

Probleme de satisfacere a restricțiilor

IA 2025/2026

Introducere

Algoritmi de căutare

Îmbunătățirea algoritmului de căutare

Structura grafului de restricții

Căutare locală

Probleme de satisfacere a constrângerilor (CSP)

- ▶ Sunt definite prin: **variabilele** X_i cu valori din **domeniul** D_i și o mulțime de **constrângeri** care specifică combinațiile permise de valori pentru submulțimi de variabile.
- ▶ O asignare este *consistentă* dacă nu sunt încălcate constrângeri. O asignare este *completă* dacă include toate variabilele.
Soluție: o asignare completă de valori variabilelor a.î. toate constrângerile sunt satisfăcute.
- ▶ NP-hard
 - ▶ algoritmi cu *scop-general*, mai puternici decât algoritmi de căutare standard
- ▶ MaxCSP: maximizează numărul de constrângeri satisfăcute

Exemplu: Colorarea unei hărți



Variabile: WA, NT, Q, NSW, V, SA, T

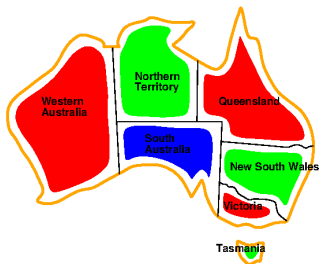
Domenii: $D_i = \{red, green, blue\}$

Constrângeri: regiunile adiacente trebuie să aibă culori diferite

$WA \neq NT$ (dacă limbajul permite), sau

$(WA, NT) \in \{(red, green), (red, blue), (green, red), (green, blue), \dots\}$

Exemplu: Colorarea unei hărți

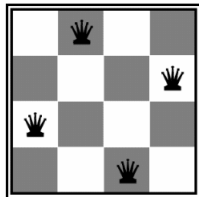


Soluțiile sunt asignări care satisfac toate restricțiile.

$\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green\}$

Exemplu: N-Regine

Formularea 1:



Variable: X_{ij}

Domenii: $\{0, 1\}$

Constrângeri

$$\sum_{i,j} X_{ij} = N$$

$$\forall i, j, k \quad (X_{ij}, X_{ik}) \in \{(0, 0), (0, 1), (1, 0)\}$$

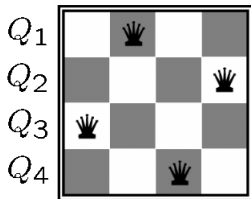
$$\forall i, j, k \quad (X_{ij}, X_{kj}) \in \{(0, 0), (0, 1), (1, 0)\}$$

$$\forall i, j, k \quad (X_{ij}, X_{i+k, j+k}) \in \{(0, 0), (0, 1), (1, 0)\}$$

$$\forall i, j, k \quad (X_{ij}, X_{i+k, j-k}) \in \{(0, 0), (0, 1), (1, 0)\}$$

Exemplu: N-Regine

Formularea 2:



Variabile: Q_k

Domenii: $\{1, 2, 3, \dots, N\}$

Constrângeri

Exprimate implicit: $\forall i, j \text{ non-threatening}(Q_i, Q_j)$

Exprimate explicit: $(Q_1, Q_2) \in \{(1, 3), (1, 4), \dots\} \dots$

Exemplu: Sudoku

Variabile: $x_{ij} \in \{1, \dots, 9\} =: N$ (valorile din celulele corespunzătoare)

Restricții de inegalitate (perechi): toate valorile de pe o linie, coloană, regiune sunt diferite

$$x_{ij} \neq x_{ik} \quad \forall k \neq i, j \in N, (\text{linie})$$

$$x_{ij} \neq x_{kj} \quad \forall k \neq i, j \in N, (\text{coloana})$$

$$x_{i_1 j_1} \neq x_{i_2 j_2} \quad \forall (i_1, j_1) \neq (i_2, j_2) \in C_{ij}, \forall i, j \in N' = \{1, 2, 3\}, (\text{regiune})$$

$$x_{ij} \in N \quad \forall i, j \in N$$

$$C_{ij} = \{(3(i-1) + i', 3(j-1) + j') \mid (i', j') \in N' \times N'\}$$

Se poate utiliza restricția globală `alldifferent` pentru o formulare mai puternică:

$$\text{alldifferent}(x_{ij} \mid j \in N) \quad \forall i \in N, (\text{linie})$$

