Proiectarea Algoritmilor

Curs 3 – Programare dinamică (continuare)

Backtracking şi propagarea restricţiilor

Algoritm generic PD

- Programare dinamică (crit_optim, problema)
  - // fie $problema_0$ $problema_1$ … $problema_n$ astfel încât
  - // $problema_n$= problema; $problema_i$ mai simpla decât $problema_{i+1}$
  - 1. Sol = soluții_inițiale(crit_optim, $problema_0$);
  - 2. **Pentru** *i* **de la** 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. s = soluție_pentru_$problema_n$(Sol); // selecție / construcție soluție finala
  - 6. **Întoarce** s;

Caracteristici PD

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

Alt exemplu: Arbori optimi la căutare (AOC)

- Def 2.1: Fie K o mulțime de chei. Un arbore binar cu cheile K este un graf orientat și aciclic A = (V,E) a.î.:
  - Fiecare nod u $\in$ V conține o singură cheie k(u) $\in$ K iar cheile din noduri sunt distincte.
  - Există un nod unic r $\in$ V a.î. i-grad(r) = 0 și $\forall$u $\neq$ r, i-grad(u) = 1.
  - $\forall$u $\in$ V, e-grad(u) $\leq$ 2; S(u) / D(u) = succesorul stânga / dreapta.

- Def 2.2: Fie K o mulțime de chei peste care există o relație de ordine $\prec$. Un arbore binar de căutare satisface:
  - $\forall$u,v,w $\in$ V avem (v $\in$ S(u) => cheie(v) $\prec$ cheie(u)) $\wedge$ (w $\in$ D(u) => cheie(u) $\prec$ cheie(w))

Căutare într-un arbore de căutare

- Caută(elem, Arb)
  - **Dacă** Arb = null
    - **Întoarce** null
  - **Dacă** elem = Arb.val // valoarea din nodul crt.
    - **Întoarce** Arb
  - **Dacă** elem $\prec$ Arb.val
    - **Întoarce** Caută(elem, Arb.st)
  - **Întoarce** Caută(elem, Arb.dr)

Complexitate: $\Theta(\log n)$

Inserţie într-un arbore de căutare

- Inserare(elem, Arb)
  - **Dacă** Arb = vid // adaug cheia în arbore
    - nod_nou(elem, null, null)

nod Stânga

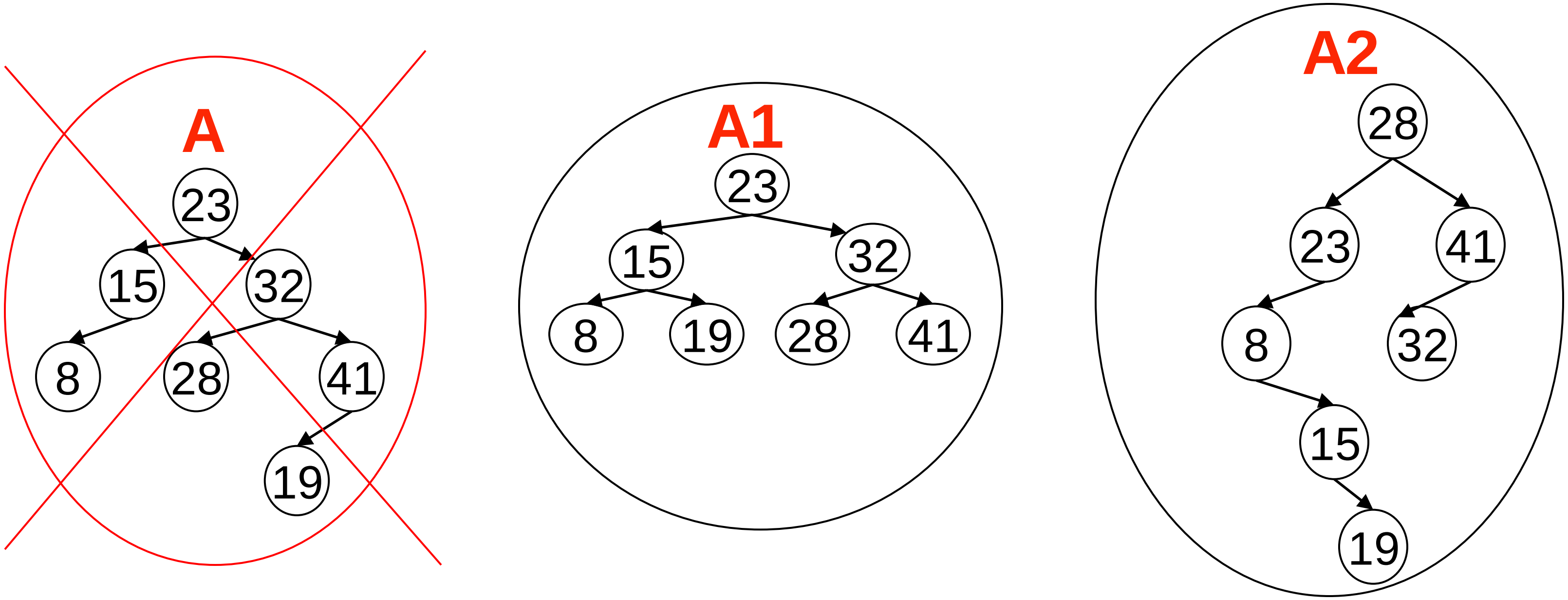
nod Dreapta

  - **Dacă** elem = Arb.val // valoarea există deja
    - **Întoarce** Arb
  - **Dacă** elem $\prec$Arb.val
    - **Întoarce** Inserare(elem, Arb.st) // adaugă în stânga
    - **Întoarce** Inserare(elem, Arb.dr) // sau în dreapta

Complexitate: Θ(logn)

