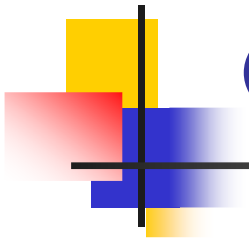
Inteligenta Artificiala

Universitatea Politehnica Bucuresti
Anul universitar 2010-2011

Adina Magda Florea
http://turing.cs.pub.ro/ia_10 si
curs.cs.pub.ro



Curs nr. 3

Strategii de rezolvare a problemelor

- Problema satisfacerii restrictiilor
- Strategii de cautare locala

1. Problema satisfacerii restrictiilor

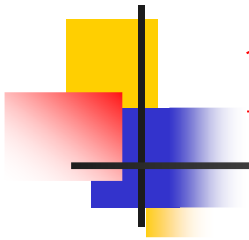
$\{X_1,...,X_n\}$ $\quad R_j \subset D_{i1} \times ... \times D_{ij}$ $\quad X_{i1},...,X_{ij}$

$\{D_1,...,D_n\}$

$\{R_1,...,R_k\}$

$\{(X_1, x_{j_1}),...,(X_n, x_{j_n})\}$

- Gradul unei variabile
- Aritatea unei restrictii
- Gradul problemei
- Aritatea problemei



1.1 Instante ale CSP

- Determinarea unei solutii sau a tuturor solutiilor
- CSP totala
- CSP partiala
- CSP binara – graf de restrictii

CSP – problema de cautare, in NP

- Sub-clase de probleme cu timp polinomial
- Reducerea timpului (reducerea sp. de cautare)

Algoritm: **Backtracking nerecursiv**

1. Initializeaza FRONTiERA cu $\{S_i\}$ /* S_i este starea initiala */
2. **daca** FRONTIERA = { }
   atunci intoarce INSUCCES /* nu exista solutie /*
3. Fie S prima stare din FRONTIERA
4. **daca** toate starile succesoare ale lui S au fost deja generate
   atunci
   4.1. Elimina S din FRONTIERA
   4.2. **repeta de la** 2
5. **altfel**
   5.1. Genereaza S', noua stare succesoare a lui S
   5.2. Introduce S' la începutul listei FRONTIERA
   5.3. Stabileste legatura S'→ S
   5.4. Marcheaza în S faptul ca starea succesoare S' a fost generata
   5.5. **daca** S' este stare finala
      atunci
      5.5.1. Afiseaza calea spre solutie urmarind legaturile S'→ S ..
      5.5.2. **întoarce** SUCCES /* s-a gasit o solutie */
   5.6. **repeta de la** 2

sfarsit.

1.2 Notatii

- X_1, …, X_N variabilele problemei, N fiind numarul de variabile ale problemei,
- U - intreg care reprezinta indicele variabilei curent selectate pentru a i se atribui o valoare
- F - vector indexat dupa indicii variabilelor, in care sunt memorate selectiile de valori facute de la prima variabila si pana la variabila curenta

$$\text{Relatie}(U_1, F[U_1], U_2, F[U_2]) = \begin{cases} \text{adevarat} & \text{daca exista restrictia } R_{U_1 U_2}(F[U_1], F[U_2]) \\ & R_{U_1 U_2} \subset D_{U_1} \times D_{U_2} \\ \text{fals} & \text{in caz contrar} \end{cases}$$

Algoritm: **Backtracking recursiv**

BKT (U, F)

pentru fiecare valoare V a lui X_U **executa**

1. F[U] ← V
2. **daca** Verifica (U,F) = adevarat
   atunci
   - 2.1. **daca** U < N
     atunci BKT(U+1, F)
   - 2.2. **altfel**
     - 2.2.1. Afiseaza valorile din vectorul F
       /* F reprezinta solutia problemei */
     - 2.2.2. **intrerupe ciclul**

sfarsit.

Verifica (U,F)

1. test ← adevarat
2. I ← U - 1
3. **cat timp** I > 0 **executa**
   3.1. test ← Relatie(I, F[I], U, F[U])
   3.2. I ← I - 1
   3.3. **daca** test = fals
   atunci intrerupe ciclul
4. **intoarce** test

sfarsit.

