Proiectarea Algoritmilor

Curs 2 – Scheme de algoritmi – Greedy continuare + Programare dinamica

Bibliografie

- Cormen – Introducere în Algoritmi cap. 17
- Giumale – Introducere in Analiza Algoritmilor cap 4.4 ,4.5
- http://www.cs.umass.edu/~barring/cs611/lecture/4.pdf
- http://thor.info.uaic.ro/~dlucanu/cursuri/tpaa/resurse/Curs6.pps
- http://www.math.fau.edu/locke/Greedy.htm
- http://en.wikipedia.org/wiki/Greedoid
- http://activities.tjhsst.edu/sct/lectures/greedy0607.pdf
- http://www.cse.ust.hk/~dekai/271/notes/L12/L12.pdf

Greedy - reminder

- Metodă de rezolvare eficientă a unor probleme de optimizare.
- Soluția trebuie să satisfacă un criteriu de optim global (greu de verificat) → optim local mai ușor de verificat.
- Se aleg soluții parțiale ce sunt îmbunătățite repetat pe baza criteriului de optim local pană ce se obțin soluții finale.
- Soluțiile parțiale ce nu pot fi îmbunătățite sunt abandonate → proces de rezolvare irevocabil (fără reveniri).

Greedy – schemă generală de rezolvare

- Schema generală de rezolvare a unei probleme folosind programarea lacoma:

- Rezolvare_lacoma(Crit_optim, Problema)
  - 1. sol_partiale = sol_initiale(problema); // determinarea soluțiilor parțiale
  - 2. sol_fin = Φ;
  - 3. while (sol_partiale ≠ Φ)
  - 4.for-each(s in sol_partiale)
  - 5. if(s este o solutie a problemei) { // daca e soluție
  - 6. sol_fin = sol_fin U {s}; // finala se salvează
  - 7. sol_partiale = sol_partiale \ {s};
  - 8. } else // se poate optimiza?
  - 9. if(optimizare_posibila(s, Crit_optim, Problema)) // da
  - 10. sol_partiale = sol_partiale \ {s} U optimizare(s,Crit_optim,Problema)
  - 11. else sol_partiale = sol_partiale \ {s}; // nu
  - 12. return sol_fin;

Comparație D&I și Greedy

- Tip abordare
  - D&I: bottom-up;
  - Greedy: top-down.
- Criteriu de optim
  - D&I: nu;
  - Greedy: da.

Alt exemplu (I)

Problema rucsacului

Trebuie să umplem un rucsac de capacitate maximă M kg cu obiecte care au greutatea m_i şi valoarea v_i. Putem alege mai multe obiecte din fiecare tip cu scopul de a maximiza valoarea obiectelor din rucsac.

- Varianta 1: putem alege fracțiuni de obiect – “problema continuă”
- Varianta 2: nu putem alege decât obiecte întregi (număr natural de obiecte din fiecare tip) – ”problema 0-1”

Alt exemplu (II)

- Varianta 1: Algoritm Greedy
  - sortăm obiectele după raportul v_i/m_i
  - adăugăm fracțiuni din obiectul cu cea mai mare valoare per kg până epuizăm stocul şi apoi adăugăm fracțiuni din obiectul cu valoarea următoare
  - Exemplu: M = 10; m_1 = 5 kg, v_1 = 10, m_2 = 8 kg, v_2 = 19, m_3 = 4 kg, v_3 = 4
  - Soluție: (m_2 , v_2) 8kg şi 2kg din (m_1,v_1) – valoarea totală: 19 + 2 * 10 / 5 = 23
- Varianta 2: Algoritmul Greedy nu funcționează => contraexemplu
  - Exemplu: M = 10; m_1 = 5 kg, v_1 = 10, m_2 = 8 kg, v_2 = 19, m_3 = 4 kg, v_3 = 4
  - Rezultat corect – 2 obiecte (m_1,v_1) – valoarea totală: 20
  - Rezultat algoritm Greedy – 1 obiect (m_2,v_2) – valoarea totală: 19

Când funcționează algoritmii Greedy? (I)

- problema are proprietatea de substructură optimă
  - soluția problemei conține soluțiile subproblemelor
- problema are proprietatea alegerii locale
  - alegând soluția optimă local se ajunge la soluția optimă global

Când funcționează algoritmii greedy? (II)