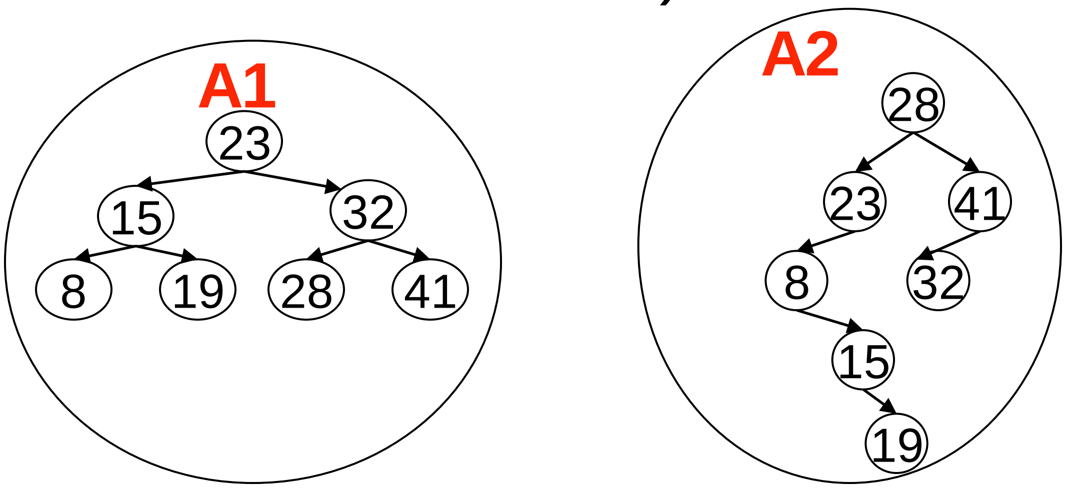
Exemplu de arbori de căutare

- Cu aceleaşi chei se pot construi arbori distincţi



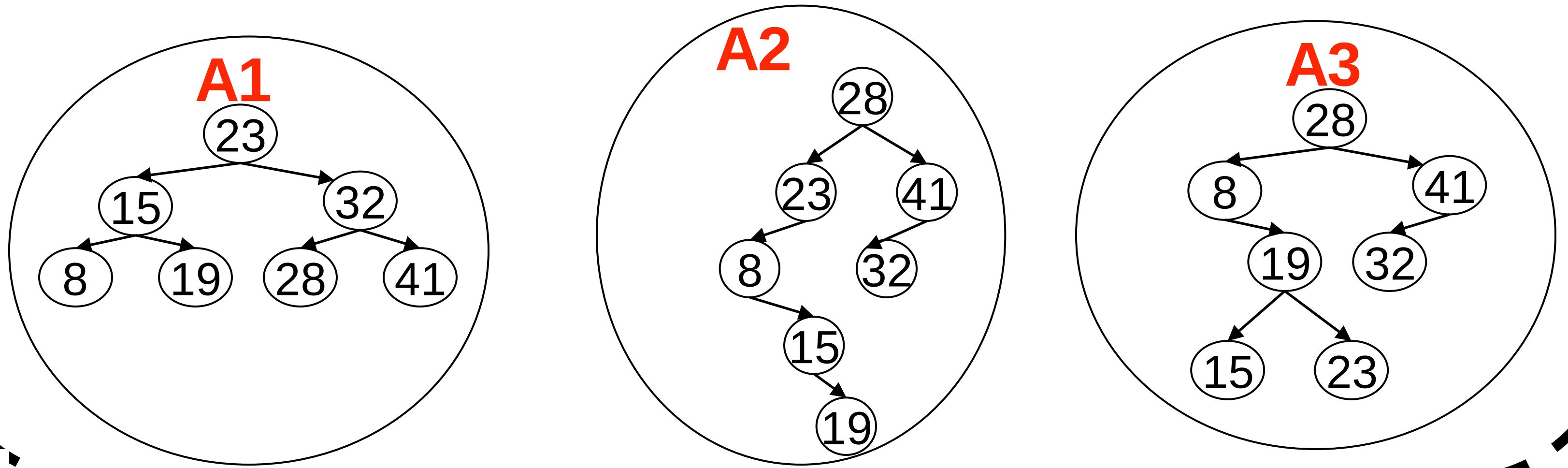
Exemplu (I)

- Presupunem că elementele din A1 şi A2 au probabilităţi de căutare egale:
  - Numărul mediu de comparaţii pentru A1 va fi: (1 + 2 + 2 + 3 + 3 + 3 + 3) / 7 = 2.42
  - Numărul mediu de comparaţii pentru A2 va fi: (1 + 2 + 2 + 3 + 3 + 4 + 5) / 7 = 2.85



Exemplu (II)

- Presupunem că elementele au următoarele probabilităţi:
  - 8: 0.2; 15: 0.01; 19: 0.1; 23: 0.02; 28: 0.25; 32: 0.2; 41: 0.22;
  - Numărul mediu de comparaţii 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 comparaţii 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 frecvenţa cu care este căutat fiecare termen.
- → Ne dorim ca termenii cei mai des căutaţi 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.

Definiţie AOC

- Definiție: Fie A un arbore binar de căutare cu chei într-o mulțime K; fie $\{x_1, x_2, \ldots x_n\}$ cheile conținute în A, iar $\{y_0, y_1, \ldots y_n\}$ chei reprezentante ale cheilor din K ce nu sunt în 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 și q_j, j = 0,n probabilitatea de a căuta o cheie reprezentată 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 $x_1, \ldots x_n$.
- Construcţia arborilor de căutare corespunzători.
- Calcularea costului pentru fiecare arbore.
- Alegerea arborelui de cost minim.
- Complexitate: $\Theta(n!)$ (deoarece sunt n! permutări).
- → căutăm altă variantă!!!