1.3 Imbunatatirea performantelor BKT

Algoritmi de imbunatatire a consistentei reprezentarii

- Consistenta locala a arcelor sau a cailor in graful de restrictii

Algoritmi hibrizi

- Imbunatatesc performantele rezolvarii prin reducerea numarului de teste.
  - *Tehnici prospective:*

  - Algoritmul de cautare cu predictie completa
  - Algoritmul de cautare cu predictie partiala
  - Algoritmul de cautare cu verificare predictiva
  - *Tehnici retrospective:*

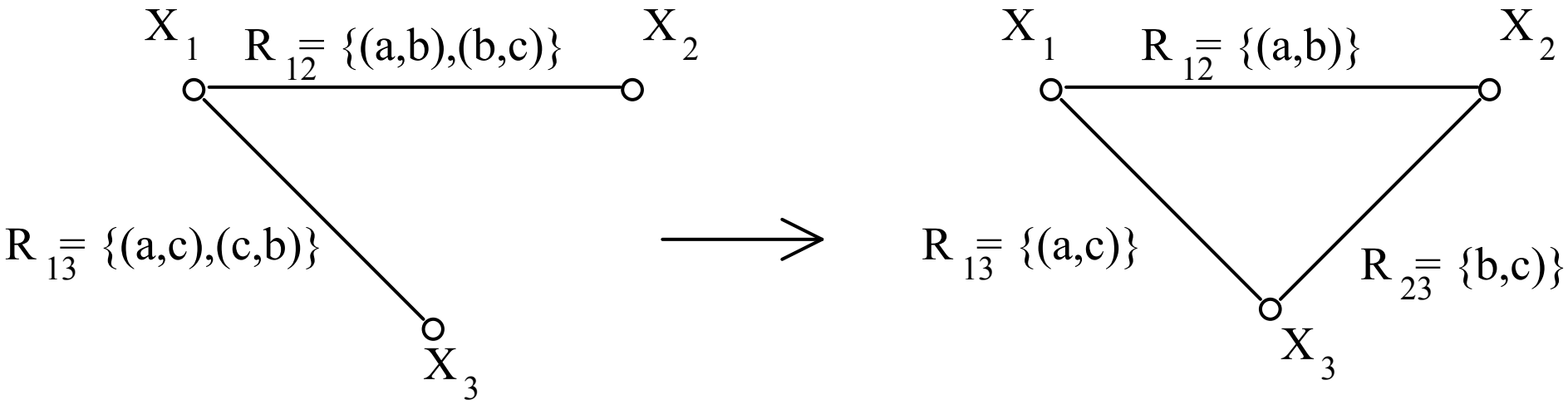
  - Algoritmul de backtracking cu salt
  - Algoritmul de backtracking cu marcare
  - Algoritmul de backtracking cu multime conflictuala

Utilizarea euristicilor

Algoritmi de imbunatatire a consistentei reprezentarii

- Propagarea restrictiilor

$$D_{X_1} = D_{X_2} = D_{X_3} = \{a, b, c\}$$



1.4 Propagarea locala a restrictiilor

- Combinatia de valori ***x*** si ***y*** pentru variabilele X_i si X_j este permisa de restrictia explicita $R_{ij}(x,y)$.

- Un **arc (X_i, X_j)** intr-un graf de restrictii orientat se numeste ***arc-consistent*** daca si numai daca pentru orice valoare $x \in D_i$, domeniul variabilei X_i, exista o valoare $y \in D_j$, domeniul variabilei X_j, astfel incat $R_{ij}(x,y)$.
- **Graf de restrictii orientat *arc-consistent***

Algoritm: **AC-3: Realizarea arc-consistentei pentru un graf de restrictii.**

1. Creeaza o coada $Q \leftarrow \{ (X_i, X_j) \mid (X_i, X_j) \in$ Multime arce, $i \neq j\}$
2. **cat timp** Q nu este vida **executa**
   - 2.1. Elimina din Q un arc (X_k, X_m)
   - 2.2. Verifica(X_k, X_m)
   - 2.3. **daca** subprogramul Verifica a facut schimbari in domeniul variabilei X_k

     atunci

     $Q \leftarrow Q \cup \{ (X_i, X_k) \mid (X_i, X_k) \in$ Multime arce, $i \neq k,m\}$

sfarsit.

Verifica (X_k, X_m)

pentru fiecare $x \in D_k$ **executa**

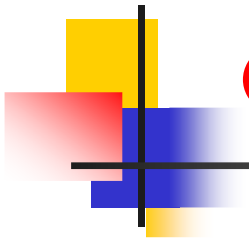
1. **daca** nu exista nici o valoare $y \in D_m$ astfel incat $R_{km}(x,y)$

   atunci elimina x din D_k

sfarsit.