- Fie E o mulțime finită nevidă şi I $\subset \mathcal{P}(E)$ a.i. $\emptyset \in I$, $\forall X \subseteq Y$ şi $Y \in I \Rightarrow X \in I$. Atunci spunem ca (E,I) sistem accesibil.
- Submulțimile din I sunt numite submulțimi "independente":
- Exemple:
  - Ex1: E = $\{e_1, e_2, e_3\}$ si I = $\{\emptyset, \{e_1\}, \{e_2\}, \{e_3\}, \{e_1, e_2\}, \{e_2, e_3\}\}$ – mulțimile ce nu conțin e_1 si e_3.
  - Ex2: E – muchiile unui graf neorientat şi I mulțimea mulțimilor de muchii ce nu conțin un ciclu (mulțimea arborilor).
  - Ex3: E set de vectori dintr-un spațiu vectorial, I mulțimea mulțimilor de vectori linear independenți.
  - Ex4: E – muchiile unui graf neorientat şi I mulțimea mulțimilor de muchii în care oricare 2 muchii nu au un vârf comun.

Când funcționează algoritmii greedy? (III)

- Un sistem accesibil este un matroid daca satisface proprietatea de interschimbare:

  X, Y $\in$ I şi |X| < |Y| => $\exists e \in Y \setminus X$ a.i. $X \cup \{e\} \in I$

- Teorema. Pentru orice subset accesibil (E,I) algoritmul Greedy rezolvă problema de optimizare dacă şi numai dacă (E,I) este matroid.

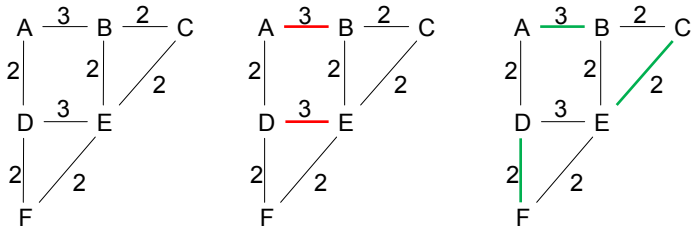
Verificăm exemplele

- Ex1: I = {∅, {e_1}, {e_2}, {e_3}, {e_1, e_2}, {e_2, e_3}}

Fie Y = {{e_1}, {e_2}, {e_3}, {e_1, e_2}} si X = {{e_1}, {e_3}}

→Y \ X = {{e_2}, {e_1, e_2}} → X ∪ {e_2} ∈ I → matroid

- Ex4:



Algoritmul Greedy

- Algoritmul generic Greedy devine:
  - X = ∅
  - *sortează* elementele din E în ordinea *descrescătoare* a ponderii
  - *pentru fiecare* element e ∈ E (sortat) *repetă*
    - X = X ∪ {e} dacă şi numai dacă (X ∪ {e}) ∈ I
  - *Întoarce* X

Greedy – tema de gândire

- Se dă un număr natural n. Să se găsească cel mai mare subset S din {1, 2, ..., n} astfel încât nici un element din S să nu fie divizibil cu nici un alt element din S.
- În plus, dacă există mai multe subseturi maximale ce respectă proprietatea de mai sus, să se determine întotdeauna cel mai mic din punct de vedere lexicografic.
- S este lexicografic mai mic decât T dacă cel mai mic element care este membru din S sau T, dar nu din amândouă, face parte din S.
- Sursa: TopCoder - Indivisible

Programare dinamică

- Programare dinamică
  - Descriere generală
  - Algoritm generic
  - Caracteristici
- Exemplificare: Înmulțirea matricilor
- Exemplificare: Arbori optimi la căutare (AOC)
  - Definiții
  - Construcția AOC

Programare dinamica

- Descriere generală
  - Soluții optime construite iterativ asamblând soluții optime ale unor probleme similare de dimensiuni mai mici.
- Algoritmi "clasici"
  - Algoritmul Floyd-Warshall care determină drumurile de cost minim dintre toate perechile de noduri ale unui graf.
  - AOC
  - Înmulțirea unui şir de matrici
  - Numere catalane
  - Viterbi

Algoritm generic

- Programare dinamică (crit_optim, problema){
  - // fie $problema_0$ $problema_1$ … $problema_n$ astfel incât
  - // $problema_n$= problema; $problema_i$ mai simpla decât $problema_{i+1}$
  - 1. Sol = soluții_initiale(crit_optim, $problema_0$);
  - 2. *Pentru* i = 1 *la* n *repetă* { // construcție soluții pentru $problema_i$
    // folosind soluțiile problemelor precedente
  - 3. Sol_i = calcul_soluții(Sol, crit_optim, $Problema_i$);
    // determin soluția $problemei_i$
  - 4. Sol = Sol U Sol_i;
    // noua soluție se adaugă pentru a fi refolosită pe viitor
  - 5. }
  - 6. s = soluție_pentru_$problema_n$(Sol);
    // selecție / construcție soluție finala
  - 7. Întoarce s;
  - 8. }