Construcţia AOC – Notaţii

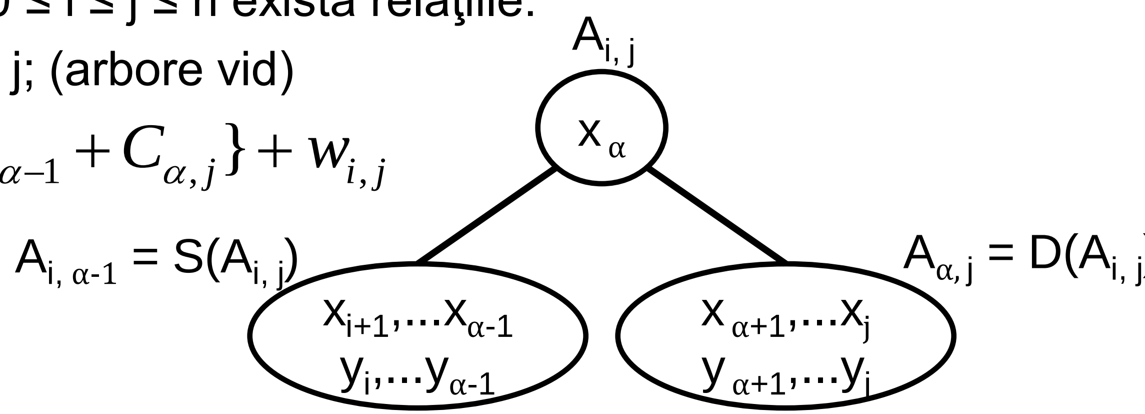
- $A_{i,j}$ desemnează un AOC cu cheile $\{x_{i+1}, x_{i+2}, \dots x_j\}$ în noduri și cu cheile $\{y_i, y_{i+1}, \dots y_j\}$ în frunzele fictive.
- $C_{i,j}$ = Cost ($A_{i,j}$). $Cost(A_{ij}) = \sum_{k=i+1}^{j} (nivel(x_k, A_{ij}) + 1) * p_k + \sum_{k=i}^{j} nivel(y_k, A_{ij}) * q_k$
- $R_{i,j}$ este indicele α al cheii x_α din rădăcina arborelui $A_{i,j}$.

$$w_{i,j} = \sum_{k=i+1}^{j} p_k + \sum_{k=i}^{j} q_k \quad w_{i,j} = \sum_{k=i+1}^{j-1} p_k + p_j + \sum_{k=i}^{j-1} q_k + q_j = w_{i,j-1} + p_j + q_j$$

- Observație: $A_{0,n}$ este chiar arborele A, $C_{0,n}$ = Cost (A) iar $w_{0,n} = 1$.

Construcţia AOC - Demonstraţie

- Lemă: Pentru orice $0 \leq i \leq j \leq n$ există relaţiile:
  - $C_{i,j} = 0$, dacă $i = j$; (arbore vid)
  - $C_{i,j} = \min\limits_{i<\alpha\leq j}\{C_{i,\alpha-1} + C_{\alpha,j}\} + w_{i,j}$



- Demonstraţie:

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

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

- $C_{i,j}$ depinde de indicele α al nodului rădăcină.

- dacă $C_{i,\alpha-1}$ şi $C_{\alpha,j}$ sunt minime (costurile unor AOC) $\rightarrow$ $C_{i,j}$ este minim.

Construcţia AOC

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

Algoritm AOC

- AOC(x, p, q, n)
- **Pentru** *i* **de la** 0 **la** *n*
- $\{C_{i,i} = 0, R_{i,i} = 0, w_{i,i} = q_i\}$ // inițializare costuri AOC vid $A_{i,i}$
- **Pentru** *d* **de la** 1 **la** *n*
- **Pentru** *i* **de la** 0 **la** *n-d* // calcul indice rădăcină și cost pentru $A_{i,i+d}$
- $j = i + d,\ C_{i,j} = \infty,\ w_{i,j} = w_{i,j-1} + p_j + q_j$
- **Pentru** *α* **de la** *i + 1* **la** *j* // ciclul critic – operații intensive
- **Dacă** $(C_{i,\alpha-1} + C_{\alpha,j} < C_{i,j})$ // cost mai mic?
- $\{C_{i,j} = C_{i,\alpha-1} + C_{\alpha,j};\ R_{i,j} = \alpha\}$ // update
- $C_{i,j} = C_{i,j} + w_{i,j}$ // update
- **Întoarce** gen_AOC(C, R, x, 0, n) // construcție efectivă arbore $A_{0,n}$
- // cunoscând indicii

Complexitate???

Exemplu constructie AOC (I)

- 8: 0.2; 15: 0.01; 19: 0.1; 23: 0.02; 28: 0.25; (58%)

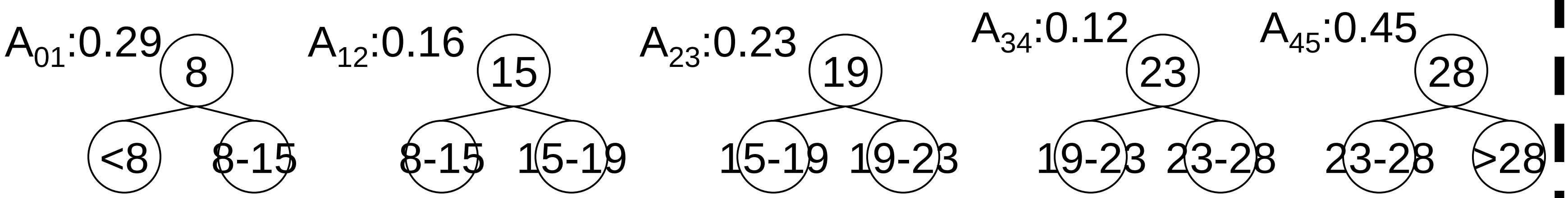
- [0:8): 0.02; (8:15): 0.07; (15:19): 0.08; (19:23): 0.05; (23:28): 0.05; (28,∞): 0.15 (42%)