Cale-consistenta

- O cale de lungime *m* prin nodurile $i_0,\ldots,i_m$ ale unui graf de restrictii orientat se numeste ***m-cale-consistenta*** daca si numai daca pentru orice valoare $x \in D_{i0}$, domeniul variabilei i_0 si o valoare $y \in D_{jm}$, domeniul variabilei i_m, pentru care $R_{i0im}(x,y)$, exista o secventa de valori $z_1 \in D_{i1} \ldots z_{m-1} \in D_{im-1}$ astfel incat $R_{i0i1}(x,z_1), \ldots, R_{im-1im}(z_{m-1},y)$
- **Graf de restrictii orientat *m-arc-consistent***
- Graf minim de restrictii
- n-cale-consistenta
- Comentarii

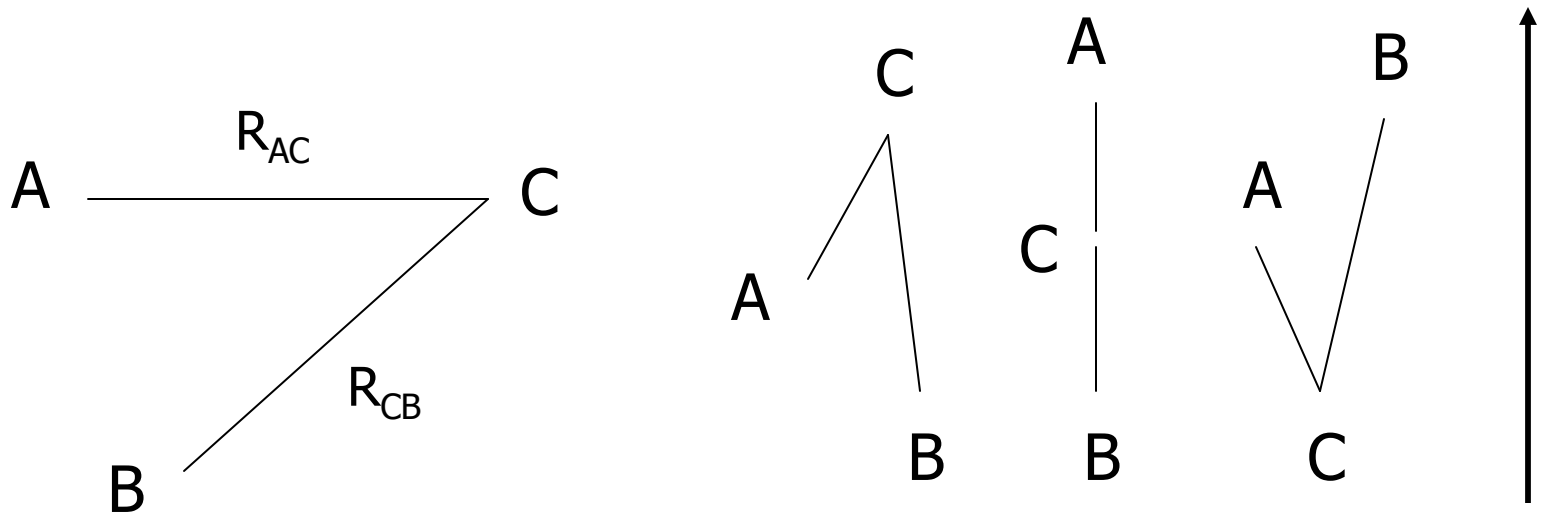


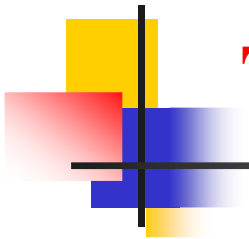
Complexitate

- **N** - numarul de variabile
- **a** - cardinalitatea maxima a domeniilor de valori ale variabilelor
- **e** - numarul de restrictii.
- Algoritmului de realizare a arc-consistentei - *AC-3*: complexitate timp este $\mathbf{O(e*a^3)}$; complexitate spatiu: **O(e+N*a)**
- S-a gasit si un algoritm de complexitate timp $\mathbf{O(e*a^2)}$ – *AC-4*
- Algoritmul de realizare a 2-cale-consistentei - *PC-4*: complexitatea timp $\mathbf{O(N^3*a^3)}$

