $first[v_k]$ la v_1

$$\begin{aligned}
B[v_1] &\geq c_{v_1 v_2} + B[v_2] \\
B[v_2] &\geq c_{v_2 v_3} + B[v_3] \\
&\ldots \\
B[v_{k-1}] &\geq c_{v_{k-1} v_k} + B[v_k] \\
B[v_k] &> c_{v_k v_1} + B[v_1]
\end{aligned}$$

sumăm, și avem:

$$0 > \sum_{i=1}^{k-1} c_{v_i v_{i+1}} + c_{v_k v_1}, \text{ ceea ce este un ciclu negativ}$$

- cum graful nu admite cicluri negative, drumul format nu are cicluri

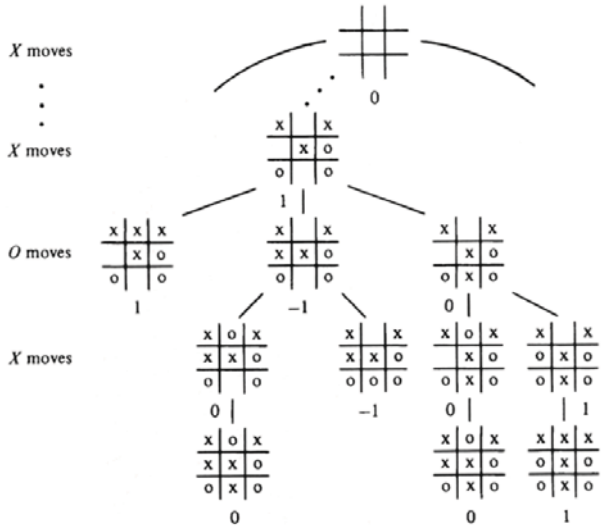
Cele mai scurte drumuri într-un graf IX

- la terminarea algoritmului, drumul este construit în $first[v]$, iar valoarea sa este determinată de $B[v]$
- în versiunea în care B este vector, e posibil ca drumul a cărui lungime este $B[v]$ după i iterații să aibă înglobate mai multe muchii decât i
- un exemplu ar fi dacă tot graful ar consta dintr-un singură succesiune de muchii de la s la t
- parcurgerea acestor muchii s-ar face integral, în for-ul după w (într-un singur pas)
- ar fi inutil să mai facem complet cele $n-1$ iterații mari
- putem observa că, dacă pentru un anumit i, vectorul $B[V]$ nu se modifică, atunci înseamnă că i este lungimea maximă a tuturor drumurilor
- $B[v]$ nu mai are cum să se modifice, pentru că se fac aceleași verificări ca și la iterația anterioară, cu aceleași valori

Backtracking I

- găsirea soluției optime folosind căutarea exhaustivă
- se generează în mod succesiv toate soluțiile posibile și se evaluează pe măsură ce ele sunt construite
- tehnica backtracking este denumită și **căutare cu revenire**, deoarece pornind de la soluția curentă se construiește următoarea soluție posibilă
- tehnică de explorare a grafurilor orientate implicite; acestea sunt de regulă arbori
- se pretează explorării arborilor corespunzînd strategiilor pentru anumite jocuri
- exemplu: strategia unui joc de șah, table, sau X-și-0
- jucătorii mută alternativ, iar starea jocului se reprezintă sub forma unei poziții pe o tablă
- jocul are o serie finită de astfel de stări (poziții) și există reguli care asigură sfârșitul jocului

Backtracking II



- rădăcina corespunde poziției de start

Backtracking III

- se poate asocia un **arbore de strategie**
- fiii unui nod reprezintă pozițiile permise pornind de la poziția curentă
- frunzelor (pozițiilor terminale) li se asociază un cost -1, 0 sau 1, corezpunzînd poziției de câștig, remiză sau pierdere
- se calculează valorile nodului prin propagare de la frunze spre rădăcină
  1. dacă nodul corespunde mutării jucătorului pentru care se consturiește arborele, valoarea nodului este maximul valorilor fiilor (presupunem că jucătorul face mutarea favorabilă lui)
  2. dacă nodul corespunde mutării celuilalt jucător, valoarea este minumul fiilor
- dacă putem calcula astfel valoarea rădăcinii ca fiind 1, spunem că primul jucător are o **strategie câștigătoare**

Funcții de tip payoff (răsplată)

- generalizează ideea ca, fiecărui nod din arborele de strategie să i se asocieze o valoare, nu neapărat în $\{-1, 0, 1\}$
- se păstrează regulile de determinare a valorii nodului:
  1. maxim, dacă urmează la mutare jucătorul 1
  2. minim, dacă urmează la mutare jucătorul 2
- jocul de șah presupune un arbore al strategiei, deși finit, cu un număr imens de noduri; stocarea sa este nefezabilă
- de regulă, pentru poziția curentă ca rădăcină, se construiesc câteva niveluri în jos
- 'frunzele' nu sunt poziții finale, așa încât se folosesc estimatori probabiliști, bazați pe diferențele în numărul și importanța pieselor, apărarea în jurul regelui

Căutarea backtracking

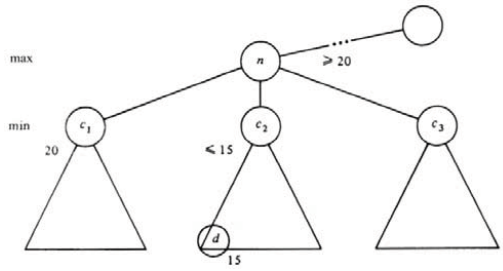
- stocarea arborelui strategiei poate fi prohibitivă
- este posibil uneori să nu stocăm decât calea directă de la rădăcină spre un anumit nod (sau frunză)
- 'calea' este realizată de un algoritm recursiv care produce nodurile doar atunci când este nevoie de ele
  1. valorile asociate sunt numere în intervalul $[-1, 1]$
  2. valoarea $+\infty$ este mai mare ca orice valoare pozitivă (la fel pentru $-\infty$)
  3. variabila *mode* ia valorile *MIN* sau *MAX* după cum la mutare urmează jucătorul 2 sau jucătorul 1
  4. există o reprezentare a sării curente (board)
  5. funcția *payoff*() calculează valoarea asociată unei frunze

Tehnica mini-max

function search(board, mode)
1: {evaluează un nod *board* presupunînd că este rîndul jucătorului 1 dacă mode = MAX sau al jucătorului 2 dacă mode = MIN }
2: **if** *board* este o frunză **then**
3: **return** payoff(board)
4: **end if**
5: **if** $mode = MAX$ **then**
6: $value \leftarrow -\infty$
7: **else**
8: $value \leftarrow +\infty$
9: **end if**
10: **for** fiecare fiu c al lui *board* **do**
11: **if** $mode = MAX$ **then**
12: $value \leftarrow \max(value, search(c, MIN))$
13: **else**
14: $value \leftarrow \min(value, search(c, MAX))$
15: **end if**
16: **end for**
17: **return** value

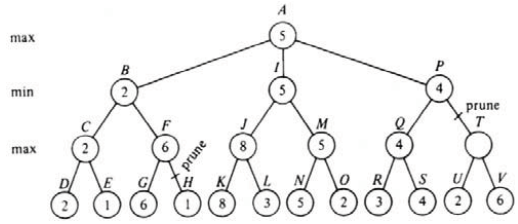
- variantă: program nerecursiv ce păstrează o stivă de stări (boards)

Retezarea alfa-beta I



- există situații în care nu e necesară continuarea calculării valorilor fiilor (în bucla for)
- pentru nodul n, s-a calculat valoarea fiului c_1 ca fiind 20
- unul din fiii lui c_2 este 15
- c_2 fiind un nod minim, nici unul din ceilalți fii ai săi nu mai are cum să determine creșterea minimului peste 15
- n fiind un nod de maxim, valoarea sa va fi peste 20
- deci ceilalți fii ai lui c_2 pot rămâne necalculați

Retezarea alfa-beta II

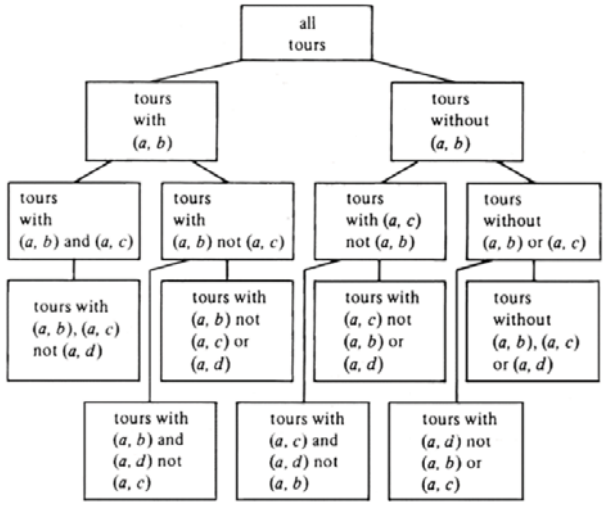


- reguli de retezare folosind valori **tentative** și **finale**:
  1. dacă toți fiii lui n au fost considerați sau retezați, valoarea tentativă a lui n devine finală
  2. dacă un max-nod n are o valoare tentativă v_1 și un fiu cu valoarea finală v_2, setează valoarea tentativă a lui n la $max(v_1, v_2)$. Dacă n este min-nod, setează valoarea sa tentativă la $min(v_1, v_2)$
  3. dacă p este un min-nod cu părintele q (max-nod), și p și q au valorile tentative v_1 și v_2, cu $v_1 \leq v_2$, atunci se pot reteza toți fiii neconsiderați ai lui p. Se pot reteza toți fiii neconsiderați ai lui p dacă el este un max-nod (și, corespunzător, q este min-nod) și $v_2 \leq v_1$

Căutarea branch-and-bound I

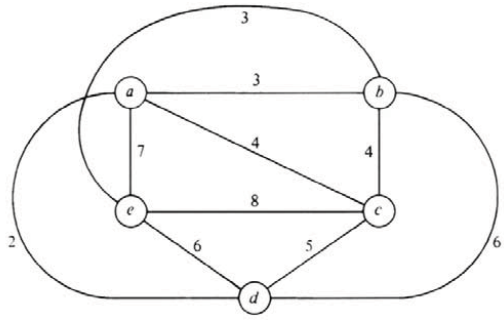
- în anumite probleme, nodurile arborelui strategiei pot fi văzute ca seturi de configurații
- printr-o selectare atentă, fiii unui nod pot reprezenta mult mai puține configurații decât părintele lor
- exemplu: TSP (travelling salesman problem), orașele $\{a, b, c, d, e\}$ poate găsi soluția optimă prin explorarea tuturor posibilelor configurații ale ciclului (câte sunt?)
- construim arborele configurațiilor, unde rădăcina reprezintă toate configurațiile posibile
- fiecare nod are doi fii - configurații ce conțin o anumită muchie, respectiv configurații ce nu conțin acea muchie
- considerăm muchiile în ordinea lexicografică (convenție)
- nu fiecare nod are exact doi fii (de exemplu, nu există ciclu cu muchiile (a, b), (a, c) și (a, d), simultan)

Căutarea branch-and-bound II



Euristici pentru branch-and-bound I

- similar cu retezarea alfa-beta
- presupunem că pentru un nod n, putem calcula o limită inferioară a valorilor soluțiilor caracteristice fiilor săi
- dacă anterior am determinat o soluție cu un cost mai mic decât limita inferioară a lui n, atunci explorarea fiilor lui n nu mai are sens



Euristici pentru branch-and-bound II

- pentru un ciclu, costul său se poate exprima ca $1/2 \sum_{v \in V}$(muchiile incidente în v)
- suma oricăror două muchii incidente în $v \geq$ suma costurilor celor mai mici două muchii incidente în v
- costul oricărui ciclu $\geq 1/2 \sum_{v \in V}$(suma costurilor celor mai mici două muchii incidente în v)
  1. astfel, nodul a are asociate (a, d), (a, b), cost 5
  2. nodul b, (a, b), (b, e), cost 6
  3. nodul c, (a, c), (b, c), cost 8
  4. nodul d, (a, d), (c, d), cost 7
  5. nodul e, (b, e), (d, e), cost 9
  6. costul unui ciclu $\geq (5 + 6 + 8 + 7 + 9)/2 = 17.5$

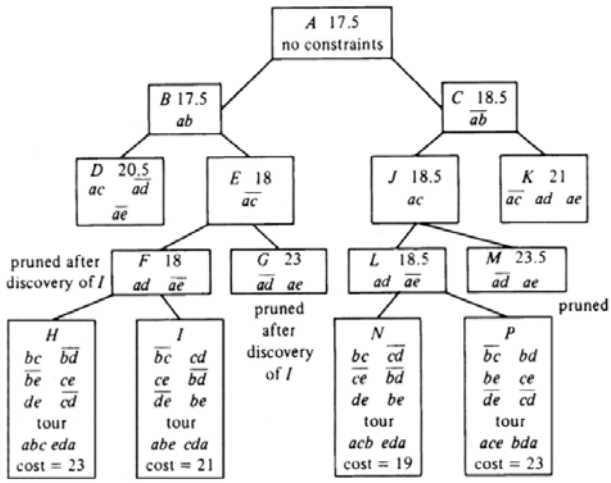
- pentru un fiu în care există constrângerile de a include (a, e) și de a exclude (b, c), suma minimelor se schimbă:
  1. nod a, (a, d), (a, e), cost 9
  2. nod b, (a, b), (b, e), cost 6
  3. nod c, (a, c), (c, d), cost 9

Euristici pentru branch-and-bound III

  4. nod d, (a, d), (c, d), cost 7
  5. nod e, (a, e), (b, e), cost 10
  6. costul unui ciclu $\geq (9 + 6 + 9 + 7 + 10)/2 = 20.5$

- la construirea ciclului, dacă muchia exclusă nu mai închide ciclul, trebuie inclusă
- dacă, din contră, includerea muchiei determină 3 muchii adiacente într-un nod, ea va fi exclusă
- muchia inclusă e de forma xy, cea exclusă de forma $\overline{xy}$

Euristici pentru branch-and-bound IV



Cursul nr. 4

STRUCTURI DE DATE AVANSATE

Cuprins

Tabele de dispersie

Arbori red–black

B-arbori

Kd-tree

Exerciții

Anexa 1: Adresarea deschisă

Anexa 2: Ștergerea din arbore red-black

Anexa 3: Ștergerea din B-arbore

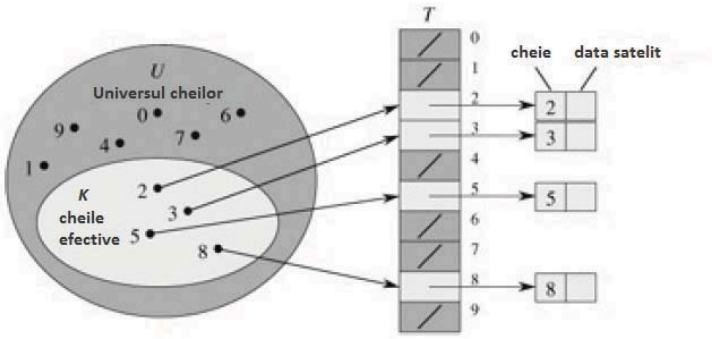
Tabele de dispersie

- Scop: generalizează noțiunea de tablou
- Denumire în limba engleză: hash table, hash map
- Permite implementarea dicționarelor: colecții de perechi de forma (cheie, valoare)
  - Tablou: cheile sunt chiar indicii elementelor, au valori 0, 1, ...
  - Tabel de dispersie: *cheile pot fi orice valori*, care vor fi aduse la forma 0, 1, ...
  - Căutarea se face după cheie
- Exemplu: perechi de forma (nume de persoana, numar de telefon); cautarea se face după numele persoanei
- Operații implementate pentru tabele de dispersie: `Inserează`, `Caută`, `Şterge` pentru elemente ale unei *mulțimi dinamice*
  - o mulțime este dinamică dacă permite inserarea și ștergerea de elemente

Tabele cu adresare directă

- Ipoteze de lucru:
  - Universul cheilor U este rezonabil de mic
  - U are forma $U = \{0, 1, \ldots, m-1\}$
- Soluție imediată: reprezentarea mulțimii dinamice prin tablou $T[0 \ldots m-1]$
- Trebuie să marcăm cumva dacă o poziție din T este ocupată =avem cheie memorată sau disponibilă = la o eventuală căutare a cheii se raportează element negăsit
  - $T[k] = NIL$: cheia k nu este memorată
  - $T[k] \neq NIL$: cheia k este memorată

Tabele cu adresare directă



Figură: Mulţime dinamică reprezentată prin tabel cu adresare directă. Fiecare cheie din $U = \{1, \ldots, 9\}$ corespunde unui indice de tablou. Locaţiile barate conţin NIL.

Tabele cu adresare directă – operaţii

```
Adresare-directa-Cauta(T, k)
    return T[k]

Adresare-directa-Insereaza(T, x)
    T[cheie[x]] ← x

Adresare-directa-Sterge(T, x)
    T[cheie[x]] ← NIL
```

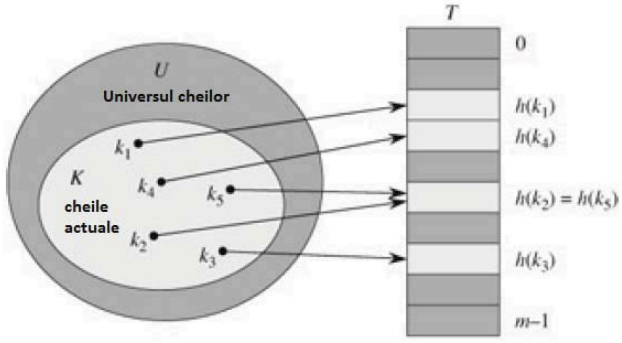
- Complexitate: $O(1)$ în cazul cel mai defavorabil
- Remarcă: se presupune că *cheie*$[x]$ produce valori diferite pentru x diferiţi

Tabele de dispersie

- Critică a tabelelor cu adresare directă:
  - în practică universul cheilor U poate fi mare
  - mulțimea cheilor efectiv memorate K ar fi mică în raport cu U $\Rightarrow$ risipă de spațiu
- Tabela de dispersie:
  - necesarul de memorie poate fi $\Theta(|K|)$
  - complexitatea *medie* a căutării: $O(1)$
- Cum?
  - în adresarea directă cheia dă chiar indicele de stocare în T
  - aici: cheia $k \to h(k)$, $h : U \longrightarrow \{0, 1, \ldots, m-1\}$
  - h: **funcție de dispersie**
  - cheia k *se dispersează* în locația $h(k)$

Tabele de dispersie

- Posibilă problemă: $k_1 \neq k_2$ dar $h(k_1) = h(k_2)$, *i.e.* h nu este injectivă
- Cu siguranță nicio funcție h nu poate fi injectivă pentru $|U| > m$
- Rezultat: coliziuni de chei



Figură: Coliziuni: cheile k_2 și k_5 dispersează în aceeași locație

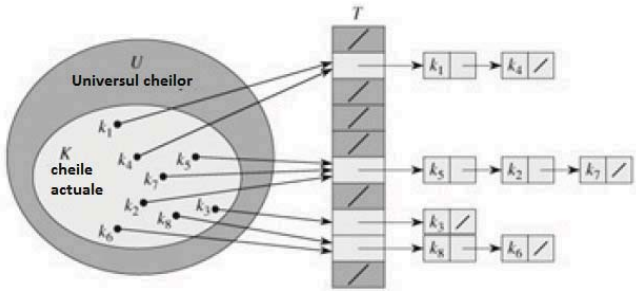
Rezolvarea coliziunilor

- Dorim ca h să producă valori cât mai dispersate, minimizând șansa de coliziune
- Tratarea coliziunilor prin:
  - înlănțuire
  - adresare deschisă

Rezolvarea coliziunilor prin înlănțuire:

- Elementele care dispersează în aceeași locație sunt stocate într–o listă (simplu / dublu) înlănțuită
- Locația $T[h(k)]$ conține referință către capul listei

Rezolvarea coliziunilor prin înlănțuire I



Figură: Rezolvarea coliziunilor prin înlănțuire. Cheile k_1 și k_4 au $h(k_1) = h(k_4) = 1$ și se regăsesc în aceeași listă înlănțuită.

`Dispersie-cu-inlantuire-Insereaza(`T`,` x`)`

Inserează x în capul listei $T[h(cheie[x])]$

Rezolvarea coliziunilor prin înlănțuire II

`Dispersie-cu-inlantuire-Cauta(`T`, `k`)`
Caută un element cu cheia k în lista $T[h(k)]$

`Dispersie-cu-inlantuire-Sterge(`T`, `x`)`
Șterge x din lista $T[h(cheie[x])]$

- Complexitate:
  - inserare: $O(1)$ în cazul cel mai nefavorabil
  - căutare: $O(\text{lungimea listei})$
  - ștergere: remarcăm că se șterge x care este nod din listă, deci pentru o listă dublu înlănțuită este $O(1)$ iar pentru simplu înlănțuită este ca la căutare;
  - ștergerea după cheie necesită căutarea în listă

Analiza dispersiei cu înlănțuire

- Cât durează căutarea în tabelă de dispersie?
- Cazul cel mai defavorabil: $h(k_1) = h(k_2) = \cdots = h(k_n) \Rightarrow$ căutarea are complexitatea $O(n)$
  - funcția de dispersie este astfel aleasă încât să reducă șansa asta
- Cantitățile considerate: $n =$ numărul de chei, $m =$ lungimea lui $T =$ numărul de locații
- Factorul de încărcare
$$\alpha = \frac{n}{m} \tag{1}$$
- α este numărul mediu de elemente stocate de o locație
- Analiza se face în funcție de α
- α este relevant în ipoteza că orice cheie poate să disperseze egal probabil în oricare din cele m locații
  - $=$ ipoteza de dispersie uniformă simplă
- Pentru fiecare locație j notăm cu n_j lungimea listei referite de $T[j]$
- $n = n_0 + n_1 + \cdots + n_{m-1}$

Analiza dispersiei cu înlănțuire

- Ipoteza de dispersie uniformă simplă - valoarea medie a lui n_j: $M(n_j)$ satisface $M(n_j) = \alpha = n/m,\ \forall j = \overline{0, m-1}$
- Presupunere: $h(\cdot)$ se poate calcula în $O(1)$

Teoremă ([1], pag. 259)

Într-o tabelă de dispersie în care coliziunile sunt rezolvate prin înlănțuire o **căutare fără succes** *are complexitatea medie* $\Theta(1+\alpha)$ *în ipoteza de dispersie uniformă simplă.*

Demonstrație: o cheie k poate ocupa cu probabilitate $1/m$ oricare din cele m locații; timpul mediu este dat de calculul lui $h(k)$ plus parcurgerea până la sfârșit a listei $T[h(k)]$ care are lungimea medie $M(n_{h(k)}) = \alpha$.

Teoremă ([1], pag. 259)

Într-o tabelă de dispersie în care coliziunile sunt rezolvate prin înlănțuire, o **căutare cu succes** *are complexitatea medie* $\Theta(1+\alpha)$ *în ipoteza de dispersie uniformă simplă.*

Interpretarea cheilor ca numere naturale

- Pentru șiruri de caractere din setul ASCII, cu coduri între 0 și 127: se consideră numărul format din codurile fiecărui caracter, interpretat în baza 128
- Exemplu: "abc" $= \overline{97\ 98\ 99}_{(128)} = 97 \cdot 128^2 + 98 \cdot 128 + 99$
- Pentru obiecte compuse din mai multe valori: se poate face compunerea (agregarea) cheilor aferente valorilor
- Recomandat: câmpurile pe baza cărora se calculează cheia să fie nemodificate, odată ce s–au setat
- În platformele actuale se prevede posibilitatea de a avea o funcție de dispersie de tip întreg încă de la baza ierarhiei de tipuri de date:
  - Java: în toate clasele standard avem implementare a metodei `public int hashCode()`, ce suprascrie polimorfic metoda omonimă moștenită din clasa `Object`
  - .NET (cod C#): analog, cu metoda `public virtual int GetHashCode()` moștenită din clasa `System.Object`

Funcții de dispersie

- Pentru chei valori numerice: cum se obține o funcție de dispersie *bună*?
- Ce este o funcție de dispersie *bună*?
  - ideal: să satisfacă cât mai fidel ipoteza dispersiei uniforme simple: fiecare cheie se poate dispersa cu aceeași probabilitate în oricare din cele m locații
  - dacă $P(k)$ este probabilitatea de alegere a cheii k, atunci ipoteza se rescrie ca:

$$\sum_{k:h(k)=j} P(k) = \frac{1}{m},\ \forall j = 0, 1, \ldots m-1 \qquad (2)$$

- Exemplu: dacă cheile sunt independente și uniform distribuite în $[0, 1)$, funcția $h(k) = \lfloor km \rfloor$ satisface (2)

Funcții de dispersie: metoda diviziunii

Metoda diviziunii

- $h(k) = k \mod m$
- Cum alegem o valoare bună a lui m?
  - preferabil: valoarea lui $h(k)$ să depindă de toți biții lui k
  - idee naivă: $m = 2^p \Rightarrow h(k)$ depinde doar de cei mai puțin semnificativi p biți
  - se poate arăta că pentru un șir de caractere interpretat în baza 2^p, $m = 2^p - 1$ este o alegere proastă, deoarece permutarea caracterelor nu modifică valoarea lui h
  - se consideră că un m prim nu prea apropiat de o putere a lui 2 este o alegere bună
  - exemplu: 1000 de chei, grad de încărcare dorit: $\alpha = 3$, adică în medie 3 elemente în fiecare listă:
    - număr orientativ de locații: $1000/3 = 333$
    - puteri ale lui 2 din jurul lui 333: $256 = 2^8$, $512 = 2^9$
    - numere prime în jurul lui 333: 313, 317, 331, 337, 347, 349
    - putem alege $m = 331$ sau $m = 337$

Funcţii de dispersie: metoda înmulţirii

Metoda înmulţirii

- Se fixează un $A \in (0, 1)$
- Calculul valorii de dispersie pentru cheia k:

$$k \to f = \underbrace{kA - \lfloor kA \rfloor}_{\text{partea fracţionară a lui } kA} \to \lfloor m \cdot f \rfloor$$

- Alegerea lui m nu este critică
- Pentru baza 2: $m = 2^p$ este o valoare ce eficientizează calculele pe sistemele binare actuale
- A: valoarea optimă depinde de cheile efectiv dispersate
- Knuth: $A = \frac{\sqrt{5}-1}{2} \approx 0.618\ldots$ = `secţiunea de aur` - 1

Funcţii de dispersie: dispersie universală

- Punct slab al funcţiilor anterioare: o alegere nefericită a funcţiei în raport cu datele (sau o alegere nefavorabilă a datelor) poate face ca $h(k) =$ constant
- Strategie: se alege aleator funcţia de dispersie dintr-o familie de funcţii
- Strategia **dispersiei universale**: *în medie* se obţin dispersii bune ale cheilor, independent de valorile lor
- $\mathcal{H}$: colecţie de funcţii de dispersie $h : U \to \{0, 1, \ldots, m-1\}$

Definiţie (Colecţie universală)

$\mathcal{H}$ este o colecţie universală dacă pentru orice $x, y \in U$ numărul funcţiilor de dispersie $h \in \mathcal{H}$ cu $h(x) = h(y)$ este exact $|\mathcal{H}|/m$

Funcții de dispersie: dispersie universală

- Considerăm p prim, $p >$ orice cheie posibilă;
- Pentru $a \in \mathbb{Z}_p^* = \{1, \ldots p-1\}$ și $b \in \mathbb{Z}_p = \{0, 1, \ldots p-1\}$ definim:

$$h_{ab}(k) = ((ak + b) \mod p) \mod m \tag{3}$$

- Considerăm

$$\mathcal{H}_{pm} = \{h_{ab} : a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\} \tag{4}$$

Teoremă ([1], pag. 267)

$\mathcal{H}_{pm}$ este o colecție universală de funcții de dispersie.

- Pentru adresarea deschisă, a se vedea anexa 1

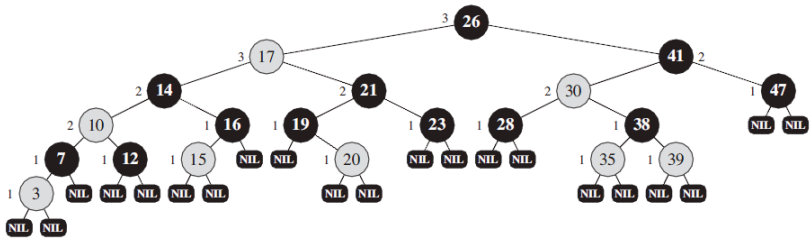
Arbori red–black: origine, comportament, particularități

- Origine: arbore binar de căutare
- Problemă: arborii binari de căutare pot degenera într–o listă dacă datele sunt deja sortate $\Rightarrow$ căutare ineficientă, în $O(n)$ pentru cazul cel mai nefavorabil
- Arborele red–black: operațiile de căutare, inserare și ștergere sunt în timp $O(\log n)$ în cazul cel mai nefavorabil
- Un arbore red–black este un arbore binar de căutare în care fiecare nod e roșu sau negru
- Dacă un nod nu are nod copil stâng sau drept, copilul lipsă se înlocuiește cu un nod fictiv, `NIL`; frunzele sunt doar copii `NIL`
- Un arbore binar de căutare este red–black dacă îndeplinește:
  1. Fiecare nod e fie roșu, fie negru.
  2. Rădăcina este neagră.
  3. Fiecare frunză (`NIL`) este neagră.
  4. Dacă un nod este roșu, atunci ambii săi fii sunt negri.
  5. Orice drum simplu de la un nod la orice frunză descendentă a sa are același număr de noduri negre.

Arbori red–black: definiție, exemplu

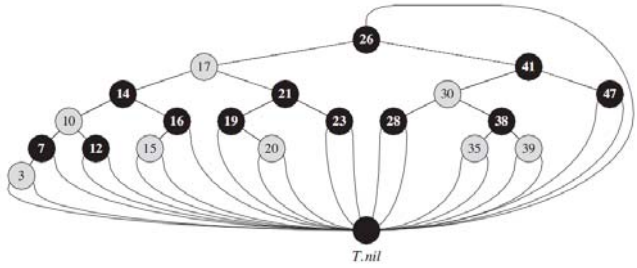
Definiție

Înălțimea neagră a unui nod x – notație: $bh(x)$ – este numărul de noduri negre (excluzându-l pe x) de pe orice drum de la x la o frunză descendentă a sa.



Figură: Exemplu de arbore red–black. Nodurile roșii sunt reprezentate cu gri. Lângă fiecare nod se află înălțimea neagră a sa.

Arbori red–black: exemplul 2



Figură: Folosirea unei santinele pentru reprezentarea nodurilor `NIL`. Părintele rădăcinii este de asemenea un nod `NIL`.

Arbori red–black: proprietate esențială

Teoremă

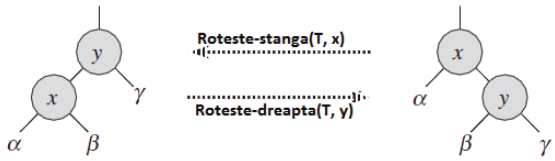
Un arbore red–black cu n noduri interne are înălțimea cel mult $2\log_2(n+1)$.

Demonstrație (schiță): Se arată prin inducție că un subarbore cu rădăcina x are cel puțin $2^{bh(x)}-1$ noduri interne. Din proprietatea 4 a arborilor red–black rezultă că cel puțin jumătate din nodurile de pe orice drum de la un nod x la o frunză (fără a număra și x) sunt negre. Pentru $x =$ rădăcina arborelui de adâncime h avem că $bh(x) \geq h/2$. Folosind rezultatul de mai sus avem că $n \geq 2^{h/2}-1$ de unde și concluzia teoremei.

- Căutarea se va executa în timp $O(\log n)$
- Inserarea și ștergerea prezentate la arborii binari de căutare trebuie modificați pentru a transforma dintr-un arbore red–black în structură de același fel

Arbori red–black: rotații

- Rotație: operație prin care se asigură respectarea condițiilor impuse arborilor de căutare
- Operația e apelată la inserare și ștergere

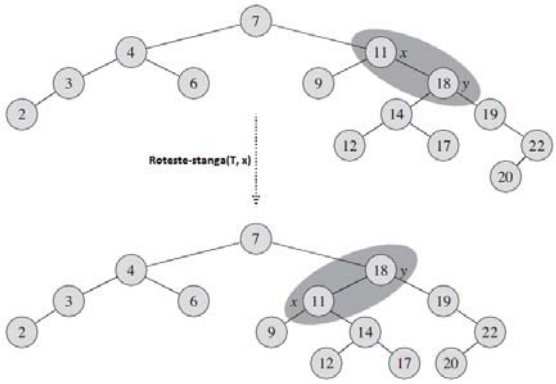


Figură: Operațiile de rotație pentru un arbore binar de căutare. α, β, γ sunt subarbori.

Arbori red–black: rotații

```
Roteste-stanga(T, x)
 1  y ← dreapta[x]
 2  dreapta[x] ← stanga[y]
 3  if stanga[y] ≠ NIL then
 4           p[stanga[y]] ← x
 5  p[y] ← p[x]
 6  if p[y] = NIL    then
 7           radacina[T] ← y
 8     else
 9           if x = stanga[p[x]] then
10                    stanga[p[x]] ← y
11              else
12                    dreapta[p[x]] ← y
13  stanga[y] ← x
14  p[x] ← y
```

Arbori red–black: rotații



Figură: Exemplu de aplicare a procedurii Roteste-dreapta pentru un arbore binar. Se observă că se pleacă de la un arbore binar de căutare și rotirea păstrează această proprietate.