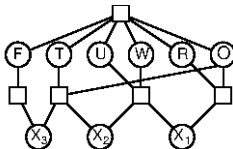
$$\text{alldifferent}(x_{ij} \mid i \in N) \quad \forall j \in N, (\text{coloana})$$

$$\text{alldifferent}(C_{ij}) \quad \forall i, j \in N', (\text{regiune})$$

$$x_{i,j} \in N \quad \forall i, j \in N$$

Exemplu: Cryptarithmic puzzle

$$\begin{array}{r} \text{TWO} \\ + \text{TWO} \\ \hline \text{FOUR} \end{array}$$



Variable: $F, T, U, W, R, O, X_1, X_2, X_3$

Domenii: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Constrângeri:

$\text{alldiff}(F, T, U, W, R, O)$

$O + O = R + 10 \cdot X_1$, etc.

Solution: $F = 1, O = 4, R = 8, T = 7, W = 3, U = 6$ ($734 + 734 = 1468$)

Exemplu: *Job-shop scheduling*

Asamblarea unei mașini

- ▶ Sarcini: instalează axele (față, spate), fixează roțile, strânge piulițele (pentru fiecare roată), fixează capacele roților, inspectează ansamblul final.

Modelare: variabila: sarcină, valoare: timpul la care începe (minute).

$X = \{Axle_F, Axle_B, Wheel_{RF}, Wheel_{LF}, Wheel_{RB}, Wheel_{LB}, Nuts_{RF}, Nuts_{LF}, Nuts_{RB}, Nuts_{LB}, Cap_{RF}, Cap_{LF}, Cap_{RB}, Cap_{LB}, Inspect\}$

- ▶ Constrângeri:
 - ▶ o sarcină trebuie să înceapă înaintea alteia (o roată trebuie instalată înaintea capacului)
 $T_1 + d_1 \leq T_2$ (T_1 trebuie să înceapă înaintea lui T_2)
 - ▶ o sarcină necesită o perioadă de timp pentru a fi finalizată

Exemplu: *Job-shop scheduling*

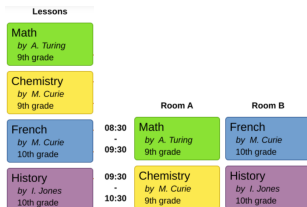
Asamblarea unei mașini. Constrângeri:

- ▶ Axele trebuie instalate înaintea roților
 $Axle_F + 10 \leq Wheel_{RF}$
- ▶ După fixarea roților, strânge piulițele și apoi atașează capacele
 $Wheel_{RF} + 1 \leq Nuts_{RF}; Nuts_{RF} + 2 \leq Cap_{RF}$
- ▶ Pentru a așeza axele, se utilizează un instrument ($Axle_F$, $Axle_B$ nu se suprapun)
 $(Axle_F + 10 \leq Axle_B)$ **or** $(Axle_B + 10 \leq Axle_F)$
- ▶ Inspectia este la sfârșit și durează 3 minute
 $X + d_X \leq Inspect$
- ▶ Ansamblul trebuie terminat în 30 de minute
 $D = \{1, 2, 3, \dots, 27\}$

Exemplu: planificarea orarului

Planificarea orarului profesorilor și a elevilor

- ▶ Profesori, materii, clase, săli, intervale orare
Un profesor predă anumite materii
O materie este ținută la anumite clase
- ▶ Constrângeri:
 - ▶ două cursuri pentru aceeași clasă nu pot fi programate în același timp
 - ▶ un profesor preferă anumite intervale orare, etc.



Exemplu: planificarea orarului

$t \in T$ timp, $r \in R$ săli, $c \in C$ cursuri, $s \in S$ studenți, $(c_i, c_j) \in SA$: c_i, c_j nu se pot suprapune (același profesor), z_s cursurile alocate grupei s