- $C_{01}=p_1+q_0+q_1=0.2+0.02+0.07=0.29$
- $C_{12}=p_2+q_1+q_2=0.01+0.07+0.08=0.16$
- $C_{23}=p_3+q_2+q_3=0.1+0.08+0.05=0.23$
- $C_{34}=p_4+q_3+q_4=0.02+0.05+0.05=0.12$
- $C_{45}=p_5+q_4+q_5=0.25+0.05+0.15=0.45$

$$w_{i,j} = \sum_{k=i+1}^{j} p_k + \sum_{k=i}^{j} q_k$$

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

Exemplu constructie AOC (II)

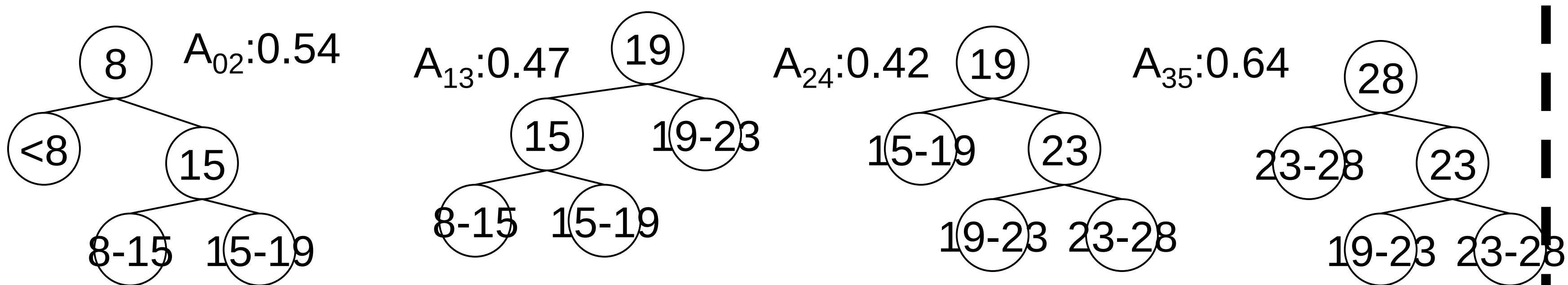


C_{02}= min {(C_{00}+C_{12}),(C_{01}+C_{22})} + w_{02} = min(0.16,0.29) + 0.38 = 0.54 R_{02}= 1 (α=1)

C_{13} = min{C_{11}+C_{23},C_{12}+C_{33}} + w_{13} = min(0.23,0.16) + 0.31 = 0.47 R_{13}= 3 (α=3)

C_{24} = min{C_{34},C_{23}} + w_{24} = min(0.12,0.23) + 0.3 = 0.42 R_{24}= 3

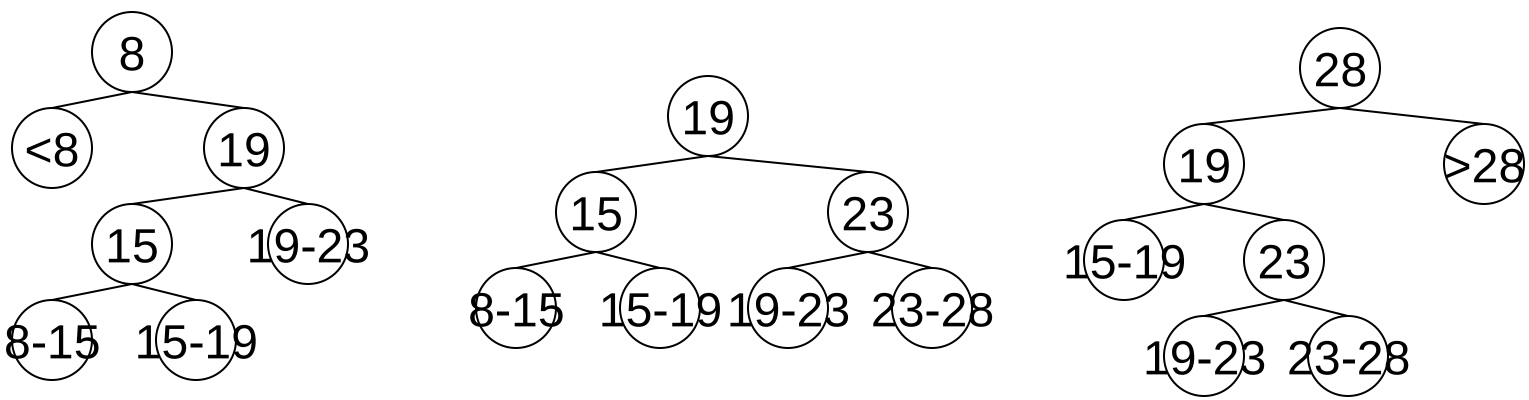
C_{35} = min{C_{45},C_{34}} + w_{35} = min(0.45,0.12) + 0.52 = 0.64 R_{35}= 5



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

Exemplu constructie AOC (III)

- $C_{03} = \min(C_{00}+C_{13}, C_{01}+C_{23}, C_{02}+C_{33}) + w_{03} = \min(0.47, 0.52, 0.54) + w_{03}$ => $R_{03} = 1$
- $C_{14} = \min(C_{11}+C_{24}, C_{12}+C_{34}, C_{13}+C_{44}) + w_{14} = \min(0.42, 0.28, 0.47) + w_{14}$ => $R_{14} = 3$
- $C_{25} = \min(C_{22}+C_{35}, C_{23}+C_{45}, C_{24}+C_{55}) + w_{25} = \min(0.64, 0.67, 0.42) + w_{25}$ => $R_{25} = 5$



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

AOC – Corectitudine (I)

- Teoremă: Algoritmul AOC construiește un arbore AOC A cu cheile $x = \{x_1, x_2, \ldots x_n\}$ conform probabilităților de căutare p_i, $i = 1,n$ și q_j, $j = 0,n$.