Arbori red–black: inserarea

```
RN-Insereaza(T, z)
 1  y ← nil[T]
 2  x ← radacina[T]
 3  while x ≠ nil[T]
 4      y ← x
 5      if cheie[z] < cheie[x]
 6      then x ← stanga[x]
 7      else  x ← dreapta[x]
 8  p[z] ← y
 9  if y = nil[T]
10      then radacina[T] ← z
11      else  if cheie[z] < cheie[y]
12            then    stanga[y] ← z
13      else  dreapta[y] ← z
14  stanga[z] ← dreapta[z] ← nil[T]
15  culoare[z] ← rosu
16  RN-Corecteaza(T, z)
```

- Inserarea se face în primă fază ca la un arbore binar de căutare
- Orice nod inserat are inițial culoarea roșie; aceasta va fi eventual corectată de către funcția `RN-Corecteaza`
- Apelul de metodă `RN-Corecteaza` este pentru a păstra proprietățile de colorare

Arbori red–black: inserarea

```
RN-Corecteaza(T, z)
 1  while culoare[p[z]] = rosu
 2      if p[z] = stanga[p[p[z]]
 3      then y ← dreapta[p[p[z]]]
 4           if culoare[y] = rosu
 5             then culoare[p[z]] ← culoare[y] ← negru //caz 1
 6                  culoare[p[p[z]]] ← rosu //caz 1
 7                  z ← p[p[z]] //caz 1
 8             else  if z = dreapta[p[z]]
 9                     z ← p[z] //caz 2
10                     Roteste-Stanga(T, z) //caz 2
11                  culoare[p[z]] ← negru //caz 3
12                  culoare[p[p[z]]] ← rosu //caz 3
13                  Roteste-Dreapta(T, p[p[z]]) //caz 3
14      else  (la fel ca la then, interschimband "stanga" cu "dreapta")
15  culoare[radacina[T]] ← negru
```

Arbori red–black: analiza inserării

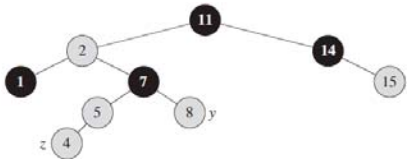
Condițiile ce trebuie îndeplinite de arborele de căutare pentru a fi red–black:

1. Fiecare nod e fie roșu, fie negru
2. Rădăcina este neagră
3. Fiecare frunză (`NIL`) este neagră
4. Dacă un nod este roșu, atunci ambii săi fii sunt negri
5. Orice drum simplu de la un nod la orice frunză descendentă a sa are același număr de noduri negre

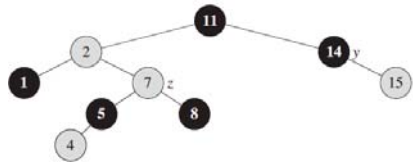
- Proprietățile 1, 2, 3, 5 se păstrează prin inserare și apel de `RN-Corecteaza`
  - ex: proprietatea 5 se păstrează deoarece nodul inserat (roșu) înlocuiește o santinelă neagră și va avea doi copii santinelă – deci negri
- Proprietatea 4 este scopul ciclului **while** din procedura `RN-Corecteaza`

Arbori red–black: exemplu de inserare

Exemple de inserare a unui nod în arborele de căutare: nodurile gri deschis reprezintă noduri roșii. Nodurile `NIL` nu sunt reprezentate.

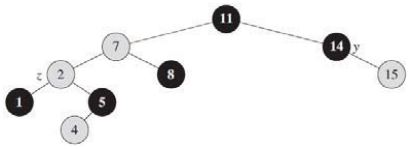


Figură: După inserarea nodului z: atât z cât și părintele său sunt roșii, deci se încalcă proprietatea 4. "Unchiul" lui z este roșu și mergem pe caz 1 din pseudocod.

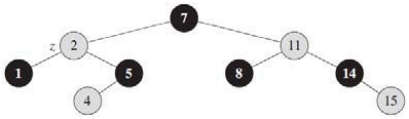


Figură: z și părintele său sunt roșii, dar unchiul lui z este negru. Deoarece z este copil drept, mergem pe cazul 2 din pseudocod.

Arbori red–black: exemplu de inserare



Figură: Ca la cazul anterior, dar cu z copil stâng. Urmează cazul 3: recolorare și rotație.



Figură: Arbore red–black; se resepctă toate proprietățile cerute.

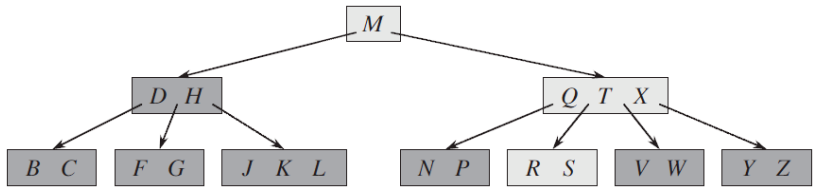
Arbori red–black: complexitatea inserării și ștergerii

- Inserarea propriu-zisă se face pe un arbore red–black de adâncime $O(\log_2 n)$.
- Eventuala corectare a colorării se face printr-un ciclu **while** care sare de la nodul z inserat către rădăcină, cel puțin cu un nivel la fiecare iterație; la fiecare pas se face un număr de operații în $O(1) \Rightarrow$ corectarea colorării are complexitatea $O(\log_2 n)$.
- A se vedea anexa 2 pentru pseudocod
- Complexitatea este $O(\log_2 n)$

B-arbori - descriere

- Arbori de căutare balansați
- Lucrează eficient cu date aflate pe medii de stocare lente (*e.g.* hard disk)
- Scop: minimizarea costului total datorat operațiilor de intrare–ieșire
- Exemplu de utilizare: indexarea în baze de date
  - datele nu pot fi încărcate în totalitate în memoria RAM
  - accesul la disc este costisitor d.p.d.v. al timpului
- Similari cu arborii red-black, dar un nod poate avea foarte mulți copii
  - se reduce drastic adâncimea unui arbore $\rightarrow$ număr mic de operații de intrare–ieșire cu discul
  - complexitatea operațiilor pentru un B-arbore cu n noduri: $O(\log n)$, **cu baza logaritmului un număr mare**
- Costul operațiilor se raportează în funcție de *numărul de accese la disc* și *timpul procesor*

B-arbori - exemplu



Figură: B-arbore având drept chei litere.

- Un nod intern cu p chei are $p + 1$ noduri copil
- Toate frunzele sunt la același nivel
- În figură: nodurile cu culoare deschisă sunt parcurse în căutarea literei "R"

B-arbori - elemente specifice

- E posibil ca dacă se dorește să se acceseze un nod, acesta să nu fie prezent în RAM, caz în care se încarcă prin acces la disc
- Modificarea conținutului unui nod va cere scrierea lui pe disc la un moment dat
- În această structură de date, doar o parte din nodurile arborelui sunt încărcate în RAM
  - De regulă nodul rădăcină e permanent prezent în RAM
  - Spre deosebire de alte structuri de date, operarea se poate face chiar dacă B-arborele nu încape în RAM, dar conținutul măcar al unui nod poate fi încărcat oricând

B-arbori - definiție I

Proprietățile unui B-arbore

1. Fiecare nod x are următoarele câmpuri:
   1.1 $n[x]$, numărul de chei memorate în x
   1.2 cele $n[x]$ chei, ordonate nedescrescător: $cheie_1[x] \leq cheie_2[x] \leq \cdots \leq cheie_{n[x]}[x]$
   1.3 valoarea logică $frunza[x]$: adevărat dacă x e nod frunză, fals altfel; un nod x ce nu e frunză este nod intern
2. Dacă x e nod intern, atunci mai are $n[x]+1$ referințe către fiii lui $c_1[x]$, $c_2[x]$, ..., $c_{n[x]+1}[x]$
3. Cheile dintr-un nod separă domeniile de chei din fiecare subarbore al nodului; pentru k_i cheie oarecare din subarbore cu rădăcina $c_i[x]$, avem:

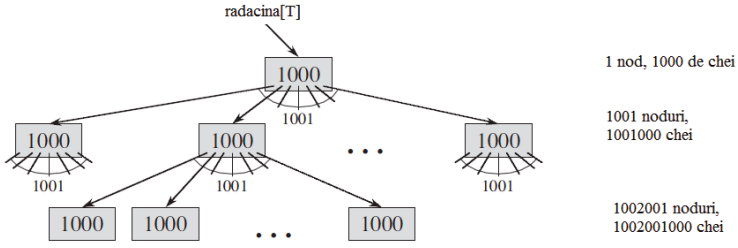
$$k_1 \leq cheie_1[x] \leq k_2 \leq cheie_2[x] \leq \cdots \leq cheie_{n[x]}[x] \leq k_{n[x]+1}$$

B-arbori - definiție (cont.)

4. Fiecare frunză se află la aceeeași adâncime, ceea ce dă înălțimea h a arborelui
5. Numărul de chei este limitat inferior/superior: pentru un $t \geq 2$ numit *gradul minim al B-arborelui* avem că:
   - fiecare nod (posibil cu excepția rădăcinii) trebuie să aibă cel puțin $t-1$ chei; dacă arborele e nevid, rădăcina are cel puțin o cheie
   - fiecare nod poate să conțină cel mult $2t-1$ chei

- Un nod cu maxim de chei se numește *plin*
- Consecință directă: orice nod intern diferit de rădăcină are între t și $2t$ noduri copil.
- În practică eficiența acestei structuri este dată de $t \gg 2$

B-arbori - alt exemplu



Figură: Un B-arbore de înălțime 2 conține peste 1 miliard de chei. Fiecare nod conține 1000 de chei. Numărul din interiorul fiecărui nod x este numărul de chei, $n[x]$.

Înălțimea unui B-arbore

Teoremă

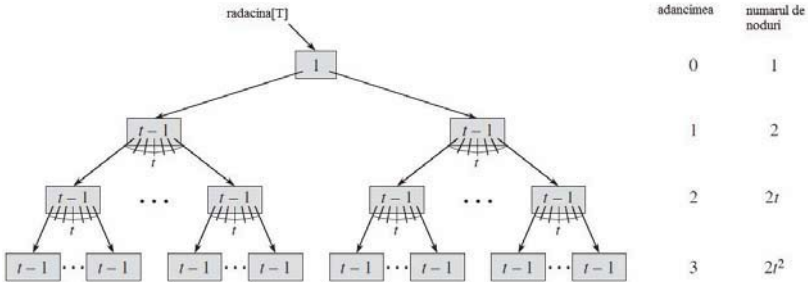
Pentru $n \geq 1$, un B-arbore T cu n chei de înălțime h și gradul minim $t \geq 2$ avem că $h \leq \log_t \frac{n+1}{2}$

Demonstrație: Dacă avem un B-arbore de adâncime h cu gradul minim $t \geq 2$, numărul minim de chei pe care îl poate conține se obține pentru cazul în care:

- rădăcina are o cheie
- fiecare nod diferit de rădăcină are numărul minim de chei, $t - 1$

(a se vedea figura de pe slide-ul următor, pentru $h = 3$). Avem deci 1 nod rădăcină, 2 noduri de adâncime 1, $2t$ noduri de adâncime 2, $2t^2$ noduri de adâncime 3, ..., $2t^{i-1}$ noduri de adâncime i (inducție matematică). Pentru un astfel de arbore numărul de chei conținute este minim, deci $n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} = 2t^h - 1$ de unde se deduce majorant pentru h.

Înălțimea unui B-arbore



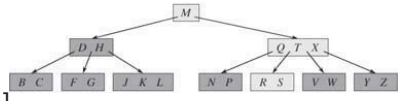
Figură: B-arbore de înălțime 3 cu număr minim de chei. În fiecare nod este trecut numărul efectiv de chei.

Căutarea în B-arbore

- Similară cu căutarea în arbore binar; la fiecare nod x trebuie verificate însă $n[x]$ chei, eventual se trece într-unul din cele $n[x]+1$ noduri copil
- Procedura `Cauta-B-Arbore`:
  - date de intrare: x - nodul de unde se efectuează căutarea, k - cheia căutată
  - ieșire: `NIL` dacă cheia nu se găsește în arbore, altfel perechea: y - un nod care conține cheia, i - indicele lui k în y

Căutarea în B-arbore

```
Cauta-B-Arbore(x, k)
 1  i ← 1
 2  while i ≤ n[x] și k > cheie_i[x]
 3      i ← i + 1
 4  if i ≤ n[x] and k = cheie_i[x]
 5      then return (x, i)
 6  if frunza[x]
 7      then return NIL
 8      else
 9  Citeste-disc(c_i[x])//dacă nodul c_i[x] nu e în memorie
10  return Cauta-B-Arbore(c_i[x], k)
```



- Cine este parametrul x dat pentru primul apel?
- Complexitate: numărul paginilor accesate este $O(h) = O(\log_t n)$; cum $n[x] \leq 2t$, timpul consumat pentru instrucțiunile barometru 2-3 este $O(t)$ și deci timpul CPU este $O(th) = O(t \log_t n)$

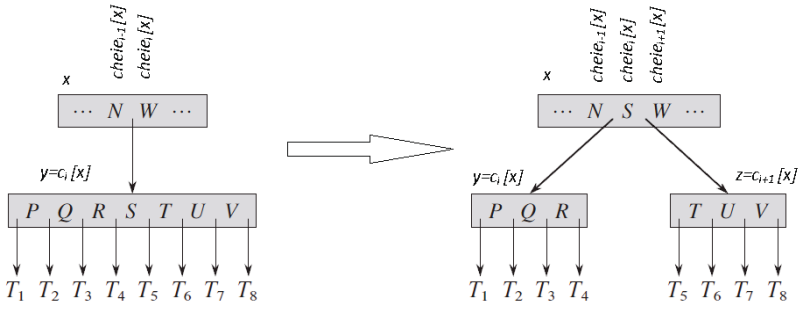
Construirea unui B-arbore: construirea unui arbore vid

```
Creeaza-B-Arbore(T)
1  x ← Aloca − Nod()
2  frunza[x] ← adevarat
3  n[x] ← 0
4  Scrie-Disc(x)
5  radacina[T] ← x
```

- Complexitate: $O(1)$, atât pentru operații cu discul și timp CPU

Construirea unui B-arbore: divizarea unui nod plin

- Nod plin y: nod care are $2t - 1$ chei $\Leftrightarrow n[y] = 2t - 1$



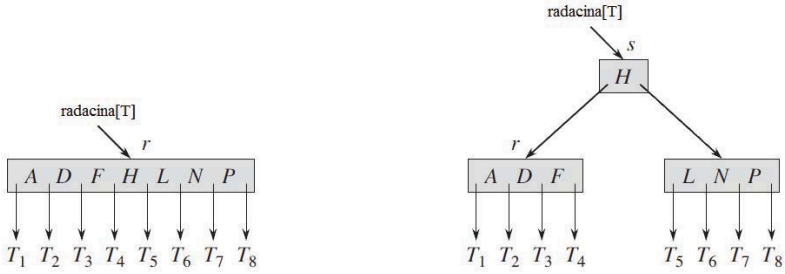
Figură: Divizarea unui nod plin (t=4). Nodul plin y este divizat în y și z, iar cheia mediană S este mutată în x, părintele lui y.

Construirea unui B-arbore: divizarea unui nod plin

- Situația poate apărea la inserare de cheie
- Presupunând că nodul părinte al lui y nu e plin – procedura de inserare va asigura acest lucru – divizarea se face astfel:
  - $cheie_t[y]$ – valoarea mediană a cheilor – urcă în părintele lui y, presupus (de fapt: sigur!) neplin
  - ultimele $t-1$ chei din y se duc într-un nod frate al lui y, anume construit la acest pas
  - nodul y rămâne doar cu primele $t-1$ chei din cele inițiale

Construirea unui B-arbore: divizarea rădăcinii

- Dacă nodul y nu are părinte, atunci se creează un părinte al său cu cheia $= cheie_t[y]$
  - Acesta e singurul mod de creștere a adâncimii B-arborelui



Figură: Divizarea unui nod rădăcină plin (t=4).

Construirea unui B-arbore: divizarea unui nod $y = c_i[x]$

```
Divide-Fiu-B-Arbore(x, i, y)
 1  z ← Aloca − Nod()
 2  frunza[z] ← frunza[y]
 3  n[z] ← t − 1
 4  for j ← 1, t − 1
 5      cheie_j[z] ← cheie_{j+t}[y]
 6  if frunza[y] = fals then
 7      for j = 1, t
 8          c_j[z] ← c_{j+t}[y]
 9  n[y] ← t − 1
10  for j ← n[x] + 1, i + 1, −1
11      c_{j+1}[x] ← c_j[x]
12  c_{i+1}[x] ← z
13  for j ← n[x], i, −1
14      cheie_{j+1}[x] ← cheie_j[x]
15  cheie_i[x] ← cheie_t[y]
16  n[x] ← n[x] + 1
17  Scrie-Disc(y)
18  Scrie-Disc(z)
19  Scrie-Disc(x)
```

- Timpul CPU este $\Theta(t)$, datorită ciclărilor din liniile 4-5 și 7-8 (celelalte iterații rulează în timp $O(t)$)
- Sunt $3 \in O(1)$ operații de lucru cu discul

Inserarea unei chei într–un B-arbore

- Se execută într-o singură parcurgere, de la rădăcină către frunze
- Cazuri: rădăcina este plină sau nu

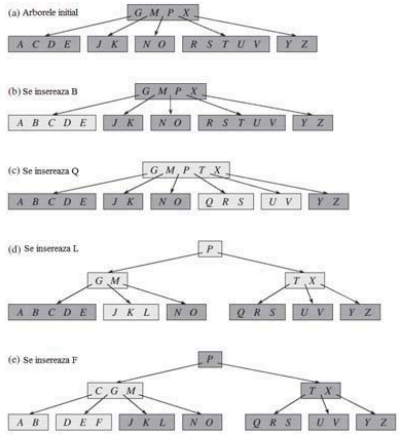
```
Insereaza-B-Arbore(T, k)
 1  r ← radacina[T]
 2  if n[r] = 2t − 1 then        //radacina e plina
 3          s ← Aloca − Nod()
 4          radacina[T] ← s
 5          frunza[s] ← fals
 6          n[s] ← 0
 7          c_1[s] ← r
 8          Divide-Fiu-B-Arbore(s, 1, r)
 9          Insereaza-B-Arbore-Neplin(s, k)//acum sigur rad.
            //nu mai e plina
10      else
11          Insereaza-B-Arbore-Neplin(r, k)
```

Inserarea unei chei într–un B-arbore (cont)

```
Insereaza-B-Arbore-Neplin(x, k)
 1  i ← n[x]
 2  if frunza[x]
 3          while i ≥ 1 and k < cheie_i[x]
 4            cheie_{i+1}[x] ← cheie[x]
 5            i ← i − 1
 6          cheie_{i+1}[x] ← k
 7          n[x] ← n[x] + 1
 8          Scrie-Disc(x)
 9     else
10          while i ≥ 1 and k < cheie_i[x]
11            i ← i − 1
12          i ← i + 1
13          Citeste-Disc(c_i[x])
14          if n[c_i[x]] = 2t − 1
15            Divide-Fiu-B-Arbore(x, i, c_i[x])
16            if k > cheie_i[x]
17                  i ← i + 1
18          Insereaza-B-Arbore-Neplin(c_i[x], k)
```

- Liniile 3-8 tratează cazul în care x este frunză
- Din modul de lucru al lui Insereaza-B-Arbore-Neplin și Insereaza-B-Arbore, garantat x nu e plin
- Dacă x nu e frunză, inserarea se face căutând un nod frunză adecvat pentru k
- Numărul de accesări de disc: $O(h)$
- Timpul total CPU: $O(th) = O(t \log_t n)$

Inserarea unei chei într–un B-arbore: exemplu



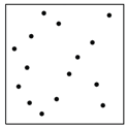
Figură: Inserare de chei într-un B-arbore cu $t = 3$ - diverse situații. Nodurile colorate mai deschis sunt cele modificate.

Ștergerea unei chei într–un B-arbore: schiță

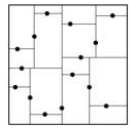
- Se cere ștergerea cheii k din subarborele de rădăcină x
- Posibil ca nodul din care se șterge să rămână cu mai puțin de $t - 1$ chei
- Se efectuează manevră de fuziune (inversa divizării) prin care o cheie este coborâtă
- Dacă prin această manevră rădăcina rămâne fără nicio cheie – deci cu un singur fiu – ea e ștearsă și acest fiu devine noua rădăcină
- Detalii: anexa 3, slide 8 și următoarele

Kd-tree

- Arbore binar care partajează spațiul în celule din ce în ce mai mici
- Separarea celulelor se face prin hiperplane, perpendiculare pe axele de coordonate
- Descompunerea este de tip ierarhic, într-o manieră recursivă
- Pentru figura de mai jos: care este prima dreaptă (presupusă verticală) de separare trasată?



Figură: Mulțimea inițială de puncte în spațiul bidimensional. Hiperplanele de separare vor fi în acest caz drepte, perpendiculare pe axele Ox și Oy.



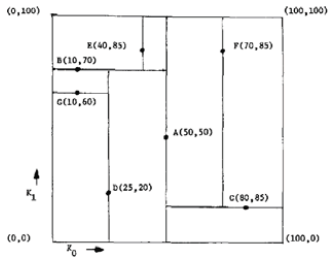
Figură: Împărțirea mulțimii de puncte. La fiecare pas se alege câte un punct, se trasează o dreapta de direcție diferită față de cea de la pasul anterior. Mulțimile rămase se procesează recursiv

Kd-tree

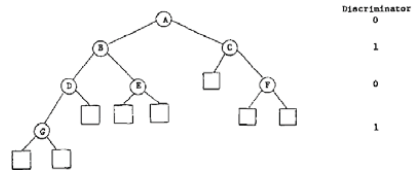
- Utilitate: căutarea rapidă a punctelor sau a regiunilor, chiar cu specificații parțiale:
  - valori precizate doar pentru unele dimensiuni, sau
  - valori ce satisfac inegalități
- Punct de plecare: o mulțime de puncte k-dimensionale
- Rezultat: un arbore binar – un kd-tree – care are câte un nod asociat fiecărui punct
- Pentru un nod P din arbore avem câmpurile:
  - $cheie_0(P), \ldots, cheie_{k-1}(P)$ sunt valorile pentru cele k dimensiuni ale lui A
  - $discriminator(P) \in \{0, \ldots, k-1\}$ – care este criteriul (dimensiunea, direcția) după care se face împărțirea în P
  - $stanga(P)$, $dreapta(P)$ – referințe către nodurile fiu stâng respectiv drept

Kd-tree

- Convenție: toate nodurile S din subarborele stâng al lui P au $cheie_{discriminator}(S) < cheie_{discriminator}(P)$
- Analog: toate nodurile D din subarborele drept al lui P au $cheie_{discriminator}(D) > cheie_{discriminator}(P)$
- Cazul cheilor egale: va urma



Figură: Puncte în spațiul bidimensional, cu coordonate precizate.



Figură: Kd-tree pentru punctele din figura alăturată. Nodurile marcate cu pătrare sunt NIL.

Kd-tree

- Toate nodurile de pe un nivel au același discriminator
- Nodul rădăcină are discriminatorul 0, nivelul 1 are discriminator 1 etc, nivelul $k-1$ are discriminatorul $k-1$, nivelul k are discriminatorul 0 – se ciclează modulo k
- Funcția $succesor(P, Q)$ pentru un nod curent P și o valoare ce se caută Q returnează fie $stanga(P)$, fie $dreapta(P)$, corespunzător direcției în care trebuie să se continue căutarea lui Q
- Cazul cheilor egale $cheie_j(P) = cheie_j(Q)$, unde $0 \leq j \leq k-1$ este indicele de discriminare:
  - se definește supercheia $S_j(P)$ a lui P ca $S_j(P) = cheie_j(P)cheie_{j+1}(P)\ldots cheie_{k-1}(P)cheie_0(P)\ldots cheie_{j-1}(P)$
  - dacă $S_j(Q) < S_j(P)$ atunci $succesor(P, Q)$ returnează $stanga(P)$, dacă $S_j(Q) > S_j(P)$ atunci $dreapta(P)$ altfel o valoare specială care să indice egalitate

Kd-tree

- Un nod Q este j–mai mic decât P dacă $succesor(P, Q)$ returnează $stanga(P)$; analog se definește j–mai mare ($j = discriminator(P)$)
- Algoritmul de inserare preia un kd–tree T și un nod P, pentru care doar cheile sunt setate
- Dacă nod cu aceeași cheie există deja în T, atunci se returnează referință la el; altfel, se inserează și se returnează NIL

Kd-tree: inserare

```
Kd-Insereaza(T, P)
 1  if radacina(T) = NIL    then
 2          radacina(T) ← P
 3          stanga(P) ← dreapta(P) ← NIL
 4          discriminator(P) ← 0
 5          return P
 6     else
 7          Q ← radacina(T)
 8  repeat
 9          if cheie_i(P) = cheie_i(Q) ∀0 ≤ i ≤ k − 1
10            then return Q
11            else
12                 fiu(Q) ← succesor(Q, P)//fiu este st. sau dr. pentru Q
13                 if fiu(Q) = NIL    then
14                        fiu(Q) ← P
15                        stanga(P) ← dreapta(P) ← NIL
16                        discriminator(P) = (discriminator(Q) + 1)  mod k
17                   else  Q ← fiu(Q)
```

Kd-tree: inserare și căutare

- Comportamentul pentru inserare: la fel ca la arborele binar de căutare, cu valori furnizate în ordine aleatoare
- Pentru căutarea unui nod într–un kd-tree cu n noduri numărul mediu de noduri vizitate este aproximativ $1.386 \log_2 n \in \Theta(log_2 n)$ (vezi [2])
- Tipuri de interogări:
  - de tip intersecție:

    $\{P | cheie_3(P) = 7\}$ sau $\{P | 1 \leq cheie_1(P) \leq 5$ şi $2 \leq cheie_2(P) \leq 4\}$

  - de tip "cea mai bună potrivire" – *e.g.* cel mai apropiat vecin al lui P

Kd-tree: căutare cu potrivire exactă

- Se caută un punct P pentru care valorile fiecărei chei sunt specificate
- Se modifică algoritmul de inserare pentru a face doar comparații de chei și eventual coborâre în arbore
- Algoritmul în pseudocod: temă
- Dacă acesta e singurul tip de interogare care se face, o tabelă de dispersie este mai eficientă

Kd-tree: căutare cu potrivire parțială

- Valorile sunt specificate doar pentru o parte din chei
- Se cere găsirea punctelor: $\{P | cheie_{s_i}(P) = v_{s_i}\}$ unde $\{s_1, \ldots, s_t\} \subset \{1, \ldots, k\}, t < k, v_{s_i}$ sunt valori precizate
- Algoritm de căutare recursiv:
  - pentru fiecare apel se furnizează ca parametru un nod P
  - pentru P se verifică dacă satisface condițiile $cheie_{s_i}(P) = v_{s_i}, \forall i = 1, \ldots, t$; dacă da, atunci se returnează
  - altfel: dacă $discriminant(P) = s \in \{s_1, \ldots, s_t\}$ se continuă căutarea recursiv în subarborele stâng (pentru $v_s < cheie_s(P)$), în subarborele drept (pentru $v_s > cheie_s(P)$), în amândoi subarborii (pentru $v_s = cheie_s(P)$).
  - dacă $discriminant(P) \notin \{s_1, \ldots, s_t\}$ atunci se va face căutarea în ambii subarbori ai lui P
- Observăm că se poate returna o mulțime de noduri ce au egalitate de valori pe dimensiunile s_i

Kd-tree: căutare cu potrivire parțială

- Complexitate: dacă arborele este "ideal" — *i.e.* numărul de noduri n este $2^{kh} - 1$ și toate frunzele apar pe nivelul kh — atunci numărul maxim de noduri vizitate este (vezi [2]):

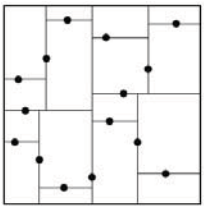
$$V(n,t) = \frac{[(t+2)2^{k-t-1} - 1]}{2^{k-t} - 1} \cdot \left[(n+1)^{\frac{k-t}{k}} - 1\right]$$

Kd-tree: căutare de puncte din regiuni

- Problemă: se cere determinarea tuturor punctelor care se găsesc într–o regiune
- Regiunea e specificată ca un vector de mărginire cu $2d$ elemente
- Cele 2 interogări anterioare sunt cazuri particulare ale acesteia
- Algoritm: [2], pagina 514

Kd-tree: optimizarea arborilor

- Scopul optimizării: obținerea unui arbore cât mai echilibrat
- Formal: pentru orice nod, diferența de adâncime între subarborele său stâng și cel drept să fie cel mult 1
- Intrare pentru algoritm: o listă de puncte A și un discriminator $0 \leq J \leq k-1$
- Ieșire: un arbore optimizat T, cu $discriminant(radacina(T)) = J$
- Strategie: divide et impera + determinare de mediană



Figură: Divizare echilibrată (optimală) a spațiului.

Kd-tree: construirea de arbori balansați

```
Contruire-Balansata(A, J)
1  if A e vid  then
2          return NIL
3  P ← J-mediana lui A
4  A_stanga ← {a ∈ A|a este J–mai mic decât P}
5  A_dreapta ← {a ∈ A|a este J–mai mare decât P}
   //nr. de elemente din A_stanga diferă de cel al lui A_dreapta cu cel mult 1
6  discriminant(P) ← J, M ← (J + 1) mod k
7  stanga(P) ← Contruire − Balansata(A_stanga, M)
8  dreapta(P) ← Contruire − Balansata(A_dreapta, M)
9  return P
```

- Complexitate: depinde de modul de determinare a medianei din linia 3; dacă se folosește algoritm liniar, atunci:

$$T(n) = 2T(n/2) + cn \in \Theta(n \log_2 n)$$

- Ce complexitate are algoritmul dacă la pasul 3, la fiecare nivel se folosește un algoritm de sortare și apoi se alege elementul din mijloc?

Exerciții

1. Pentru structurile de date din acest curs, care sunt tipurile de date sau bibliotecile care conțin implementări pentru ele? Căutați variante pentru C++, Java, Python. Alcătuiți o scurtă descriere a lor – ex: dacă implementează exact structurile de date prezentate sau au particularități, eventualele restricții legale sau tehnice, eventuala eficiență măsurată prin teste.
2. Scrieți un algoritm care să returneze cel mai apropiat vecin pentru un punct dat, plecând de la kd-tree. Punctul pentru care se caută vecinul se găsește în arbore.

Adresarea deschisă

- Toate elementele sunt memorate în tabelă, nu ^ in liste externe
- Factorul de încărcare α nu poate depăși 1
- Dacă există coliziuni la inserare se caută alte locații libere pentru obiectul curent = se fac probe (încercări) succesive
- Șirul de poziții în care se caută alternative depinde de cheia ce se inserează
- Funcția de dispersie considerată include numărul de încercare:

$$h : U \times \{0, 1, \ldots, m-1\} \longrightarrow \{0, 1, \ldots, m-1\}$$

- Secvența de verificare pentru căutare (de cheie sau de poziție disponibilă pentru inserare):

$$< h(k, 0), h(k, 1), \ldots, h(k, m-1) >$$

Adresarea deschisă

```
DISPERSIE-INSEREAZA(T, k)
1   i ← 0
2   repeat j ← h(k, i)
3         if T[j] = NIL
4           then T[j] ← k
5                return j
6           else i ← i + 1
7   until i = m
8   error depasire tabela
```

```
DISPERSIE-CAUTA(T, k)
1   i ← 0
2   repeat
3         j ← h(k, i)
4         if T[j] = k
5           then return j
6         i ← i + 1
7   until T[j] = NIL or i = m
8   return NIL
```

Adresarea deschisă

- Ştergerea: dificilă
- Dacă am seta la NIL locaţia j golită, la o căutare s–ar putea raporta în mod eronat că tabela nu conţine cheia k, pentru cazul în care poziţia actuală a lui k este $h(k, i')$ şi $\exists i < i'$: $h(k, i) = j$
- Alternativă: se marchează locaţia j cu o valoare "STERS" care să arate că s–a făcut ştergere
- Se modifică doar `Dispersie-Insereaza` pentru a permite inserarea într–o locaţie din care s–a şters

Adresarea deschisă: moduri de verificare

- Folosim funcțiile de dispersie $h'(\cdot)$, $h'_1(\cdot)$, $h'_2(\cdot)$
- Verificare liniară:

$$h(k,i) = (h'(k,i) + i) \mod m$$

- Verificare pătratică:

$$h(k,i) = (h'(k,i) + c_1 i + c_2 i^2) \mod m$$

- Dispersia dublă:

$$h(k,i) = (h'_1(k) + i h'_2(k)) \mod m$$

Adresarea deschisă: complexitate

Teoremă ([1], pag. 274)

Pentru o tabelă de dispersie cu adresare deschisă și factor de încărcare $\alpha < 1$, numărul mediu de probe într–o căutare fără succes este cel mult $1/(1-\alpha)$, în ipoteza dispersiei uniforme[1].

Teoremă ([1], pag. 275)

Pentru o tabelă de dispersie cu adresare deschisă și factor de încărcare $\alpha < 1$, numărul mediu de probe într–o inserare este cel mult $1/(1-\alpha)$, în ipoteza dispersiei uniforme.

[1] Dispersie uniformă: secvența de probe pentru fiecare cheie poate produce oricare din cele $m!$ permutări ale mulțimii $\{0, 1, \ldots, m-1\}$ cu aceeași probabilitate.