- ▶ Variabile: $x_c = t$ cursul c este programat la t
 $y_c = r$ cursul c are loc în r
- ▶ Constrângeri:
 - ▶ $x_c = t$, for $c \in C$, ...
 - ▶ două cursuri nu se pot desfășura în același timp (același profesor)
If $x_{c_1} = t_1$ and $(c_1, c_2) \in SA$ then $x_{c_2} \neq t_2$ if $overlap(t_1, t_2)$
 - ▶ două cursuri alocate aceleiași grupe de studenți nu se pot desfășura în același timp
If $z_s = c_1$ and $z_s = c_2$ and $x_{c_1} = t_1$ then $x_{c_2} \neq t_2$ if $overlap(t_1, t_2)$
 - ▶ două cursuri nu pot fi programate în același timp în aceeași sală
If $(x_{c_1} = t_1)(x_{c_2} = t_2)(y_{c_1} = r_1)$ then $y_{c_2} \neq r_1$ if $overlap(t_1, t_2)$

Probleme din lumea reală

- ▶ Planificarea orarului personalului medical
- ▶ Planificarea transportului în comun, a construcției unei fabrici, *Floorplanning*, etc.
- ▶ *Meeting scheduling*
- ▶ Probleme de asignare (a profesorilor la clase)
- ▶ Configurare hardware, etc.

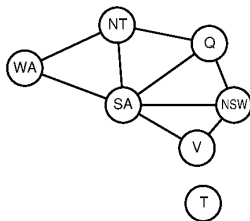
Obs: multe probleme din lumea reală implică variabile cu valori reale
<https://www.csplib.org/Problems/>

Tipuri de variabile și constrângeri

- ▶ Variabile discrete (domenii finite (dimensiune domeniu $d \implies O(d^n)$ asignări complete) și infinite) și continue
- ▶ Constrângeri
 - ▶ **unare** implică o singură variabilă
ex: $SA \neq green$
 - ▶ **binare** implică perechi de variabile
ex: $SA \neq WA$
Probleme **CSP binare**: fiecare constrângere se referă la cel mult două variabile
 - ▶ **de ordin superior** implică 3 sau mai multe variabile
ex: *cryptarithmic puzzle*
 - ▶ **Preferințe** (restricții *soft*) ex: *roșu* este mai bun decât *verde*
reprezentate prin costuri asociate asignărilor \rightarrow probleme de optimizare

Graful de restricții

Nodurile sunt variabile, muchiile reprezintă restricții.



Algoritmii cu scop-general utilizează structura grafului pentru a accelera căutarea (ex: Tasmania este o subproblemă independentă)

Introducere

Algoritmi de căutare

Îmbunătățirea algoritmului de căutare

Structura grafului de restricții

Căutare locală

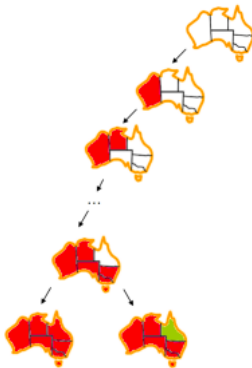
Formularea standard (căutare incrementală)

Stările sunt definite prin valorile asiguate.

- ▶ **Starea inițială:** asignarea vidă, \emptyset
- ▶ **Funcția succesori:** atribuie o valoare unei variabile neasiguate
- ▶ **Testarea obiectivului:** asignarea curentă este completă

Formularea standard (căutare incrementală)

- ▶ Soluțiile se găsesc la adâncimea n în arbore (n variabile asignate)
⇒ utilizează DFS
- ▶ Factorul de ramificare este nd , la următorul nivel $(n - 1)d$, etc. → $n!d^n$ frunze, d^n asignări posibile



Backtracking

Asignările variabilelor sunt **comutative**

$[WA = red, NT = green]$ la fel ca $[NT = green, WA = red]$

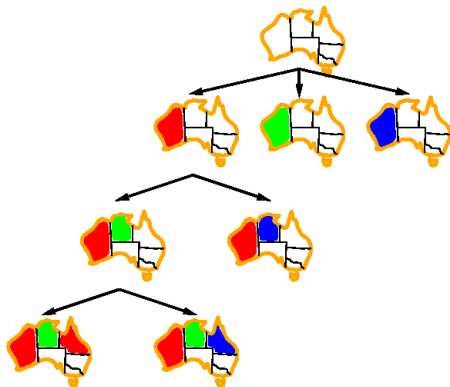
La fiecare nod considerăm asignarea unei singure variabile
(nu intră în conflict cu asignarea curentă)

$\implies d$ noduri pe nivel și d^n frunze

Căutarea DFS pentru CSP cu asignări pentru o singură variabilă:
backtracking. Backtracking este algoritmul de bază neinformatic pentru CSP.