1.5 CSP fara bkt - conditii

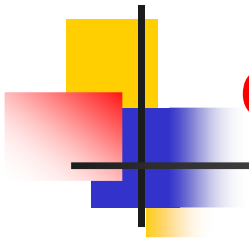
- Graf de restrictii ordonat
- Latimea unui nod
- Latimea unei ordonari a nodurilor
- Latimea unui graf de restrictii





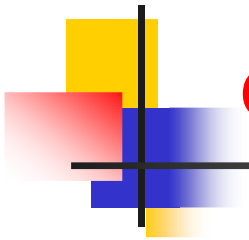
Teoreme

- Daca un graf de restrictii **arc-consistent** are ***latimea egala cu unu*** (i.e. este un arbore), atunci problema asociata grafului admite o solutie fara backtracking.
- Daca un graf de restrictii **2-cale-consistent** are ***latimea egala cu doi***, atunci problema asociata grafului admite o solutie fara backtracking.



d-consistenta

- Fiind data o ordonare *d* a variabilelor unui graf de restrictii R, **graful R este *d-arc-consistent*** daca toate arcele avand directia *d* sint arc-consistente.
- Fie un graf de restrictii R, avind ordonarea variabilelor *d* **cu latimea egala cu unu**. Daca R este d-arc-consistent atunci cautarea dupa directia *d* este fara backtracking.



d-consistenta

Algoritm: **Realizarea d-arc-consistentei unui graf de restrictii cu ordonarea variabilelor (X_1,…,X_N)**

pentru I ← N **la** 1 **executa**

1. **pentru** fiecare arc (X_j, X_i) cu j < i **executa**

   1.1. Verifica(X_j, X_i)

sfarsit.

- Complexitatea timp: $O(e*a^2)$

1.6 Tehnici prospective

- Conventii
- Notatii: U, N, F (F[U]), T (T[U] … X_U), TNOU, LINIE_VIDA

$$\text{Relatie}(U1, L1, U2, L2) = \begin{cases} \text{adevarat} & \text{daca } R_{U_1 U_2}(L_1, L_2) \\ \text{fals} & \text{în caz contrar} \end{cases}$$

- *Verifica_Inainte*
- *Verifica_Viitoare*

- **Predictie completa**
- **Predictie partiala**
- **Verificare predictiva**

Algoritm: **Backtracking cu predictie completa**

Predictie(U, F, T)

pentru fiecare element L din T[U] **executa**

1. F[U] ← L
2. **daca** U < N **atunci** //*verifica consistenta atribuirii*
   - 2.1 TNOU ← **Verifica_Inainte** (U, F[U], T)
   - 2.2 **daca** TNOU ≠ LINIE_VIDA
     atunci TNOU ← **Verifica _Viitoare** (U, TNOU)
   - 2.3 **daca** TNOU ≠ LINIE_VIDA
     atunci **Predictie** (U+1, F, TNOU)
3. **altfel** afiseaza atribuirile din F

sfarsit

Verifica_Inainte (U, L, T)

1. TNOU ← tabela vida
2. **pentru** U2 ← U+1 pana la N **executa**
    2.1 **pentru** fiecare element L2 din T[U2] **executa**
        2.1.1 daca Relatie(U, L, U2, L2) = adevarat
            atunci introduce L2 in TNOU[U2]
    2.2 **daca** TNOU[U2] este vida
        atunci intoarce LINIE_VIDA
3. **intoarce** TNOU

sfarsit

Verifica_Viitoare (U, TNOU)

daca U+1 < N **atunci**

1. **pentru** U1 ← U+1 pana la N **executa**
    1.1 **pentru** fiecare element L1 din TNOU[U1] **executa**
        1.1.1 **pentru** U2 ← U+1 pana la N, U2≠U1 **executa**
            i. **pentru** fiecare element L2 din TNOU[U2] **executa**
                - **daca** Relatie (U1, L1, U2, L2) = adevarat
                  atunci intrerupe ciclul //dupa L2
            ii. **daca** nu s-a gasit o valoare consistenta pentru U2
                atunci
                - elimina L1 din TNOU[U1]
                - **intrerupe ciclul** // dupa U2
    1.2 **daca** TNOU[U1] este vida
        atunci intoarce LINIE_VIDA
2. **intoarce** TNOU

sfarsit

BKT cu predictie partiala

- Se modifica Verifica_Viitoare pasii marcati cu rosu

Verifica_Viitoare (U, TNOU)
daca U+1 < N **atunci**
1. **pentru** U1 ← U+1 pana la N - 1 **executa**
    1.1 **pentru** fiecare element L1 din TNOU[U1] **executa**
        1.1.1 **pentru** U2 ← U1+1 pana la N **executa**
            i. **pentru** fiecare element L2 din TNOU[U2] **executa**
                - **daca** Relatie (U1, L1, U2, L2) = adevarat
                  atunci intrerupe ciclul //dupa L2
            ii. **daca** nu s-a gasit o valoare consistenta pentru U2
                atunci
                - elimina L1 din TNOU[U1]
                - **intrerupe ciclul** // dupa U2
    1.2 **daca** TNOU[U1] este vida
        atunci intoarce LINIE_VIDA
2. **intoarce** TNOU
sfarsit