Ștergerea unei chei dintr–un arbore red-black (1)

- Funcția `RB-Transplant` este utilizată pentru a înlocui subarborele având ca rădăcină pe u cu subarborele cu rădăcina v

```
RB-Transplant(T, u, v)
1  if p[u] = nil[T]
2  radacina[T] ← v
3      else if u == stanga[p[u]]
4  stanga[p[u]] ← v
5      else dreapta[p[u]] ← v
6  p[v] ← p[u]
```

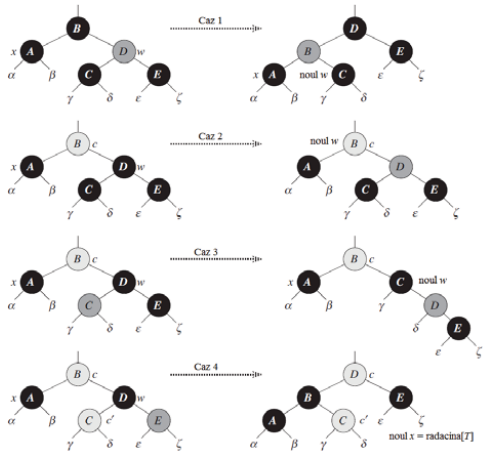
Ștergerea unei chei dintr–un arbore red-black (2)

```
RB-Sterge(T, z)
 1  y ← z
 2  culoare − originala − y ← culoare[y]
 3  if stanga[z] = nil[T]
 4  x ← dreapta[z]
 5  RB-Transplant(T, z, dreapta[z])
 6      else if dreapta[z] = nil[T]
 7  x ← stanga[z]
 8  RB-Transplant(T, z, stanga[z])
 9      else y ← Minim − Arbore(dreapta[z])
10  culoare − originala − y ← culoare[y]
11  x ← dreapta[y]
12  if p[y] = z
13  p[x] ← y
14      else RB-Transplant(T, y, dreapta[y])
15  dreapta[y] ← dreapta[z]
16  parinte[dreapta[y]] ← y
17  RB-Transplant(T, y, dreapta[y])
18  stanga[y] ← stanga[z]
19  parinte[stanga[y]] ← y
20  culoare[y] ← culoare[z] if culoare − originala − y = negru
21  Corectie-Stergere-RB(T, x)
```

Ştergerea unei chei dintr–un arbore red-black (3)

```
Corectie-Stergere-RB(T, x)
 1  while x ≠ radacina[T] and culoare[x] = negru
 2  if x == stanga[p[x]]
 3  w ← dreapta[p[x]]
 4  if culoare[w] = rosu
 5  culoare[w] ← negru //caz 1
 6  culoare[p[x]] ← rosu //caz 1
 7  Roteste-Stanga(T, p[x]) //caz 1
 8  w ← dreapta[p[x]] //caz 1
 9  if culoare[stanga[w]] = negru and culoare[dreapta[w]] = negru
10  culoare[w] ← rosu //caz 2
11  x ← p[x] //caz 2
12      else if culoare[dreapta[w]] = negru
13  culoare[stanga[w]] ← negru //caz 3
14  culoare[w] ← rosu //caz 3
15  Roteste-Dreapta(T, w) //caz 3
16  w ← dreapta[p[x]] //caz 3
17  culoare[w] ← culoare[p[x]] //caz 4
18  culoare[p[x]] = negru //caz 4
19  culoare[dreapta[w]] = negru //caz 4
20  Roteste-Stanga(T, p[x]) //caz 4
21  x ← radacina[T] //caz 4
22      else la fel ca la then dar interschimband "stanga" cu "dreapta"
23  culoare[x] ← negru
```

Ştergerea unei chei dintr–un arbore red-black (4)

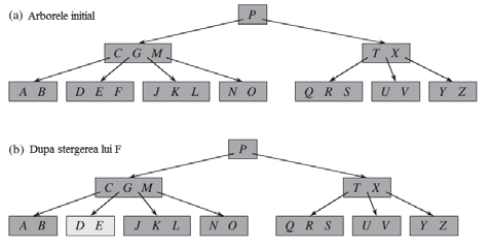


Figură: Cele 4 cazuri tratate in cadrul algoritmului `Corectie-Stergere-RB`.

Ștergerea unei chei dintr–un B-arbore (1)

Metoda de ștergere ar fi `Sterge-B-Arbore(`x`, `k`)`, cu x dat inițial ca *radacina*(T)

1. Dacă cheia k este în nodul x iar x e nod frunză, se șterge k din x
   - Posibil ca în frunză să rămână mai puțin de t chei, trebuie făcut transfer de cheie către / dinspre un nod frate al frunzei sau unire de noduri frate.

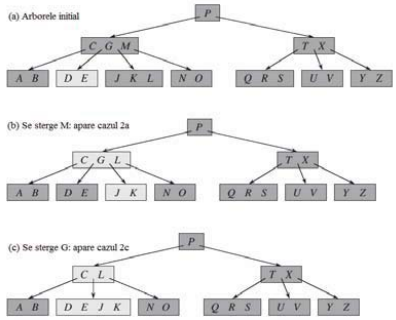


Figură: Ștergerea unuei chei din nod frunză, pentru un B-arbore cu $t = 3$ și $x =$ nod frunză care conține k.

Ștergerea unei chei dintr–un B-arbore (2)

2. Dacă cheia k este în nodul x și x nod intern:
   a. Dacă fiul y care precede cheia k are cel puțin t chei, se caută predecesorul k' al cheii k în subarborele de rădăcină y; se șterge k' și se înlocuiește k cu k', după care se aplică mai departe, recursiv, aceeași regulă
   b. Simetric, dacă fiul z care succede cheia k în nodul x are cel puțin t chei
   c. Dacă atât y cât și z au câte $t - 1$ chei, cele două noduri fuzionează: în y intră k și toate cheile din z, după care y va conține $2t - 1$ chei. Se șterge nodul z și recursiv se șterge cheia k din y.

Ștergerea unei chei dintr–un B-arbore (3)

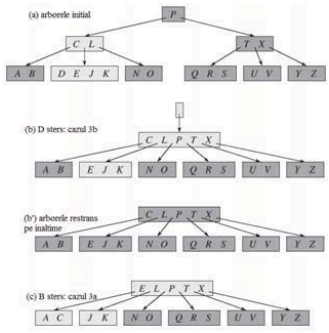


Figură: Stergere de cheie din nod interior, $t = 3$. În cazul (b) x e nodul care conține cheile C, G, M; se aplică 2a: predecesorul L al lui M urcă pentru a lua locul lui M. În cazul (c) x se presupune a fi nodul cu cheile C, G, L; este cazul 2c: coborâm G pentru a produce nodul DEGJK și apoi ștergem G din frunză = cazul 1.

Ștergerea unei chei dintr–un B-arbore (4)

3. Dacă cheia k nu e prezentă în nodul x, se determină rădăcina $c_i[x]$ pentru subarborele ce conține k (se presupune că cheia k chiar se regăsește în B-arbore). Dacă $c_i[x]$ are numai $t - 1$ chei, se execută pașii 3a sau 3b pentru a se asigura numărul minim de chei. Apoi se aplică recursiv procedura la fiul potrivit al lui x.
   a dacă $c_i[x]$ are numai $t - 1$ chei, dar are un nod frate în stânga sau în dreapta lui care are t chei, atunci are loc mutarea unei chei din x în $c_i[x]$, apoi mutarea unei chei în x din fratele din stânga sau dreapta lui $c_i[x]$
   b Dacă $c_i[x]$ și toți frații lui au câte $t - 1$ chei, fuzionează c_i cu unul din frați, ceea ce implică mutarea unei chei din x în nodul nou fuzionat ca și cheie mediană.

Ştergerea unei chei dintr–un B-arbore (5)



Figură: Cazul 3, $t = 3$. (b) Ştergerea cheii D: cazul 3b, recurenţa nu poate să coboare din nodul CL deoarece are numai două chei, deci P este împins în jos şi prin fuzionarea lui CL cu TX se obţine nodul CLPTX; apoi se şterge D in frunza = cazul 1. (b') Se şterge rădăcina şi înălţimea scade cu 1. (c) Ştergerea cheii B: cazul 3a, C este mutat în vechiul loc al lui B şi E este mutat în locul lui C.

Bibliografie

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein *Introduction to Algorithms*. Addison Wesley, Massachusetts, 3rd Edition, MIT Press 2009
- Jon Louis Bentley *Multidimensional Binary Search Trees Used for Associative Searching*. Communications of the ACM, September 1975, Number 9

Cursul nr. 5

COMPLEXITATE NP

Cuprins

Timpul polinomial

Verificări în timp polinomial

NP-completitudine și reductibilitate

Demonstrații ale NP-completitudinii

Enunțuri de probleme NP-complete

Probleme accesibile

- în general, problemele pot fi rezolvate în timp polinomial sau nu:
  1. probleme **accesibile**, timp de calcul polinomial (clasa P, timp în $O(n^k)$, $k \in \mathbb{N}$ este o constantă)
  2. probleme **inaccesibile**, timp suprapolinomial (zise NP complete)
- problemele rezolvabile în timp polinomial necesită în practică un timp mult mai mic ca $\Theta(n^{100})$
- problemele rezolvabile în timp polinomial cu ajutorul unui model de calcul (pe mașini seriale cu acces direct) pot fi rezolvate polinomial pe orice alt model de calcul (pe o mașina Turing, de exemplu)
- clasa P are o serie de proprietăți de închidere:
  1. compunerea a doi algoritmi polinomiali (rezultatul primului este intrarea celuilalt) este un algoritm polinomial
  2. dacă un algoritm polinomial apelează de un număr constant de ori subrutine polinomiale, algoritmul rezultat e de tip polinomial

Probleme abstracte

- o **problemă abstractă** este o relație binară între mulțimea instanțelor problemei și mulțimea soluțiilor sale
  - problema drumului minim este o relație binară ce asociază unei instanțe, cuplul $(G = (V, M), \{a, b\})$ cu o soluție posibilă - cel mai scurt drum între vârfurile a și b
  - o instanță a problemei poate avea mai multe soluții
- o **problemă de decizie** este o problemă abstractă ce are o soluție de tipul da/nu
  - problema drumului minim se poate reformula ca $(G = (V, M), \{a, b\}, k)$, $k \in \mathbb{Z}^+$, problema existenței unui drum între vârfurile $u, v \in V$ de lungime cel mult k
  - tipul de probleme la care se referă teoria NP-completitudinii

- în general, problemele abstracte sunt de multe ori probleme de optim; ele se transformă în probleme de decizie prin impunerea unei limite superioare, în cazul problemelor de minim, sau a unei limite inferioare, pentru cele de maxim
- în practică, dacă o problemă de optim este ușor de rezolvat, la fel este și problema de decizie asociată
- d.p.d.v. al NP-completitudinii, dacă se demonstrează că o problemă de decizie este greu de rezolvat, atunci se poate demonstra că și problema de optim asociată este greu de rezolvat
  - dacă pentru o problemă de decizie ușor de rezolvat, putem rezolva problema de optim folosindu-ne de o succesiune finită de probleme de decizie, înseamnă că putem rezolva problema de optim de asemenea ușor

Codificări

- instanțele problemei abstracte sunt codificate pentru ca mașina să poată lucra cu ele
- o **codificare** a unei mulțimi S de obiecte este o funcție e ce mapează fiecare obiect $s \in S$ într-un șir binar, $e : S \to \{0,1\}^*$
  - mulțimea numerelor naturale $\mathbb{N} = \{0,1,2,3,4,\dots\}$ poate fi codificată în mulțimea de șiruri $\{0,1,10,11,100,\dots\}$
  - $e(23) = 10111$
  - poligoanele, grafurile, funcțiile, perechile ordonate, toate pot fi codificate ca șiruri binare
- o **problemă concretă** este o problemă ce are ca mulțime a instanțelor o mulțime de șiruri binare
- un algoritm **rezolvă** o problemă concretă în timp $O(T(n))$ dacă pentru orice instanță de lungime n a problemei, algoritmul găsește soluție într-un timp cel mult $O(T(n))$
  - o problemă concretă este **rezolvabilă în timp polinomial** dacă există un algoritm ce găsește o rezolvare într-un timp în $O(n^k)$, k fiind o constantă

- codificări
  - codificarea în binar a lui 6 este 110
  - codificarea în unar a lui 6 este 111111
- codificarea problemei influențează eficiența algoritmului de rezolvare
  - considerăm un algoritm ce are timpul de execuție în $\Theta(k)$, unde k este un număr întreg ce reprezintă intrarea algoritmului
  - dacă numărul este reprezentat unar, complexitatea este $\Theta(n)$, unde $n = k$ este lungimea intrării
  - dacă numărul este reprezentat binar, lungimea intrării este $n = \lfloor log\ k \rfloor + 1$, iar complexitatea devine $\Theta(k) = \Theta(2^n)$, adică un timp exponențial față de lungime intrării
  - de regulă vom evita codificarea unară, considerată 'costisitoare'
- o funcție $f : \{0,1\}^* \to \{0,1\}^*$ este **calculabilă în timp polinomial** dacă există un algoritm în timp polinomial pentru care, dându-se intrarea $x \in \{0,1\}^*$, ieșirea este $f(x)$

- două codificări ale unei instanțe i a problemei $e_1(i), e_2(i)$ sunt **asociate polinomial** dacă există două funcții f_{12}, f_{21} calculabile în timp polinomial pentru care $f_{12}(e_1(i)) = e_2(i)$ și $f_{21}(e_2(i)) = e_1(i)$
  - orice codificare într-o bază $b \geq 2$ poate fi transformată într-o codificare într-o altă bază $d \geq 2$ într-un timp polinomial
- dacă codificarea $e_1(Q)$ a instanțelor problemei Q poate fi rezolvată într-un timp polinomial, $e_1(Q) \in P$, iar $e_1(Q)$ și $e_2(Q)$ sunt asociate polinomial, atunci și $e_2(Q) \in P$
  - fiind asociate polinomial, există un algoritm polinomial pentru trecerea de la $e_2(Q)$ la $e_1(Q)$
  - dar pentru codificarea $e_1(Q)$ avem un algoritm polinomial
  - de remarcat că, pentru schimbarea codificării, lungimea mărimii cazului nu se schimbă, va rămâne tot în ordinul unui logaritm de mărimea cazului
  - prin compunere, rezultă un algoritm de rezolvare pentru codificarea $e_2(Q) \in P$
- codificarea ne-unară a instanțelor problemei nu modifică natura polinomială/suprapolinomială a calculabilității sale
- astfel, nu se mai face distincție între probleme abstracte și probleme concrete

Limbaje formale

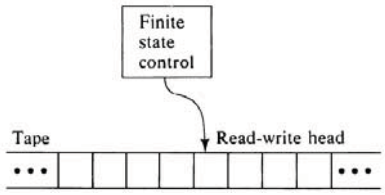
- limbajele formale reprezintă un instrument facil pentru exprimarea problemelor de decizie asupra codificărilor
- un **limbaj** L peste un **alfabet** Σ este orice mulțime de șiruri formate cu simboluri din Σ
  - $\Sigma = \{0, 1\}$, $L = \{10, 11, 101, 111, 1011, 1101, \dots\}$, L - limbajul reprezentării binare a numerelor prime
- limbajul tuturor șirurilor peste Σ este notat cu L^*
  - $L^* = \{\lambda, 0, 1, 00, 01, 10, 11, 000, \dots\}$, unde λ este șirul vid
- complementul lui L, $\overline{L} = \Sigma^* - L$
- concatenarea limbajelor L_1 și L_2 este limbajul $L = \{x_1x_2 \mid x_1 \in L_1, x_2 \in L_2\}$
- închiderea sau operatorul Kleene-star este limbajul $L^* = \lambda \cup L \cup L^2 \cup L^3 \cup \dots$, unde L^k este limbajul obținut prin concatenarea limbajului L cu el însuși de k ori

- mulțimea instanțelor unei probleme Q este o submulțime a lui Σ^*, unde $\Sigma = \{0, 1\}$
- mai precis, Q poate fi privită ca un limbaj L peste Σ, $L = \{x \in \Sigma^* \mid Q(x) = 1\}$, adică acele șiruri din limbaj (instanțe) pentru care problema de decizie are un rezultat afirmativ (1)
  - de exemplu, problema de decizie a drumului determină limbajul $DRUM = \{G = (V, M), \{u, v\}, k\}$, unde graful $G = (V, M)$ este un graf neorientat, $u, v \in V$ sunt cele două vârfuri între care există un drum a cărui lungime este cel mult k, cu $k \geq 0$
- un algoritm A **acceptă** un șir de intrare $x \in \{0, 1\}^*$ dacă, pentru intrarea x, algoritmul dă la ieșire un răspuns afirmativ, $A(x) = 1$
- limbajul acceptat de algoritmul A este $L = \{x \in \{0, 1\}^* \mid A(x) = 1\}$, adică mulțimea șirurilor pe care le acceptă
- un algoritm A **respinge** șirul x dacă $A(x) = 0$
  - nu e obligatoriu pentru un algoritm să accepte sau să respingă un șir de intrare; de exemplu, el poate intra într-o buclă infinită

- un limbaj L este **clarificat** de un algoritm A dacă fiecare șir ce aparține limbajului este acceptat de A iar fiecare șir ce nu aparține limbajului este respins de A
- un limbaj L este **acceptat în timp polinomial** de algoritmul A dacă există o constantă $k \geq 0$ a.î. pentru orice $x \in L$ de lungime n, algoritmul îl acceptă pe x într-un timp $O(n^k)$
- un limbaj L este **clarificat în timp polinomial** dacă pentru orice șir $x \in \{0,1\}^*$ algoritmul decide în mod corect dacă $x \in L$ într-un timp $O(n^k)$, unde k este o constantă
  - exemplu: limbajul *DRUM* poate fi acceptat în timp polinomial
  - un algoritm poate funcționa astfel:
    - găsește cel mai scurt drum dintre u și v prin căutarea în lățime
    - compară lungimea găsită cu k
    - dacă distanța este cel mult k, algoritmul întoarce 1 și se oprește
    - altfel, intră într-o buclă infinită
  - algoritmul acceptă limbajul *DRUM* în timp polinomial, dar el nu poate clarifica limbajul *DRUM*, deoarece nu returnează 0 dacă cel mai scurt drum este mai mare decât k
  - totuși, pentru limbajul *DRUM*, este ușor de proiectat im algoritm ce îl clarifică

- **clasa de complexitate** se definește ca mulțimea limbajelor $L \in \Sigma^*$ pentru care algoritmul ce clarifică limbajul L are o anumită complexitate
- **clasa aloritmilor polinomiali** P este dată de mulțimea limbajelor L, unde L este un limbaj ce este acceptat de un algoritm în timp polinomial
  - fie A un algoritm ce acceptă L în timp polinomial
  - există o constantă c pentru care A acceptă L în cel mult $T = cn^k$ pași
  - construim un algoritm A' care simulează acțiunile realizate de A pentru T pași
  - dacă după T pași A se termină cu acceptare, A' va întoarce și el cu acceptare
  - în caz contrar, A' va opri simularea algoritmului A și va respinge instanța problemei
  - astfel, am construit un algoritm care clarifică, în timp polinomial, limbajul L, pornind de la un algoritm care acceptă L în timp polinomial
  - clasa limbajelor acceptate în timp polinomial este aceeași cu clasa limbajelor clarificate în timp polinomial

Mașina Turing



- MT formalizează noțiunea de algoritm
- **Mașina Turing** (MT) este cuplul $TM = (\Sigma, Q, f, \#, q_0)$, unde:
  - Σ este alfabetul simbolurilor ce pot fi scrise pe banda MT
  - Q este un set finit de stări
  - funcția de tranziție $f : \Sigma \times Q \to \Sigma \times Q \times \{left, right, -\}$, care, condiționată de simbolul și starea curentă determină înscrierea simbolului pe poziția curentă a capului, starea următoare și comandă avansul (sau nu) al capului de citire/scriere (left/right/stă pe loc)
  - $\# \in \Sigma$ este simbolul ce delimitează locațiile utile ale benzii
  - $q_0 \in Q$ este starea inițială a automatului

Adunarea numerelor

- pentru fiecare problemă va trebui compus un alt algoritm de funcționare (automat cu stări)
- reprezentarea unui număr pozitiv n în unar constă din $n + 1$ cifre de 1 delimitate de zerouri
- reprezentarea numărului 3 pe banda MT este #011110#, a lui 0 este #010#
- calculul se termină când MT se află într-o stare finală, rezultatul pe bandă iar capul de citire/scriere indică prima cifră a numărului
- considerăm că inițial pe bandă se află cele două numere, #$\boxed{0}$111011110#, capul de citire/scriere aflîndu-se în dreptul primului 0

Automatul MT pentru adunare

$f(q_0, 0) = (q_0, 0, R)$ $\#\boxed{0}111011110\#$ {se citește primul 0}
$f(q_0, 1) = (q_1, 1, R)$ $\#0\boxed{1}11011110\#$ {se găsește primul 1}
$f(q_1, 1) = (q_1, 1, R)$ $\#01\boxed{1}1011110\#$ {parcurge primul număr}
$f(q_1, 0) = (q_2, 1, R)$ $\#0111\boxed{0}11110\#$ {s-a găsit zeroul dintre numere}
$f(q_2, 1) = (q_2, 1, R)$ $\#01111\boxed{1}1110\#$ {parcurge al doilea număr}
$f(q_2, 0) = (q_3, \#, L)$ $\#011111111\boxed{0}\#$ {suprascrie ultimul 0}
$f(q_3, -) = (q_4, \#, L)$ $\#01111111\boxed{1}\#\#$ {suprascrie ultimul 1}
$f(q_4, -) = (q_5, 0, L)$ $\#0111111\boxed{1}\#\#\#$ {suprascrie penultimul 1}
$f(q_5, 1) = (q_5, 1, L)$ $\#011111\boxed{1}0\#\#\#$ {caută începutul numărului}
$f(q_5, 0) = (q_6, 0, -)$ $\#\boxed{0}1111110\#\#\#$ {starea finală}
$f(q_6, -) = (q_6, 0, -)$

Conversia unar-binar

$f(q_0, 0)$	$= (q_0, 0, R)$	{citește primul 0}
$f(q_0, 1)$	$= (q_1, 1, R)$	{caută ultimul 0}
$f(q_1, 1)$	$= (q_1, 1, R)$	
$f(q_1, 0)$	$= (q_2, 0, L)$	{a găsit zeroul, merge invers}
$f(q_2, -)$	$= (q_3, X, R)$	{pune X peste ultimul 1}
$f(q_3, s \neq \#)$	$= (q_3, s, R)$	{sare succesiv}
$f(q_3, \#)$	$= (q_4, \#, L)$	{a găsit $\#$}
$f(q_4, 1)$	$= (q_4, 0, L)$	{incrementează, în loc de 1 pune 0}
$f(q_4, s \neq 1)$	$= (q_5, 1, L)$	{se pune c.m.s. cifră binară}
$f(q_5, X)$	$= (q_5, X, L)$	{se sare peste toți X-ii spre stânga}
$f(q_5, s \neq X)$	$= (q_6, s, -)$	{s-a găsit un non-X}
$f(q_6, 1)$	$= (q_3, X, R)$	{pune X peste acel 1, și repetă}
$f(q_6, 0)$	$= (q_7, \#, R)$	{pune $\#$ peste primul 0}
$f(q_7, s \neq 1)$	$= (q_7, \#, R)$	{pune $\#$ până la primul 1}
$f(q_7, 1)$	$= (q_8, 1, L)$	{s-a găsit primul 1}
$f(q_8, -)$	$= (q_8, 0, -)$	{starea finală}

Mașina Turing și clasa P

- pentru ca MT să corespundă noțiunii de algoritm, trebuie să se oprească pentru fiecare șir al alfabetului, fie că este acceptat sau respins de acesta
- MT poate rezolva orice calcul ce poate fi programat pe o mașină convențională (dar mult mai încet)
- o mașină Turing M **rezolvă** o problemă de decizie folosind program $e(M)$ (automatul cu stări ce caracterizează MT) dacă acea mașină M se oprește pentru orice șir din alfabetul de intrare (definiție echivalentă cu **acceptarea** limbajului de către algoritmul ce definește funcționarea MT)
- clasa **algoritmilor polinomiali din perspectiva MT** este dată de mulțimea limbajelor ce pot fi rezolvate de MT cu un algoritm al cărui număr de pași este în ordin de timp polinomial

Mașina Turing Universală

- algoritmul MT trebuie definit pentru fiecare problemă în parte
- ne-ar trebui o mașină al cărei algoritm să nu depindă de problemă
- mașina universală ar trebui să interpreteze o reprezentare a unei MT clasice stocată pe aceeași bandă (zona de program)
- **Mașina Turing Universală** (MTU) este o mașină Turing $U(e(M), T)$, T este banda MT iar $e(M)$ este șirul (programul) ce caracterizează MT, stocat pe aceeași bandă ca și datele programului (șirul ce trebuie procesat)
- funcția de tranziție (programul) poate fi redus la un set de perechi de forma:

$$f(q_i, Tape = x) = (q_j, y, c_t)$$
$$f(q_i, Tape \neq x) = (q_k, z, c_f)$$

unde $q_i, q_j \in Q$, $x, y, z \in \Sigma$ iar $c_t, c_f \in \{left, right, -\}$

- fiecare astfel de pereche va fi asociată cu o stare a automatului, reprezentată ca o succesiune de tuple de forma:

$$\&, q_i, x, q_j, y, c_t, q_k, z, c_f$$

  unde & este simbolul ce indică începutul specificării stării q_i
- banda MTU va fi divizată în două secțiuni, una pentru șirul T de procesat și alta pentru descrierea $e(M)$ a MT ce trebuie interpretată (simulată)
- față de MT, MTU va avea în plus un cap de citire/scriere ce parcurge descrierea $e(T)$ a MT și un registru R ce servește stocării instrucțiunii curente
- R poate fi comandat să încarce sau nu instrucțiunea din ciclul curent, și este folosit la căutarea pe bandă a stării specificate în instrucțiune

Problema opririi

- pentru o mașină Turing M, se pune întrebarea dacă mașina M se va opri după un număr finit de pași sau nu
- considerăm că problema opririi este calculată de o MTU și întoarce 1 dacă mașina M cu conținutul inițial al benzii T se oprește, altfel întoarce 0:

$$H(e(M), T) = 1 \text{ dacă } M(T) \text{ se oprește}$$
$$H(e(M), T) = 0 \text{ dacă } M(T) \text{ rulează indefinit}$$

- Funcția $H(e(M), T)$ **nu este calculabilă**.
  - presupunem că există o MT H pentru orice descriere de MT și pentru orice conținut al benzii de intrare
  - astfel, se poate realiza o MT G astfel încât pentru orice MT F, $G(e(F))$ se oprește dacă $H(e(G), e(F)) = 0$, respectiv $G(e(F))$ nu se oprește dacă $H(e(G), e(F)) = 1$
  - mașina G este fezabilă deoarece am presupus că funcția H e calculabilă

- încercăm să vedem ce putem spune despre $H(e(G), e(G))$ - punem mașina să lucreze pe propria ei descriere
- presupunem că $H(e(G), e(G)) = 1$, atunci înseamnă că mașina $G(e(G)$ se oprește, însă prin ipoteza făcută, mașina se oprește doar dacă $H(e(G), e(G)) = 0$
- presupunem că $H(e(G), e(G)) = 0$, asta înseamnă că mașina nu se oprește, dar plecînd de la ipoteza făcută, mașina rulează indefinit doar pentru $H(e(G), e(G)) = 1$
- aplicarea funcției H la mașina G generează o contradicție
- deoarece H este definită pentru orice descriere și pentru orice conținut al benzii, concluzionăm că ea nu e realizabilă

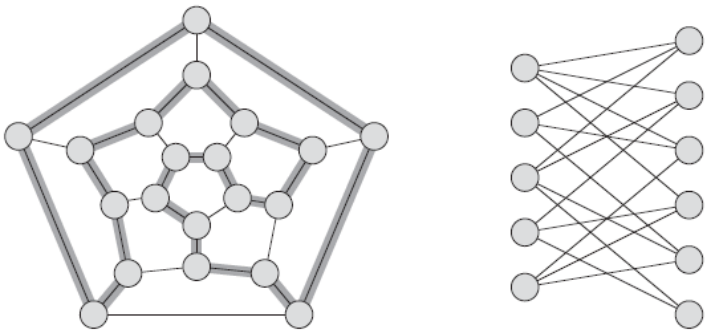
- nu există un calcul prin care, dîndu-se $e(F)$ și T, să aflăm dacă acel calcul se poate face
- în general autoreferirea duce la paradoxuri
  - afirmația 'eu mint', este adevărată sau falsă?
  - foaie de hârtie, față = 'prop. de pe verso e falsă', verso = 'prop. de pe față este adevărată'

Teza Church-Turing

- **teza Church-Turing:** orice funcție discretă ce poate fi calculată de o mașină realizabilă, este de asemenea calculabilă de către o Mașină Turing
  - nu avem o dovadă pentru ea
  - doar un contraexemplu ar putea dovedi inconsistența acestei teze (nu s-a găsit nici unul până acum)
  - modelul Mașinii Turing a devenit un model de referință pentru calculabilitate
  - calculabilitatea în sens Turing demonstrează validitatea unui model de calcul propus
  - orice mașină care poate executa funcțiile realizate de o MT poate fi folosită pentru a implementa o Mașină Universală

Cicluri hamiltonieie

- presupunem că, pentru o instanță dată $(G, \{u, v\}, k)$ a problemei *DRUM*, avem dat și un drum de la u la v
  - verificarea se face foarte ușor, nu va necesita un ordin de timp mai mare decât ordinul de timp al problemei inițiale
  - clarificarea se face cu un algoritm polinomial
- **ciclul hamiltonian** al unui graf $G = (V, M)$ este un ciclu elementar care conține fiecare nod din V
- un graf este denumit **graf hamiltonian** dacă conține un ciclu hamiltonian
  - dodecaedrul este un graf hamiltonian
- problema ciclului hamiltonian: pentru un graf G dat, să se determine dacă graful este un graf hamiltonian
  - un posibil algoritm de clarificare ar verifica fiecare permutare a vârfurilor G, dacă este un ciclu hamiltonian
  - dacă n este numărul vârfurilor, generarea tuturor permutărilor este în $\Omega(n!) = \Omega(2^n)$, care este suprapolinomial



  - (a) un dodecaedru este un graf hamiltonian
  - (b) un graf bipartit cu număr impar de vârfuri; orice astfel de graf nu este hamiltonian
- este o problemă NP completă

Algoritmi de verificare

- dacă ciclul hamiltonian este dat, verificarea corectitudinii acestuia se face într-un timp în $O(n^2)$
- un **algoritm de verificare** $A(x, y)$ verifică dacă, pentru o instanță a problemei x, șirul y dat drept probă poate constitui o soluție a problemei
- **limbajul verificat** de un algoritm de verificare A este

$$L = \{x \in \{0,1\}^* \mid \text{există} y \in \{0,1\} * \text{pentru care} A(x, y) = 1\}$$

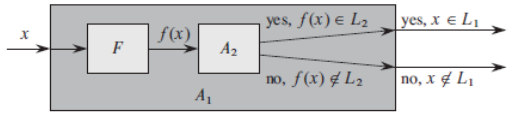
- pentru un șir $x \notin L$, nu există nici o probă y pentru care algoritmul de verificare $A(x, y) = 1$

Clasa de complexitate NP

- vom considera că un **algoritm nedeterminist** rezolvă problema deciziei în două etape:
  1. algoritmul 'ghicește' o structură S
  2. se verifică dacă structura găsită S este o soluție pentru problema inițială, într-o manieră deterministă, returnînd fie da/nu, fie rulînd la infinit
- un algoritm nedeterminist **rezolvă** o problemă de decizie Q dacă, pentru toate instanțele $x \in \{0,1\}^*$:
  1. dacă $x \in L$, atunci există o structură S care, atunci când este 'ghicită' pentru intrarea x, va determina $A(x, y) = 1$
  2. dacă $x \notin L$, atunci nu există nici o structură S care să determine acest lucru

- **clasa de complexitate NP** este clasa limbajelor ce pot fi verificate de un algoritm în timp polinomial
  - problema deciziei ciclului hamiltonian este o problemă NP
  - dacă o problemă $L \in P$, atunci $L \in NP$, deoarece dacă există un algoritm polinomial care clarifică L, atunci acesta poate fi modificat într-unul care acceptă doar acele probe care sunt și soluții ale lui L
- putem concluziona că $P \subseteq NP$
- problemă deschisă: **este P = NP sau P $\neq$ NP** ?
- intuitiv:
  - clasa P este cea a problemelor ce pot fi rezolvate rapid
  - clasa NP este cea a problemelor ce pot fi verificate rapid

Reductibilitate

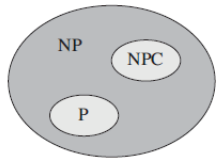


- un limbaj L_1 este **reductibil în timp polinomial** la un limbaj L_2 dacă există o funcție calculabilă în timp polinomial $f : \{0,1\}^* \to \{0,1\}^*$ astfel că oricare ar fi $x \in \{0,1\}^*$:

$$x \in L_1 \text{ dacă și numai dacă } f(x) \in L_2$$

- **închiderea**: dacă $L_1, L_2 \in \{0,1\}^*$ sunt limbaje astfel că L_1 este reductibil în timp polinomial la L_2, atunci dacă $L_2 \in P$ avem că și $L_1 \in P$
  - algoritmul se execută în timp polinomial deoarece atât f cât și A_2 se execută în timp polinomial

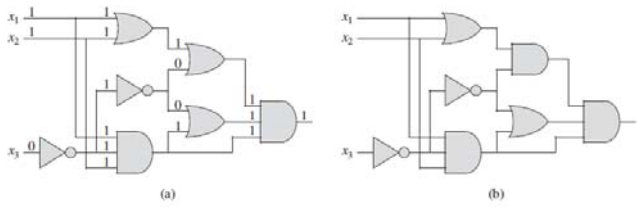
NP-completitudine



- reducerea în timp polinomial arată că o problemă este cel puțin la fel de dificilă ca alta
- un limbaj L este **NP-complet** dacă:
  1. $L \in NP$
  2. oricare ar fi $L' \in NP$, L' este reductibil în timp polinomial la L
- un limbaj este **NP-dificil** dacă satisface numai cea de-a doua din proprietățile de mai sus
- **clasa NPC** este clasa problemelor NP-complete

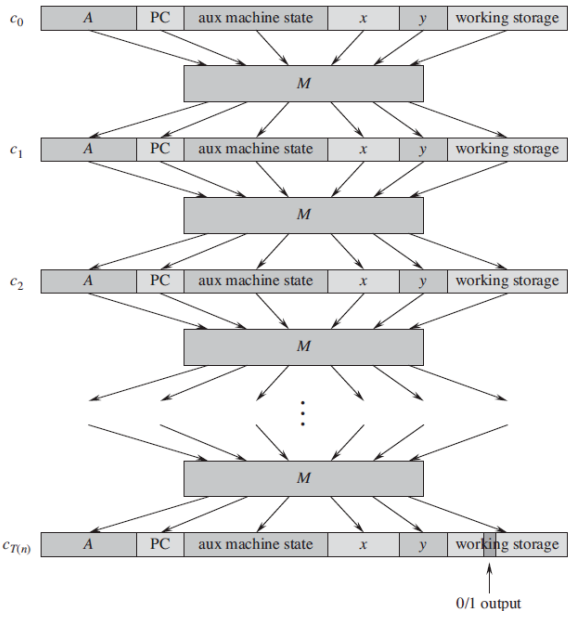
- dacă orice problemă NP-completă este rezolvabilă în timp polinomial, atunci $P = NP$
  - fie $L \in P$ o problemă NP-completă
  - oricare ar fi $L' \in NP$, L' este reductibilă în timp polinomial la L
  - atunci și acest $L' \in P$
- deocamdată se crede că $P \neq NP$
- nu s-a exclus varianta găsirii unui algoritm care să rezolve o problemă NP-completă în timp polinomial - s-ar demonstra că $P = NP$
- o demonstrație a faptului că o problemă este NP-completă este acceptată ca dovadă a inaccesibilității sale

Satisfiabilitatea circuitului



- un circuit este **satisfiabil** dacă are o atribuire de adevăr pe variabilele de intrare care determină ieșirea 1
  - (a) circuitul este satisfiabil
  - (b) nu există nici o atribuire pentru care circuitul să fie satisfiabil
- **problema satisfiabilității circuitului:** dându-se un circuit combinațional logic format din porți AND, OR, NOT, este el satisfiabil?