Exemplu Backtracking

Când un nod este extins, se verifică dacă fiecare stare următoare este consistentă, înainte de a fi adăugată.



Backtracking

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
  return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment given CONSTRAINTS[csp] then
      add { var = value } to assignment
      result ← RECURSIVE-BACKTRACKING(assignment, csp)
      if result ≠ failure then return result
      remove { var = value } from assignment
  return failure
```

Backtracking = DFS + ordonarea variabilelor + *fail-on-violation*

Poate rezolva problema celor n regine pentru $n \approx 25$.

Introducere

Algoritmi de căutare

Îmbunătățirea algoritmului de căutare

Structura grafului de restricții

Căutare locală

Îmbunătățirea algoritmului de căutare

1. Ce variabilă trebuie asignată?
2. În ce ordine trebuie verificate valorile?
3. Putem detecta eșecul mai devreme?
4. Putem profita de structura problemei?

1. Minimum-remaining-values

Minimum-remaining-values (MRV): alege variabila cu cele mai puține valori permise (variabila cea mai constrânsă).

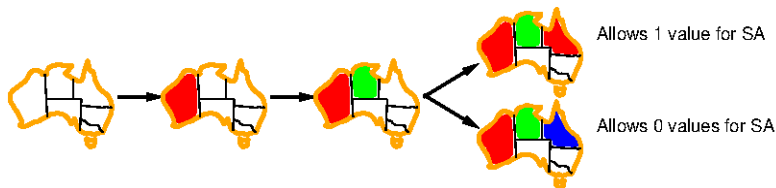


"Fail-first" heuristic

2. Least-constraining-value

Least-constraining-value: alege valoarea cea mai puțin constrânsă (cea care exclude cele mai puține valori)

Exemplu: ce valoare alegem pentru Q ?



Combinarea acestor euristici face posibilă rezolvarea problemei 1000-regine.

Chestionar



Question!

Variable order WA, NT, Q, NSW, V, T, SA . **Tightest upper bound on naïve backtracking search space size?**

(A): 145

(B): 382

(C): 433

(D): 3^7

Chestionar



Question!

Variable order SA, NT, Q, NSW, V, WA, T . **Tightest upper bound on naïve backtracking search space size?**

(A): 52

(B): 145

(C): 382

(D): 433

Forward checking

Idee: actualizează domeniul variabilelor neasignate

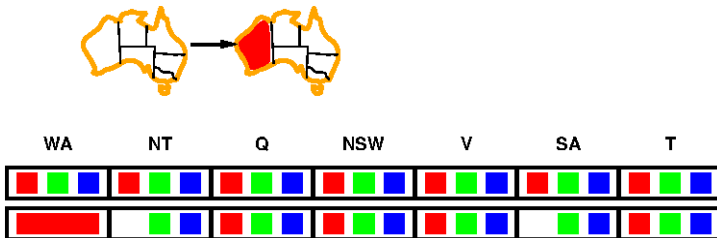
- ▶ atunci când selectăm o valoare pentru o variabilă, elimină valoarea din domeniul variabilelor neasignate, conectate cu aceasta



Forward checking

Idee: actualizează domeniul variabilelor neasignate

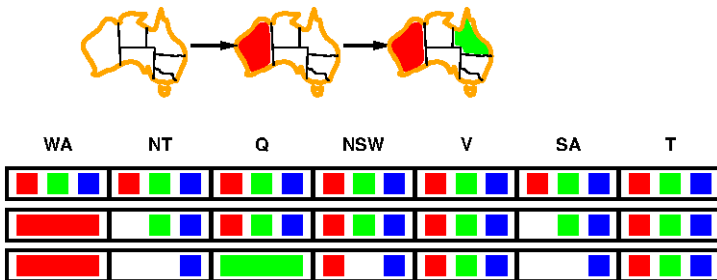
- ▶ atunci când selectăm o valoare pentru o variabilă, elimină valoarea din domeniul variabilelor neasignate, conectate cu aceasta



Forward checking

Idee: actualizează domeniul variabilelor neasignate

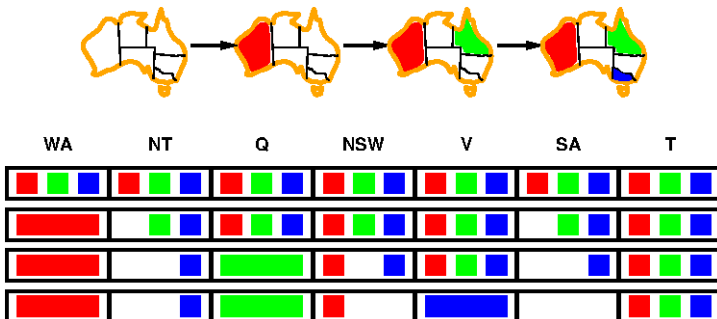
- ▶ atunci când selectăm o valoare pentru o variabilă, elimină valoarea din domeniul variabilelor neasignate, conectate cu aceasta



Forward checking

Idee: actualizează domeniul variabilelor neasignate

- ▶ atunci când selectăm o valoare pentru o variabilă, elimină valoarea din domeniul variabilelor neasignate, conectate cu aceasta



Forward checking - algorithm