- Demonstrație: prin inducție după etapa de calcul a costurilor arborilor cu d noduri.

- Caz de bază: 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: $d \geq 1$. Ip. ind.: pentru orice d' < d, algoritmul AOC calculează costurile $C_{i,\,i+d'}$ și indicii $R_{i,\,i+d'}$, ai rădăcinilor unor AOC $A_{i,\,i+d'}$ cu i = 0, n-d' cu cheile $\{x_{i+1}, x_{i+2}, \ldots x_{i+d'}\}$. Trebuie să arătam că valorile $C_{i,\,i+d}$ și $R_{i,\,i+d}$ corespund unor AOC $A_{i,\,i+d}$ cu i = 0, n-d cu cheile $\{x_{i+1}, x_{i+2}, \ldots x_{i+d}\}$.

AOC – Corectitudine (II)

- Pentru d și 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 $C_{i,\,\alpha-1}$ și $C_{\alpha,\,i+d}$ corespund unor arbori cu un număr de noduri d' = α – 1 – i în cazul $C_{i,\,\alpha-1}$ și d' = i + d – α în cazul $C_{\alpha,\,i+d}$.

- 0 ≤ d' ≤ d – 1 → aceste valori au fost deja calculate în etapele d' < d și conform ipotezei inductive → sunt costuri și indici ai rădăcinilor unor AOC.

- Conform Lemei 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 și cu rădăcina de indice $R_{0,\,n}$.

AOC – Concluzii

- Câte subprobleme sunt folosite în soluţia optimală într-un anumit pas?

  AOC: pentru fiecare din cei n – d + 1 arbori sunt folosite 2 subprobleme $C_{i, \alpha-1}$ și $C_{\alpha, i+d}$

- Câte variante de ales avem de făcut pentru determinarea alegerii optimale într-un anumit pas?

  AOC: pentru fiecare din cei n – d + 1 arbori avem j – i candidaţi pentru rădăcină

- Informal, complexitatea = Ns * Na (Ns = număr subprobleme; Na = număr alegeri)

  Complexitate AOC: $O(n) * O(n^2) = O(n^3)$

  Optimizare – Knuth: $O(n^2)$

Proiectarea Algoritmilor

Curs 3 – Backtracking şi propagarea restricţiilor

Bibliografie

http://ktiml.mff.cuni.cz/~bartak/constraints/intro.html

Problema

	2		8	1		7	4	
7					3	1		
	9				2	8		5
		9		4			8	7
4			2		8			3
1	6			3		2		
3		2	7				6	
		5	6					8
	7	6		5	1		9	

SUDOKU

- Joc foarte la modă cu reguli foarte simple.
- Fiecare rând, coloană sau regiune nu trebuie să conţină decât o dată cifrele de la unu la nouă (Wikipedia).
- Prin trecerea în revistă a soluţiilor posibile pentru acest joc vom explora tehnicile de rezolvare backtracking şi propagarea restricţiilor.

Soluţia 1 – generează şi testează

- Generăm toate soluţiile posibile şi le testăm.
- 43 spaţii de completat, 9 posibilităţi de completare pentru fiecare căsuţă => 9^{43} soluţii de testat.

1-9	2	1- 9	8	1	1-9	7	4	1-9
7	1-9	1-9	1-9	1-9	3	1	1-9	1-9
1-9	9	1-9	1-9	1-9	2	8	1-9	5
1-9	1-9	9	1-9	4	1-9	1-9	8	7
4	1-9	1-9	2	1-9	8	1-9	1-9	3
1	6	1-9	1-9	3	1-9	2	1-9	1-9
3	1-9	2	7	1-9	1-9	1-9	6	1-9
1-9	1-9	5	6	1-9	1-9	1-9	1-9	8
1-9	7	6	1-9	5	1	1-9	9	1-9

Soluţia 2 – Backtracking cronologic (orb) (I)

- Construieşte soluţiile iterativ.
- Menţine evidenţa alegerilor făcute.
- În momentul în care se ajunge la o contradicţie se revine la ultima decizie luată şi se încearcă alegerea unei alte variante.

Soluţia 2 – Backtracking cronologic (orb) (II)

~~1~~ ~~2~~ ~~3~~ ~~4~~ 5	2		8	1		7	4	
7					3	1		
	9				2	8		5
		9		4			8	7
4			2		8			3
1	6			3		2		
3		2	7				6	
		5	6					8
	7	6		5	1		9	

5	2	~~1~~ ~~2~~ 3	8	1	~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~ 6	7	4	... 9
7	4	8	5	9	3	1	2	6
6	9	1	4	7	2	8	3	5
2	3	9	1	4	5	6	8	7
4	5	7	2	6	8	9	1	3
1	6	∅		3		2		
3		2	7				6	
		5	6					8
	7	6		5	1		9	

Nu se găseşte nici o soluţie pe această cale deci trebuie să revenim.

Soluţia 2 – Backtracking cronologic (orb) (III)

Se încearcă schimbarea ultimei alegeri (în acest caz nu se poate deci revenim iar la alegerea precedentă).

5	2	~~1~~ ~~2~~ 3	8	1	~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~ 6	7	4	... 9
7	4	8	5	9	3	1	2	6
6	9	1	4	7	2	8	3	5
2	3	9	1	4	5	6	8	7
4	5	7	2	6	8	9	∅	3
1	6	∅		3		2		
3		2	7				6	
		5	6					8
	7	6		5	1		9	

Găsim o nouă soluţie posibilă şi reluăm avansul.