- problema satisfiabilității circuitului (CIRC-SAT) aparține clasei NP.
  - se poate imagina un algoritm A ce primește la intrare, pe de o parte descrierea polinomială a circuitului, ca funcție de intrări, iar pe de altă parte, o probă reprezentînd valori ale intrărilor circuitului
  - algoritmul A de verificare se execută în timp polinomial
  - avem că problema $CIRC - SAT \in NP$
- pentru demonstrarea CIRC-SAT este NP-dificilă, ne bazăm pe funcționarea hardware-ului calculatorului
- o **configurație** a calculatorului este dată de starea memoriei la un moment dat
- putem vedea execuția unei instrucțiuni ca o funcție între două configurații
- partea hardware ce realizează această funcție ca un CLC, notat cu M



- problema satisfiabilității circuitului (CIRC-SAT) este NP-dificilă.
  - fie $L \in NP$ un limbaj; vom descrie un algoritm polinomial pentru calculul funcției de reducere ce realizează corespondența dintre șirul x și un circuit $C = f(x)$, astfel încât $x \in L$ dacă și numai dacă $C \in CIRC - SAT$
  - deoarece $L \in NP$, există un algoritm polinomial A ce verifică L
  - reprezentăm calculele făcute de A ca o serie de configurații (programul A, PC, intrarea x, proba y, workspace)
  - algoritmul are un număr de pași în $T(n) = O(n^k)$, algoritm de decizie (ieșirea 0/1)
  - algoritmul de reducere construiește un singur CLC ce calculează toate configurațiile - $T(n)$ copii ale circuitului M
  - algoritmul de reducere construiește circuitul $C = f(x)$ care este satisfiabil dacă și numai dacă există o probă y pentru care $A(x, y) = 1$

- dacă există o probă, circuitul calculează $A(x, y) = 1$, circuitul furnizează ieșirea 1, deci este satisfiabil
- invers, dacă este satisfiabil, trebuie că există o intrare y pentru care circuitul returnează 1, deci există $A(x, y) = 1$
- numărul de biți ce reprezintă o configurație e funcție polinomială de n
- programul are lungime constantă
- algoritmul are $O(n^k)$ pași, deci spațiul de memorie pentru A e în ordin polinomial
- proba y este de lungime polinomială
- circuitul C constă în cel mult $O(n^k)$ copii ale lui M, iar M este polinomial, depinzînd de lungimea configurației
- construcția lui C din x se face în timp polinomial, fiecare pas necesită timp polinomial (număr finit, polinomial, de pași)

- cum orice problemă se poate reduce la CIRC-SAT, iar $CIRC - SAT \in NP$, CIRC-SAT este NP-dificilă

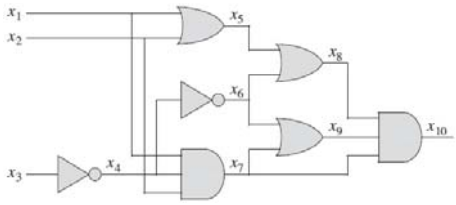
Demonstrații ale NP-completitudinii

- se poate realiza o metodologie de a arăta că un limbaj L este NP complet fără a reduce fiecare limbaj din clasa NP la el
- Dacă L este un limbaj astfel că există un limbaj $L' \in NPC$ care se reduce la el, atunci L este NP-dificil. Dacă $L \in NP$, atunci $L \in NPC$
  - $L' \in NPC$, deci orice limbaj L'' din NP se reduce la L'
  - dar L' se reduce la L
  - deci orice limbaj $L'' \in NP$ se reduce la L, deci L este NP-dificil
- va fi suficient să arătăm că un limbaj L este în NP, că un $L' \in NPC$ se reduce la el și că funcția de reducere este polinomială, atunci $L \in NPC$

Satisfiabilitatea formulelor

- problema de satisfacere a unei formule logice
- instanța SAT este formată din:
  1. variabilele logice $x_1, x_2, \ldots$
  2. funcții booleene cu una sau două intrări și o singură ieșire, precum $\wedge$ (AND), $\vee$(OR), $\neg$(NOT), $\leftarrow$(implică), $\leftrightarrow$(dacă și numai dacă)
  3. paranteze
- o **atribuire satisfiabilă** este o mulțime de valori pentru care evaluarea expresiei are rezultatul 1
  $((x_1 \leftarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$
  are atribuirea satisfiabilă
  $(x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1)$
- algoritmul naiv poate verifica 2^n combinații de intrare

- (Teorema Cook) Problema satisfiabilității formulei logice (SAT) este NP-completă.
  - demonstrăm mai întâi că $SAT \in NP$ și apoi că o problemă NP-completă (CIRC-SAT) se reduce la ea
  - pentru fiecare probă (colecție de valori x) formula se poate evalua în timp polinomial, așadar $SAT \in NP$
  - dacă, pornind de la ieșire, înlocuim fiecare poartă cu o funcție logică și apoi intrările porții le înlocuim cu formule, putem obține creșterea exponențială a formulei generate (dați un exemplu)
  - formula unei porți AND -$x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)$ - poate fi descrisă de conjuncția dintre varibila ce reprezintă firul de ieșire și predicatul format de intrări



- se obține formula

$$x_{10} \wedge (x_4 \leftrightarrow \neg x_3) \wedge (x_5 \leftrightarrow (x_1 \vee x_2)) \wedge (x_6 \leftrightarrow \neg x_4) \wedge$$
$$(x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4)) \wedge (x_8 \leftrightarrow (x_5 \vee x_6)) \wedge (x_9 \leftrightarrow (x_6 \vee x_7)) \wedge$$
$$(x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9))$$

- formula se obține în timp polinomial
- circuitul este satisfiabil dacă și numai dacă formula este satisfiabilă
- am arătat că problema SAT se reduce în timp polinomial la SAT-CIRC; mai mult, $SAT \in NP$, deci $SAT \in NPC$

Problema satisfiabilității 3-FNC

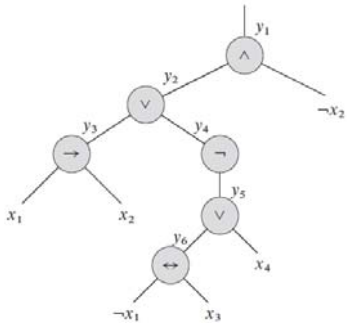
- o serie de probleme NP pot fi demonstrate ca fiind NP-complete prin demonstrarea reductibilității 3-FNC la ele
- O formulă logică se află în **formă normală conjunctivă**(FNC) dacă este formată prin conjuncția mai multor propoziții, fiecare propoziție fiind formată din disjuncția mai multor literali
- O formulă logică se află în **a treia formă normală conjunctivă**(3-FNC) dacă fiecare propoziție conține exact trei literali distincți

$$(x_1 \vee \neg x_1 \vee x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee x_3 \vee x_4)$$

- se cere să determinăm dacă o propoziție 3-FNC este satisfiabilă

- Problema satisfiabilității formulelor logice 3-FNC este NP-completă.
  - încercăm să reducem SAT (problemă NP-completă) la ea
  - construim un arbore 'gramatical' (fiecare nod va avea unul sau doi fii)

$$((x_1 \leftarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$



  - la fel ca anterior, introducem o variabilă y_i pentru ieșirea fiecărui nod intern

$$\phi = y_1 \wedge (y_1 \leftrightarrow (y_2 \vee \neg x_2)) \wedge (y_2 \leftrightarrow (y_3 \vee y_4)) \wedge (y_3 \leftrightarrow (x_1 \leftarrow x_2)) \\ \wedge (y_4 \leftrightarrow \neg y_5) \wedge (y_5 \leftrightarrow (y_6 \vee x_4)) \wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3))$$

  - se obține o conjuncție de propoziții, fiecare dintre ele având cel mult trei literali
  - fiecare propoziție ϕ_i va fi convertită apoi în FNC
  - putem face tabelul de adevăr pentru fiecare propoziție și, pe baza zerourilor, să scriem FND (forma normală disjunctivă, SAU de ȘI-uri) echivalent cu $\neg\phi_i$

$$\neg\phi_1 = (y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee \\ (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2) \text{ cu relațiile lui de Morgan:}$$
$$\phi_1 = (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \wedge \\ (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2)$$

- ϕ ajunge să fie o conjuncție de propoziții de maxim trei literali
- fiecare propoziție de doi literali, $l_1 \vee l_2$ se poate transforma în $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$, unde p este o variabilă auxiliară, rezultatul nedepinzînd de valoarea ei
- fiecare propoziție de un literal l poate fi scrisă ca $(l \vee p \vee q) \wedge (l \vee p \vee \neg q) \wedge (l \vee \neg p \vee q) \wedge (l \vee \neg p \vee \neg q)$
- forma finală 3-FNC este satisfiabilă dacă și numai dacă ϕ este satisfiabilă
- reducerea poate fi făcută în timp polinomial
  1. se introduc variabilele corespunzătoare propozițiilor ('liniile' circuitului)
  2. construcția unei formule FNC echivalente introduce cel mult 8 propoziții pentru fiecare propoziție anterioară
  3. la construcția formei finale se introduc cel mult 4 propoziții
- mărimea formulei depinde, după o formă polinomială, de formula inițială
- așadar SAT se reduce în timp polinomial la 3-FNC
- de asemenea $3-FNC \in NP$ (o probă se verifică în timp polinomial), deci $3-FNC \in NPC$

Probleme NP-complete

- Problema clicii
  - **clica**, într-un graf neorientat $G = (V, M)$, este o submulțime $V' \subseteq V$ cu proprietatea că între fiecare pereche de vârfuri din V există o muchie în M - subgraf complet al lui G
  - **problema clicii** constă în găsirea clicii cu cel mai mare număr posibil de vârfuri
- Problema acoperirii cu vârfuri
  - o **acoperire cu vârfuri** a unui graf neorientat $G = (V, M)$ este o submulțime $V' \subseteq V$ cu proprietatea că, dacă $(u, v) \in M$, cel puțin unul din vârfurile u, v face parte din V'
  - o astfel de mulțime de vârfuri acoperă toate muchiile din G
  - dimensiunea acoperirii este dată de numărul vârfurilor continute
  - **problema acoperirii** constă în determinarea unei acoperiri de dimensiune minimă

Probleme NP-complete (2)

- Problema sumei submulțimii
  - pentu o mulțime finită $S \subset \mathbb{N}$ și o țintă $t \in \mathbb{N}$, se cere verificarea faptului că există o submulțime $S' \subset S$ pentru care suma elementelor componente este t
- Problema ciclului hamiltonian
- Problema comis-voiajorului (TSP)
  - parcurgerea ciclului hamiltonian de cost minim

Cursul nr. 6

COMPLEXITATE PARALELĂ

Cuprins

Complexitatea paralelă

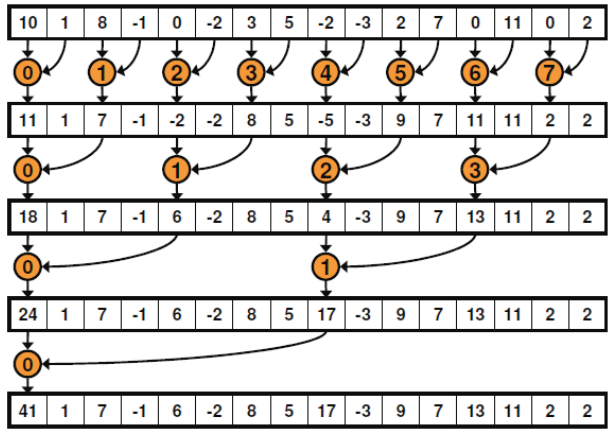
Modelul PRAM

Teza calculului paralel

O problemă de sortare

- algoritmul de sortare pe o mașină secvențială pentru un caz de mărime n realizează un timp în $O(n \log n)$
- am dori să accelerăm acest lucru folosind o mașină paralelă
  - arhitectură simplă
  - colecție apreciabilă de procesoare identice
  - procesoarele pot citi/scrie dintr-o/într-o memorie partajată pentru un cost constant
- mașina are $p(n)$ procesoare
- prima abordare, pentru $p(n) < n$ și constant:
  - împarte șirul de sortat $x_1, x_2, \ldots, x_n$ în $p(n)$ șiruri
  - sortează pe fiecare procesor șirul format din $n/p(n)$ elemente, în $O((n/p(n))log(n/p(n)))$
  - interclasează rezultatele pe perechi de procesoare, într-un timp dominat de interclasarea ultimelor două șiruri, $O(n)$
  - timpul total - $O\left(n + \frac{n}{p(n)} log \frac{n}{p(n)}\right)$
  - pentru $p(n)$ o constantă independentă de n, timpul tinde asimptotic la $O(n \log n)$
- a doua abordare:
  - pentru $p(n) \leq log\ n$, număr de procesoare mic relativ la mărimea problemei, timpul este în $O((n \log n)/p(n))$, astfel că adăugarea de procesoare până la acest prag accelerează rezolvarea față de timpul secvențial
- a treia abordare:
  - adăugarea de procesoare, $p(n) > \log n$, timpul va tinde la $O(n)$
  - adăugarea de procesoare nu mai crește eficiența algoritmului
- pentru o îmbunătățire exponențială a algoritmului, ar trebui să obținem un timp de sortare în $O(log\ n)$, sau măcar în $O((log\ n)^2)$

- a patra abordare:
  - dacă pentru valorile $x_1, x_2, \ldots, x_n$ am avea calculate rangurile $r_1, r_2, \ldots, r_n$, într-un singur pas, în paralel, procesoarele ar rearanja valorile, procesorul p_i ar realiza atribuirea $x_i \leftarrow x_{r_i}$
  - am putea face ranking-ul pe n^2 procesoare astfel:
    1. fiecare procesor realizează testul $x_i > x_j$ și setează $c_{ij} \leftarrow 1$ dacă este adevărat sau 0 în caz contrar
    2. pentru fiecare i, n procesoare calculează $\sum_{j=1}^{n} c_{ij} = r_i$; calculul sumei se face în $O(log\ n)$ (arbore de reducere)



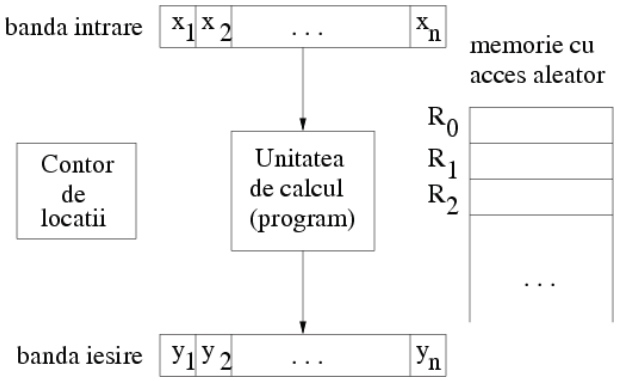
  1. pasul 1 merge în timp constant
  2. pasul 2 se realizează în timp logaritmic
- folosind n^2 procesoare, se realizează sortarea în $O(log\ n)$

- în general soluțiile paralele găsite pentru rezolvarea problemelor vor folosi un **număr polinomial de procesoare** și vor avea un **timp de rulare polilogaritmic**
- în general acești algoritmi sunt **fezabili**, pentru că folosesc un număr polinomial de procesoare și sunt **paralel rapizi** în sensul că timpul este în $O(log^k\ n)$ (polilogaritmici)
- există însă situații în care probleme cu algoritmi secvențiali polinomiali nu pot fi rezolvate în timp polilogaritmic (paralel rapid) pe un număr polinomial de procesoare
- am dori o teorie similară cu teoria NP-completitudinii, însă pe cazul paralel
- o problemă P-completă este inerent secvențială, adică este puțin probabil să găsim o soluție paralelă rapidă pentru orice caz

Fezabilitate paralelă

- o problemă are o soluție **fezabilă** doar dacă avem un algoritm secvențial polinomial care o rezolvă, adică în timp $n^{O(1)}$
- dihotomia între probleme rezolvabile polinomial și cele rezolvabile în timp suprapolinomial este un instrument foarte util al fezabilității
- pentru programarea paralelă, și procesoarele sunt considerate resursă, similar cu $s(n)$ - mărimea memoriei - pentru cazul secvențial
  - o problemă de mărime n este **paralel fezabilă** dacă se poate rezolva pe o mașină paralelă folosind $n^{O(1)}$ procesoare și cu timpul pentru cel mai nevaforabil caz tot în $n^{O(1)}$
  - o problemă este **fezabilă cu grad ridicat de paralelism** dacă poate fi rezolvată de un algoritm cu timpul pentru cazul cel mai nefavorabil în $log^{O(1)}n$ și procesoare în $n^{O(1)}$
  - o problemă este **inerent secvențială** dacă este fezabilă dar nu are un algoritm cu grad ridicat de paralelism pentru găsirea soluției
- clasa problemelor **paralel fezabile** este aceeași cu clasa P (clasa problemelor rezolvabile în timp polinomial cu algoritmi secvențiali)

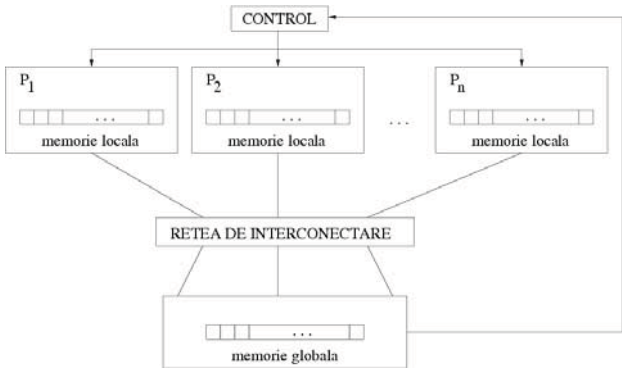
Modelul RAM



- Random Access Machine, model de calcul uzual pentru proiectarea algoritmilor secvențiali
- programul se realizează sub forma de automat cu stări, nu poate fi modificat de mașină în funcționarea ei

- programul citește un întreg de pe banda de intrare, apoi avansează capul de citire
- se scrie un întreg pe banda de ieșire, apoi se avansează capul de scriere
- memoria este alcătuită dintr-o secvență infinită de regiștri R_0, R_1, ..., fiecare putînd conține un singur întreg
- instrucțiuni RAM: load, store, read, write, add, subtract, multiply, divide, test, jump, halt
- orice operație se execută în unitatea de timp
- modelul este echivalent cu o Mașină Turing
- spre deosebire de MT, memoria pentru modelul RAM are acces aleator
- clasa problemelor rezolvabile pe un RAM este aceeași cu clasa problemelor rezolvabile pe MT (din teza Church-Turing)

Modelul PRAM



- un model PRAM constă dintr-o unitate de control, o memorie globală și o mulțime infinită de procesoare, fiecare cu propria sa memorie locală

- fiecare procesor p_i poate fi activat/dezactivat pentru ciclul curent
- toate procesoarele active execută instrucțiuni identice
- calculul începe cu un singur procesor activ ce citește o locație de memorie locală/globală
- pe parcursul calculului, un procesor activ poate citi o singură locație de memorie locală/globală, procesează operația, rezultatul fiind scris într-o locație de memorie locală/globală
- pe parcursul unui pas de calcul, un procesor activ poate activa alt procesor
- toate procesoarele active trebuie să execute aceeași instrucțiune
- calculul se oprește când se oprește ultimul procesor (halt)

Spațiul de calcul al algoritmilor

- spațiul de calcul $s(n)$ necesar unei mașini Turing este definit ca maximul numărului de celule parcurs de capul de citire/scriere dintre pentru toate cazurile de mărime n
- distincția între spațiul ocupat de cazul de mărime n și spațiul de lucru este realizată clar de modelul RAM, pentru că avem banda de intrare $x_1, x_2, \ldots, x_n$ separată de regiștrii infiniți $R_1, R_2, \ldots$
- $DSPACE(s(n))$ este clasa limbajelor ce pot fi acceptate în spațiu $s(n)$ de o mașină Turing deterministă
  - un program care calculează minimul unui șir de n întregi are o complexitate a spațiului în $s(n) = n + 1 \in O(n)$
  - calculul produsului a două matrice pătrate de dimensiuni $n \times n$ are $s(n) \in O(n^2)$
- $NSPACE(s(n))$ este clasa limbajelor ce pot fi acceptate în spațiu $s(n)$ de o MT nedeterministă

- $PSPACE$ este clasa de complexitate a problemelor de decizie ce pot fi rezolvate de o Mașină Turing într-un spațiu polinomial

$$PSPACE = \cup_{k \in \mathbb{N}} DSPACE(n^k)$$

- un algoritm $L \in P$, nu poate folosi spațiu mai mult decât polinomial, deoarece el funcționează în timp polinomial și face un acces la o singură locație odată

$$P \subseteq NP \subseteq PSPACE$$

- $DSPACE(\log n) \subseteq P$, deoarece o MT deterministă ce folosește spațiu în $O(\log n)$ are cel mult un număr de configurații în ordin polinomial, fiecare putînd apărea cel mult o dată, altfel în funcționarea MT ar apărea bucle infinite

- Clasa problemelor calculabile în spațiu polilogaritmic, $L \in DSPACE(log^k n)$ este numită și POLYLOGSPACE
  - problema $P \subseteq POLYLOGSPACE$ este o problemă deschisă
  - egalitatea ar fi satisfăcută dacă s-ar găsi o problemă P-completă care să facă parte din POLYLOGSPACE
- Un limbaj $L \subseteq \Sigma^*$ este **reductibil în spațiu logaritmic** la un limbaj $L' \subseteq \Sigma^*$ dacă există o funcție f calculabilă în spațiu logaritmic astfel ca pentru orice $x \in \Sigma^*$, $x \in L$ dacă și numai dacă $f(x) \in L'$.
- Pentru o clasă C de limbaje, spunem că un limbaj B este **log-space dificil pentru C** dacă orice limbaj $A \in C$ se reduce în spațiu logaritmic la B. Mai mult, dacă $B \in C$, atunci numim B log-space complet pentru C.
- dacă mulțimea C este P - mulțimea limbajelor calculabile (clarificabile) în timp polinomial, vom denumi acel limbaj B ca fiind P-dificil (log-space dificil pentru P), respectiv P-complet (log-space complet pentru P)

- Un limbaj L este log-space dificil pentru P sau **P-dificil** dacă orice limbaj $L' \in P$ se reduce în spațiu logaritmic la L
  - În plus, dacă $L \in P$, atunci el este **P-complet**
- noțiunea de reductibilitate în spațiu logaritmic este echivalentă celei de reductibilitate în timp polinomial
- la fel cum reductibilitatea în timp polinomial era folosită pentru demonstrația NP-completitudinii, reductibilitatea în spațiu logaritmic e folosită pentru demonstrația P-completitudinii
- un alt fel de a denumi o problemă NP-completă ar fi 'polinomial-time completă pentru NP'

Circuit value problem

- **Problema valorii circuitului (CVP)**: fiind dat un circuit C format din porți logice, CLC - fără bucle de reacție, intrările $x_1, x_2, \ldots, x_n$ și o ieșire y, să se determine dacă ieșirea y are valoarea 1 (true) pentru intrările date.
- **Teoremă (Ladner)**: problema CVP este log-space completă pentru P.
  - orice problemă $L \in P$ se reduce astfel în spațiu logaritmic la CVP
  - problema CVP este pentru teoria P-completitudinii ceea ce este problema SAT (teorema Cook) pentru teoria NP-completitudinii

Teza calculului paralel

- **Teoremă (teza calculului paralel)**: Clasa problemelor rezolvabile pe un PRAM într-un timp $t(n)^{O(1)}$ este egală cu clasa problemelor rezolvabile într-un spațiu de lucru în $t(n)^{O(1)}$ pe un RAM, dacă $t(n) \geq log\ n$
- teorema a fost demonstrată numai pentru cazurile în care $t(n)$ este o funcție polinomială
- astfel, un PRAM poate recunoaște în timp polinomial toate limbajele recunoscute de un RAM în spațiu polinomial
- teorema nu afirmă nimic despre timpul necesar pe RAM; astfel, nu exclude timpul exponențial pe RAM
- teza afirmă că timpul paralel e în relație polinomială cu spațiul secvențial
- **Consecință**: un PRAM poate rezolva probleme NP-complete (pe un RAM) în timp polinomial
  - exemplu: colorarea unui graf cu un număr exponențial de procesoare

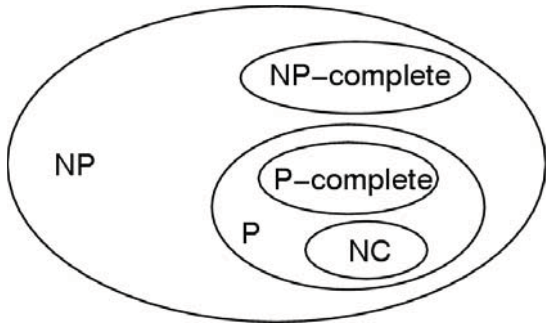
Clasa NC (Nick's Class)

- **Teoremă:** dacă numărul de procesoare într-un PRAM este polinomial, atunci problemele rezolvabile în timp polinomial sunt cele din mulțimea P (cele rezolvabile în timp polinomial pe un RAM).
- **Clasa NC** este clasa de probleme de decizie rezolvabile în timp polilogaritmic ($O(log^c n)$) folosind un număr polinomial de procesoare ($O(n^k)$).

$$NC \subseteq P$$

- exemplu de problemă NC: circuitele hardware procesează operații în timp logaritmic, în funcție de numărul de biți ai operațiunilor, în timp paralel
- este o problemă deschisă dacă $P = NC$
- este însă improbabil ca fiecare problemă din P să fie în NC

- o problemă $L \in P$ este **P-completă** dacă orice altă problemă din P poate fi redusă în timp polilogaritmic la L pe un PRAM cu un număr polinomial de procesoare.
- problemele P complete sunt cele care par să nu admită o rezolvare paralelă eficientă (adică într-un timp polilogaritmic pe PRAM)
  - exemplu de problemă P-completă: parcurgerea depth-first într-un graf



Cursul nr. 7

TEHNICI DE PRELUCRARE A SECVENȚELOR DE ȘIRURI

Cuprins

Algoritmi de căutare în șiruri de caractere

Cea mai lungă subsecvență comună a două șiruri

Distanțe de editare

Alinierea de secvențe

Enunțul problemei, notații

- Problemă: dându–se un text T și un pattern P ca șiruri de caractere, să se determine toate pozițiile din T în care apare P
- Aplicații:
  - găsirea unui subșir într–un document în editoare de text
  - căutarea de subșir într–un șir ADN
  - căutarea de viruși informatici după semnătură
- Textul $T = T[1 \ldots n]$
- Pattern-ul $P = P[1 \ldots m]$, $m \leq n$
- Spunem că P apare cu deplasamentul s în T dacă $0 \leq s \leq n - m$ și $T[s+1 \ldots s+m] = P[1 \ldots m]$
  - $T[s+1 \ldots s+m] = P[1 \ldots m]$ se definește ca: $\forall j \in \{0, \ldots, m-1\},\ T[s+1+j] = P[1+j]$

Enunțul problemei, notații

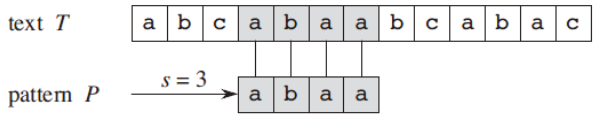


Fig: Un exemplu de căutare cu găsire a unui pattern într–un text. Deplasamentul este $s = 3$: cu 1 mai puțin decât poziția de la care începe P în T.

- Problema căutării cere determinarea *tuturor* deplasamentelor s ale lui P în T

Notații

- mulțimea simbolurilor (caracterelor) din care se compun P și T este alfabetul Σ;
- $|\Sigma|$ e numărul de simboluri din Σ
- Σ^* e mulțimea tuturor șirurilor – inclusiv șirul ε de lungime 0 – formate cu simboluri din Σ
- lungimea unui șir $x \in \Sigma^*$: $|x|$
- pentru $x, y \in \Sigma^*$, $x + y$ e concatenarea lui x și y: caracterele lui x urmate de caracterele lui y
- w e **prefix** al lui x – notat: $w \sqsubset x$ dacă $x = wy$ pentru $y \in \Sigma^*$
  - $abba \sqsubset abbabbb$
- w e **sufix** al lui x – notat: $w \sqsupset x$ dacă $x = yw$ pentru $y \in \Sigma^*$
  - $abbb \sqsupset abbabbb$
- pentru un șir $X[1 \ldots p]$, notăm $X_k = X[1 \ldots k]$, $k = 0, \ldots, p$; $X_0 = \varepsilon$

Rezultate auxiliare

Lema (Lema sufixelor suprapuse)

Să presupunem că avem 3 șiruri x, y, z astfel încât $x \sqsupset z$ și $y \sqsupset z$. Dacă $|x| \leq |y|$, atunci $x \sqsupset y$. Dacă $|x| \geq |y|$, atunci $y \sqsupset x$. Dacă $|x| = |y|$ atunci $x = y$.

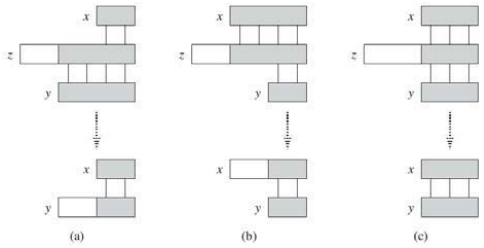


Fig: Exemplificarea celor trei cazuri din lema sufixelor suprapuse. $x, y \sqsupset z$ în fiecare caz.