Caracteristici

- O soluție optimă a unei probleme conține soluții optime ale subproblemelor.
- Decompozabilitatea recursivă a problemei P in subprobleme similare P = P_n, P_{n-1}, … P_0 care acceptă soluții din ce in ce mai simple.
- Suprapunerea problemelor (soluția unei probleme P_i participă in procesul de construcție a soluțiilor mai multor probleme P_k de talie mai mare k > i) – memoizare (se foloseşte un tablou pentru salvarea soluțiilor subproblemelor pentru a nu le recalcula)
- In general se folosește o abordare bottom-up, de la subprobleme la probleme.

Diferențe Greedy – programare dinamică

Programare lacomă	*Programare dinamică*
Sunt menținute doar soluțiile parțiale curente din care evoluează soluțiile parțiale următoare	Se păstrează toate soluțiile parțiale
Soluțiile parțiale anterioare sunt eliminate	La construcția unei soluții noi poate contribui orice altă soluție parțială generată anterior
Se poate obține o soluție neoptimă. (trebuie demonstrat că se poate aplica)	Se obține soluția optimă dacă soluțiile parțiale sunt **independente** între ele.

Diferențe divide et impera – programare dinamică

Divide et impera

- abordare top-down – problema este descompusă în subprobleme care sunt rezolvate independent

Programare dinamică

- abordare bottom-up - se porneşte de la sub-soluții elementare şi se combină sub-soluțiile mai simple în sub-soluții mai complicate, pe baza criteriului de optim
- se evită calculul repetat al aceleiaşi subprobleme prin memorarea rezultatelor intermediare

Exemplu: Parantezarea matricilor (Chain Matrix Multiplication)

- Se dă o şir de matrice: A_1, A_2, ..., A_n.
- Care este numărul minim de înmulțiri de scalari pentru a calcula produsul:

  $A_1 \times A_2 \times \ldots \times A_n$?

- Să se determine una dintre parantezările care minimizează numărul de înmulțiri de scalari.

Înmulțirea matricilor

- A(p, q) x B (q, r) => pqr înmulțiri de scalari.
- Dar înmulțirea matricilor este asociativă (deşi nu este comutativă).
- A(p, q) x B (q, r) x C(r, s)
  (AB)C => pqr + prs înmulțiri
  A(BC) => qrs + pqs înmulțiri
- Ex: p = 5, q = 4, r = 6, s = 2
  (AB)C => 180 înmulțiri
  A(BC) => 88 înmulțiri
- Concluzie: Parantezarea este foarte importantă!

Soluția banală

- Matrici: $A_1, A_2, ..., A_n$.
- Vector de dimensiuni: $p_0, p_1, p_2, ... , p_n$.
- $A_i(p_{i-1}, p_i) \rightarrow A_1(p_0, p_1), A_2(p_1, p_2), \ldots$
- Dacă folosim căutare exhaustivă şi vrem să construim toate parantezările posibile pentru a determina minimul: $\Omega(4^n / n^{3/2})$.
- Vrem o soluție polinomială folosind P.D.

Descompunere în subprobleme

- Încercăm să definim subprobleme identice cu problema originală, dar de dimensiune mai mică.
- $\forall$ $1 \le i \le j \le n$:
  - Notăm $A_{i,j} = A_i \times \ldots \times A_j$. $A_{i,j}$ are p_{i-1} linii si p_j coloane: $A_{i,j}(p_{i-1}, p_j)$
  - m[i, j] = numărul optim de înmulțiri pentru a rezolva subproblema $A_{i,j}$
  - s[i, j] = poziția primei paranteze pentru subproblema $A_{i,j}$
  - Care e parantezarea optimă pentru $A_{i,j}$?
- Problema inițială: $A_{1,n}$

Combinarea subproblemelor

- Pentru a rezolva $A_{i,j}$
  - trebuie găsit acel indice $i \le k < j$ care asigură parantezarea optimă:

$$A_{i,j} = (A_i \times \ldots \times A_k) \times (A_{k+1} \times \ldots \times A_j)$$

$$A_{i,j} = A_{i,k} \times A_{k+1,j}$$

Alegerea optimală

- Căutăm optimul dintre toate variantele posibile de alegere ($i \le k < j$)
- Pentru aceasta, trebuie însă ca şi subproblemele folosite să aibă soluție optimală (adică $A_{i,\,k}$ şi $A_{k+1,\,j}$)

Substructura optimală

- Dacă ştim că alegerea optimală a soluției pentru problema $A_{i,\,j}$ implică folosirea subproblemelor ($A_{i,\,k}$ şi $A_{k+1,\,j}$) şi soluția pentru $A_{i,\,j}$ este optimală, atunci şi soluțiile subproblemelor $A_{i,\,k}$ şi $A_{k+1,\,j}$ trebuie să fie optimale!
- Demonstrație: Folosind metoda cut-and-paste (metodă standard de demonstrare a substructurii optimale pentru problemele de programare dinamică).
- Observație: Nu toate problemele de optim posedă această proprietate! Ex: drumul maxim dintr-un graf orientat.