5	2	~~1~~ ~~2~~ 3	8	1	~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~ 6	7	4	... 9
7	4	8	5	9	3	1	2	6
6	9	1	4	7	2	8	3	5
2	~~3~~ 5	9	~~1~~	4	~~5~~	~~6~~	8	7
4	~~5~~	~~7~~	2	~~6~~	8	~~9~~	∅	3
1	6	∅		3		2		
3		2	7				6	
		5	6					8
	7	6		5	1		9	

Soluţia 2 – Backtracking cronologic (orb) (IV)

Schema Backtracking

- Soluție-parţială ← INIT // inițializez
- EȘEC-DEFINITIV ← fals // nu am ajuns (încă) la eșec
- **Cât timp** Soluție-parţială nu este soluţie finală **şi nu avem** EȘEC-DEFINITIV
  - Soluție-parţială ← AVANS (Soluție-parțială) // avansez
  - **Dacă** EȘEC (Soluție-parțială) // nu mai pot avansa
    - **Atunci** REVENIRE (Soluție-parțială) // mă întorc
- **Dacă** EȘEC-DEFINITIV
  - **Atunci** Întoarce EȘEC // nu s-a găsit nicio soluție
  - **Altfel** Întoarce SUCCES // am ajuns la soluția problemei
- **Sfârșit.**

Procedura AVANS (Soluție-parțială)

- **Dacă** există alternativă de extindere // pot avansa?
  - **Atunci** Soluție-parțială ← Soluție-parțială ∪ alternativă de extindere // avansez
  - **Altfel Dacă** Soluție-parțială este INIT
    - **Atunci** EȘEC-DEFINITIV ← adevărat // nu s-au găsit soluții pentru problemă
    - **Altfel** EȘEC (Soluție-parțială) // ramura curentă a dus la eșec

Backtracking – optimizări posibile (I)

- Alegerea variabilelor în altă ordine.
- Îmbunătăţirea revenirilor.
  - Necesită detectarea cauzei producerii erorii.
- Evitarea redundanţelor în spaţiul de căutare (îmbunătăţirea avansului).
  - Evitarea repetării unei căutări care ştim că va duce la un rezultat greşit.

Backtracking – optimizări posibile (II)

Îmbunătăţirea revenirilor

Revenire la alegerea variabilei care a cauzat eşecul (8 nu poate fi pus decât la poziţia indicată).

5	2	~~1~~ ~~2~~ 3	8	1	~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~ 6	7	4	... 9
7	4	8	~~5~~	~~9~~	3	1	~~2~~	~~6~~
~~6~~	9	~~1~~	~~4~~	~~7~~	2	8	~~3~~	5
~~2~~	~~3~~	9	~~1~~	4	~~5~~	~~6~~	8	7
4	~~5~~	~~7~~	2	~~6~~	8	~~9~~	~~1~~	3
1	6	∅		3		2		
3		2	7				6	
		5	6					8
	7	6		5	1		9	

Backtracking – optimizări posibile (III)

Evitarea redundanţelor în spaţiul de căutare

Alegerea lui 8 pe această poziţie va produce un eşec în viitor indiferent de celelalte alegeri făcute deci în cazul revenirii în această poziţie nu are sens să mai facem această alegere.

5	2	~~1~~ ~~2~~ 3	8	1	~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~ 6	7	4	... 9
7	4	~~8~~	~~5~~	~~9~~	3	1	~~2~~	~~6~~
~~6~~	9	~~1~~	~~4~~	~~7~~	2	8	~~3~~	5
~~2~~	~~3~~	9	~~1~~	4	~~5~~	~~6~~	8	7
4	~~5~~	~~7~~	2	~~6~~	8	~~9~~	~~1~~	3
1	6	∅		3		2		
3		2	7				6	
		5	6					8
	7	6		5	1		9	

Restricţii, reţele de restricţii, probleme de prelucrarea restricţiilor

- Definiție: O restricţie c este o relaţie între una sau mai multe variabile $v_1, \ldots, v_m$, (denumite nodurile sau celulele restricţiei). Fiecare variabila v_i poate lua valori într-o anumita mulţime D_i, denumită domeniul ei (ce poate fi finit sau nu, numeric sau nu).

- Definiție: Se spune că un tuplu (o atribuire) de valori $(x_1, \ldots, x_m)$ din domeniile corespunzătoare celor m variabile satisface restricţia $c(v_1, \ldots, v_m)$, dacă $(x_1, \ldots, x_m) \in c(v_1, \ldots, v_m)$.

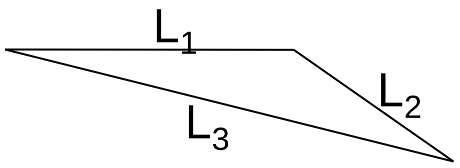
Exprimarea restricţiilor

- Enumerarea tuplelor restricţiei.
  - **(5,2,3,8,1,6,7,4,9); (6,2,3,8,1,5,7,4,9); etc.**
- Formule matematice, cum ar fi ecuaţiile sau inecuaţiile.
  - $0 < V_{1j} < 10$; $V_{1j} \neq V_{1k}$; $\forall\, j \neq k,\ 0 < j, k < 10$
- Precizarea unei mulţimi de reguli.

	2		8	1		7	4	
7					3	1		
	9				2	8		5
		9		4			8	7
4			2		8			3
1	6			3		2		
3		2	7				6	
		5	6					8
	7	6		5	1		9	

Tipuri de restricţii

- Unare
  - Specificarea domeniului variabilei.
  - $V_{11} \in \{1, 2, 3, 4, 5, 6, 7, 8, 9\} \setminus \{2, 8, 1, 7, 4, 3, 9\}$
- Binare
  - Între 2 variabile.
  - $V_{1j} \neq V_{1k}\ \forall j \neq k,\ 0 < j, k < 10$
- N-are
  - Între n variabile.
  - Regula triunghiului: $L_1 + L_2 > L_3$.