Algoritmi de căutare de subșiruri

Algoritm	Timp de preprocesare	Timpul de căutare
Naiv	0	$O((n-m+1)m)$
Rabin-Karp	$\Theta(m)$	$O((n-m+1)m)$
Boyer–Moore	$O(\|\Sigma\|+m)$	$O(n)$

Tabel: Algoritmii de căutare de subșiruri

- Presupunem că avem implementată testarea egalității a două șiruri A și B cu $|A| = |B|$ cu complexitate $\Theta(t+1)$ unde t e lungimea celui mai lung șir z ce e prefix atât pentru A și B

Algoritmul naiv de căutare

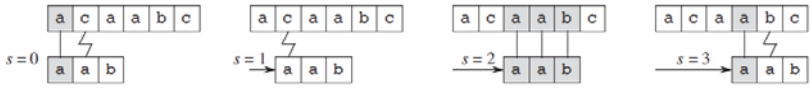


Fig: Exemplu de funcționare pentru căutarea naivă.

```
Cautare-naiva(T, P)
1  m ← length[P]
2  n ← length[T]
3  for s = 0 to n − m
4      if P[1 . . . m] = T[s + 1 . . . s + m]
5          print "pattern-ul apare cu deplasamentul " s
```

Algoritmul naiv de căutare

- Abordare de tip *brute–force*
- Cazul cel mai nefavorabil: $T = \underbrace{aa\ldots a}_{n \text{ ori}} = a^n$, $P = a^m$
- Complexitate: $O((n-m+1)m)$
- Nu necesită pregătiri prealabile, deci timpul de preprocesare este 0
- Deficiența algoritmului: când se trece de la s la $s+1$ se ignoră informația câștigată la pasul s
  - dacă $P = aaab$ și P se găsește în $T = aaab\ldots$ la poziția 1 atunci nu are sens să se caute o eventuală apariție și pentru pozițiile 2, 3, 4 din T

Algoritmul Rabin–Karp

- Algoritm care se comportă bine în practică, chiar dacă complexitatea pentru cazul cel mai nefavorabil este aceeași ca la algoritmul naiv
- Plecând de la niște presupuneri rezonabile, timpul mediu este însă mai bun
- Mecanism esențial: echivalența modulo q a două numere: $a \equiv b \pmod q \Leftrightarrow a \mod q = b \mod q$
  - $23 \mod 4 = 3 = 19 \mod 4$
- Pentru expunere: presupunem că $\Sigma = \{0, 1, \ldots, 9\}$
  - cazul mai general, $|\Sigma| = d$: fiecare simbol va fi interpretat ca un număr în baza d
- Interpretăm un șir de k caractere ca un număr cu k cifre
- Exemplu: șirul de caractere "31415" va fi intepretat ca numărul trezeci și una de mii patru sute cinsprezece

Algoritmul Rabin–Karp

- Pentru pattern-ul $P[1 \ldots m]$ notăm p valoarea sa, calculată de exemplu cu schema lui Horner:

$$p = P[m]+10(P[m-1]+10(P[m-2]+\cdots+10(P[2]+10P[1])\cdots))$$

- Complexitatea calculului valorii lui p: $\Theta(m)$
- Notăm cu t_s valoarea numerică a subșirului $T[s+1 \ldots s+m]$, $0 \leq s \leq n-m$
- Folosim observația esențială că $p = t_s \Leftrightarrow P = T[s+1 \ldots s+m]$
- Tot prin schema lui Horner se poate calcula t_0 în timp $\Theta(m)$
  - t_0 este valoarea numerică a lui $T[1 \ldots m]$
- Pentru calculul lui t_{s+1} folosim informația de la pasul anterior:

$$t_{s+1} = 10(t_s - 10^{m-1} T[s+1]) + T[s+m+1] \qquad (1)$$

Algoritmul Rabin–Karp

- scăderea înlătură prima cifră din t_s, multiplicarea cu 10 deplasează la stanga cifrele rămase cu o poziție iar adunarea ultimului termen duce la concatenarea cifrei următoare
- Considerăm pentru moment că putem opera cu $p, t_0, t_1, \ldots$ în timp constant, *i.e.* nu sunt numere mai mari decât cele care încap într–un cuvânt de calculator
- Valoarea 10^{m-1} poate fi precalculată în timp $\Theta(\log_2 m)$ sau *brute-force* în $\Theta(m)$
- t_{s+1} poate fi calculat din t_s conform ecuației (1) în timp constant
- În ipoteza că operarea cu numerele p și t_s se face în timp constant, căutarea lui $P[1 \ldots m]$ în $T[1 \ldots n]$ presupune:
  - preprocesare de complexitate $\Theta(m)$: calculul lui p, t_0 și 10^{m-1}
  - căutare lui P în T prin compararea succesivă a lui p cu $t_0, \ldots t_{n-m}$, operație (pentru moment) în timp constant, deci $\Theta(n-m+1)$

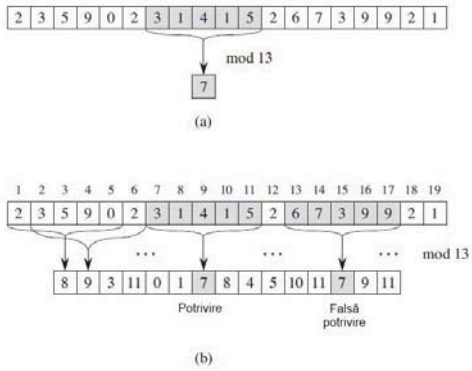
Algoritmul Rabin–Karp

- Presupunerea că numerele p și t_s pot fi manipulate în timp constant este nerezonabilă în practică; se lucrează cu numere cu m cifre, m posibil mare
- Vom calcula p și t_s modulo un număr q; valorile rezultate vor fi din $\{0, 1, \ldots q-1\}$, manipulabile în timp constant
- Comparațiile dintre șirurile de caractere se vor face prin intermediul reprezentării lor numerice modulo q
  - dacă $p \mod q \neq t_s \mod q$ atunci sigur $p \neq t_s$
  - pentru $p \mod q = t_s \mod q$ s–ar putea totuși ca $p \neq t_s$, *i.e.* să avem o falsă potrivire; verificarea trebuie făcută caracter cu caracter
  - dar: comparația modulo q este o euristică rapidă prin care se pot elimina multe subșiruri ale lui T care nu pot fi egale cu P
  - pentru q mare, sperăm ca în practică numărul de false potriviri să se reducă drastic
- Datorită folosirii operației $\mod q$, ecuația (1) devine:

$$t_{s+1} = (d(t_s - hT[s+1]) + T[s+m+1]) \mod q \quad (2)$$

unde $h = d^{m-1} \mod q$

Algoritmul Rabin–Karp - exemplu



Fiecare caracter e o cifră în baza 10, $q = 13$, $P = 31415$, $P \mod 13 = 7$. Pentru fiecare subsecvență de 5 cifre din T se calculează valoarea modulo 13 a numărului corespunzător, apoi se face compararea valorilor asociate subsecvențelor cu 7. Toate potrivirile modulo 13 trebuie verificate prin comparația caracter cu caracter între P și subsecvențele respective. Euristica duce la mult mai puține comparații între P și subșiruri ale lui T față de algoritmul naiv.

Algoritmul Rabin–Karp

```
Cautare-Rabin-Karp(T, P, d, q)
 1  n ← length[P]
 2  m ← length[T]
 3  h ← d^(m−1) mod q
 4  p ← t_0 ← 0
 5  for i = 1 to m //preprocesare
 6      p ← (dp + P[i]) mod q
 7      t_0 ← (dt_0 + T[i]) mod q
 8  for s = 0 to n − m //cautare
 9      if p = t_s
10          if P[1...m] = T[s + 1...s + m]
11              print "pattern-ul apare cu deplasamentul " s
12      if s < n − m
13          t_{s+1} ← (d(t_s − hT[s + 1]) + T[s + m + 1]) mod q
```

Algoritmul Rabin–Karp

- Complexitate: $\Theta(m)$ pentru preprocesare
- Cazul cel mai nefavorabil: avem echivalență modulo q la fiecare valoare a lui s, fiecare necesitând verificarea făcută în linia 10 a algoritmului $\Rightarrow \Theta((n-m+1)m)$
- Un astfel de caz nefavorabil: $T = a^n$, $P = a^m$
- Cazul cel mai defavorabil duce la același comportament ca algoritmul naiv
- Dar în medie ne putem aștepta ca doar mică parte din subșirurile de m elemente din T să fie echivalente modulo q cu p

Algoritmul Boyer–Moore

- Este un algoritm de căutare eficient pentru P lung și Σ cu multe simboluri
- În cazul unei nepotriviri de simboluri se efectuează salturi peste un număr (posibil) mare de caractere pentru care comparațiile sunt inutile, fără a rata eventualele apariții
- Elemente deosebite față de algoritmii precedenți:
  - compararea șirurilor se face dinspre dreapta spre stânga
  - euristica bazată pe caracterul slab
  - euristica sufixului bun

Algoritmul Boyer–Moore

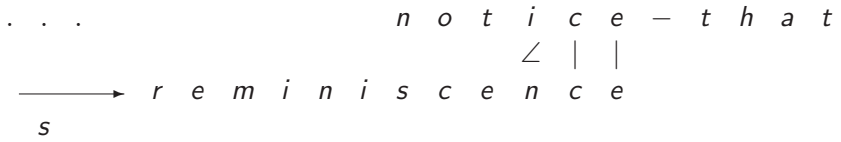
```
Cautare-Boyer-Moore(T, P, Σ)
 1  n ← length[P]
 2  m ← length[T]
 3  λ ← Calcul − ultima − aparitie(P, m, Σ)
 4  γ ← Calcul − sufix − bun(P, m)
 5  s ← 0
 6  while s ≤ n − m
 7      j ← m
 8      while j > 0 and P[j] = T[s + j]
 9            j ← j − 1
10      if j = 0
11            Tipareste "pattern-ul apare cu deplasamentul " s
12            s ← γ[0]
13      else
14            s ← s + max(γ[j], j − λ[T[s + j]])
```

Algoritmul Boyer–Moore

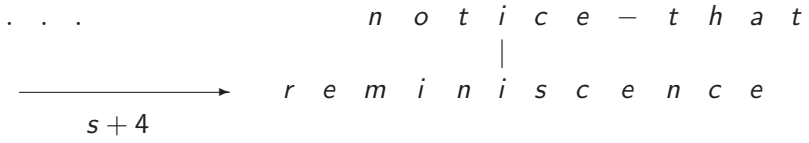
- Observație: dacă în liniile 12 și 14 din pseudocod atribuirile pentru s ar fi schimbate cu: $s \leftarrow s + 1$ atunci s–ar obține algoritmul naiv de căutare
- Variabilele λ și γ sunt tablouri, iar valorile lor sunt utile pentru a realiza saltul peste pozițiile care nu merită să fie comparate
- Când apare o nepotrivire între simboluri, fiecare din cele două euristici propune câte un salt în cadrul lui T, **fără a pierde însă deplasamente valide!**

Algoritmul Boyer–Moore: exemplificare

- Euristica bazată pe caracterul slab:

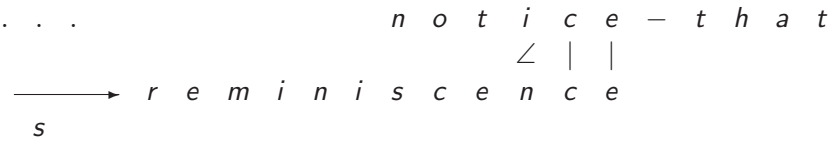


Se compară de la dreapta la stânga. Sufixul "ce" este valid iar caracterul "i" este invalid. Se propune deplasarea lui P spre dreapta cu 4 poziții astfel încât cel mai din dreapta "i" din P să se fie adus peste caracterul slab "i" din T.

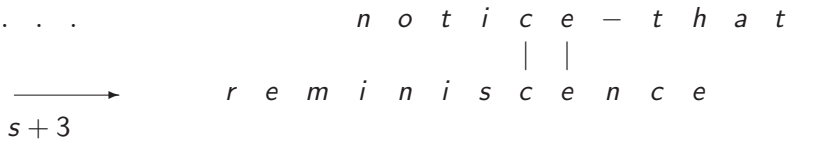


Algoritmul Boyer–Moore: exemplificare

- Euristica sufixului bun:



Euristica sufixului bun propune deplasarea pattern-ului spre dreapta până la prima apariție a lui "ce" în P:



Algoritmul Boyer-Moore alege maximul dintre cele două decalaje propuse, deci 4.

Algoritmul BM: euristica bazată pe caracterul slab

- Eficiența euristicii caracterului slab: pentru cazul cel mai favorabil, $j = m$ iar $T[s + m]$ nu apare în P
  - exemplu: dorim să căutăm $P = a^m$ în $T = b^n$
  - la fiecare nepotrivire, detectată chiar la prima comparație făcută, algoritmul "împinge" P la dreapta cu m poziții prin $s \leftarrow s + m$
  - scenariul arată eficiența comparației începute de la dreapta
- Mai general: presupunem că avem nepotrivire $P[j] \neq T[s + j]$ pentru un j, $1 \leq j \leq m$
- Fie k cel mai mare index din intervalul $1 \leq k \leq m$ astfel încât $T[s + j] = P[k]$
  - Dacă un astfel de k nu există, atunci setăm $k = 0$
- Pentru $k < j$ deplasarea lui P la dreapta este dictată de euristica bazată pe caracterul slab; pentru $k > j$ intervine euristica sufixului bun

Algoritmul BM: euristica bazată pe caracterul slab

Afirmație: este corect să se mărească s cu $j - k$

1. Cazul $k = 0$, *i.e.* caracterul $T[s + j]$ nu apare deloc în P: putem deplasa pattern–ul P la prima poziție de după caracterul "h", adică să mărim pe s cu $j = j - k$ fără ca prin asta să riscăm să pierdem o apariție a lui P în T.

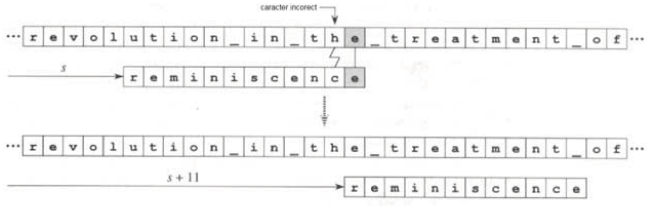


Fig: Cazul euristicii caracterului slab, situația $k = 0$. Caracterul slab "h" nu apare deloc în model și astfel modelul poate fi avansat cu $j = 11$ poziții până când trece peste caracterul slab.

Algoritmul BM: euristica bazată pe caracterul slab

Afirmație: este corect să se mărească s cu $j - k$

2. Cazul $k < j$, $k \neq 0$: cea mai din dreapta apariție a caracterului slab din T în cadrul lui P este la stânga poziției j, deci $j - k > 0$ și P poate fi deplasat cu $j - k$ poziții spre dreapta, fără a risca pierderea vreunei soluții

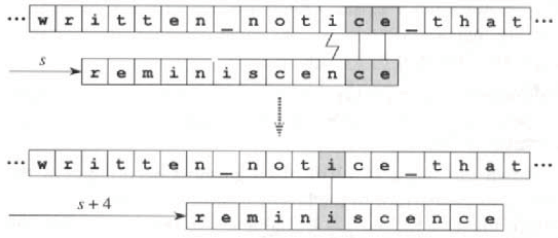


Fig: Cazul euristicii caracterului slab, situația $k < j$. $j = 10$, $k = 6$ pentru caracterul slab "i" și deci P poate fi avansat cu 4 poziții până când pozițiile lui "i" din T și P se suprapun.

Algoritmul BM: euristica bazată pe caracterul slab

Afirmație: este corect să se mărească s cu $j - k$

3. Cazul $k > j$:

- linia 14 din algoritmul Cautare-Boyer-Moore este:
  $s \leftarrow s + \max(\gamma[j], j - \underbrace{\lambda[T[s+j]]}_{k})$
- Euristica bazată pe caracterul slab ar propune decrementarea lui s. Recomandarea va fi ignorată de algoritm, deoarece intervine euristica sufixului bun care va propune totuși deplasare spre dreapta; $\gamma[j]$ e tot timpul un număr pozitiv.

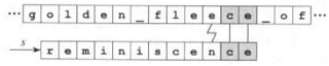


Fig: Cazul euristicii caracterului slab, situația $k > j$: $j = 10$, $k = 12$. Cea mai din dreapta apariție a caracterului slab "e" în cadrul lui P este după poziția curentă j. Dacă s–ar folosi doar euristica bazată pe caracterul slab s–ar ajunge la o deplasare a lui P în stânga. Aceasta este o situație în care e utilă euristica sufixului bun.

Algoritmul BM: euristica bazată pe caracterul slab

- Calculul lui λ, folosit în euristica bazată pe caracterul slab:

```
Calcul-ultima-aparitie(P, m, Σ)
1  for fiecare caracter a ∈ Σ
2      λ[a] ← 0
3  for j ← 1, m
4      λ[P[j]] ← j
5  return λ
```

- Complexitate: $O(|\Sigma| + m)$; este pas de preprocesare

Algoritmul BM: euristica sufixului bun

- Definim relația $Q \sim R$ ("Q este similar cu R") astfel:

$$Q \sim R \Longleftrightarrow Q \sqsupset R \text{ sau } R \sqsupset Q$$

- Relația "$\sim$" este simetrică
- Datorită lemei sufixelor suprapuse de la slide-ul 6 avem:

$$Q \sqsupset R \text{ si } S \sqsupset R \Rightarrow Q \sim S$$

- Două șiruri similare se pot alinia la dreapta, cu potrivire completă pe cel mai scurt din ele.
- Dacă $P[j] \neq T[s+j]$, $j < m$, atunci, euristica sufixului bun afirmă că putem incrementa pe s cu:

$$\gamma[j] = m - max\{k | 0 \leq k < m, P[j+1..m] \sim P_k\}$$

Algoritmul BM: euristica sufixului bun

- Definim funcția prefix care arată cum se potrivește pattern–ul cu decalări ale lui însuși
- $\Pi : \{1, 2, \ldots, m\} \rightarrow \{0, 1, \ldots, m-1\}$,

$$\Pi(q) = \max\{k | k < q \text{ şi } P_k \sqsupset P_q\} \quad (3)$$

- $\Pi(q)$ este lungimea celui mai mare prefix al lui P care este și sufix propriu al lui P_q
- Definim P' ca *inversul* lui P: $P'[i] = P[m-i+1]$ pentru $i = 1, 2, ..., m$.
- Fie Π' funcția prefix a lui P'. Se poate arăta că:

$$\gamma[j] = min(\{m - \Pi[m]\} \cup \{l - \Pi'[l] \mid 1 \leq l \leq m \text{ şi } j = m - \Pi'[l]\}).$$

Algoritmul BM: euristica sufixului bun

```
Functie-prefix(P)
 1  m ← length[P]
 2  π[1] ← 0
 3  k ← 0
 4  for q ← 2 to m
 5      while k > 0 and P[k + 1] ≠ P[q]
 6            k ← π[k]
 7      if P[k + 1] = P[q]
 8            k ← k + 1
 9      π[q] ← k
10  return π
```

```
Sufix-bun(P, m)
 1  π ←Functie-prefix(P)
 2  P' ←invers(P)
 3  π' ←Functie-prefix(P')
 4  for j ← 0 to m
 5      γ[j] ← m − π[m]
 6  for l ← 1 to m
 7      j ← m − π'[l]
 8      if γ[j] > l − π'[l]
 9            π[j] ← l − π'[l]
10  return γ
```

- Complexitatea lui `Sufix-bun`: $O(m)$; reprezintă pas de preprocesare
- Suport intuitiv pentru euristica sufixului bun: [3], secțiunea 2.2

Algoritmul Boyer-Moore: complexitate

- Preprocesarea are complexitatea $O(|\Sigma| + m) + O(m) = O(|\Sigma| + m)$
- Restul algoritmului are complexitatea $O(n)$: prin demonstrații complexe se arată că în cel mai defavorabil caz – indiferent dacă T conține sau nu pe P – este $3n$ ([3]).
- **În practică este deseori cel mai eficient algoritm de căutare**

Definiție, utilitate

Definiție (Subsecvență)

Dându–se o secvență $X = \langle x_1, x_2, \ldots, x_m \rangle$*, o secvență* $Z = \langle z_1, z_2, \ldots, z_k \rangle$ *se numește subsecvență a lui* X *dacă există o secvență strict crescătoare de indici de elemente din* X*,* $\langle i_1, i_2, \ldots, i_k \rangle$ *astfel încât* $\forall j = 1, 2, \ldots, k$*:* $z_j = x_{i_j}$*.*

- Exemplu: $X = \langle A, B, C, B, D, A, B \rangle$ are ca subsecvență pe $Z = \langle B, C, D, B \rangle$ pentru indicii $\langle 2, 3, 5, 7 \rangle$ din X
- Pentru două secvențe X, Y ne punem problema determinării celei mai lungi secvențe comune Z ce e subsecvență atât pentru X, cât și pentru Y; abreviat: CMLSC
- Exemplu: $X = \langle A, B, C, B, D, A, B \rangle$, $Y = \langle B, D, C, A, B, A \rangle$: $\langle B, C, A \rangle$ este o subsecvență comună, iar $\langle B, C, B, A \rangle$ este o CMLSC; nu e singura CMLSC
- Utilitate: dă o măsură a similarității dintre două șiruri; cu cât CMLSC a lui X și Y e mai lungă, cu atât X și Y sunt mai similare

O abordare

- Varianta *brute-force*: se enumeră toate subsecvențele lui $X = \langle x_1, x_2, \ldots, x_m \rangle$ și se determină care din ele este și subsecvență de lungime maximă a lui Y
- Avem $2^m - 1$ subsecvențe, *i.e.* toate submulțimile nevide ale setului de indici $\{1, 2, \ldots, m\}$; timpul rezultat este cel puțin exponențial – mai apare operația de verificare a faptului că o subsecvență a lui X e și subsecvență pentru Y
- Abordare mai eficientă bazată pe programare dinamică
- Notație: pentru $i = 0, 1, \ldots, m$, definim X_i ca fiind prefixul $\langle x_1, x_2, \ldots, x_i \rangle$
  - pentru $X = \langle A, B, C, B, D, A, B \rangle$, $X_4 = \langle A, B, C, B \rangle$
  - putem avea și secvență vidă

Suport teoretic pentru rezolvarea eficientă a problemei CMLSC

Teoremă (Substructura optimală pentru CMLSC)

Pentru $X = \langle x_1, x_2, \ldots x_m \rangle$ și $Y = \langle y_1, y_2, \ldots, y_n \rangle$ fie $Z = \langle z_1, z_2, \ldots, z_k \rangle$ o CMLSC a lor. Avem cazurile:

1. *Dacă $x_m = y_n$, atunci $z_k = x_m = y_n$ și Z_{k-1} este o CMLSC a lui X_{m-1} și Y_{n-1}*
2. *Dacă $x_m \neq y_n$, atunci din $z_k \neq x_m$ rezultă că Z este o CMLSC a lui X_{m-1} și Y_n*
3. *Dacă $x_m \neq y_n$, atunci din $z_k \neq y_n$ rezultă că Z este o CMLSC a lui X_m și Y_{n-1}*

Demonstrație: prin reducere la absurd; vezi și [1].

- O CMLSC a lui X și Y conține o CMLSC a unor prefixe ale lui X și Y. Se aplică deci principiul optimalității: orice subsoluție a unei soluții optimale trebuie să fie ea însăși o soluție optimală.

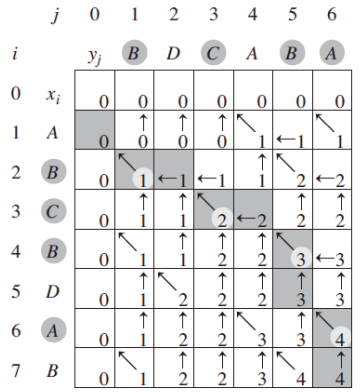
Rezolvarea problemei CMLSC – varianta brute-force

- Formularea problemei îndeamnă la o primă abordare recursivă
  - dacă $x_m = y_n$ atunci se găsește CMLSC Z pentru X_{m-1} și Y_{n-1} și soluția este Zx_m
  - dacă $x_m \neq y_n$ atunci se abordează două subprobleme: se determina CMLSC pentru X_m și Y_{n-1} respectiv pentru X_{m-1} și Y_n; cea mai lungă din acestea este CMLSC pentru X_m și Y_n
  - avem subprobleme comune: determinarea CMLSC pentru X_{m-1} și Y_{n-1} e subproblemă ce apare pentru X_{m-1} și Y_n, respectiv pentru X_m și Y_{n-1}
  - ecuația care dă lungimea CMLSC pentru X_i și Y_j, $0 \leq i \leq m$, $0 \leq j \leq n$:

$$c[i,j] = \begin{cases} 0, & \text{dacă } i = 0 \text{ sau } j = 0 \\ c[i-1,j-1]+1, & \text{dacă } i,j > 0 \text{ şi } x_i = y_j \\ \max(c[i,j-1], c[i-1,j]), & \text{dacă } i,j > 0 \text{ şi } x_i \neq y_j \end{cases} \quad (4)$$

  - lungimea CMLSC a lui $X = X_m$ și $Y = Y_n$ este $c[m,n]$
  - dacă se implementează recursiv, algoritmul este de complexitate exponențială (demonstrați)

Rezolvarea problemei CMLSC – exemplu



i		j	0	1	2	3	4	5	6
			y_j	B	D	C	A	B	A
0	x_i		0	0	0	0	0	0	0
1	A		0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B		0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C		0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B		0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D		0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A		0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B		0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Fig: Exemplu de calcul a lungimii CMLSC pentru $X = \langle A, B, C, B, D, A, B\rangle$ și $Y = \langle B, D, C, A, B, A\rangle$. CMLSC are lungimea $c[7, 6]$ și este $\langle B, C, B, A\rangle$. Săgeata dintr-o celulă arată pe baza cărei celule s–a obținut valoarea corespunzătoare.

Rezolvarea problemei CMLSC – programare dinamică

- Se calculează valorile $c[i, j]$, $0 \leq i \leq m$ și $0 \leq j \leq n$, linie cu linie
- Se menține o tabelă auxiliară b de $(m + 1) \times (n + 1)$ pentru reconstituirea CMLSC în final

```
Lungime-CMLSC(X, Y)
 1  m ← length[X]
 2  n ← length[Y]
 3  alocă spațiu pentru tablourile c[0 . . . m, 0 . . . n] și b[0 . . . m, 0 . . . n]
 4  for i ← 1 to m
 5      c[i, 0] ← 0
 6  for j ← 1 to n
 7      c[0, j] ← 0
 8  for i ← 1 to m
 9      for j ← 1 to n
10          if x_i = y_j
11              c[i, j] ← c[i − 1, j − 1] + 1
12              b[i, j] ← "stanga sus"
13  elseif c[i − 1, j] ≥ c[i, j − 1]
14              c[i, j] ← c[i, j − 1]
15              b[i, j] ← "sus"
16      else    c[i, j] ← c[i, j − 1]
17              b[i, j] ← "stanga"
```

Rezolvarea problemei CMLSC – programare dinamică

- Complexitate: $\Theta(mn)$
- $c[m, n]$ este lungimea CMLSC a lui X și Y
- Reconstituirea CMLSC se face folosind matricea b
  - se începe din colțul dreapta jos al matricei b, cu $i = m$ și $j = n$
  - dacă valoarea stocată în $b[i, j]$ este "stanga sus", atunci $x_i = y_j$ și acest caracter se adaugă într-o stivă
  - dacă $b[i, j]$ este "sus" (respectiv "stanga") atunci $i \leftarrow i - 1$ (respectiv $j \leftarrow j - 1$)
  - procedeul se continuă până când i sau j devin 0
  - scoțând din stivă elementele și concatenându-le se obține CMLSC

Reconstituirea CMLSC

```
Tipareste-CMLSC(b, X, i, j)
1  if i = 0 or j = 0
2     return
3  if b[i, j] = "stanga sus"
4     Tipareste-CMLSC(b, X, i − 1, j − 1)
5     print x_i
6  elseif b[i, j] = "sus"
7     Tipareste-CMLSC(b, X, i − 1, j)
8     else  Tipareste-CMLSC(b, X, i, j − 1)
```

- Complexitate: $O(m + n) = O(\max\{m, n\})$, deoarece la fiecare pas cel puțin una din valorile i și j scade cu o unitate

Îmbunătățiri ale codului

- Se poate renunța la matricea b; se poate determina pentru fiecare $i, 1 \leq i \leq m$ și j, $1 \leq j \leq n$ cum s–a calculat $c[i,j]$: sunt de urmărit doar variantele:
  $c[i,j] \in \{c[i-1,j-1]+1, c[i-1,j], c[i,j-1]\}$
- Dacă se dorește doar lungimea CMLSC, atunci în funcția Lungime-CMLSC(X, Y) se poate menține la liniile 8-17 doar linia $i-1$ și linia curentă
  - reconstruirea CMLSC folosind spațiu de memorie liniar este făcută în algoritmul lui Hirschberg [2]
- Există posibilitate de paralelizare a algoritmului, dar sunt dependențe mari între date; pentru o propunere de paralelizare folosind CUDA, a se vedea prezentarea de aici.

Distanțe de editare

- Ne interesează numărul de operații de editare care trebuie aplicate pe un șir de caractere pentru a fi transformat într–un alt șir
- În funcție de tipul de operații de editare permise — înlocuire, ștergere, inserare — avem distanțele:
  - Hamming
  - Levenshtein
- Distanțele de editare sunt folosite în teoria informației și bioinformatică

Distanțe de editare: distanța Hamming

Definiție (Distanța Hamming)

Pentru două șiruri de lungimi egale, distanța Hamming este numărul de poziții pe care caracterele corespunzătoare diferă.

- Exemplu: $d_H(0100101, 0010001) = 3$
- Putem folosi simbolul lui Kronecker pentru scrierea formulei distanței Hamming:

$$\delta_{ab} = \begin{cases} 1, & \text{dacă } a = b \\ 0, & \text{altfel} \end{cases}$$

- Pentru $X = x_1 x_2 \dots x_n$, $Y = y_1 y_2 \dots y_n$

$$d_H(X, Y) = \sum_{i=1}^{n} (1 - \delta_{x_i, y_i}) = n - \sum_{i=1}^{n} \delta_{x_i, y_i}$$

- Complexitatea calculului lui $d_H(X, Y)$: $\Theta(n)$

Distanțe de editare: distanța Levenshtein

Definiție (Distanța Levenshtein)

Distanța Levenshtein dintre două șiruri X și Y este numărul minim de operații de editare prin care X poate fi transformat în Y. Operațiile de editare sunt: inserare, ștergere și substituire de caracter.

- Exemplu: pentru transformarea din "kitten" în "sitting" e nevoie de 3 transformări
  - **k**itten $\rightarrow$ **s**itten: substituire
  - sitt**e**n $\rightarrow$ sitt**i**n: substituire
  - sittin $\rightarrow$ sittin**g**: inserarea lui 'g' la final
- Legătura cu distanța Hamming: la aceasta din urmă se permit doar substituiri – de aceea cele două șiruri măsurate prin distanța Hamming trebuie să fie de lungime egală
- Legătura cu problema celei mai lungi subsecvențe comune: la CMLSC operațiile permise erau doar inserarea și ștergerea

Distanțe de editare: distanța Levenshtein

- Aplicații:
  - spell checking în editarea de text
  - corectarea cuvintelor după un proces OCR
  - asistare în traducere automată
- Considerăm $X = x_1x_2 \dots x_m$, $Y = y_1y_2 \dots y_n$
- Analog cu problema CMLSC, distanța Levenshtein se poate calcula prin programare dinamică
- Pentru subșirurile $X_i = x_1x_2 \dots x_i$ și $Y_j = y_1y_2 \dots y_j$, notăm cu $D[i, j]$ numărul minim de operații prin care se transformă X_i în Y_j
- $D[m, n]$ este distanța de editare Levenshtein căutată

Distanțe de editare: distanța Levenshtein

- Pentru calculul lui $D[i, j]$ cu $i, j > 0$ putem avea posibilitățile:
  - $x_i = y_j$ și atunci din X_i se obține Y_j prin tot atâtea operații câte sunt necesare pentru a transforma pe X_{i-1} în Y_{j-1} cu număr minim de operații după care se adaugă caracterul comun $x_i = y_j$ – operație ce nu costă nimic
  - $x_i \neq y_j$ și atunci avem variantele:
    1. transformăm cu număr minim de operații pe X_{i-1} în Y_j și **ștergem** x_i
    2. transformăm cu număr minim de operații pe X_i în Y_{j-1} și apoi **inserăm** caracterul y_j
    3. transformăm cu număr minim de operații pe X_{i-1} în Y_{j-1} și apoi **modificăm** pe x_i în y_j
- Evident, pentru transformarea dintr-un șir cu k caractere într-un șir vid avem nevoie de exact k ștergeri deci $D[k, 0] = k$ pentru $0 \leq k \leq i$; analog $D[0, k] = k$ inserări pentru $0 \leq k \leq j$

Distanțe de editare: distanța Levenshtein

Teoremă (Calculul distanței Levenshtein)

Transformarea unui șir $X = x_1x_2 \ldots x_m$ într–un șir $Y = y_1y_2 \ldots y_n$ se face cu $D[m, n]$ operații, unde $D[i, 0] = i$, $D[0, j] = j$ pentru $0 \leq i \leq m$, $0 \leq j \leq n$ iar pentru $1 \leq i \leq m$, $1 \leq j \leq n$ $D[i, j]$ se definește ca:

$$D[i,j] = \begin{cases} D[i-1, j-1], \ dacă\ x_i = y_j \\ \min\{D[i-1,j], D[i,j-1], D[i-1,j-1]\} + 1, \\ \ dacă\ x_i \neq y_j \end{cases} \tag{5}$$

Demonstrație: Se face prin reducere la absurd; pentru o demonstrație completă, a se vedea [3]. □

- Relația (5) se mai poate scrie și ca:

$$D[i,j] = \min\{D[i-1,j], D[i,j-1], D[i-1,j-1] - \delta_{x_i,y_j}\} + 1$$

Distanțe de editare: distanța Levenshtein

- Complexitate în timp și spațiu de memorie: $\Theta(mn)$
- Pentru determinarea unei liste de operații care fac transformarea lui X în Y se poate folosi ca la problema CMLSC un tablou auxiliar care să mențină operațiile efectuate pentru determinarea lui $D[i, j]$, sau se poate folosi direct tabloul D
- Exemplu: distanța Levenshtein între cuvintele "kitten" și "sitting" este 3

		k	i	t	t	e	n
	0	1	2	3	4	5	6
s	1	1	2	3	4	5	6
i	2	2	1	2	3	4	5
t	3	3	2	1	2	3	4
t	4	4	3	2	1	2	3
i	5	5	4	3	2	2	3
n	6	6	5	4	3	3	2
g	7	7	6	5	4	4	3

Tabel: Matricea D pentru cuvintele "kitten" și "sitting". Care este lista de transformări?