Definirea recursivă

- Folosind descompunerea in subprobleme, combinarea subproblemelor, alegerea optimală şi substructura optimală putem să rezolvăm problema prin programare dinamică.
- Următorul pas este să definim recursiv soluția unei subprobleme.
- Vrem să găsim o formulă recursivă pentru m[i, j] şi s[i, j].

Definirea recursivă (II)

$$m[i,j] = \begin{cases} 0 & i = j, \\ \min_{i \le k < j} (m[i,k] + m[k+1,j] + p_{i-1} p_k p_j) & i < j \end{cases}$$

- Cazurile de bază sunt m[i, i]
- Noi vrem să calculăm m[1, n]
- Cum alegem s[i, j] ?
- Bottom-up de la cele mai mici subprobleme la cea inițială

Rezolvare bottom-up

$m[1,2], m[2,3], m[3,4], \ldots, m[n-3,n-2], m[n-2,n-1], m[n-1,n]$

$m[1,3], m[2,4], m[3,5], \ldots, m[n-3,n-1], m[n-2,n]$

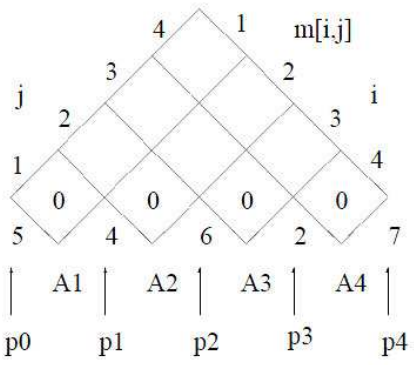
$m[1,4], m[2,5], m[3,6], \ldots, m[n-3,n]$

$\vdots$

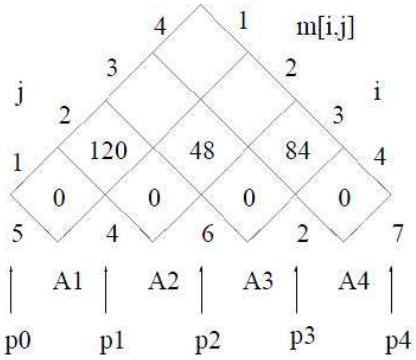
$m[1,n-1], m[2,n]$

$m[1,n]$

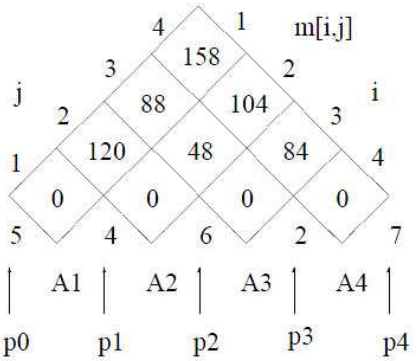
Rezolvare - inițializare



Rezolvare – pas intermediar



Rezolvare – final



Pseudocod

```
Matrix-Chain(p, n)
{  for (i = 1 to n) m[i, i] = 0;
   for (l = 2 to n)
   {
      for (i = 1 to n − l + 1)
      {
         j = i + l − 1;
         m[i, j] = ∞;
         for (k = i to j − 1)
         {
            q = m[i, k] + m[k + 1, j] + p[i − 1] ∗ p[k] ∗ p[j];
            if (q < m[i, j])
            {
               m[i, j] = q;
               s[i, j] = k;
            }
         }
      }
   }
   return m and s;
}
```


Complexitate

- Spațială: $\Theta(n^2)$
  - Pentru memorarea soluțiilor subproblemelor
- Temporală: $O(n^3)$
  - Ns: Număr total de subprobleme: $O(n^2)$
  - Na: Număr total de alegeri la fiecare pas: $O(n)$
  - Complexitatea este de obicei egala cu Ns x Na