BKT cu verificare predictiva

- Se elimina apelul Verifica_Viitoare(U, TNOU) in subprogramul Predictie

Algoritm: **Backtracking cu verificare predictiva**

Predictie(U, F, T)

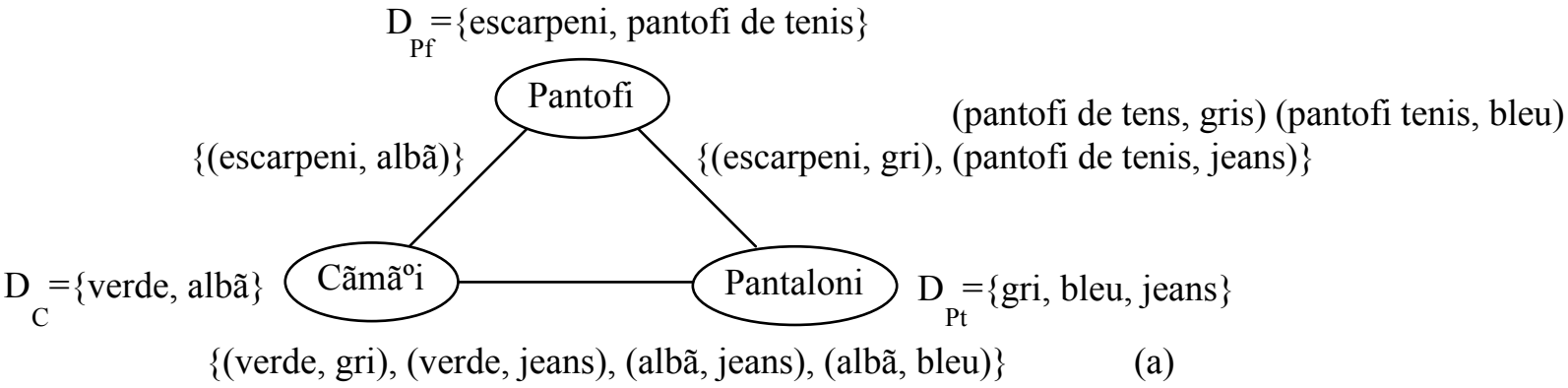
pentru fiecare element L din T[U] **executa**

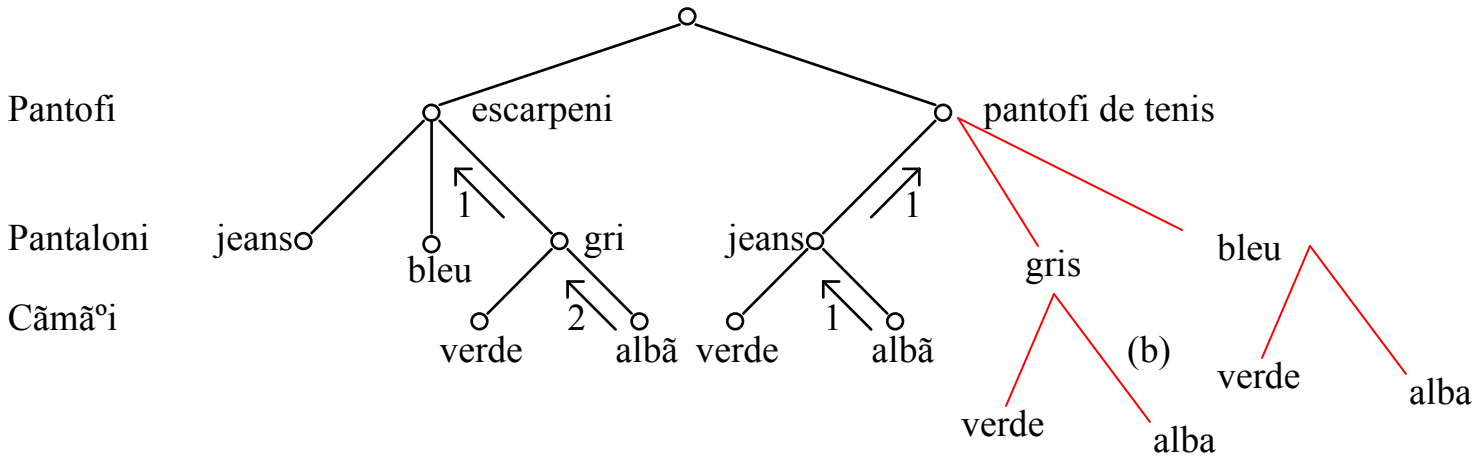
1. F[U] ← L
2. **daca** U < N **atunci** //*verifica consistenta atribuirii*
   - 2.1 TNOU ← Verifica_Inainte (U, F[U], T)
   - ~~2.2 **daca** TNOU ≠ LINIE_VIDA~~
   - ~~**atunci** TNOU ← Verifica _Viitoare (U, TNOU)~~
   - 2.2 **daca** TNOU ≠ LINIE_VIDA
   - **atunci** Predictie (U+1, F, TNOU)
3. **altfel** afiseaza atribuirile din F