Distanțe de editare: distanța Levenshtein

```
Distanţa-editare-Levenshtein(X, Y)
 1  m ← length[X]
 2  n ← length[Y]
 3  alocă spațiu pentru tabloul D[0 . . . m, 0 . . . n]
 4  for i ← 1 to m
 5      D[i, 0] ← i
 6  for j ← 1 to n
 7      D[0, j] ← j
 8  for i ← 1 to m
 9      for j ← 1 to n
10              if x_i = y_j
11                  D[i, j] ← D[i − 1, j − 1]
12      else
13              D[i, j] ← min {D[i − 1, j], D[i, j − 1], D[i − 1, j − 1]} + 1
```

Exemplificare

- Pornim de la cuvintele: "occurrence" și "ocurrance"
- Folosind operațiile:
  - inserare de spațiu, *i.e.* deplasarea următoarelor caractere cu o poziție la dreapta
  - modificarea unui caracter în altul

  care sunt modalitățile de a ajunge de la primul cuvânt la al doilea?
- Dacă secvențele se transformă până ajung identice, atunci spunem că ele sunt aliniate

Exemplificare

- Varianta 1: o aliniere aproape perfectă; "-" înseamnă inserare de spațiu; mai avem o diferență de caractere pentru perechea (a – e)

```
o - c u r r a n c e
o c c u r r e n c e
```

- Varianta 2: o aliniere perfectă cu trei inserări de spații

```
o - c u r r - a n c e
o c c u r r e - n c e
```

- Care secvență de operații este mai bună? o inserare de spațiu și o nepotrivire sau trei inserări de spațiu?
- Mai important: care e secvența de operații de cost minim care aliniază două secvențe?
- Utilitate: determinarea similarității dintre două secvențe de proteine sau nucleotide
- Este cronologic prima aplicare a programării dinamice în bioinformatică

Notații, definiții

- Se dau secvențele $X = x_1 \dots x_m$, $Y = y_1 \dots y_n$
- Considerăm mulțimile de indici pentru fiecare secvență: $I_x = \{1, \dots, m\}$ respectiv $I_y = \{1, \dots, n\}$
- Un "matching" M peste mulțimile I_x și I_y este o mulțime de perechi (i, j), $i \in I_x$, $j \in I_y$ și pentru care fiecare element din I_x și I_y apare cel mult o dată
  - exemplu: $I_x = \{1, 2, 3\}$, $I_y = \{1, 2, 3, 4\}$; $M = \{(1, 1), (3, 2), (2, 4)\}$ este un matching, dar nu și $\{(1, 1), (3, 2), (3, 4)\}$ și nici $\{(1, 1), (2, 1), (3, 3)\}$

Notații, definiții

- **Aliniere:** matching M pentru care dacă $(i,j),(i',j') \in M$ și $i < i'$ atunci $j < j'$
- Intuitiv: o aliniere (în sensul de mai sus) corespunde unei modalități de aliniere de secvențe
- Exemplu:

1	2	3	4	
s	t	o	p	-
-	t	o	p	s
	1	2	3	4

corespunde alinierii $\{(2,1),(3,2),(4,3)\}$

Notații, definiții

- Pentru o aliniere M a două secvențe:
  - se consideră o penalizare de inserare de spațiu, de cost δ[1]; pentru fiecare poziție care nu are corespondent în M avem o penalizare δ
  - pentru fiecare pereche p, q din alfabetul de simboluri acem un cost α_{pq} de modificare a lui p în q; putem presupune că $\alpha_{pp} = 0$ pentru fiecare literă p
  - costul alinierii date de M este suma costurilor δ și α_{pq}
- Pentru situațiile:

o - c u r r a n c e
o c c u r r e n c e

și

o - c u r r - a n c e
o c c u r r e - n c e

costurile sunt $\delta + \alpha_{ae}$ respectiv 3δ. Alegerea soluției optime depinde de relația dintre α_{ae} și 2δ.

[1] A nu se confunda cu simbolul δ al lui Kronecker.

Aliniere optimă

- Similaritatea între două secvențe e dată de costul minim al alinierii lor
- Se cere determinarea costului minim și a secvenței de operații care produce costul minim
- Evident, dacă M este o aliniere de cost minim (optimă), atunci fie $(m, n) \in M$, fie $(m, n) \notin M$

Teoremă
Dacă M este o aliniere oarecare si $(m, n) \notin M$, atunci m sau n nu au corespondent în M.

Demonstrație: Să presupunem prin absurd că $(m, n) \notin M$ și atât m cât și n au corespondent în M: $\exists i, j$: $i < m$, $j < n$ cu $(m, j) \in M$ și $(i, n) \in M$. Asta însă contrazice proprietatea de aliniere a lui M: dacă $i < m$ atunci ar trebui și ca $n < j$, contradicție cu presupunerea asupra lui j. □

Aliniere optimă

- Combinând rezultatele anterioare, obținem:

Teoremă
Într–o aliniere optimală M a lui X, Y cu $|X| = m$, $|Y| = n$, cel puțin una din următoarele este adevărată:
1. $(m, n) \in M$
2. poz. m a lui X nu apare în vreo pereche din M ca prim element
3. poziția n a lui Y nu apare în vreo pereche din M ca al 2-lea element

- Notăm $OPT(i, j)$ costul minim al alinierii între $x_1 \dots x_i$ și $y_1 \dots y_j$. Dacă considerăm prima variantă din cele 3 de mai sus, atunci plătim penalizarea $\alpha_{x_m y_n}$ și aliniem secvențele $x_1 \dots x_{m-1}$ și $y_1 \dots y_{n-1}$; obținem costul $OPT(m, n) = \alpha_{x_m y_n} + OPT(m-1, n-1)$. Pentru al doilea caz plătim costul unei inserări de spațiu deoarece ultimul caracter din X nu are potrivire și aliniem secvențele $x_1 \dots x_{m-1}$ și $y_1 \dots y_n$; $OPT(m, n) = OPT(m-1, n) + \delta$. Al treilea caz e analog cu al doilea.

Aliniere optimă

- Costul minim pentru alinierea de secvențe satisface:

$$OPT(i,j) = \begin{cases} i\delta \text{ pt. } j = 0 \\ j\delta \text{ pt. } i = 0 \\ \min\{\alpha_{x_i y_j} + OPT(i-1, j-1), \\ \delta + OPT(i, j-1), \delta + OPT(i-1, j)\} \\ \text{ pt. } i, j \geq 1 \end{cases} \tag{6}$$

Algoritmul pentru calculul costului de aliniere optimă

```
Cost-minim-aliniere(X, Y)
1  Alocă spațiu pentru tabloul OPT[0..m, 0..n]
2  for i ← 0 to m
3      OPT[i, 0] ← iδ
4  for j ← 0 to n
5      OPT[0, j] ← jδ
6  for i ← 1 to m
7      for j ← 1 to n
8          folosește ultima ramură din ec. (6) pentru calcul OPT[i, j]
```

- Complexitate: $\Theta(mn)$
- Determinarea suitei de operații pentru alinierea optimă: se folosește o matrice auxiliară sau se reface drumul plecând de la valoarea din colțul de indici (m, n) cu complexitate $O(m + n)$.

Bibliografie

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*. Addison Wesley, Massachusetts, 3rd Edition, MIT Press 2009
- D. S. Hirschberg, *A linear space algorithm for computing maximal common subsequences*. Communications of the ACM, Volume 18 Issue 6, June 1975
- Dan Gusfield, *Algoritms on String, Trees, and Sequences*. Cambridge University Press, 1997
- Jon Kleinberg, Éva Tardos, *Algorithm Design*, Pearson Education, 2006

Cursul nr. 8

PATTERN-URI STRUCTURALE

Cuprins

Pattern matching, Arbori și grafuri

Pattern matching în arbori

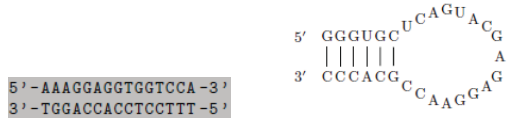
Pattern matching în grafuri

Definiții: Pattern matching

- *Pattern* – secvență nenulă și finită de simboluri; model
- *Pattern matching* – identificarea unui pattern într-o secvență mai complexă, de exemplu găsirea unui cuvânt într-un text; în unele aplicații se caută și apariții aproximative ale pattern-ului
- *Pattern matching în arbori* – atât pattern-ul căutat cât și secvența în care se face căutarea sunt arbori
- *Pattern matching în grafuri* – atât pattern-ul căutat cât și secvența în care se face căutarea sunt grafuri
- În general, problema de matching este NP-completă
- Exemplu:
  - Basic Local Alignment Search Tool (BLAST) este o aplicație care găsește similarități între secvențe biologice. Se compară secvențe de proteine sau nucleotide cu secvențe dintr-o bază de date și se calculează gradul de potrivire dintre acestea.

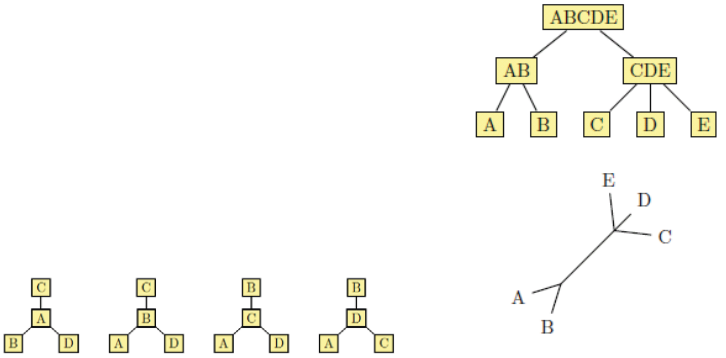
Definiții: Arbori și grafuri

- *Secvența* – listă ordonată de elemente
- *Graf neorientat* – o mulțime de noduri între care există relații binare descrise sub forma muchiilor
- *Graf orientat* – este un graf în care muchiile sunt orientate de la un nod sursă către un nod destinație
- *Arbore* – graf neorientat, conex și fără cicluri; în unele aplicații nodurile și muchiile sunt etichetate
- Aplicații: reprezentarea macromoleculelor (ADN, ARN, proteine) în biologia computațională folosind un alfabet specific pentru etichetarea nodurilor, reprezentarea formulelor chimice, reprezentarea circuitelor electrice



Figură: Reprezentări folosite în biologia computațională.

Arbori

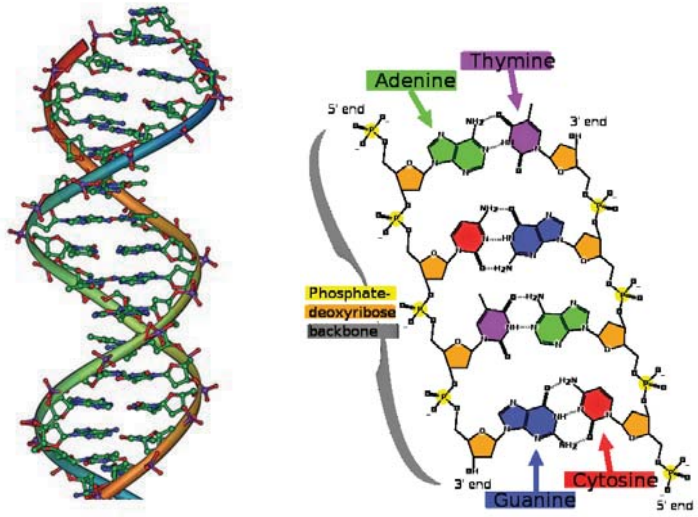


Figură: a) Cei patru arbori sunt identici ca arbori neetichetați, dar sunt diferiți dacă se consideră și etichetele; b) Reprezentare simplificată a unui arbore.

Arbori filogenetici (1)

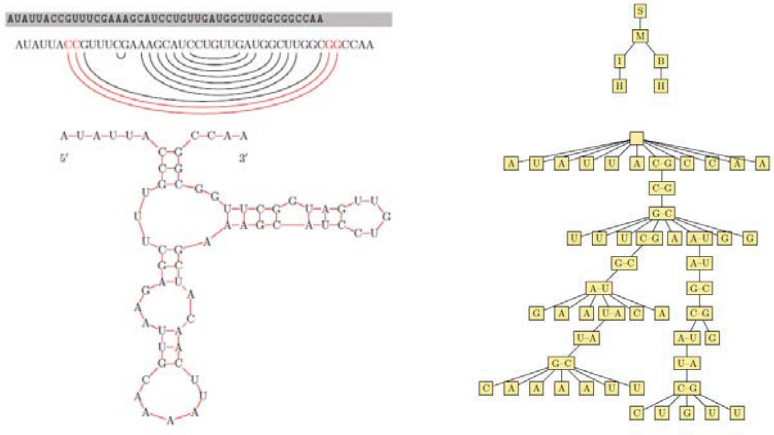
- În biologia computațională sunt folosiți în mod frecvent *arborii filogenetici*
- Arborii filogenetici sunt arbori cu sau fără rădăcină ai căror noduri terminale sunt etichetate cu denumiri ale unor entități din biologie
- Exemplu: reprezentarea secvențelor ADN și ARN
  - *Genomul* descrie toată informația ereditară conținută de cromozomi. Genomul uman conține 46 de cromozomi
  - *Cromozomul* reprezintă forma condensată a unei moecule de ADN
  - *ADN* (acid dezoxiribonucleic) și *ARN* (acid ribonucleic) sunt formate din molecule organice. ADN-ul se găsește în fiecare celulă a ființelor vii și este esențial pentru identitatea oricărui organism.
  - O moleculă ADN conține zone numite *gene* și are o structură dublu elicoidală între care se găsesc perechi de molecule organice de tip adenină (A), citozină (C), guanină (G) și timină (T). În ARN timina este înlocuită cu uracil (U).

Arbori filogenetici (2)



Figură: Structură ADN.

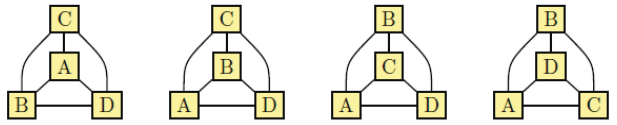
Arbori filogenetici (3)



Figură: Modalități de reprezentare a unei secvențe ARN. În reprezentarea arborescentă sunt folosite etichete care simbolizează elementele structurale de bază: H (hairpin loop), B (bulge), I (internal loop), M (multiple bifurcation loop), S (single-stranded region).

Grafuri

- Un graf neorientat este format dintr-o mulțime de noduri și muchii
- Unele aplicații folosesc grafuri etichetate în care nodurile și muchiile au atribute suplimentare



Figură: Cele patru grafuri sunt identice dacă sunt neetichetate, dar sunt diferite ca și grafuri etichetate.

Rețele filogenetice

- *Rețelele filogenetice* sunt folosite în biologia computațională și reprezintă grafuri aciclice cu sau fără rădăcină, direcționate, ale căror noduri terminale sunt etichetate
- Nodurile interne sunt fie de tip arbore dacă au un singur părinte, fie hibride dacă au doi sau mai mulți părinți
- O rețea filogenetică se numește *fully resolved* dacă fiecare nod intern de tip arbore are doi copii și fiecare nod hibrid are doi părinți și un copil

Rețele filogenetice - exemplu (1)

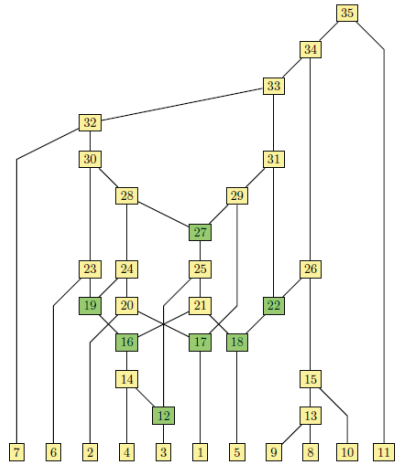
- Pentru o populație în care se regăsesc 11 specii ale unei gene care conțin nucleotide mutante, se poate elabora o rețea filogenetică pentru a descrie segregările și recombinările
- Cea mai frecventă nucleotidă este considerată de bază, iar cele mai puțin frecvente sunt mutante
- Nucleotidele de bază sunt notate cu 0, iar cele mutante cu 1

```
1  CCGCAATAATGGCGCTACTCTCACAATAACCCACTAGACAGCCT
2  CCCCAATATGGGCGCTACTTTCACAATAACCCACTAGACAGCCT
3  CCGCAATATGGGCGCTACCCCCCGGAATCTCCACTAAACAGTCA
4  CCGCAATATGGGCGCTGTCCCCCGGAATCTCCACTAAACTACCT
5  CCGAGATAAGTCCGAGGTCCCCCGGAATCTCCACTAGCCAGCCT
6  CCCCAATATGGGCGCGACCCCCCGGAATCTCTATTCACCAGCTT
7  CCCCAATATGGGCGCGACCCCCCGGAATCTGTCTCCGCCAGCCT
8  TGCAGATAAGTCGGCGACCCCCCGGAATCTGTCTCCGCGAGCCT
9  TGCAGATAAGTCGGCGACCCCCCGGAATCTGTCTCCGCGAGCCT
10 TGCAGATAAGTCGGCGACCCCCCGGAATCTGTCTCCGCGAGCCT
11 TGCAGGGGAGGGCTCGACCCCACGGGATCTGTCTCCGCCAGCCT
```

```
1  00100000010000010010101110111010101010000000
2  00000000100000010011101110111010101010000000
3  00100000100000010000000000000001010111000101
4  00100000100000011100000000000001010111011000
5  00111000001100101100000000000001010100000000
6  00000000100000000000000000000000000010000010
7  00000000100000000000000000000001010100000000
8  11011000001110000000000000000001010100010000
9  11011000001110000000000000000001010100010000
10 11011000001110000000000000000001010100010000
11 11011111000001000000010001000010101000000000
```

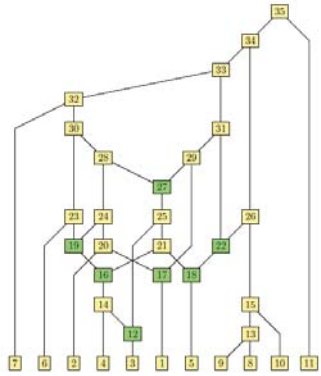
Rețele filogenetice - exemplu (2)

- Nodurile terminale - etichetate cu numerele asociate genelor
- Muchiile sunt direcționate de sus în jos și reprezintă mutații
- Muchia care pornește de la nodul 23 către frunza 6 reprezintă o mutație a lui C în T pe poziția 43



Rețele filogenetice - exemplu (3)

- Muchiile care pornesc din nodurile 21 (corespunzător secvenței 00100000100000011100000000000001010100000000) și 22 (secvența 00111000001100000000000000000001010100000000) către nodul 18 reprezintă o recombinare prin încrucișare a prefixului celei de-a doua (primele 16 poziții) cu sufixul ultimei (ultimele 28 poz.) formând secv. 00111000001100001100000000000001010100000000



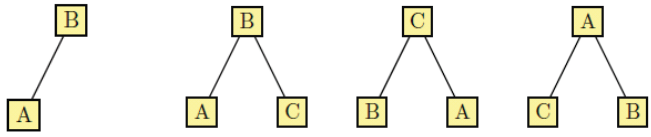
Pattern matching în arbori

Probleme de pattern matching în arbori:

- Găsirea drumurilor dintre două noduri ale unui arbore – utilă pentru calcularea distanțelor într-un arbore sau pentru calcularea distanțelor dintre doi arbori
- *Combinatorial pattern matching* – căutarea unei apariții exacte sau aproximative a unui pattern:
- (i) căutarea unui arbore filogenetic în alt arbore filogenetic poate fi un element util în determinarea similarităților sau a diferențelor
- (ii) scanarea unei structuri ARN pentru determinarea prezenței unui pattern cunoscut

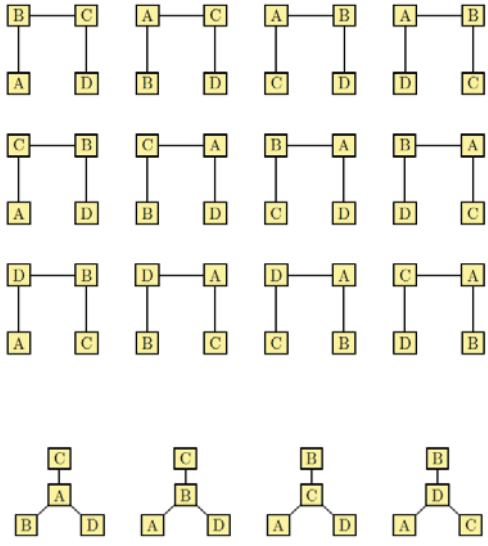
Numărarea arborilor (1)

- Numărarea arborilor - determinarea numărului de arbori cu o anumită proprietate, de exemplu toți arborii etichetați
- $n \geq 2$ noduri etichetate pot fi conectate în n^{n-2} moduri pentru a forma un arbore



Figură: a) Vârfurile A și B se pot conecta în $2^{2-2} = 1$ moduri; b) Vârfurile A, B și C se pot conecta în $3^{3-2} = 3$ moduri.

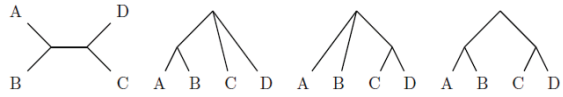
Numărarea arborilor (2)



Figură: Vârfurile A, B, C și D se pot conecta în $4^{4-2} = 16$ moduri.

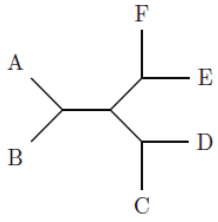
Distanțe în arbori fără rădăcină

- Distanța dintre oricare două noduri teminale într-un arbore filogenetic fără rădăcină
- *Distanța de partiționare* – este o măsură care reflectă diferența dintre doi subarbori obținuți în urma divizării unui arbore filogenetic fără rădăcină
- *Nodal distance* – distanța dintre doi arbori filogenetici fără rădăcină
- Într-un arbore fără rădăcină, oricare două noduri sunt legate prin exact un singur drum întrucât nu există niciun arc parcurs de mai multe ori
- Un arbore filogenetic fără rădăcină poate fi transformat într-unul cu rădăcină prin plasarea rădăcinii într-unul dintre noduri sau prin introducerea unui nod nou care va fi rădăcina



Distanța dintre două noduri teminale într-un arbore filogenetic fără rădăcină

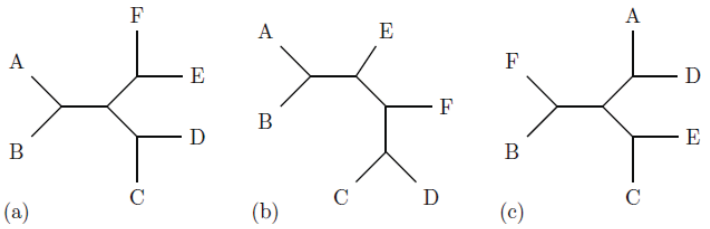
- Distanța dintre două noduri terminale dintr-un arbore filogenetic fără rădăcină este egală cu lungimea drumului dintre cele două noduri
- În arborii în care arcele au lungimi, distanța se calculează ca sumă a lungimilor arcelor



	A	B	C	D	E	F
A	0	2	4	4	4	4
B	2	0	4	4	4	4
C	4	4	0	2	4	4
D	4	4	2	0	4	4
E	4	4	4	4	0	2
F	4	4	4	4	2	0

Distanța de partiționare între arbori fără rădăcină (1)

- *Distanța de partiționare* – este o măsură care reflectă diferența dintre doi subarbori obținuți în urma divizării unui arbore filogenetic fără rădăcină
- Se bazează pe partiția nodurilor terminale determinată de fiecare arc intern din arborii care se compară
- Tăieturile arcelor interioare determină partiționarea în doi subarbori, în timp ce tăieturile arcelor exterioare determină separarea unui singur nod



Distanța de partiționare între arbori fără rădăcină (2)

- Partițiile celor trei arbori sunt:
- (A, B) și (C, D, E, F); (A, B, C, D) și (E, F); (A, B, E, F) și (C, D)
- (A, B) și (C, D, E, F); (A, B, E) și (C, D, F); (A, B, E, F) și (C, D)
- (A, B, D, F) și (C, E); (A, C, D, E) și (B, F); (A, D) și (B, C, E, F)
- *Distanța de partiționare* = nr. partițiilor diferite dintre arbori
- Arborii (a) și (b) au două partiții identice și două partiții diferite, deci distanța de partiționare dintre ei este 2; sunt cei mai asemănători
- Arborii (a) și (c) nu au nicio partiție comună, deci distanța de partiționare dintre ei este 6 (idem pt. arborii (b) și (c))
- Algoritmul se bazează pe funcția care construiește partițiile unui arbore T generate de fiecare muchie internă (v, w)
- O partiție generată de o astfel de muchie constă din etichetele tuturor nodurilor terminale care sunt descendente ale lui w și etichetele tuturor celorlalte noduri terminale

Distanța de partiționare între arbori fără rădăcină (3)

```
function partition(T)
1  P ← ∅
2  for each internal node v of T do
3          A ← taxa of all descendants of v in T
4          B ← taxa of all other leaves of T
5          P ← P ∪ {(A, B)}
6  return P

function partition distance(T1, T2)
1  P1 ← partition(T1)
2  P2 ← partition(T2)
3  d ← 0
4  for (A, B) ∈ P1 do
5          if (A, B) ∉ P2
6                  d ← d + 1
7  for (A, B) ∈ P2 do
8          if (A, B) ∉ P1
9                  d ← d + 1
10 return d
```

Distanța nodală între arbori fără rădăcină (1)

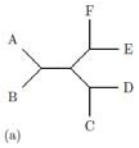
- *Distanța nodală* – se bazează pe distanța dintre perechi de noduri terminale din cei doi arbori care sunt comparați
- Fie $D(T)$ vectorul de lungime $n(n-1)/2$ al distanțelor nodale dintre toate perechile de noduri terminale ale unui arbore filogenetic T:

  $$D(T) = (d_T(1,2), d_T(1,3)...d_T(1,n), d_T(2,3)...d_T(n-1,n)),$$

  unde cele n noduri terminale ale lui T sunt numerotate de la $1...n$
- Distanța nodală $d_N(T_1, T_2)$ dintre arborii T_1 și T_2 este:

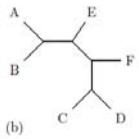
$$d_N(T_1, T_2) = \sum_{1 \le i < n, 1 < j \le n} |d_{T_1}(i,j) - d_{T_2}(i,j)|$$

Distanța nodală între arbori fără rădăcină (2)



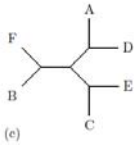
(a)

	A	B	C	D	E	F
A	0	2	4	4	4	4
B	2	0	4	4	4	4
C	4	4	0	2	4	4
D	4	4	2	0	4	4
E	4	4	4	4	0	2
F	4	4	4	4	2	0



(b)

	A	B	C	D	E	F
A	0	2	5	5	3	4
B	2	0	5	5	3	4
C	5	5	0	2	4	3
D	5	5	2	0	4	3
E	3	3	4	4	0	3
F	4	4	3	3	3	0



(c)

	A	B	C	D	E	F
A	0	4	4	2	4	4
B	4	0	4	4	4	2
C	4	4	0	4	2	4
D	2	4	4	0	4	4
E	4	4	2	4	0	4
F	4	2	4	4	4	0

	(a)	(b)	(c)	$\lvert(a)-(b)\rvert$	$\lvert(a)-(c)\rvert$	$\lvert(b)-(c)\rvert$
AB	2	2	4	0	2	2
AC	4	5	4	1	0	1
AD	4	5	2	1	2	3
AE	4	3	4	1	0	1
AF	4	4	4	0	0	0
BC	4	5	4	1	0	1
BD	4	5	4	1	0	1
BE	4	3	4	1	0	1
BF	4	4	2	0	2	2
CD	2	2	4	0	2	2
CE	4	4	2	0	2	2
CF	4	3	4	1	0	1
DE	4	4	4	0	0	0
DF	4	3	4	1	0	1
EF	2	3	4	1	2	1
				9	12	19

Arborii filogenetici (a) și (b) sunt cei mai asemănători pentru că distanța nodală dintre ei este 9, în timp ce distanțele dintre arborii (a) și (c), respectiv (b) și (c) sunt 12 și 19.

Distanța nodală între arbori fără rădăcină (3)

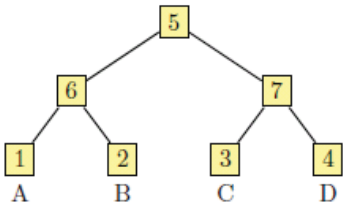
- Distanța nodală se obține calculând distanțele dintre fiecare pereche de noduri terminale în fiecare arbore , după care se obține diferența absolută dintre cei doi vectori ai distanțelor nodale

function nodal distance(T_1, T_2)

```
1  L ← terminal node labels in T1 and T2
2  n ← length(L)
3  d ← 0
4  for i = 1, ··· , n − 1 do
5          i1 ← terminal node of T1 labeled L[i]
6          i2 ← terminal node of T2 labeled L[i]
7          for j = i + 1, ··· , n do
8                  j1 ← terminal node of T1 labeled L[j]
9                  j2 ← terminal node of T2 labeled L[j]
10                 d1 ← distance(T1, i1, j1)
11                 d2 ← distance(T2, i2, j2)
12                 d ← d + |d1 − d2|
13 return d
```

Distanțe în arbori cu rădăcină (1)

Oricare două noduri sunt conectate prin exact un drum într-un arbore cu rădăcină atâta timp cât niciun arc nu este traversat de mai multe ori de un drum dintre cele două noduri, iar acest unic drum traversează cel mai apropiat parinte comun al celor două noduri



	A	B	C	D
A	1	6	5	5
B	6	2	5	5
C	5	5	3	7
D	5	5	7	4

Distanțe în arbori cu rădăcină (2)

- Distanța dintre două noduri teminale într-un arbore cu rădăcină – este suma lungimilor drumurilor dintre două noduri și cel mai apropiat părinte comun
- *Distanța de partiționare* – se calculează și pentru arborii cu rădăcină obținând mai întâi partițiile generate de arcele interne după care se numără partițiile care diferă. Partițiile induse de un arc intern (v,w) sunt toate nodurile terminale care sunt descendente nodului w, respectiv celelalte noduri terminale
- *Distanța nodală* – distanța dintre doi arbori filogenetici cu rădăcină se obține calculând distanța dintre fiecare pereche de noduri terminale din fiecare arbore, după care se calculează diferența absolută dintre cei doi vectori de distanțe nodale

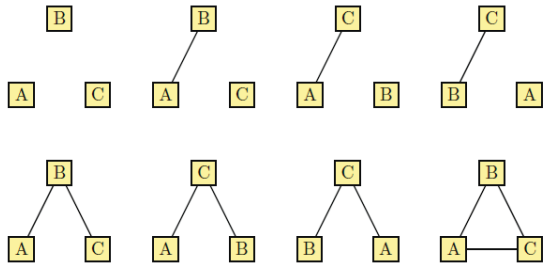
Pattern matching în grafuri

Probleme de pattern matching în grafuri:

- Găsirea unor pattern-uri mai simple în grafuri, cum ar fi drumurile sau arborii
  - Exemplu: Identificarea unuor structuri mai simple în rețelele filogenetice
  - *Rețelele filogenetice* – folosite în biologia computațională pentru a modela structuri de ARN, proteine, interacțiuni între gene și proteine
- Calculul distanțelor în grafuri: *path multiplicity distance*, *distanța tripartită*, *distanța nodală*

Numărarea grafurilor

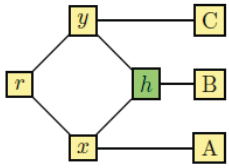
- În general, $n \geq 1$ noduri etichetate pot fi aranjate în $2^{\frac{n(n-1)}{2}}$ moduri pentru a forma un graf pentru că muchiile unui graf cu n noduri sunt o submulțime a mulțimii de $C_n^2 = \frac{n(n-1)}{2}$ perechi de vârfuri
- Unele aplicații folosesc grafuri etichetate în care nodurile și muchiile au atribute suplimentare



Figură: Trei noduri etichetate A, B și C pot fi aranjate în $2^{\frac{3 \cdot 2}{2}} = 8$ moduri pentru a forma un graf.

Găsirea drumurilor în grafuri (1)

- Oricare două noduri pot fi conectate prin mai mult de un drum într-un graf, chiar dacă nu se traversează nicio muchie mai mult de o singură dată într-un drum dintre două noduri ale grafului

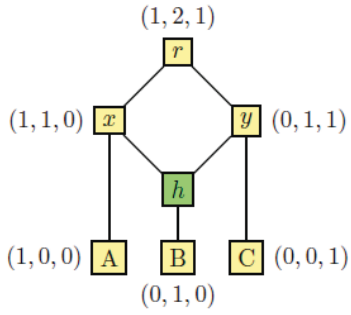


Figură: Există două drumuri între rădăcină și nodul terminal etichetat cu B: r-x-h-B și r-y-h-B.