Arbori optimi la căutare

- Def 2.1: Fie K o mulțime de chei. Un arbore binar cu cheile K este un graf orientat si aciclic A = (V,E) a.i.:
  - Fiecare nod u ∈ V conține o singură cheie k(u) ∈ K iar cheile din noduri sunt distincte.
  - Există un nod unic r ∈ V a.i. i-grad(r) = 0 si ∀u ≠ r, i-grad(u) = 1.
  - ∀u ∈ V, e-grad(u) ≤ 2; S(u) / D(u) = subarbore stânga / dreapta.
- Def 2.2: Fie K o mulțime de chei peste care exista o relație de ordine ≺. Un arbore binar de căutare satisface:
  - ∀u,v,w ∈ V avem (v ∈ S(u) => cheie(v) ≺ cheie(u)) ∧ (w ∈ D(u) => cheie(u) ≺ cheie(w))

Căutare

- Caută(elem, Arb)
  - dacă Arb = null
    - Întoarce null
  - dacă elem = Arb.val // valoarea din nodul crt.
    - Întoarce Arb
  - dacă elem ≺ Arb.val
    - Întoarce Caută(elem, Arb.st)
  - Întoarce Caută(elem, Arb.dr)
- Complexitate: Θ(logn)

Inserție în arbore de căutare

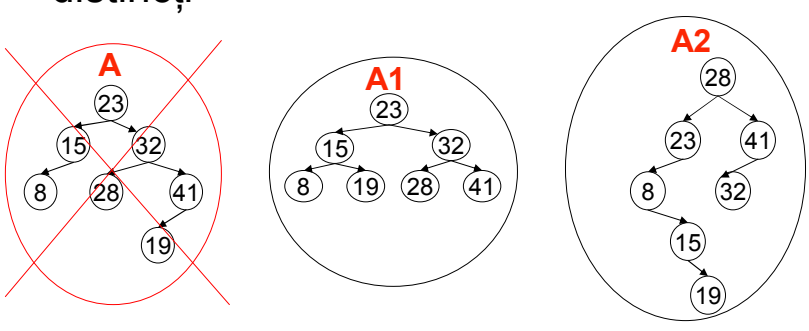
- Inserare(elem, Arb)
  - dacă Arb = vid // adaug cheia in arbore
    - nod_nou(elem, null, null)
  - dacă elem = Arb.val // valoarea există deja
    - Întoarce Arb
  - dacă elem ≺ Arb.val
    - Întoarce Inserare(elem, Arb.st) // adaugă in stânga
    - Întoarce Inserare(elem, Arb.dr) // sau in dreapta
- Complexitate: Θ(logn)

nod Stânga

nod Dreapta

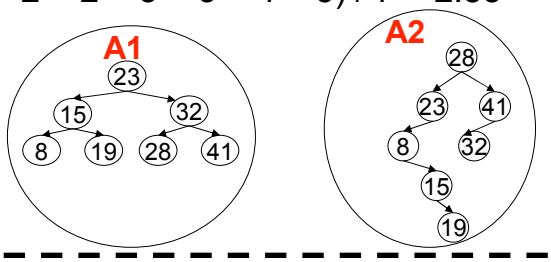
Exemplu de arbori de căutare

- Cu aceleaşi chei se pot construi arbori distincți



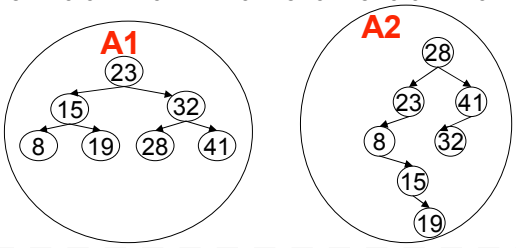
Exemplu (I)

- presupunem că elementele din A1 şi A2 au probabilități de căutare egale:
  - numărul mediu de comparații pentru A1 va fi: (1 + 2 + 2 + 3 + 3 + 3 + 3) / 7 = 2.42
  - numărul mediu de comparații pentru A2 va fi: (1 + 2 + 2 + 3 + 3 + 4 + 5) / 7 = 2.85



Exemplu (II)

- presupunem că elementele au următoarele probabilități:
  - 8: 0.2; 15: 0.01; 19: 0.1; 23: 0.02; 28: 0.25; 32: 0.2; 41: 0.22;
  - numărul mediu de comparații pentru A1:
    - 0.02*1+0.01*2+0.2*2+0.2*3+0.1*4+0.25*3+0.22*3=2.85
  - numărul mediu de comparații pentru A2:
    - 0.25*1+0.02*2+0.22*2+0.2*3+0.2*3+0.01*4+0.1*5=2.47