sfarsit

1.7 Tehnici retrospective

- Backtracking cu salt





Algoritm: **Backtracking cu salt**

BacktrackingCuSalt(U, F) /* intoarce Nivel – blocare */

/* NrBlocari, NivelVec, I, test– var locale */

1. NrBlocari ← 0, I ← 0, Nivel ← U
2. **pentru** fiecare element V a lui X_U **executa**
   - 2.1 F[U] ← V
   - 2.2 test, NivelVec[I] ← Verifica**1** (U, F)
   - 2.3 **daca** test = adevarat **atunci**
     - 2.3.1 **daca** U < N **atunci**
       - i. Nivel ← BacktrackingCuSalt (U+1, F)
       - ii. **daca** Nivel < U **atunci** salt la pasul 4
     - 2.3.2 **altfel** afiseaza valorile vectorului F // solutia
   - 2.4 **altfel** NrBlocari ← NrBlocari + 1
   - 2.5 I ← I + 1
3. **daca** NrBlocari = numar valori ale lui X[U] **si**
   toate elementele din NivelVec sunt egale
   atunci Nivel ← NivelVec[1]
4. **intoarce** Nivel

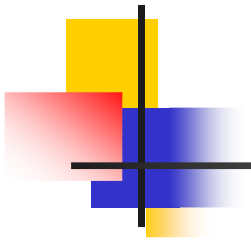
sfarsit

Verifica**1** (U, F)

/* intoarce test si nivelul la care s-a produs blocarea sau 0 */

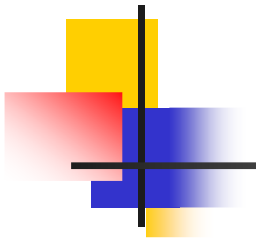
1. test ← adevarat
2. I ← U-1
3. **cat timp** I>0 **executa**
   - 3.1 test ← Relatie(I, F[I], U, F[U])
   - 3.2 **daca** test = fals
     - **atunci** intrerupe ciclul
   - 3.3 I ← I –1
4. NivelAflat ← I
5. **intoarce** test, NivelAflat