- *Înalțimea* unui nod dintr-o rețea filogenetică este lungimea celui mai lung drum de la nod către un nod terminal

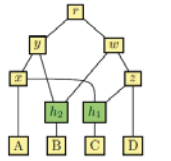
Găsirea drumurilor în grafuri (2)

- În rețeaua filogenetică din figură, nodurile terminale au înlălțimea 0, nodul hibrid are înlălțimea 1, nodurile interne de tip arbore au înlălțimea 2 și rădăcina are înlălțimea 3. Există câte un drum de la rădăcină la nodurile terminale A și C și două la nodul terminal B, iar acest lucru se poate reprezenta prin vectorul (1, 2, 1) de distanțe
- Copiii rădăcinii au ca vectori de distanțe (1, 1, 0) și (0, 1, 1), de unde rezultă (1, 2, 1) = (1, 1, 0) + (0, 1, 1)

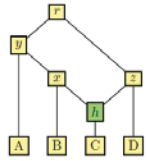


Distanța *path multiplicity* dintre grafuri

- Similaritățile sau diferențele dintre două rețele filogenetice pot fi determinate prin calcularea unei distanțe între rețele
- Distanța *path multiplicity* se bazează pe numărul de drumuri de la nodurile interne la nodurile terminale în cele două rețele
- Se calculează între două rețele filogenetice ale căror noduri terminale au aceleași etichete și eprezintă numărul de vectori de distanțe prin care diferă cele două rețele (6 în exemplu)



nod	h	vector
A	0	(1, 0, 0, 0)
B	0	(0, 1, 0, 0)
C	0	(0, 0, 1, 0)
D	0	(0, 0, 0, 1)
h_1	1	(0, 0, 1, 0)
h_2	1	(0, 1, 0, 0)
x	2	(1, 0, 1, 0)
z	2	(0, 0, 1, 1)
w	3	(0, 1, 1, 1)
y	3	(1, 1, 1, 0)
r	4	(1, 2, 2, 1)



nod	h	vector
A	0	(1, 0, 0, 0)
B	0	(0, 1, 0, 0)
C	0	(0, 0, 1, 0)
D	0	(0, 0, 0, 1)
h	1	(0, 0, 1, 0)
x	2	(0, 1, 1, 0)
z	2	(0, 0, 1, 1)
y	3	(1, 1, 1, 0)
r	4	(1, 1, 2, 1)

Distanța *path multiplicity* dintre grafuri (2)

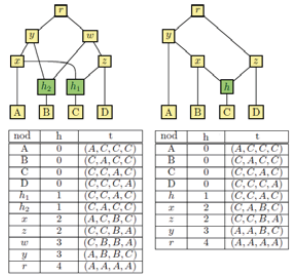
- Algoritmic, se contorizează numărul de vectori partajați de cele două rețele, parcurgând simultan vectorii sortați din rețele

function path_multiplicity_distance(N_1, N_2)

```
1  path_multiplicity(N1, μ1)
2  path_multiplicity(N2, μ2)
3  sort μ1 and μ2
4  n1 ← number of nodes of N1
5  n2 ← number of nodes of N2
6  i1 ← 1;  i2 ← 1
7  c ← 0
8  while i1 ≤ n1 and i2 ≤ n2      do
9          if μ1[i1] < μ2[i2]   then
10                 i1 ← i1 + 1
11         else  if μ1[i1] > μ2[i2]   then
12                 i2 ← i2 + 1
13         else
14                 i1 ← i1 + 1;      i2 ← i2 + 1
15                 c ← c + 1
16 return n1 + n2 − 2c
```

Distanța tripartită dintre grafuri

- *Distanța tripartită* dintre grafuri se bazează pe clasificarea nodurilor terminale în descendenți direcți, indirecți și nedescendenți relativ la nodurile celor două rețele filogenetice care se compară
- Cele două rețele filogenetice diferă prin vectorii tripartiți (A,A,B,C), (A,B,B,C), (A,C,B,C), (C,A,B,C), (C,A,C,C), (C,B,B,A), iar distanța tripartită dintre ele este 6



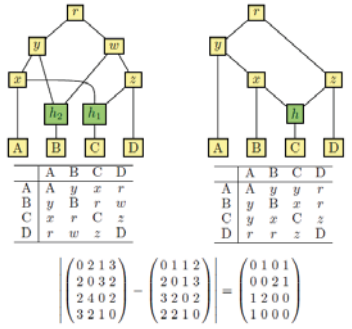
nod	h	t
A	0	(A,C,C,C)
B	0	(C,A,C,C)
C	0	(C,C,A,C)
D	0	(C,C,C,A)
h_1	1	(C,C,A,C)
h_2	1	(C,A,C,C)
x	2	(A,C,B,C)
z	2	(C,C,B,A)
w	3	(C,B,B,A)
y	3	(A,B,B,C)
r	4	(A,A,A,A)

nod	h	t
A	0	(A,C,C,C)
B	0	(C,A,C,C)
C	0	(C,C,A,C)
D	0	(C,C,C,A)
h	1	(C,C,A,C)
x	2	(C,A,B,C)
z	2	(C,C,B,A)
y	3	(A,A,B,C)
r	4	(A,A,A,A)

Figură: Nodurile sunt sortate ascendent după înălțime în cele două tabele. A: descendent direct, B: descendent indirect, C: nedescendent.

Distanța nodală dintre grafuri (1)

- *Distanța nodală* dintre grafuri se bazează pe cel mai scurt drum dintre nodurile terminale în cele două rețele filogenetice care se compară
- Similaritățile și diferențele dintre cele două rețele sunt determinate din matricele de distanțe dintre fiecare pereche de noduri terminale și cei mai apropiați ascendenți comuni
- Adunând valorile din matricea rezultat se obține că distanța nodală dintre cele două grafuri este 9



	A	B	C	D
A	A	y	x	r
B	y	B	r	w
C	x	r	C	z
D	r	w	z	D

	A	B	C	D
A	A	y	y	r
B	y	B	x	r
C	y	x	C	z
D	r	r	z	D

$$\left|\begin{pmatrix}0&2&1&3\\2&0&3&2\\2&4&0&2\\3&2&1&0\end{pmatrix}-\begin{pmatrix}0&1&1&2\\2&0&1&3\\3&2&0&2\\2&2&1&0\end{pmatrix}\right|=\begin{pmatrix}0&1&0&1\\0&0&2&1\\1&2&0&0\\1&0&0&0\end{pmatrix}$$

Distanța nodală dintre grafuri (2)

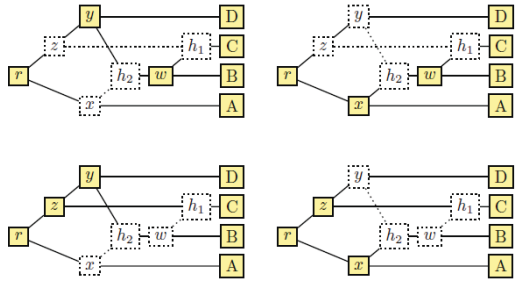
- Cel mai apropiat ascendent comun al fiecărei perechi de noduri terminale este calculat o singură dată, cu scopul de a obține cele două distanțe nodale dintre ele

function nodal distance(N_1, N_2)

```
1  L ← terminal node labels in N1 and N2
2  n ← length(L)
3  d ← 0
4  for i = 1, ··· , ndo
5          i1 ← terminal node of N1 labeled L[i]
6          i2 ← terminal node of N2 labeled L[i]
7          for j = i + 1, ··· , n do
8                  j1 ← terminal node of N1 labeled L[j]
9                  j2 ← terminal node of N2 labeled L[j]
10                 l1 ← LCSA(N1, i1, j1)
11                 l2 ← LCSA(N2, i2, j2)
12                 d1 ← distance(N1, l1, i1)
13                 d2 ← distance(N2, l2, i2)
14                 d ← d + |d1 − d2|
15                 d1 ← distance(N1, l1, j1)
16                 d2 ← distance(N2, l2, j2)
17                 d ← d + |d1 − d2|
18 return d
```

Găsirea arborilor în grafuri (1)

- Găsirea arborilor conținuți de un graf este utilă pentru calculul distanțelor dintre două grafuri
- O rețea filogenetică *full resolved* care are n noduri terminale și m noduri hibride conține 2^m arbori filogenetici, fiecare dintre ei cu n noduri terminale, care rezultă din tăierea uneia dintre cele două muchii care converg către fiecare nod hibrid și contractarea drumurilor



Găsirea arborilor în grafuri (2)

- Muchiile șterse sunt adăugate din nou la rețeaua N după ce au fost generați toți arborii care nu conțin acele muchii, pentru a evita crearea unor copii locale ale rețelelor fologenetice după fiecare pas
- Rezultatele sunt introduse în T care inițial este gol

procedure explode(N, T)

```
1  if N has no hybrid nodesthen
2          contract any elementary paths in N
3          T ← T ∪ N
4     else
5          v ← a hybrid node of N
6          for all parents u of node v  do
7                  delete edge (u, v) from N
8                  explode(N, T)
9                  add edgde (u, v) to N
```

Bibliografie

- Gabriel Valiente - *Combinatorial Pattern Matching Algorithms in Computational Biology Using Perl and R*. Chapman and Hall/CRC, 2009.

Cursul nr. 9

ALGORITMI RANDOMIZAȚI

Cuprins

Introducere

Un exemplu: RandQS

Monte Carlo versus Las Vegas

Algoritmi Monte Carlo
- Calculul valorii lui π
- Verificarea produsului a două matrice
- Problema găsirii motivului

Introducere

- randomizat, stocastic sau probabilist?
- Algoritmii clasici sunt *determiniști*
- Algoritmii randomizați - algoritmi în care se iau anumite decizii în mod aleatoriu, în timpul execuției implementării acestor algoritmi
- Caracteristica principală: un anumit aleatorism introdus în mod *intenționat* și *controlat*.
- **Presupunere**: alegerea sau decizia aleatoare este o operație *atomică* (realizată în unitatea de timp)
- **Consecință**: ordinul de complexitate al algoritmului nu este afectat, față de algoritmul determinist echivalent
- **Atenție!** generarea de valori aleatoare, conform cu o distribuție oarecare, influențează timpul de execuție, rezultatul fiind însă același (e.g. șirul sortat, pentru algoritmul RandQS)

Algoritmi determiniști vs. algoritmi randomizați

Algoritmi determiniști

- **Scop**: de a dovedi că un algoritm rezolvă *corect* o problemă (întotdeauna) și *repede* (în mod tipic, ordinul de complexitate să fie polinomial în funcție de dimensiunea intrării)

Algoritmi randomizați

- În plus față de intrările sau datele problemei, un algoritm randomizat va face o serie de alegeri, în mod aleatoriu, în timpul execuției, pe baza unei surse de numere aleatoare
- Comportamentul algoritmului poate varia chiar pentru același set de date de intrare
- **Scop**: de a proiecta algoritmul și de a dovedi că răspunsul algoritmului este probabil să fie cel corect, pentru fiecare set de date de intrare

Avantajele algoritmilor randomizați

A nu se confunda

- Algoritm randomizat $\neq$ Analiza probabilistică a algoritmilor, pentru care:
  - Datele de intrare sunt generate conform cu o distribuție aleatoare
  - Se arată că algoritmul funcționează pentru majoritatea seturilor de date de intrare

Avantaje:

- Simplitate
- Performanțe mai bune
- Pentru multe probleme, un algoritm randomizat poate fi cel mai simplu sau cel mai eficient, sau uneori ambele

Domenii de aplicativitate (după Raghavan)

- Algoritmi numerici - testul de numere prime
- Structuri de date - sortare, căutare, geometrie computațională
- Identități algebrice - verificarea identității a două polinoame sau matrice
- Optimizare (programare matematică) - algoritmi mai rapizi de programare liniară (optimizarea unei funcții obiectiv liniare) e.g. maximizarea expresiei $c^T x$ atunci când $Ax \leq b$
- Algoritmi în grafuri - cea mai scurtă cale, tăieturi minime, arborele parțial de cost minim
- Calcul paralel și distribuit - evitarea *deadlock-ului*
- etc.

Algoritmi randomizați

- Principiile ce stau la baza algoritmilor randomizați pot fi identificate chiar în metodele de tip Monte Carlo utilizate în analiza numerică, statistică sau simulare
- Alegerile aleatoare realizate de către un algoritm randomizat sunt *independente* de valorile de intrare ale algoritmului
- Conceptul de mașină Turing probablistică a fost propus încă din 1955 de către Leeuw, Moore, Shannon și Shapiro [4], de Rabin în 1963 și Gill în 1977
- Exemplele prezentate sunt inspirate din Karp [2], Motwani [3].

Randomized Quick Sort (RandQS)

RandQS

Se bazează pe algoritmul propus de Hoare în 1962

- **Input**: Un șir S de numere, de lungime n
- **Output**: Șirul S ordonat crescător

1 Se alege în mod aleator un element y din S
2 Se compară fiecare element din S cu y și se determină mulțimile:
$S_1 = \{s \in S | s \leq y\}$
$S_2 = \{s \in S | s > y\}$
3 Se sortează în mod recursiv mulțimile S_1 si S_2
4 Se tipărește: mulțimea S_1 ordonată, y, mulțimea S_2 ordonată

Observație: alegerea elementului s se face conform cu o distribuție uniformă, astfel că fiecare element din S are șanse egale să fie ales.

Analiza de timp pentru RandQS

- Cel mai rău caz: $O(y^2)$ (vs. $O(n^2)$ pentru algoritmul clasic)
- Ordinul de timp estimat (expected): $O(ylogy)$ (vs. $O(nlogn)$)
- Ordinul de timp estimat este, de regulă, o bună metrică pentru măsurarea performanței unui algoritm randomizat (mai bună decât cel mai rău caz)
- Algoritmul RandQS va da întotdeauna un răspuns corect

Monte Carlo vs. Las Vegas

Monte Carlo

Un algoritm de tip *Monte Carlo* ruleaza într-un număr fix de pași, iar rezultatul este corect cu o probabilitate $\geq \frac{1}{3}$

Las Vegas

Un algoritm de tip *Las Vegas* întotdeauna conduce la un rezultat corect, dar timpul de rulare este o variabilă aleatoare (cu valoare medie finită)

Repetări independente ale algoritmului de tip Monte Carlo conduc la reducerea exponențială a probabilității de a da un răspuns greșit.

Calculul valorii lui π

Problema acului lui Buffon

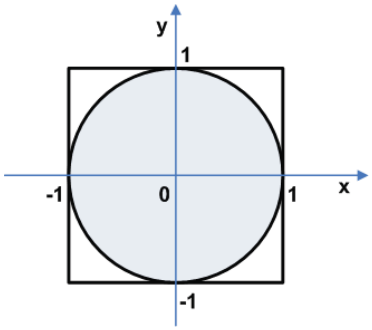
Experimentul clasic (Georges-Louis Leclerc, conte de Bouffon, 1777), al aruncării unui ac pe o hârtie cu linii paralele. Soluția, folosind geometrie integrală, pentru calculul probabilității ca acul să cadă între 2 linii paralele poate fi folosită pentru proiectarea unui algoritm de tip *Monte Carlo* pentru aproximarea valorii lui π.

- În geometria Euclidiană, π reprezintă prin definiție raportul dintre circumferința unui cerc și diametrul său, acest raport fiind constant indiferent de raza cercului
- π mai poate fi definit ca raportul dintre aria unui cerc și aria pătratului cu latura egală cu raza cercului respectiv

Algoritmul randomizat de calcul al valorii lui π

- Se alege în mod aleator un punct (x, y), unde x și y au valori în intervalul $[-1, 1]$
- Probabilitatea ca acest punct să fie în interiorul cercului unitate este dată de proporția dintre aria cercului unitate și cea a pătratului de latură 1:

$$P\{x^2 + y^2 < 1\} = \frac{A_{cerc}}{A_{pătrat}} = \frac{\pi r^2}{r^2} = \pi \qquad (1)$$



Algoritmul randomizat de calcul al valorii lui π (2)

- Această probabilitate poate fi estimată dacă alegem în mod aleator acel punct de N ori, atunci în M cazuri din cele N posibile punctul se va afla în interiorul cercului unitate, astfel:

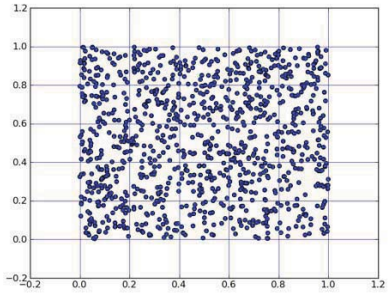
$$P\{x^2 + y^2 < 1\} = \frac{M}{N} \tag{2}$$

- Pentru N tinzând la infinit, cele două probabilități vor fi egale, de unde valoarea lui π se poate calcula ca fiind:

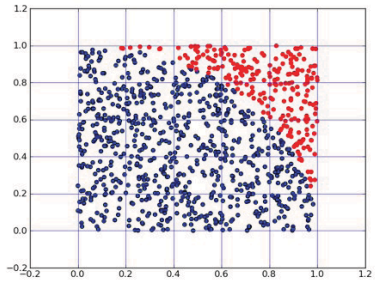
$$\pi = \frac{M}{N}|_{N\to\infty} \tag{3}$$

Algoritmul randomizat de calcul al valorii lui π (3)

- Se generează în mod aleator un număr N cât mai mare de perechi de valori (x, y) cuprinse în domeniul $[-1, 1]x[-1, 1]$ după o distribuție uniformă (valorile x și y sunt realizări alea unor variabile aleatoare ξ și η independente)
- Se numără câte perechi de puncte (M la număr) care sunt în interiorul cercului unitate și apoi se calculează valoarea lui π ca raport între M și N



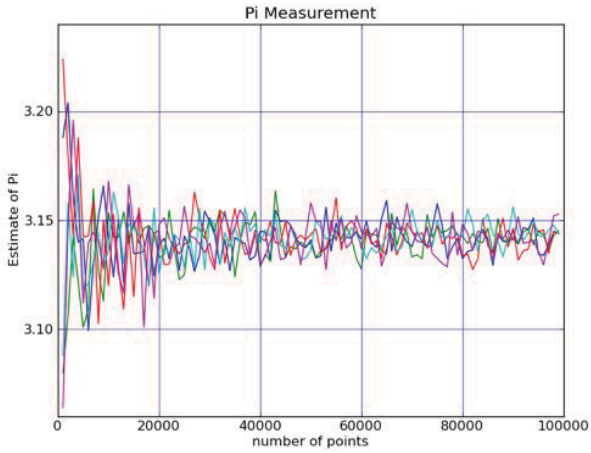
(a)



(b)

Precizia de calcul

- Precizia cu care se calculează valoarea lui π depinde în mod direct și evident de valoarea lui N
- Cu cât N este mai mare, cu atât valoarea calculată pentru π este mai aproape de valoarea de 3.1415926...



Verificarea produsului a două matrice

Problemă

Fie A, B și C trei matrice de dimensiune $n \times n$, cu valori aparținând unui câmp finit (câmp Galois) $\mathcal{F}$. Vrem să verificăm că $AB = C$

- Multiplicarea matricilor (abordare deterministă) este în $O(n^3)$
- Folosind algoritmi sofisticați, ordinul de complexitate poate fi redus la $O(n^{2.38})$

Algoritmul randomizat

1 Se alege un vector x de lungime n cu valori aleatoare din $\mathcal{F}$
2 Se calculează $z = A(Bx)$
3 **if** $z = Cx$ **then** "$AB = C$"
4 **else** "$AB \neq C$"

- Algoritmul randomizat are un ordin de complexitate în $O(n^2)$
- Dacă $AB = C$ atunci rezultatul algoritmului randomizat va fi "$AB = C$"
- Dacă $AB \neq C$ atunci rezultatul va fi "$AB = C$" cu o probabilitate de cel mult $\frac{1}{card\,\mathcal{F}}$

Problema găsirii motivului

Problema găsirii motivului [1]

Fiind dată o listă t de secvențe de lungime n, să se găsească *cel mai bun* pattern (motiv) de lungime l care apare în fiecare din cele t secvențe

- în capitolele precedente au fost prezentați algoritmi determiniști de rezolvare a problemei (Greedy, Branch and Bound)
- Algoritmul randomizat: selectează în mod aleator locațiile posibile și apoi găsește o modalitate (Greedy) prin care să modifice aceste locații până la aflare pattern-ului (motivului) căutat
- rezolvarea problemei se bazează pe calculul de *profile*

Profile. Evaluarea secvențelor pe bază de profile

- Fie $s = (s_1, s_2, \cdots, s_t)$ o mulțime de poziții de început pentru $l - meri$ în cele t secvențe
- Subșirurile corespunzătoare acestor poziții de început vor conduce la formarea:
  - $t \times l$ matrice de aliniere
  - $4 \times l$ matrice de profil P (definite pe baza frecvențelor relative de apariție a literelor din secvențe)
- $Prob(a|P)$ este prin definiție probabilitatea ca un $l - mer$ a să fie creat de profilul P

$$Prob(a|P) = \prod_{i=1}^{n} p_{ai} \tag{4}$$

- Dacă a este foarte asemănător cu **consensus string** al profilului P, atunci $Prob(a|P)$ va fi mare
- Dacă a este foarte diferit, atunci $Prob(a|P)$ va fi mică

Exemplu

Fie un profil P de forma:

A	**1/2**	**7/8**	**3/8**	0	1/8	0
C	1/8	0	1/2	**5/8**	**3/8**	0
T	1/8	1/8	0	0	1/4	**7/8**
G	1/4	0	1/8	3/8	1/4	1/8

Probablitatea pentru consensus string va fi:

$$Prob(aaacct|P) = \frac{1}{2} \times \frac{7}{8} \times \frac{3}{8} \times \frac{5}{8} \times \frac{3}{8} \times \frac{7}{8} = 0.033646$$

$$Prob(atacag|P) = \frac{1}{2} \times \frac{1}{8} \times \frac{3}{8} \times \frac{5}{8} \times \frac{1}{8} \times \frac{1}{8} = 0.001602$$

A	**1/2**	7/8	**3/8**	0	**1/8**	0
C	1/8	0	1/2	**5/8**	3/8	0
T	1/8	**1/8**	0	0	1/4	7/8
G	1/4	0	1/8	3/8	1/4	**1/8**

Cel mai P probabil $l-mer$

Prin definiție cel mai P probabil $l-mer$ dintr-o secvență este un $l-mer$ care are probabilitatea cea mai mare de a fi creat dintr-un profil P

A	1/2	7/8	3/8	0	1/8	0
C	1/8	0	1/2	5/8	3/8	0
T	1/8	1/8	0	0	1/4	7/8
G	1/4	0	1/8	3/8	1/4	1/8

Problemă: Fiind dată o secvență **ctataaaccttacatc** să se găsească cel mai P probabil $l-mer$
Se calculează $Prob(a|P)$ pentru toate variantele de $6-meri$:
1: **c t a t a a a c c t t a c a t c**
2: **c t a t a a a a c c t t a c a t c**
3: **c t a t a a a a c c t t a c a t c**
...

Cel mai P probabil $l-mer$ (2)

Secvență	Calcule	$Prob(a\|P)$
ctataaaccttacat	1/8 x 1/8 x 3/8 x 0 x 1/8 x 0	0
ctataaaccttacat	1/2 x 7/8 x 0 x 0 x 1/8 x 0	0
ctataaaccttacat	1/2 x 1/8 x 3/8 x 0 x 1/8 x 0	0
ctataaaccttacat	1/8 x 7/8 x 3/8 x 0 x 3/8 x 0	0
ctataaaccttacat	1/2 x 7/8 x 3/8 x 5/8 x 3/8 x 7/8	**0.0336**
ctataaaccttacat	1/2 x 7/8 x 1/2 x 5/8 x 1/4 x 7/8	0.0299
ctataaaccttacat	1/2 x 0 x 1/2 x 0 1/4 x 0	0
ctataaaccttacat	1/8 x 0 x 0 x 0 x 0 x 1/8 x 0	0
ctataaaccttacat	1/8 x 1/8 x 0 x 0 x 3/8 x 0	0
ctataaaccttacat	1/8 x 1/8 x 3/8 x 5/8 x 1/8 x 7/8	0.0004

aaacct este cel mai P probabil $l-mer$ din **ctataaaccttacatc** deoarece $Prob(aaacct|P) = 0.0336$ este mai mare decât probabilitățile tuturor celorlalți $l-meri$

Cei mai P-probabili l-meri din mai multe secvențe

Găsirea celui mai P-probabil l-mer în fiecare secvență

A	1/2	7/8	3/8	0	1/8	0
C	1/8	0	1/2	5/8	3/8	0
T	1/8	1/8	0	0	1/4	7/8
G	1/4	0	1/8	3/8	1/4	1/8

ctataaacgttacatc
atagcgattcgactg
cagcccagaaccct
cggtataccttacatc
tgcattcaatagctta
tatcctttccactcac
ctccaaatcctttaca
ggtcatcctttatcct

Cei mai P-probabili l-meri din mai multe secvențe (2)

1	a	a	a	c	g	t	ctataaacgttacatc
2	a	t	a	g	c	g	atagcgattcgactg
3	a	a	c	c	c	t	cagcccagaaccct
4	g	a	a	c	c	t	cggtgaaccttacatc
5	a	t	a	g	c	t	tgcattcaatagctta
6	g	a	c	c	t	g	tgtcctgtccactcac
7	a	t	c	c	t	t	ctccaaatcctttaca
8	t	a	c	c	t	t	ggtctaccttatcct

Un nou profil P

A	**5/8**	5/8	**4/8**	0	0	0
C	0	0	4/8	**6/8**	**4/8**	0
T	1/8	**3/8**	0	0	**3/8**	6/8
G	2/8	0	0	2/8	1/8	**2/8**

negru bold - frecvență mai mare; albastru - frecvență mai mică

Căutarea Greedy a motivului unui profil

Soluție

Folosirea de cei mai P-probabili l-meri pentru ajustarea pozițiilor de început în vederea obținerii celui *mai bun* profil (adică a motivului)

1. Se selectează în mod aleator pozițiile de început
2. Se crează un profil P pe baza subsecvențelor ce încep cu aceste poziții
3. Se calculează cel mai P-probabil l-mer a în fiecare secvență și se modifică poziția de început la poziția de început a lui a
4. Se calculează un nou profil pe baza noilor poziții de început după fiecare iterație și se repeta procedeul până când nu se mai poate obține o ameliorare a evaluării l-merilor

Algoritmul - pseudocod

GreedyProfileMotifSearch(DNA, t, n, l)

1 selectează aleator pozițiile de început $s = (s_1, \cdots, s_t)$ din DNA
2 $bestScore \leftarrow 0$
3 **while** $Score(s, DNA) > bestScore$
4 calculează profilul P din s
5 $bestScore \leftarrow Score(s, DNA)$
6 **for** $i \leftarrow 1$ **to** t
7 găsește cel mai P-probabil l-mer a din secvența i
8 $s_i \leftarrow$ poziția de început a lui a
9 **return** $bestScore$

Analiza algoritmului Greedy propus

- Deoarece pozițiile de început au fost alese în mod aleator, sunt puține șanse de a ne afla în preajma soluției optime - prin urmare va dura destul de mult până la găsirea motivului optim
- Este puțin probabil că pozițiile aleatoare de început vor conduce la soluția corectă
- În practică acest algoritm este rulat de mai multe ori cu speranța că pozițiile aleatoare vor fi aproape de soluția optimă (din pură întâmplare)
- GreedyProfileMotifSearch - nu este cea mai bună metodă de găsit motive, dar poate fi îmbunătățită prin utilizarea **eșantionării Gibbs** (Gibbs sampling): o procedură iterativă care înlocuiește câte un l-mer cu altul nou, după fiecare iterație
- eșantionarea Gibbs este o metodă mai lentă, care alege noi l-meri în mod aleator, crescând șansele ca metoda să conveargă spre soluția corectă

Algoritmul "Gibbs sampling"

1. Se selectează în mod aleator pozițiile de început $s = (s_1, \cdots, s_t)$ și se formează o mulțime de l-meri asociați acestor poziții
2. Se alege în mod aleator una din cele t secvențe
3. Se crează un profil P din celelalte $t - 1$ secvențe
4. Pentru fiecare poziție în secvența aleasă spre a fi eliminată, se calculează probabilitatea ca l-merul ce începe la acea poziție să fie generat de profilul P
5. Se alege o nouă poziție pentru secvența respectivă pe baza probabilității calculate la pasul 4
6. Se repetă pașii 2-5 până când nu se mai obține nici o îmbunătățire

Gibbs sampling - un exemplu

Date de intrare

$t = 5$ secvențe, lungimea motivului $l = 8$

1. GTAAACAATATTTATAGC
2. AAAATTTACCTCGCAAGG
3. CCGTACTGTCAAGCGTGG
4. TGAGTAAACGACGTCCCA
5. TACTTAACACCCTGTCAA

1) Se selectează în mod aleator pozițiile de început $s = (s_1, s_2, s_3, s_4, s_5)$ în cele 5 secvențe:

$s_1 = 7$ GTAAACAATATTTATAGC
$s_2 = 11$ AAAATTTACCTTAGAAGG
$s_3 = 9$ CCGTACTGTCAAGCGTGG
$s_4 = 4$ TGAGTAAACGACGTCCCA
$s_5 = 1$ TACTTAACACCCTGTCAA

Gibbs sampling - un exemplu (2)

2) Se alege în mod aleator una din cele 5 secvențe: secvența 2 - AAAATTTACCTTAGAAGG
3) Se crează un profil P din celelalte 4 secvențe:

1	A	A	T	A	T	T	T	A
3	T	C	A	A	G	C	G	T
4	G	T	A	A	A	C	G	A
5	T	A	C	T	T	A	A	C
A	1/4	2/4	2/4	3/4	1/4	1/4	1/4	2/4
C	0	1/4	1/4	0	0	2/4	0	1/4
T	2/4	1/4	1/4	1/4	2/4	1/4	1/4	1/4
G	1/4	0	0	0	1/4	0	3/4	0
consensus string	T	A	A	A	T	C	G	A

Gibbs sampling - un exemplu (3)

4) Se calculează probabilitatea $Prob(a|P)$ pentru toți cei 8-meri posibili în secvența eliminată (nr. 2):

AAAATTTACCTTAGAAGG	0.000732
AAAATTTACCTTAGAAGG	0.000122
AAAATTTACCTTAGAAGG	0
AAAATTTACCTTAGAAGG	0
AAAATTTACCTTAGAAGG	0
AAAATTTACCTTAGAAGG	0
AAAATTTACCTTAGAAGG	0
AAAATTTACCTTAGAAGG	0.000183
AAAATTTACCTTAGAAGG	0
AAAATTTACCTTAGAAGG	0
AAAATTTACCTTAGAAGG	0

Gibbs sampling - un exemplu (4)

5) Se alege o nouă poziție pentru secvența respectivă pe baza probabilității calculate la pasul precedent (4):

- Se normează probabilitățile la cea mai mică:
  Poziția de început 1: $Prob(AAAATTTA|P) = \frac{0.000732}{0.000122} = 6$
  Poziția de început 2: $Prob(AAATTTAC|P) = \frac{0.000122}{0.000122} = 1$
  Poziția de început 8: $Prob(ACCTTAGA|P) = \frac{0.000183}{0.000122} = 1.5$
- Se calculează probabilitățile pozițiilor de început pe baza rapoartelor calculate anterior:
  Probablitatea de a selecta poziția 1 de început:
  $\frac{6}{6+1+1.5} = 0.706$
  Probablitatea de a selecta poziția 2 de început:
  $\frac{1}{6+1+1.5} = 0.118$
  Probablitatea de a selecta poziția 8 de început:
  $\frac{1.5}{6+1+1.5} = 0.176$

Gibbs sampling - un exemplu (5)

În ipoteza că alegem poziția de început cu cea mai mare probabilitate (poziția 1), se obțin următoarele secvențe și poziții de început:

$s_1 = 7$ GTAAACAATATTTATAGC
$s_2 = 1$ AAAATTTACCTTAGAAGG
$s_3 = 9$ CCGTACTGTCAAGCGTGG
$s_4 = 4$ TGAGTAAACGACGTCCCA
$s_5 = 1$ TACTTAACACCCTGTCAA

6) Se repetă pașii 2-5 până când nu se mai obține nici o îmbunătățire

Analiza algoritmului "Gibbs sampling"

- Algoritmul trebuie modificat pentru cazul când secvențele au distribuții diferite (inegale) ale nucleotidelor
- Gibbs sampling converge adesea la motive optime local mai degrabă decât la optime globale
- Trebuie rulat cu multe poziții de început alese în mod aleator, pentru a obține rezultate bune

Bibliografie

Neil C. Jones and Pavel A. Pevzner.
An Introduction to Bioinformatics Algorithms.
MIT Press, 2004.

Richard M. Karp.
An introduction to randomized algorithms.
Discrete Appl. Math., 34(1-3):165–201, 1991.

Rajeev Motwani and Prabhakar Raghavan.
Randomized Algorithms.
Cambridge University Press, 1995.

C. E. Shannon and J. McCarthy.
Automata Studies.
Princeton University Press, Princeton, NJ, USA, 1956.

Cursul nr. 10

ALGORITMI DE APROXIMARE PENTRU PROBLEME NP COMPLETE

Cuprins

Utilitatea algoritmilor de aproximare, definiţii, notaţii

Problema acoperirii cu vârfuri

Problema comis–voiajorului

Problema submulţimii de sumă maximă

Problema acoperirii mulţimii

Problema încărcării balansate

Exemple de probleme NP complete

- *Problema celui mai lung superstring*: se dau câteva șiruri de caractere; să se găsească cel mai scurt șir care le are pe toate drept subșiruri; problema se regăsește frecvent în analiza secvențelor ADN
- *Asignarea sarcinilor într–un sistem multiprocesor*: se dau o mulțime de programe care trebuie executate (pentru fiecare știm durata de execuție) și niște procesoare; cum trebuie distribuite sarcinile astfel încât durata totală de execuție să fie minimă?
- *Deservirea clienților*: se dau niște clienți în diferite locații fixe, avem o flotă de mașini care trebuie să le facă livrări; se cere determinarea unui plan de deservire care să ducă la costuri minime
- O listă de probleme NP-complete se găsește aici

Utilitatea algoritmilor de aproximare

- *Although this may seem a paradox, all exact science is dominated by the idea of approximation. When a man tells you that he knows the exact truth about anything, you are safe in inferring that he is an inexact man.* Bertrand Russel (1872-1970)
- Multe probleme de utilitate practică sunt NP-complete
- Variante de lucru:
  1. dacă dimensiunile problemelor sunt mici, atunci se poate aplica un algoritm brute–force de timp exponențial
  2. se izolează cazuri particulare care se pot rezolva în timp polinomial
  3. se găsește un algoritm care să găsească soluții *aproape optimale* în timp polinomial
- **Algoritm de aproximare:** algoritm care returnează soluții aproape optimale

Definiții, notații

- Presupunem că lucrăm cu probleme de optimizare: se cere determinarea unei soluții de cost minim sau maxim
- Notăm C^* costul soluției optime; putem considera că avem $C^* > 0$

Definiție (Raport de aproximare)

Un algoritm de aproximare are raportul de aproximare $\rho(n)$ dacă pentru orice dimensiune n a problemei:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n) \tag{1}$$

- Dacă problema este de maximizare: $C \leq C^*$, $\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) = \frac{C^*}{C} \geq 1$; analog pentru o problemă de minimizare; deci $\rho(n) \geq 1$

Definiții, notații

- Dacă algoritmul de aproximare realizează un raport de aproximare $\rho(n)$, spunem că este un **$\rho(n)$–algoritm de aproximare**
- Dorim ca algoritmul de aproximare să se execute în timp polinomial și $\rho(n)$ să fie cât mai apropiat de 1
- Uneori ρ nu depinde de n
- Alteori $\rho(n)$ poate să scadă dacă algoritmul este lăsat să ruleze mai mult

Definiție (Schemă de aproximare)

O schemă de aproximare pentru o problemă de optimizare este un algoritm de aproximare care primește la intrare o instanță a problemei și o valoare $\varepsilon > 0$ astfel încât pentru orice ε fixat, schema este un $(1+\varepsilon)$–algoritm de aproximare.