Probleme

- Costul căutării depinde de frecvența cu care este căutat fiecare termen.
- → Ne dorim ca termenii cei mai des căutați să fie cât mai aproape de vârful arborelui pentru a micşora numărul de apeluri recursive.
- Dacă arborele este construit prin sosirea aleatorie a cheilor putem ajunge la o simplă listă cu n elemente

Definiție AOC

- Definiție: Fie A un arbore binar de căutare cu chei intr-o mulțime K, fie $\{x_1, x_2, \ldots x_n\}$ cheile conținute in A, iar $\{y_0, y_1, \ldots y_n\}$ chei reprezentante ale cheilor din K ce nu sunt in A astfel încât: $y_{i-1} \prec x_i \prec y_i, i = \overline{1,n}$. Fie p_i, i = 1,n probabilitatea de a căuta cheia x_i si q_j, j = 0,n probabilitatea de a căuta o cheie reprezentata de y_j. Vom avea relația: $\sum_{i=1}^{n} p_i + \sum_{j=0}^{n} q_j = 1$. Se numește arbore de căutare probabilistică, un arbore cu costul:

$$Cost(A) = \sum_{i=1}^{n} (nivel(x_i, A) + 1) * p_i + \sum_{j=0}^{n} nivel(y_j, A) * q_j$$

- Definiție: Un arbore de căutare probabilistică având cost minim este un arbore optim la căutare (AOC).

Algoritm AOC naiv

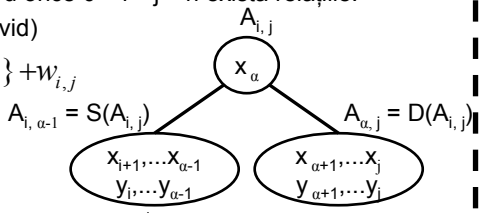
- Generarea permutărilor x_1, ... x_n.
- Construcția arborilor de căutare corespunzători.
- Calcularea costului pentru fiecare arbore.
- Alegerea arborelui de cost minim.
- Complexitate: Θ(n!) (deoarece sunt n! permutări).
- → căutăm altă variantă!!!

Construcția AOC – Notații

- $A_{i,j}$ desemnează un AOC cu cheile $\{x_{i+1}, x_{i+2}, \ldots x_j\}$ in noduri si cu cheile $\{y_i, y_{i+1}, \ldots y_j\}$ in frunzele fictive.
- $C_{i,j}$ = Cost ($A_{i,j}$). $Cost(A_{ij}) = \sum_{k=i+1}^{j} (nivel(x_k, A_{ij}) + 1) * p_k + \sum_{k=i}^{j} nivel(y_k, A_{ij}) * q_k$
- $R_{i,j}$ este indicele α al cheii x_α din radacina arborelui $A_{i,j}$.
- $w_{i,j} = \sum_{k=i+1}^{j} p_k + \sum_{k=i}^{j} q_k$
- Observatie: $A_{0,n}$ este chiar arborele A, $C_{0,n}$ = Cost (A) iar $w_{0,n}$ = 1.

Construcția AOC - Demonstrație

- Lemă (alegere optimală): Pentru orice 0 ≤ i ≤ j ≤ n există relațiile:
  - $C_{i,j}$ = 0 , daca i = j; (arbore vid)
  - $C_{i,j} = \min_{i<\alpha\le j}\{C_{i,\alpha-1} + C_{\alpha,j}\} + w_{i,j}$



- Demonstrație:

$$Cost(A_{ij}) = \sum_{k=i+1}^{j}(nivel(x_k, A_{ij}) + 1) * p_k + \sum_{k=i}^{j} nivel(y_k, A_{ij}) * q_k$$

→ $C_{i,j} = C_{i,\alpha-1} + C_{\alpha,j} + w_{i,j}$

- $C_{i,j}$ depinde de indicele α al nodului rădăcină
- dacă $C_{i,\alpha-1}$ şi $C_{\alpha,j}$ sunt minime (costurile unor AOC) → $C_{i,j}$ este minim

Construcția AOC

- 1. In etapa d, d = 1, 2, … n se calculează costurile si indicele cheilor din rădăcina arborilor AOC $A_{i, i+d}$, i = 0, n-d cu d noduri si d + 1 frunze fictive
- Arborele $A_{i, i+d}$ conține in noduri cheile $\{x_{i+1}, x_{i+2}, \dots x_{i+d}\}$, iar in frunzele fictive sunt cheile $\{y_i, y_{i+1}, \dots y_{i+d}\}$. Calculul este efectuat pe baza rezultatelor obținute in etapele anterioare
- Conform lemei avem $C_{i,i+d} = \min_{i<\alpha\le i+d}\{C_{i,\alpha-1} + C_{\alpha,i+d}\} + w_{i,i+d}$
- rădăcina $A_{i, i+d}$ are indicele $R_{i, j}$ = α care minimizează $C_{i, i+d}$.
- 2. Pentru d = n, $C_{0, n}$ corespunde arborelui AOC $A_{0, n}$ cu cheile $\{x_1, x_2, \dots x_n\}$ in noduri si cheile $\{y_0, y_1, \dots y_n\}$ in frunzele fictive