sfarsit



1.8 Euristici

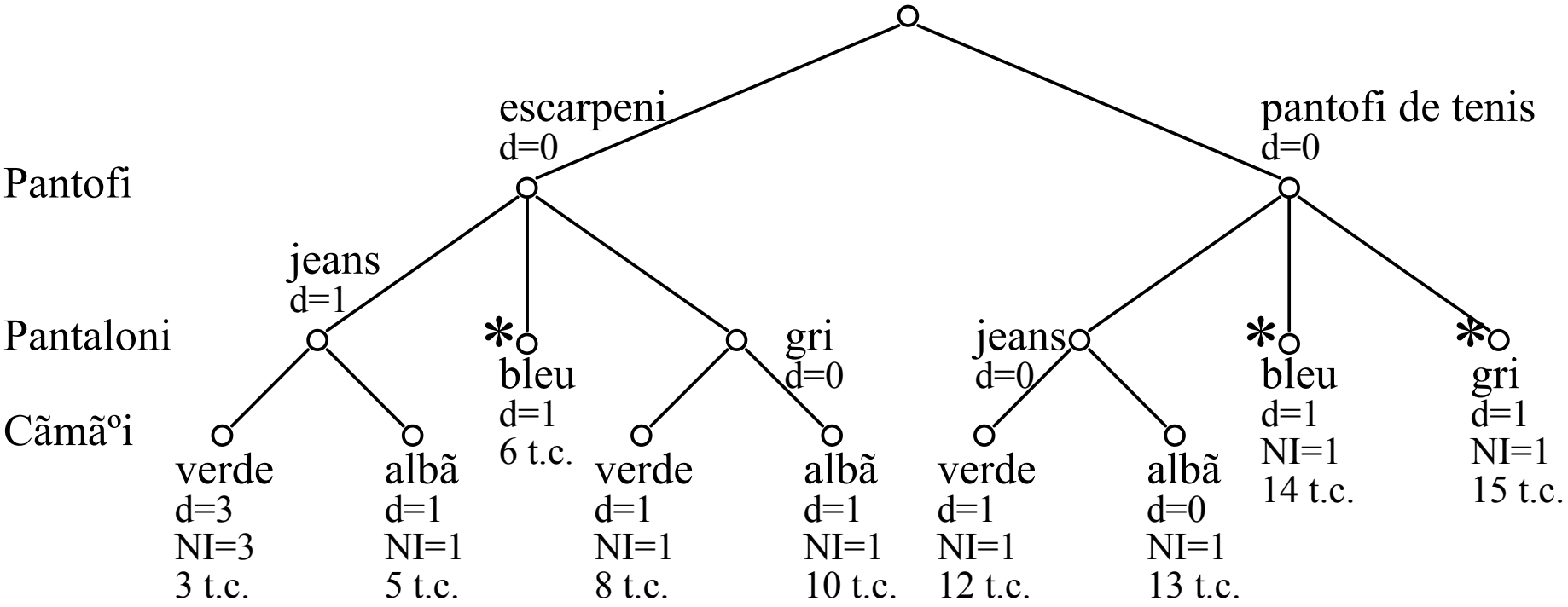
- **Ordonarea variabilelor**
- **Ordonarea valorilor**
- **Ordonarea testelor**

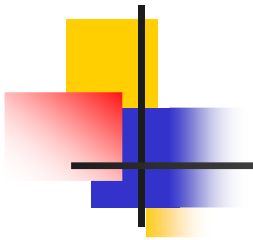


1.9 CSP partiala

- Memoreaza cea mai buna solutie gasita pana la un anumit moment (gen IDA*) – *distanta d* fata de solutia perfecta
- Abandoneaza calea de cautare curenta in momentul in care se constata ca acea cale de cautare nu poate duce la o solutie mai buna
- NI - numarul de inconsistente gasite in "cea mai buna solutie" depistata pana la un moment dat – *limita necesara*

CSP partiala





CSP partiala

- S - *limita suficienta* - specifica faptul ca o solutie care violeaza un numar de S restrictii (sau mai putine), este acceptabila.
- PBKT(Cale, Distanta,Variablie,Valori)
  - Semnificatie argumente
  - Rezultat: GATA sau CONTINUA
- variabile globale: CeaMaiBuna, NI, S

Algoritm: **CSP Partiala**
PBKT(Cale, Distanta, Variabile, Valori)

/ intoarce GATA sau CONTINUA */*

1. **daca** Variabile = {}
   atunci
   1.1 CeaMaiBuna ← Cale
   1.2 NI ← Distanta
   1.3 **daca** NI ≤ S **atunci intoarce** GATA
   altfel intoarce CONTINUA
2. **altfel**
   2.1 **daca** Valori = {} **atunci** CONTINUA

/ s-au incercat toate valorile si se revine la var ant. */*

   2.2 **altfel**
   2.2.1 **daca** Distanta ≥ NI
   atunci intoarce CONTINUA

/ revine la var ant pentru gasirea unei solutii mai bune*/*

2.2.2 **altfel**

i. Var ← first(Variabile)

ii. Val ← first(Valori)

iii. DistNoua ← Distanta

iv. Cale1 ← Cale

v. **cat timp** Cale1 <> {} **si** DistNoua < NI **executa**

- (VarC, ValC) ← first(Cale1)
- **daca** Rel(Var,Val,VarC,ValC) = fals
- **atunci** DistNoua ← DistNoua + 1
- Cale1 ← Rest(Cale1)