Definiții, notații

Definiție (Schemă de aproximare în timp polinomial)

Dacă pentru orice $\varepsilon > 0$ fixat o schemă rulează în timp polinomial, atunci ea se numește schemă de aproximare în timp polinomial.

- Ideal: scăderea lui ε de câteva ori ar trebui să ducă la creșterea timpului de calcul tot de câteva ori[1]
- Caz în care nu avem situația ideală: $O(n^{2/\varepsilon})$

Definiție (Schemă de aproximare în timp complet polinomial)

O schemă de aproximare în timp polinomial pentru care timpul de rulare este polinomial atât în $(1/\varepsilon)$ cât și în n se numește schemă de aproximare în timp complet polinomial.

- Exemplu: schemă de aproximare cu complexitatea $O\left((1/\varepsilon)^2 n^3\right)$

[1] Nu neapărat de același număr de ori.

Problema acoperirii cu vârfuri

- Se dă un graf neorientat $G = (V, E)$
- O acoperire cu vârfuri este un subset $V' \subseteq V$ astfel încât dacă (u, v) este o muchie din G, atunci $u \in V'$ sau $v \in V'$ (sau ambele)
- Dimensiunea unei acoperiri cu vârfuri este numărul de vârfuri din V'
- Problema cere determinarea unui V' ca mai sus cu număr minim de vârfuri
- Problema este NP-completă

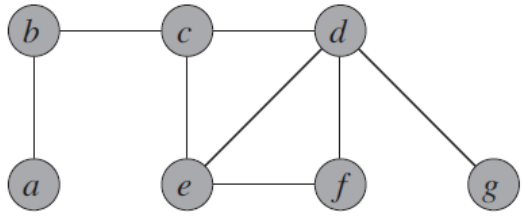


Figura: Acoperirea cu vârfuri optimală este mulțimea $\{b, d, e\}$.

Algoritm aproximativ pentru problema acoperirii cu vârfuri

```
Acoperire-cu-varfuri-aprox(G)
1  C ← ∅
2  E' ← E[G]
3  while E' ≠ ∅
4      fie (u, v) o muchie arbitrară din E'
5      C ← C ∪ {u, v}
6      șterge din E' orice muchie incidentă cu u sau v
7  return C
```

- Complexitatea algoritmului este dată de numărul de execuții al ciclului **while**. Dacă se folosesc liste de adiacență pentru reprezentarea lui G atunci complexitatea este $O(|V| + |E|)$.

Problema acoperirii cu vârfuri

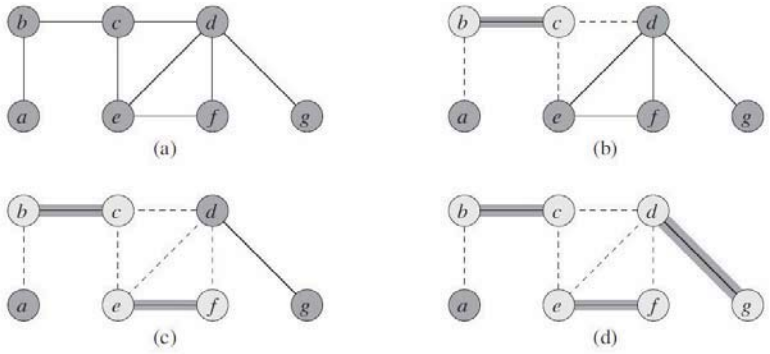


Figura: Aplicarea algoritmului `Acoperire-cu-varfuri-aprox` pentru un graf.(a) Graful de intrare. (b) Muchia (b, c) este prima munchie aleasă de algoritm. Vârfurile b, c sunt adăugate acoperirii cu vârfuri C. Muchiile (a, b), (c, e), (c, d) sunt șterse deoarece sunt acoperite de vârfurile b sau c. (c) Muchia (e, f) este adăugată la C. (d) Muchia (d, g) este adăugată la C. Mulțimea $C = \{b, c, d, e, f, g\}$ este acoperirea convexă produsă de algoritm; acoperirea optimală este $\{b, d, e\}$.

Problema acoperirii cu vârfuri

Teoremă

Algoritmul `Acoperire-cu-varfuri-aprox` *are este un 2-algoritm de aproximare polinomial.*

- Altfel zis: soluția furnizată de algoritm are de cel mult două ori mai multe vârfuri decât soluția optimă C^*.

Demonstrație: Mulțimea C este o acoperire cu vârfuri a lui G deoarece ea se completează atâta timp cât mai există muchii în E'. Notăm cu A mulțimea de muchii alese la linia 4 din algoritm; A crește la fiecare iterație. Cum C^* este o acoperire a lui G, C^* include cel puțin unul din capetele fiecărei muchii din A, deci $|C^*| \geq |A|$. Pe de altă parte, pentru u, v alese la linia 4 avem că niciunul din ele nu este deja în C, datorită eliminării muchiilor făcute anterior la linia 6, deci $|C| = 2|A|$. De aici $|C| = 2|A| \leq 2|C^*|$. □

Problema comis–voiajorului

- Se dă un graf neorientat $G = (V, E)$; fiecare muchie (u, v) are un cost $c(u, v) \geq 0$
- Se cere găsirea unui ciclu hamiltonian A de cost minim:

$$A = \arg\min_{A'} \sum_{(u,v)\in A'} c(u, v)$$

  unde A' este ciclu hamiltonian al lui G
- Considerăm că $c(\cdot, \cdot)$ satisface inegalitatea triunghiului:

$$c(u, w) \leq c(u, v) + c(v, w)\ \forall (u, w), (u, v), (v, w) \in E$$

  - *e.g.* distanța Euclidiană, distanța Manhattan
- Problema este NP-completă, cu sau fără această presupunere

Problema comis–voiajorului cu inegalitatea triunghiului

`PCV-tur-aproximativ(`G`)`

1 selectează un vârf $r \in V[G]$ pentru a fi rădăcină
2 calculează un arbore minim de acoperire T al lui G pornind de la r cu algoritmul lui Prim
3 fie H lista de vârfuri obținute prin parcurgere în preordine a lui T
4 **return** H

- Chiar folosind o variantă neeficientă a algoritmului lui Prim, complexitatea lui `PCV-tur-aproximativ` este $\Theta(|V|^2)$

Problema comis–voiajorului cu inegalitatea triunghiului

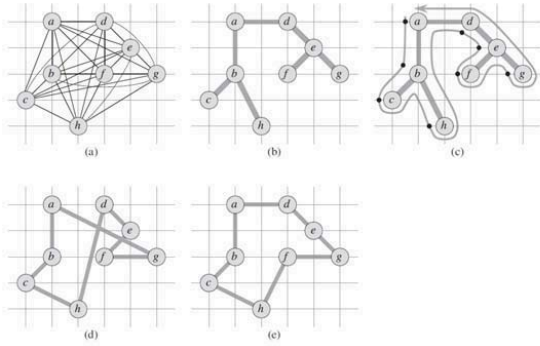


Figura: (a) Un graf neorientat. Distanța dintre vârfuri este distanța Euclidiană. (b) Un arbore minim de acoperire pentru graful dat, folosind vârful a ca rădăcină. (c) O parcurgere a lui T plecând din rădăcina a. Parcurgerea în preordine este a, b, c, h, d, e, f, g. (d) Un ciclu al vârfurilor obținut pe baza parcurgerii în preordine, soluția returnată de algoritmul de aproximare. Costul lui este aproximativ 19.074. (e) Un ciclu hamiltonian optimal H^*, cu costul aproximativ 14.715, cu circa 23% mai mic decât dat de algoritmul de aproximare.

PCV cu inegalitatea triunghiului - rezultat teoretic I

Teoremă

Algoritmul de aproximare `PCV-tur-aproximativ(G)` este un 2-algoritm de aproximare polinomial.

Demonstrație: Fie H^* un ciclu hamiltonian de cost minim. Ștergând o muchie se obține un arbore de acoperire; luând în considerare că muchia ștearsă are cost nenegativ și că arborele T determinat în linia 2 a algoritmului este de cost minim, obținem

$$c(T) \leq c(H^*) \tag{2}$$

O parcurgere completă W al lui T se obține prin enumerarea vârfurilor în preordine, ori de câte ori vârfurile sunt atinse. De exemplu, pentru parcurgerea de la slide-ul 14 $W = a, b, c, b, h, b, a, d, e, f, e, g, e, d, a$. O parcurgere completă conține fiecare muchie a grafului de exact două ori, avem:

$$c(W) = 2c(T) \tag{3}$$

PCV cu inegalitatea triunghiului - rezultat teoretic II

Din ecuațiile (2) și (3) obținem

$$c(W) \leq 2c(H^*) \tag{4}$$

Datorită vârfurilor repetate, parcurgerea completă W nu este un ciclu Hamiltonian. Datorită inegalității triunghiului, putem scoate vârfuri din W fără ca să creștem costul total al drumului. Fiecare a doua apariție de vârf în W este ștersă din W. Pentru exemplul considerat obținem exact a, b, c, h, d, e, f, g și se poate arăta că acesta este chiar H returnat de algoritmul `PCV-tur-aproximativ`. Deoarece H se obține ștergând vârfuri din W avem că:

$$c(H) \leq c(W) \tag{5}$$

Combinând (4) și (5) obținem $c(H) \leq 2c(H^*)$. □

Problema generală a comis–voiajorului

- Dacă renunțăm la presupunerea că funcția de cost c satisface inegalitatea triunghiului obținem forma generală a PCV

Teoremă
Dacă $P \neq NP$ atunci pentru orice constantă $\rho \geq 1$ nu există un ρ-algoritm polinomial de aproximare.

Demonstrație: Vezi [1], secțiunea 35.2.2. □

Problema submulțimii de sumă maximă

- Se dau o mulțime $S = \{x_1, x_2, \ldots, x_n\}$ de numere naturale și un număr natural t
- Problemă de decizie: să se determine dacă există o submulțime a lui S pentru care suma elementelor conținute este *exact* t
- Problema de optimizare derivată: să se determine o submulțime a lui S pentru care suma elementelor este cât mai mare posibilă, fără a depăși pe t
- Abordare brute–force: se determină toate submulțimile S' ale lui S și se calculează suma elementelor conținute
- Complexitatea este exponențială, deoarece mulțimea părților lui S are $2^{|S|}$ submulțimi

Problema submulțimii de sumă maximă

- Notație: pentru L listă de întregi și x un număr natural, $L + x$ este lista obținută prin adunarea lui x la fiecare element din L
  - exemplu: $L = \langle 1, 2, 3, 5, 9 \rangle$, $x = 2$, $L + x = \langle 3, 4, 5, 7, 11 \rangle$
- Folosim o procedură care preia două liste ordonate crescător și returnează lista obținută prin interclasarea lor, fără duplicate
- Complexitatea interclasării: $\Theta(|L| + |L'|)$
- Algoritm care implementează ideea brute-force:

```
Submultime-suma-maxima(S, t)
1  n ← |S|
2  L_0 = ⟨0⟩
3  for i = 1, n
4      L_i ← Interclaseaza − liste(L_{i−1}, L_{i−1} + x_i)
5      scoate din L_i elementele mai mari decât t
6  return cel mai mare element din L_n
```

Problema submulțimii de sumă maximă

- Notăm cu P_i mulțimea tuturor valorilor obținute prin selectarea unei submulțimi – posibil vide – a lui $\{x_1, x_2, \ldots, x_i\}$ și însumarea membrilor săi
  - exemplu: $S = \{1, 4, 5\}$; $P_1 = \{0, 1\}$; $P_2 = \{0, 1, 4, 5\}$, $P_3 = \{0, 1, 4, 5, 6, 9, 10\}$.
- Observăm că $P_i = P_{i-1} \cup (P_{i-1} + x_i)$
- Prin inducție după i se poate arăta că L_i conține fiecare element al lui P_i mai mic sau egal decât t
- Tot prin inducție: $|L_i| \leq 2^i$ deci algoritmul `Submultime-suma-maxima` este de complexitate exponențială

Problema submulțimii de sumă maximă

- Euristica pentru algoritmul de aproximare: dacă două numere dintr–o listă sunt suficient de apropiate, atunci poate fi menținut doar unul din ele
- Considerăm un parametrul al algoritmului: $0 < \delta < 1$
- Filtrarea unei liste L înseamnă obținerea listei L' cu număr minim de elemente, a.i. pentru fiecare element y care a fost scos din L să avem un $z \in L'$, $z \leq y$ ce aproximează pe y :

$$\frac{y - z}{y} \leq \delta \tag{6}$$

- Obținem: $(1 - \delta)y \leq z \leq y$
- Fiecare z devine astfel reprezentantul valorii y și deci y va fi scoasă din L
- Exemplu: pentru $L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle$ și $\delta = 0.1$ lista filtrată este $L' = \langle 10, 12, 15, 20, 23, 29 \rangle$

Problema submulțimii de sumă maximă

- Algoritmul de filtrare:

```
Filtreaza(L, δ)
1  m ← |L|
2  L' ← ⟨y_1⟩
3  ultimul ← y_1
4  for i = 2, m
5     if ultimul < (1 − δ)y_i
6           adaugă y_i la sfârșitul lui L'
7           ultimul ← y_i
8  return L'
```

- Intuitiv: un număr din L este adăugat la L' doar dacă nu poate fi reprezentat de cel mai recent număr introdus în L'
- Evident: orice element care apare în L' este din L; dacă L e sortată, atunci la fel e și L'
- Complexitate: $\Theta(|L|)$

Problema submulțimii de sumă maximă

```
Suma-submultimi-aprox(S, t, ε)
1  n ← |S|
2  L_0 ← ⟨0⟩
3  for i ← 1, n
4      L_i ← Interclaseaza − liste(L_{i−1}, L_{i−1} + x_i)
5      L_i ← Filtreaza(L_i, ε/n)
6      șterge din L_i toate elementele mai mari decât t
7  Fie z maximul din L_n
8  return z
```

Problema submulțimii de sumă maximă

- Exemplu: $L = \langle 104, 102, 201, 101 \rangle$, $t = 308$, $\varepsilon = 0.2 \Rightarrow \delta = \varepsilon/4 = 0.05$

linia 2	$L_0 = \langle 0 \rangle$
linia 4	$L_1 = \langle 0, 104 \rangle$
linia 5	$L_1 = \langle 0, 104 \rangle$
linia 6	$L_1 = \langle 0, 104 \rangle$
linia 4	$L_2 = \langle 0, 102, 104, 206 \rangle$
linia 5	$L_2 = \langle 0, 102, 206 \rangle$
linia 6	$L_2 = \langle 0, 102, 206 \rangle$
linia 4	$L_3 = \langle 0, 102, 201, 206, 303, 407 \rangle$
linia 5	$L_3 = \langle 0, 102, 201, 303, 407 \rangle$
linia 6	$L_3 = \langle 0, 102, 201, 303 \rangle$
linia 4	$L_4 = \langle 0, 101, 102, 201, 203, 302, 303, 404 \rangle$
linia 5	$L_4 = \langle 0, 101, 201, 302, 404 \rangle$
linia 6	$L_4 = \langle 0, 101, 201, 302 \rangle$

Problema submulțimii de sumă maximă: rezultat teoretic I

Teoremă

Algoritmul `Suma-submultimi-aprox` este o schemă de aproximare în timp complet polinomial pentru problema sumei submulțimii.

Demonstrație: Operațiile de filtrare din liniile 5 și 6 ale algoritmului duc de la un L_i submulțime a lui P_i la o mulțime cu aceeași proprietate, deci valoarea z returnată de algoritm este suma unei anumite submulțimi a lui S. Vom arăta că z nu este de $(1-\varepsilon)$ ori mai mică decât suma unei mulțimi optimale:
$C^*(1-\varepsilon) \leq C = z$.
Prin filtrare inducem o eroare de cel mult ε/n între valorile reprezentative rămase în L_i și valorile de dinainte de filtrare. Prin inducție după i putem arăta că $\forall y \in P_i, y \leq t\ \exists z \in L_i$:
$(1-\varepsilon/n)^i y \leq z \leq y$. Dacă $y^* \in P_n$ este optimul căutat, atunci $\exists z \in L_n$: $(1-\varepsilon/n)^n y^* \leq z \leq y^*$.
$(1-\varepsilon/n)^n$ este crescătoare cu n, deci pentru $n > 1$:
$1-\varepsilon < (1-\varepsilon/n)^n$ și deci $(1-\varepsilon)y^* \leq z$.

Problema submulțimii de sumă maximă: rezultat teoretic II

Pentru complexitatea algoritmului căutăm un majorant al lui $|L_i|$. După filtrare, oricare două elemente succesive z și z' din L_i sunt în relația $z/z' > 1/(1-\varepsilon/n)$ și de aici, numărul maxim al elementelor din orice L_i este:

$$\log_{1/(1-\varepsilon/n)} t = \frac{\ln t}{-\ln(1-\varepsilon/n)} \leq \frac{n \ln t}{\varepsilon}$$

Marginea e deci polinomială în n, $1/\varepsilon$ și $\log t$ = numărul de biți necesari pentru reprezentarea lui t. Timpul de execuție al algoritmului `Suma-submultimi-aprox` este polinomial în lungimea lui L_i și de aici `Suma-submultimi-aprox` este o schemă de aproximare în timp complet polinomial. □

Problema acoperirii mulțimii

- Problemă de optimizare, privind alocarea de resurse
- O instanță $(X, \mathcal{F})$ a problemei acoperirii mulțimii constă într–o mulțime finită X și o familie $\mathcal{F}$ de submulțimi ale lui X astfel încât
$$X = \bigcup_{S \in \mathcal{F}} S$$
- Se cere determinarea unei submulțimi $\mathcal{C} \subseteq \mathcal{F}$ cu $|\mathcal{C}|$ minim a.i.
$$X = \bigcup_{S \in \mathcal{C}} S$$
- $|\mathcal{C}|$ = dimensiunea lui $\mathcal{C}$

Problema acoperirii mulțimii

- Problema are o formă generală care se regăsește în multe alte situații
- Exemplu concret: $X = \{1, 2, 3, 4, 5\}$, $\mathcal{F} = \{\{1, 2, 3\}, \{2, 4\}, \{3, 4\}, \{4, 5\}\}$; evident, $\mathcal{F}$ realizează o acoperire a lui X, dar acoperire de dimensiune minimă este $\{\{1, 2, 3\}, \{4, 5\}\}$
- Problemă practică: avem un set de abilități și de oameni care dispun de anume abilități; trebuie găsit un comitet format din cât mai puțini indivizi pentru care abilitățile prezente să acopere abilitățile cerute.

Problema acoperirii mulțimii: exemplu

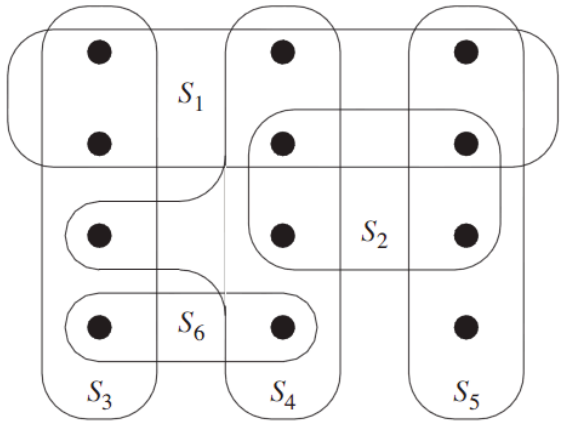


Figura: O instanță a problemei acoperirii mulțimii pentru U formată din 12 elemente și $\mathcal{S} = \{S_1, \ldots, S_5\}$. O acoperire a lui U cu număr minim de mulțimi din $\mathcal{S}$ este $\{S_3, S_4, S_5\}$.

Problema acoperirii mulțimii: algoritm de aproximare

- Ideea de bază: la fiecare iterație se alege în stil greedy mulțimea S care conține cele mai multe elemente neacoperite anterior

```
Acoperire-multimi-greedy(X, F)
1  U ← X      //multimea de elemente inca neacoperite
2  C ← ∅      //acoperirea ce se construieste iterativ
3  while U ≠ ∅
4      selectează un S ∈ F care maximizează |S ∩ U| //pas greedy
5      U ← U − S
6      C ← C ∪ {S}
7  return C
```

- Complexitate:
  - numărul de iterații este $O(\min(|X|, |\mathcal{F}|))$
  - corpul ciclului are complexitatea $O(|X||\mathcal{F}|)$
  - total: $O(|X||\mathcal{F}| \min(|X|, |\mathcal{F}|))$
- Se poate ajunge la algoritm de complexitate $O(\sum_{S \in \mathcal{F}} |S|)$

Problema acoperirii mulțimii

- Pentru exemplul dat în figura de mai jos, algoritmul alege în ordine: S_1, S_4, S_5 urmat de S_3 sau S_6.

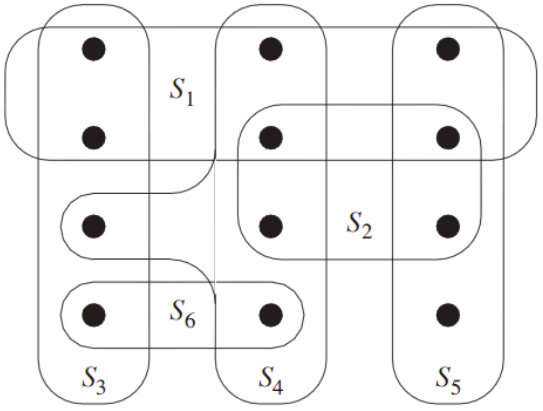


Figura: Subsetul de acoperire cu număr minim de mulțimi este $\{S_3, S_4, S_5\}$.

Problema acoperirii mulțimii

- Notație: $H(m) = \sum_{i=1}^{m} \frac{1}{i}$
- $H(m) \in \Theta(\log m)$

Teoremă

`Acoperire-multimi-greedy` este un $\rho(n)$–algoritm de aproximare polinomial, unde $\rho(n) = H(\max\{|S| : S \in \mathcal{F}\})$

Demonstrație: A se vedea secțiunea 35.3 din [1]. □

Corolar

`Acoperire-multimi-greedy` este un $(\ln |X| + 1)$–algoritm de aproximare polinomial.

- Cum funcția ln nu crește foarte repede, algoritmul este util.

Problema încărcării balansate

- Problema:
  - avem m mașini $M_1, \ldots M_m$
  - avem n sarcini, fiecare cu un timp de execuție $t_j > 0$
  - vrem plasarea sarcinilor pe mașini astfel încât încărcarea mașinilor să fie cât mai echilibrată
- Notăm $A(i)$ mulțimea sarcinilor repartizate mașinii M_i
- Timpul de lucru necesar pentru mașina M_i:

$$T_i = \sum_{j \in A(i)} t_j$$

- Dorim minimizarea *duratei totale* $T = \max_i T_i$
- Problema este NP-completă [2]

Problema încărcării balansate

- Algoritm greedy:

```
Balansare-Greedy(m, n, t_{1≤j≤n})
 1  for i = 1, m
 2      T_i ← 0
 3      A_i ← ∅
 4  for j = 1, n
 5      i ← arg min_{1≤k≤m} T_k
 6      //indicele mașinii cu cea mai mică încărcare
 7      //Asignează sarcina j mașinii M_i:
 8      A_i ← A_i ∪ {j}
 9      T_i ← T_i + t_j
10  return A_i, 1 ≤ i ≤ m
```

Problema încărcării balansate

- Ideea de bază: cea mai puțin încărcată mașină preia următoarea sarcină
- Exemplu: $n = 6$, $t_j \in \{2, 3, 4, 6, 2, 2\}$, $m = 3$

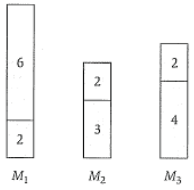


Figura: Rezultatul furnizat de algoritmul `Balansare-Greedy`

- Complexitate: dacă se folosește căutarea în mod repetat a minimului în linia 5, complexitatea este $\Theta(mn)$; prin folosirea unui min-heap se poate micșora complexitatea
  - calculați complexitatea algoritmului pentru cazul în care se folosește un min–heap; considerați efortul de actualizare a heap–ului în linia 8

Problema încărcării balansate

- Notăm cu T^* durata totală optimă (minimă)
- Considerăm timpul total $\sum_j t_j$; avem cel puțin o mașină care să execute cel puțin o fracție $1/m$ din munca totală
- Obținem deci că durata totală minimă îndeplinește condiția:

$$T^* \geq \frac{1}{m} \sum_{j=1}^{n} t_j \tag{7}$$

- Există un caz extrem în care inecuația (7) dă o margine depărtată: una din sarcini are durata foarte mare comparativ cu toate celelalte
  - dacă această sarcină este prima, algoritmul `Balansare-Greedy` determină comportamentul optim alocând sarcina unei mașini care nu va mai executa altă sarcină
  - această mașină va fi și ultima care se va opri
  - deci în acest caz T^* = timpul celei mai lungi sarcini
- Deducem încă o margine inferioară a lui T^*:

$$T^* \geq \max_{1 \leq j \leq n} t_j \tag{8}$$

Problema încărcării balansate I

Teoremă

Algoritmul `Balansare-Greedy` *produce o asignare de sarcini cu durata totală* $T \leq 2T^*$

Demonstrație: Considerăm mașina M_i care are încărcarea maximă; timpul total de execuție a sarcinilor alocate acestei mașini îl dă totdată pe T, deci $T = T_i$. Considerăm ultima sarcină t_j care este alocată lui M_i. Dacă t_j nu e prea mare față de celelalte sarcini, atunci prima margine inferioară dată în (7) este cea mai apropiată. Dacă t_j este disproporționat de mare față de restul, atunci a doua margine inferioară (8) are acuratețe mai mare. Când se asignează sarcina j lui M_i, M_i are cea mai mică încărcare – conform strategiei de alegere din algoritm. Încărcarea lui M_i înaintea asignării sarcinii j este $T_i - t_j$; orice altă mașină are cel

Problema încărcării balansate II

puțin încărcarea $T_i - t_j$. Însumând încărcarea tuturor mașinilor avem:

$$\sum_{k=1}^{m} T_k \geq m(T_i - t_j) \Leftrightarrow T_i - t_j \leq \frac{1}{m} \sum_{k=1}^{m} T_k$$

Evident, $\sum_{k=1}^{m} T_k = \sum_{j=1}^{n} t_j$; aplicând și inegalitatea (7) avem că

$$T_i - t_j \leq T^*$$

Din inecuația (8) avem că $t_j \leq T^*$. Obținem în final:

$$T = T_i = (T_i - t_j) + t_j \leq 2T^*$$

□

Problema încărcării balansate

- Caz în care raportul de aproximare este oricât de apropiat de 2:
  - m mașini, $n = m(m-1)+1$ sarcini
  - primele $n-1$ sarcini au durata $t_j = 1$, ultima are durata $t_n = m$
  - algoritmul distribuie echilibrat primele $n-1$ sarcini și pune a n–a sarcină pe o mașină oarecare
  - obținem $T = 2m-1$
  - distribuirea optimă este: sarcina n separat pe o mașină, primele $n-1$ sarcini distribuite echilibrat pe restul de $m-1$ mașini $\Rightarrow T^* = m$
  - Avem
$$\lim_{m\to\infty} \frac{T}{T^*} = 2$$
  - Analiza efectuată dă un rezultat de mărginire "strâns"

Problema încărcării balansate

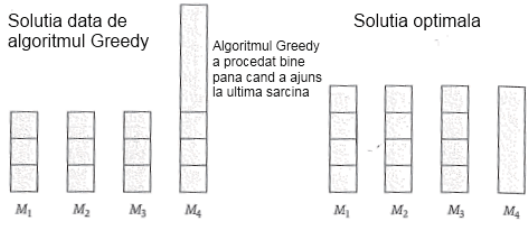


Figura: Soluția dată de algoritmul Greedy vs. soluția optimală

- Putem îmbunătăți algoritmul de aproximare
- Intuitiv: cazul cel mai nefavorabil este când ultima sarcină din listă are durata cea mai mare iar mașinile sunt deja ocupate cu alte sarcini
- Distribuirea sarcinilor cu timp t_j mai mare primele ar putea micșora durata totală

Problema încărcării balansate

```
Balansare-Sortata(m, n, t_{1≤j≤n})
 1  for i = 1, m
 2      T_i ← 0
 3      A_i ← ∅
 4  sortează sarcinile în ordinea descrescătoare a duratelor de execuție
 5  for j = 1, n
 6      i ← arg min_{1≤k≤m} T_k
 7      //indicele mașinii cu cea mai mică încărcare
 8      //Asignează sarcina j mașinii M_i:
 9      A_i ← A_i ∪ {j}
10      T_i ← T_i + t_j
11  return A_i, 1 ≤ i ≤ m
```

Problema încărcării balansate

- Dacă $m \geq n$ atunci algoritmul `Balansare-Sortata` este optimal, deoarece asignează fiecărei mașini o singură sarcină

Lema

Dacă sunt mai mult de m sarcini, atunci $T^* \geq 2t_{m+1}$.

Demonstrație: Considerăm doar primele $m+1$ sarcini în ordinea descrescătoare a duratelor. Avem deci $t_1 \geq t_2 \geq \cdots \geq t_m \geq t_{m+1}$. Primele m sarcini sunt distribuite câte una pe mașină; a $(m+1)$–a sarcină se distribuie unei mașini care deja are ceva alocat. Timpul de execuție este dat de aceasta și deci este cel puțin $2t_{m+1}$. □

Problema încărcării balansate

Teoremă
Algoritmul `Balansare-Sortata` *are raportul de aproximare* $3/2$.

Demonstrație: Trebuie să arătăm că $T \leq \frac{3}{2}T^*$. Considerăm mașina care are încărcarea maximă – și deci cea care dă durata întregului proces. Dacă M_i are o singură sarcină, atunci alocarea este optimală. Presupunem deci că mașina M_i are două sarcini alocate. Fie t_j durata ultimei sarcini alocate lui M_i. Avem că $j \geq m+1$ – deoarece primele m sarcini au fost asignate primelor m mașini – și deci $t_j \leq t_{m+1} \leq \frac{1}{2}T^*$.
Se procedează ca în demonstrația de la teorema pentru algoritmul `Balansare-Greedy`, se obține $T_i - t_j \leq T^*$ și folosind inegalitatea anterioară obținem $T = T_i \leq \frac{3}{2}T^*$.

□

Bibliografie

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*. Addison Wesley, Massachusetts, 3rd Edition, MIT Press 2009
- Jon Kleinberg, Éva Tardos, *Algorithm Design*, Pearson Education, 2006
- Vijay V. Vazirani, *Approximation Algorithms*, Springer, 2001
- Dorit Hochbaum, *Approximation Algorithms for NP-Hard Problems*, PWS Publishing Company, 1997

Titlul programului: Programul Operațional Sectorial Dezvoltarea Resurselor Umane 2007 - 2013
Titlul proiectului: Tehnici de Analiză, Modelare și Simulare pentru Imagistică, Bioinformatică și Sisteme Complexe (ITEMS)
Editorul materialului: Universitatea POLITEHNICA din București și Universitatea TRANSILVANIA din Brașov
Data publicării: martie 2013

Răzvan Andonie este profesor universitar, conducător de doctorat la Universitatea Transilvania din Brașov. Este absolvent al Facultății de Matematică-Informatică, Universitatea Babeș-Bolyai din Cluj-Napoca. A obținut în 1984 doctoratul în matematică-informatică la Universitatea București, sub conducerea Acad. Solomon Marcus. Este activ în următoarele domenii de cercetare: inteligență computațională, sisteme de învățare, calculul paralel și distribuit, chimie computațională.

Angel Cațaron este cadru didactic titular la Departamentul de Electronică și Calculatoare al Universității Transilvania din Brașov. A obținut titlul de doctor în anul 2004 la Universitatea Politehnica din București. Domeniile sale de interes sunt data mining și inteligență computațională.

Honorius Gâlmeanu este inginer software la Siemens România și cadru didactic asociat la Universitatea Transilvania din Brașov. A obținut titlul de doctor în 2008 la aceeași universitate, subiectul tezei fiind "support vector machines". În cadrul specializării Calculatoare predă cursurile de algoritmi și arhitecturi paralele și algoritmi și calculabilitate. Domeniile sale de interes sunt data mining și bioinformatică, în special alinierea secvențelor.

Mihai Ivanovici este conferențiar universitar la Departamentul de Electronică și Calculatoare al Universității Transilvania din Brașov, titular al cursurilor de bazele prelucrării semnalelor și prelucrarea și analiza imaginilor. În 2006 a obținut titlul de doctor în electronică și telecomunicații la Universitatea Politehnica din București, sub îndrumarea prof. Vasile Buzuloiu. Domeniile sale de interes includ analiza texturilor, segmentarea imaginilor și prelucrarea și analiza imaginilor biomedicale.

Lucian Sasu este doctor in știinta calculatoarelor din 2006 și lector universitar în cadrul Universității Transilvania din Brașov, Facultatea de Matematică și Informatică. Principalele sale domenii de interes sunt algoritmii și structurile de date, inteligența artificială, machine learning și data mining.

ISBN 978-606-19-0206-4