Algoritm AOC

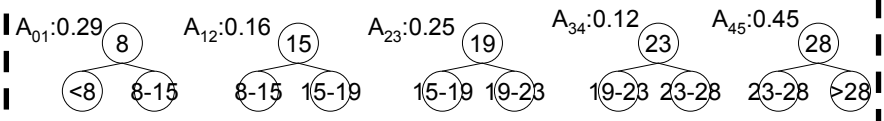
```
AOC(x, p, q, n){
    for( i = 0; i ≤ n ; i++)
        {C_{i,i} = 0, R_{i,i} = 0, w_{i,i} = q_i}        // initializare costuri AOC vid A_{i,i}
    for( d = 1; d ≤ n ; d++){
        for( i = 0; i ≤ n-d ; i++){ // calcul indice radacina si cost pentru A_{i,i+d}
            j = i + d,   C_{i,j} = ∞,   w_{i,j} = w_{i,j-1} + p_j + q_j
            for (α = i + 1;  α ≤ j; α++) {// ciclul critic – operatii intensive
                if (C_{i,α-1} + C_{α,j} < C_{i,j})  // am gasit un cost mai mic?
                    { C_{i,j} = C_{i,α-1} + C_{α,j} ; R_{i,j} = α} // update
            }
            C_{i,j} = C_{i,j} + w_{i,j} // update
        }
    }
    return gen_AOC(C, R, x, 0, n)        // constructie efectiva arbore A_{0,n}
                                         // cunoscand indicii
}
```

Exemplu constructie AOC (I)

- 8: 0.2; 15: 0.01; 19: 0.1; 23: 0.02; 28: 0.25; (58%)
- [0:8): 0.02; (8:15): 0.07; (15:19): 0.08; (19:23): 0.05; (23:28): 0.05; (28,∞): 0.15 (42%)
- $C_{01}=p_1+q_0+q_1=0.29$
- $C_{12}=p_2+q_1+q_2=0.16$
- $C_{23}=p_3+q_2+q_3=0.25$
- $C_{34}=p_4+q_3+q_4=0.02+0.05+0.05=0.12$
- $C_{45}=p_5+q_4+q_5=0.25+0.05+0.15=0.45$

$$w_{i,j} = \sum_{k=i+1}^{j} p_k + \sum_{k=i}^{j} q_k \qquad C_{i,i+d} = \min_{i<\alpha\le i+d}\{C_{i,\alpha-1} + C_{\alpha,i+d}\} + w_{i,i+d}$$

Exemplu constructie AOC (II)

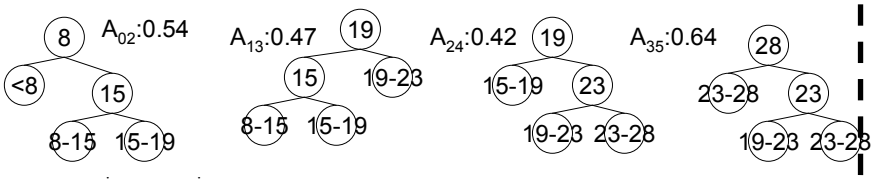


C_{02}= min {(C_{00}+C_{12}),(C_{01}+C_{22})} + w_{02} = min(0.16,0.29) + 0.38 = 0.54 R_{02}= 1 (α=1)

C_{13} = min{C_{23},C_{12}} + w_{13} = min(0.25,0.16) + 0.31 = 0.47 R_{13}= 3

C_{24} = min{C_{34},C_{23}} + w_{24} = min(0.12,0.25) + 0.3 = 0.42 R_{24}= 3

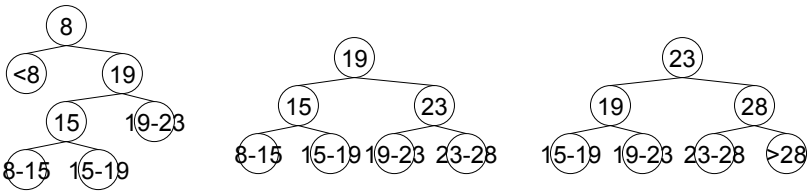
C_{35} = min{C_{45},C_{34}} + w_{35} = min(0.45,0.12) + 0.52 = 0.64 R_{35}= 5



$$w_{i,j} = \sum_{k=i+1}^{j} p_k + \sum_{k=i}^{j} q_k \qquad C_{i,i+d} = \min_{i<\alpha\le i+d}\{C_{i,\alpha-1} + C_{\alpha,i+d}\} + w_{i,i+d}$$