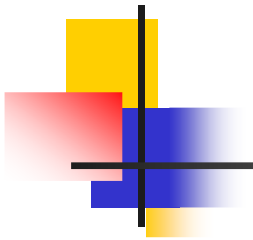
vi. **daca** DistNoua < NI **si**
PBKT(Cale+(Val,Var),DistNoua,Rest(Variabile),ValoriNoi)
= GATA

/* ValoriNoi - domeniul de valori asociat primei variabile din Rest(Variabile) */

atunci intoarce GATA

altfel intoarce
PBKT(Cale,Distanta,Variabile,Rest(Valori)

sfarsit



2 Cautari locale

- Calea catre solutie nu are importanta
- Cautari locale – opereaza asupra starii curente generand succesorii
- Gasire solutie – CSP, nu conteaza calitatea solutiei
- Gasire solutie optima – functie de evaluare sau de cost
- Probleme in gasirea solutiei optime pe baza de cautari locale (maxim global, maxim local, platou, umar)
- Hill climbing

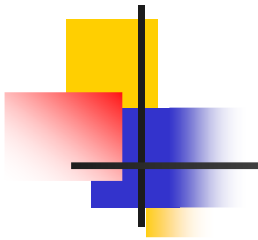
Algoritm: **Hill climbing**

/* intoarce o stare cu functia de evaluare un maxim local */

1. S ← stare initiala
2. S' ← S_j=Succ(S), S_j cu Eval(S_j) maxim pt succesorii S_j a lui S
3. **daca** Eval(S') ≤ Eval(S) **atunci intoarce** S
4. S ← S'
5. **repeta** de la 2

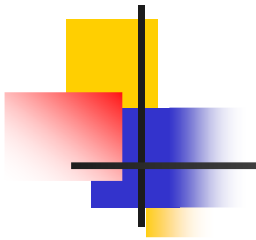
sfarsit

- Cautarea se orienteaza permanent in directia cresterii valorii;
- Se termina cand ajunge la un maxim (nici un succesor nu are valoarea mai mare)



Hill Climbing

- Problema 8 regine
- S – tabla completa
- Succesori – toate starile posibil de generat prin mutarea a 1 regina intr-un alt patrat in coloana
- o stare are 8 x 7 = 56 succesori
- Fct de evaluare = nr de perechi de regine care se ataca
- Minim global = 0
- 86% instante nu gaseste solutia (3 pasi)
- 14% gaseste soluta (4 pasi)
- Spatiul starilor $8^8 \approx 17$ milioane de stari
- Limitari



Hill Climbing - imbunatatiri

- Miscari laterale – se spera sa fie “umar” si nu “platou”
- Daca “platou” – risc de bucla infinita – necesita limita in cautare
- Cu aprox. 100 miscari laterale – problema 8 regine – 94% instante pt care se gaseste solutia

Hill Climbing - imbunatatiri

Hill climbing stohastic

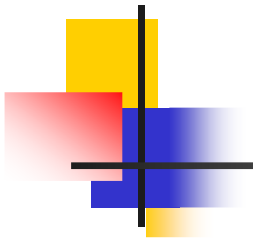
- Dintre starile succesoare cu Eval(S) ≥ Eval(S_j), se alege aleator un S_j

First choice hill climbing

- Genereaza aleator succesori pana gaseste Eval(S) ≥ Eval(S_j), continua cautarea cu S_j

Random restart Hill Climbing

- Repeta diferite HC cu stari initiale generate aleator
- Daca fiecare HC are o probabilitate de succes de p → 1/p repetitii
- 8 regine 14% → p ≈ 0.14 → aprox 7 iteratii



Simulated annealing

- Simuleaza un proces fizic (calirea)
- T – temperatura
- Scaderea gradientului – solutii de cost minim
- Alege o miscare la intamplare
- Daca starea este mai buna, cauta in continuare de la aceasta
- Altfel alege starea mai "proasta" cu o probabilitate
- Scade probabilitatea de selectie a starilor mai "proaste" pe masura ce temperatura scade

Algoritm: **Cautare Simulated Annealing**

1. T ← temperatura initiala
2. S ← stare initiala
3. v ← Eval(S)
2. **cat timp** T > temp finala **executa**
    2.1 **pentru** i=1,n **executa**
        S' ← Succ(S)
        v' ← Eval(S')
        daca v' < v **atunci S** ← S'
        altfel S ← S' cu prob. exp(-(v'-v)/kT)
    2.2 T ← 0.95 * T

sfarsit