Probleme de satisfacerea restricţiilor

- Definiție: O problemă de satisfacere a restricţiilor (PSR) este un triplet <V,D,C>, format din:
  - o mulţime V formată din n variabile $V = \{v_1, \ldots, v_n\}$;
  - mulţimea D a domeniilor de valori corespunzătoare acestor variabile: $D = \{D_1, \ldots D_n\}$;
  - o mulţime C de restricţii $C = \{c_1, \ldots, c_p\}$ între submulţimi ale V ($c_i (v_{i1}, \ldots, v_{ij}) \subseteq D_{i1} \times D_{i2} \times \ldots \times D_{ij}$).

- Conform Definiției tuplului, o restricţie $c_i (v_{i1}, \ldots, v_{ij})$, este o submulţime a produsului cartezian $D_{i1} \times D_{i2} \times \ldots \times D_{ij}$, constând din toate tuplele de valori considerate că satisfac restricţia pentru $(v_{i1}, \ldots, v_{ij})$.

Soluții ale PSR

- Definiție: O soluţie a unei PSR <V,D,C> este un tuplu de valori $<x_1, \ldots , x_n>$ pentru toate variabilele V, din domeniile corespunzătoare D, astfel încât toate restricţiile din C să fie satisfăcute.

- Definiție: PSR binară este o PSR ce conţine doar restricţii unare şi binare.

Probleme de satisfacere a restricţiilor - Reprezentare

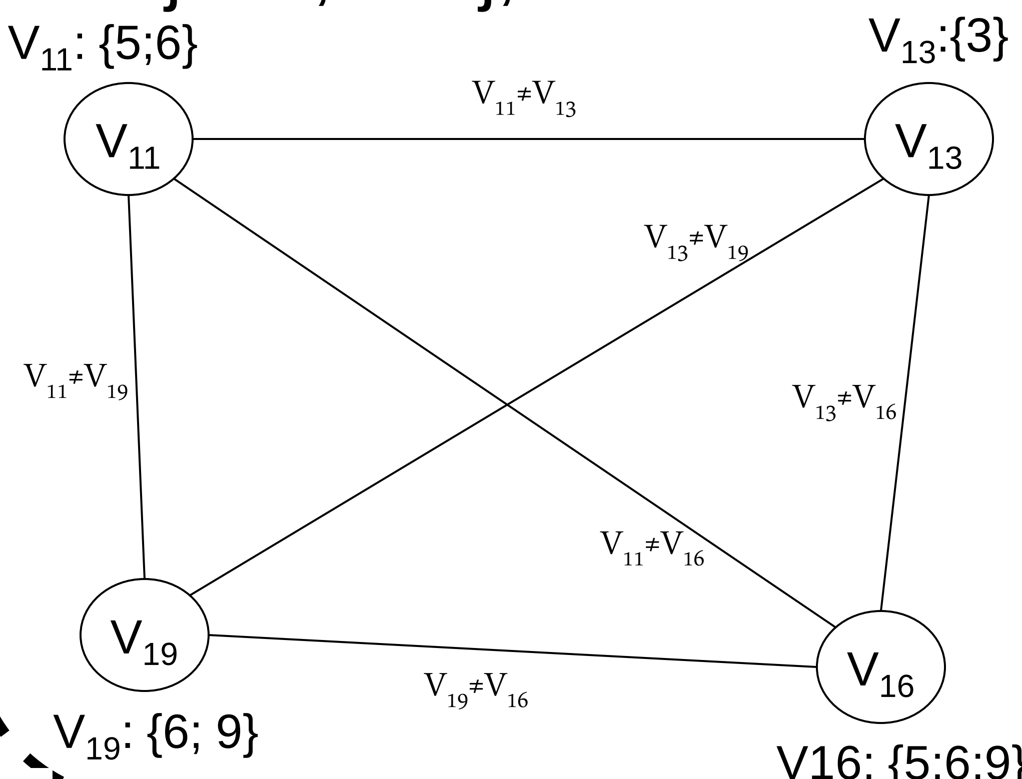
- Reprezentare PSR prin reţele de restricţii.

- Reprezentarea folosită: graf
  - Nodurile: variabilele restricţiei
  - Arcele: restricţiile problemei
  - Valabilă doar pentru PSR binare

- Altă reprezentare posibilă: graf
  - Nodurile: restricţiile problemei
  - Arcele: variabilele restricţiei
  - Valabilă pentru orice tip de PSR

Exemplu de reprezentare a PSR

- $0 < V_{1j} < 10;\ V_{1j} \neq V_{1k};$
  $\forall j \neq k,\ 0 < j, k < 10$



5;6	2	3	8	1	5;6;9	7	4	6;9
7					3	1		
	9				2	8		5
		9		4			8	7
4			2		8			3
1	6			3		2		
3		2	7				6	
		5	6					8
	7	6		5	1		9	

Algoritmi de rezolvare a PSR – Propagarea restricțiilor

- Caracteristici
  - Rezolvă PSR binare;
  - Variabilele au domenii finite de valori;
  - Prin propagarea restricţiilor se filtrează mulţimile de valori (se elimină elementele din domeniu conform unui criteriu dat);
  - Procesul de propagare se opreşte când:
    - O mulţime de valori este vidă → EȘEC;
    - Nu se mai modifică domeniul vreunei mulţimi.

Algoritmi de rezolvare a PSR – Propagarea restricțiilor

Notaţii:

- n = număr variabile = număr restricţii unare;
- r = număr restricţii binare;
- G = reţeaua de restricţii cu variabile drept noduri și restricţii drept arce;
- D_i = domeniul variabilei i;
- Q_i = predicat care verifică restricţia unară pe variabila i;
- P_{ij} = predicatul care reprezintă restricţia binară pe variabilele i și j (O muchie între i și j se înlocuiește cu arcele orientate de la i la j și de la j la i);
- a = max $|D_i|$.