Exemplu constructie AOC (III)

- C_{03} = min(C_{00}+C_{13}, C_{01}+C_{23}, C_{02}+C_{33}) + w_{03} = min (0.47,0.54, 0.54) + w_{03} => R_{03} =1
- C_{14}= min(C_{11}+C_{24},C_{12}+C_{34},C_{13}+C_{44}) + w_{14} = min(0.42,0.28, 0.47) + w_{14} => R_{14}= 3
- C_{25} = min(C_{22}+C_{35},C_{23}+C_{34}, C_{24}+C_{55}) + w_{25} = min(0.64, 0.37, 0.42) + w_{25} => R_{25} = 4



$$w_{i,j} = \sum_{k=i+1}^{j} p_k + \sum_{k=i}^{j} q_k \qquad C_{i,i+d} = \min_{i<\alpha\le i+d}\{C_{i,\alpha-1} + C_{\alpha,i+d}\} + w_{i,i+d}$$

AOC – Corectitudine (I)

- Teoremă: Algoritmul AOC construiește un arbore AOC A cu cheile x = $\{x_1, x_2, \dots x_n\}$ conform probabilităților de căutare p_i, i = 1,n si q_j, j = 0,n.
- Demonstrație: prin inducție după etapa de calcul al costurilor arborilor cu d noduri.
- Caz de baza: d = 0. Costurile $C_{i, i}$ ale arborilor vizi $A_{i, i}$, i = 0, n sunt 0, așa cum sunt inițializate de algoritm.
- Pas de inducție: d ≥ 1. Ip. ind.: pentru orice d' < d, algoritmul AOC calculează costurile $C_{i, i+d'}$ si indicii $R_{i, i+d'}$, ai rădăcinilor unor AOC $A_{i, i+d'}$, i = 0, n-d' cu cheile $\{x_{i+1}, x_{i+2}, \dots x_{i+d'}\}$. Trebuie sa arătam ca valorile $C_{i, i+d}$ si $R_{i, i+d}$ corespund unui AOC $A_{i, i+d}$, i = 0, n-d cu cheile $\{x_{i+1}, x_{i+2}, \dots x_{i+d}\}$.

AOC – Corectitudine (II)

- Pentru d si i fixate, algoritmul calculează:

$$C_{i,i+d} = \min_{i<\alpha\le i+d} \{C_{i,\alpha-1} + C_{\alpha,i+d}\} + w_{i,i+d}$$

- unde costurile $C_{i, \alpha-1}$ si $C_{\alpha, i+d}$ corespund unor arbori cu un număr de noduri d' = α – 1 – i in cazul $C_{i, \alpha-1}$ si d'' = 1 + d – α in cazul $C_{\alpha, i+d}$.
- 0 ≤ d', d'' ≤ d – 1 → aceste valori au fost deja calculate in etapele d', d'' < d si conform ipotezei inductive → sunt costuri si indici ai rădăcinilor unor AOC.
- Conform Lemei (alegerii optimale) anterioare, $C_{i, j}$ este costul unui AOC. Conform algoritmului → rădăcina acestui arbore are indicele r = $R_{i, j}$, iar cheile sunt $\{x_{i+1}, x_{i+2}, \dots x_{r-1}\}$ $\{x_r\}$ $\{x_{r+1}, x_{r+2}, \dots x_j\}$ = $\{x_{i+1}, x_{i+2}, \dots x_j\}$
- Pentru d = n, costul $C_{0, n}$ corespunde unui AOC $A_{0, n}$ cu cheile x si cu rădăcina de indice $R_{0, n}$.

AOC – Concluzii

- Câte subprobleme sunt folosite în soluția optimală intr-un anumit pas?
  - AOC: 2 subprobleme
- Câte variante de ales avem de făcut pentru determinarea alegerii optimale intr-un anumit pas?
  - AOC: j – i + 1 candidați pentru rădăcină
- Informal, complexitatea = Ns * Na (Ns = număr subprobleme; Na = număr alegeri)
  - Complexitate AOC: $O(n^2) * O(n) = O(n^3)$
  - Knuth: Se poate îmbunătăți algoritmul => $O(n^2)$

Întrebări?