NC-1 (Node Consistency -1)

- Algoritm de consistenţa nodurilor (pentru restricţii unare).
- Procedura NC(i) este:
  - **Pentru fiecare** x $\in D_i$
    - **Dacă** not Q_i (x) // nu este satisfăcută restricția unară
      - **Atunci** șterge x din D_i
  - **Sfârșit**.
- Algoritm NC-1 este:
  - **Pentru** i **de la** 1 **la** *n* **Execută** NC(i) // pentru fiecare var
  - **Sfârșit**.

Complexitate NC-1? na

NC-1: Exemplu (I)

- Elimină din domeniul de valori al fiecărui nod valorile care nu satisfac restricţiile care au ca argument variabila din nodul respectiv.
- În cazul Sudoku variabilele iniţial iau valori între 1-9.
- Algoritmul NC-1 elimină pentru fiecare variabilă acele valori din domeniu care nu sunt consistente cu valorile fixe (celulele deja fixate).

NC-1: Exemplu (II)

5;6	2	3;	8	1	5;6; 9	7	4	6;9;
7	4;5; 8;	4;8;			3	1		
6;	9	1;3; 4;			2	8		5
		9		4			8	7
4			2		8			3
1	6			3		2		
3		2	7				6	
		5	6					8
	7	6		5	1		9	

Algoritmi de consistenţă a arcelor

- Algoritmii de consistenţă a arcelor înlătură toate inconsistenţele submulţimilor de 2 elemente ale reţelei de restricţii.

- Funcţia REVISE ((i,j)) este:
  - ȘTERS ← fals // nu am modificat domeniul de valori
  - **Pentru fiecare** $x \in D_i$
    - **Dacă** nu există $y \in D_j$ a.î. $P_{ij}(x,y)$ // nu se respectă restricția
      - **Atunci**
        - Șterge x din D_i;
        - ȘTERS ← adevărat; // am făcut modificări
  - **Întoarce** ȘTERS.

Exemplu funcţionare REVISE

x

y

5;~~6~~	2	3;	8	1	5;6;9	7	4	6;9;
7	4;5;8;	4;8;			3	1		
6;	9	1;~~3~~;4;			2	8		5
5;6;2;		9		4			8	7
4			2		8			3
1	6			3		2		
3		2	7				6	
9;		5	6					8
8;	7	6		5	1		9	

Complexitate Revise ? a^2

REVISE - Concluzie

- Funcţia Revise este apelată pentru un arc al grafului de restricţii (binare) și șterge acele valori din domeniul de definiţie al unei variabile pentru care nu este satisfăcută restricţia pentru nici o valoare corespunzătoare celeilalte variabile a restricţiei.

- Complexitate Revise: $O(a^2)$

AC-1 (Arc Consistency -1)

- Algoritm AC-1 este:
  - NC-1; // reduc domeniul de valori
  - Q ← {(i,j) | (i,j) ∈ arce(G), i ≠ j} // adaug restricțiile
  - **Repetă**
    - SCHIMBAT ← fals // nu am modificat niciun domeniu
    - **Pentru fiecare** (i,j) ∈ Q // pentru fiecare restricție
      - SCHIMBAT ← (REVISE ((i,j)) sau SCHIMBAT)
  - **Până când *non*** SCHIMBAT // nu am mai făcut // modificări
  - **Sfârșit.**

AC-1 Caracteristici & Complexitate

- Se aplică algoritmul de consistenţa nodurilor şi apoi se aplică REVISE până nu se mai realizează nici o schimbare.
- Complexitate: $O(na * 2r * a^2)$

La fiecare iteraţie eliminăm o singură valoare (şi avem maxim na valori posibile).

Numărul maxim de apelări al Revise.

Complexitate Revise.

Exemplu AC-1

5,~~6~~	2	3;	8	1	5;6;9	7	4	6;9;
7	4;5;8;	4;8;			3	1		
6;	9	1;~~3~~;4;			2	8		5
~~5~~;~~6~~;2;	3;~~5~~	9		4			8	7
4	5	7	2		8			3
1	6	8		3		2		
3		2	7				6	
9;		5	6					8
8;	7	6		5	1		9	

AC-3 (Arc Consistency -3)

- Algoritm AC-3 este:
  - NC-1; // reduc domeniul de valori
  - Q $\leftarrow$ {(i,j) | (i,j) $\in$ arce(G), i $\neq$ j} // adaug restricțiile
  - **Cât timp** Q nevid
    - Selectează și șterge un arc (k,m) din Q;
    - **Dacă** REVISE ((k,m)) // am modificat domeniul
      - **Atunci** Q $\leftarrow$ Q $\cup$ {(i,k) | (i,k) $\in$ arce(G), i $\neq$ k, i $\neq$ m}

        // verific dacă nu se modifică și alte domenii

AC-3 Caracteristici

- Se elimină pe rând arcele (constrângerile).
- Dacă o constrângere aduce modificări în reţea adăugăm pentru reverificare nodurile care punctează către nodul de plecare al restricţiei verificate.
  - Scopul: Reverificarea nodurilor direct implicate de o constrângere din reţea.
- Avantaj: Se fac mult mai puţine apeluri ale funcţiei REVISE.
- Complexitate: $O(a^3r)$.

Backtracking + Propagarea restricţiilor

- În general, propagarea restricţiilor nu poate rezolva complet problema dată.
- Metoda ajută la limitarea spaţiului de căutare (foarte importantă în condiţiile în care backtracking-ul are complexitate exponenţială!).
- În cazul în care propagarea restricţiilor nu rezolvă problema se foloseşte:
  - Backtracking pentru a genera soluţii parţiale;
  - Propagarea restricţiilor după fiecare pas de backtracking pentru a limita spaţiul de căutare (şi eventual găsi că soluţia nu este validă).

ÎNTREBĂRI?