```
procedure SELECTVALUE-FORWARD-CHECKING
  while  $D'_i$  is not empty
    select an arbitrary element  $a \in D'_i$ , and remove  $a$  from  $D'_i$ 
    empty-domain  $\leftarrow$  false
    for all  $k, i < k \leq n$ 
      for all values  $b$  in  $D'_k$ 
        if not CONSISTENT( $\vec{a}_{i-1}, x_i = a, x_k = b$ )
          remove  $b$  from  $D'_k$ 
      end for
      if  $D'_k$  is empty      ( $x_i = a$  leads to a dead-end)
        empty-domain  $\leftarrow$  true
      if empty-domain    (don't select  $a$ )
        reset each  $D'_k, i < k \leq n$  to value before  $a$  was selected
      else
        return  $a$ 
    end while
    return null          (no consistent value)
end procedure
```

Generalized look ahead

procedure GENERALIZED-LOOKAHEAD

Input: A constraint network $P = (X, D, C)$

Output: Either a solution, or notification that the network is inconsistent.

$D'_i \leftarrow D_i$ for $1 \leq i \leq n$ (copy all domains)

$i \leftarrow 1$ (initialize variable counter)

while $1 \leq i \leq n$

 instantiate $x_i \leftarrow \text{SELECTVALUE-XXX}$

if x_i is null (no value was returned)

$i \leftarrow i - 1$ (backtrack)

 reset each $D'_k, k > i$, to its value before x_i was last instantiated

else

$i \leftarrow i + 1$ (step forward)

end while

if $i = 0$

return “inconsistent”

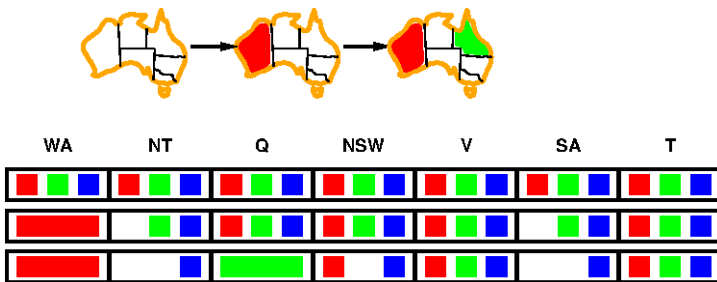
else

return instantiated values of $\{x_1, \dots, x_n\}$

end procedure

Propagarea constrângerilor

Forward checking propagă informații variabilelor neasignate, dar nu asigură detectarea timpurie a eșecurilor:

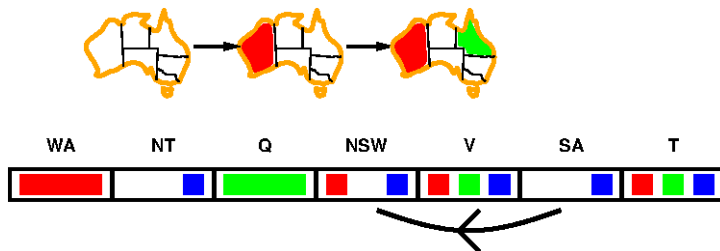


NT și *SA* nu pot fi colorate ambele cu albastru!
Nu există propagare între variabilele neasignate!

Arc consistency

Arc consistency: cea mai simplă formă de propagare; face ca fiecare arc să fie **consistent**.

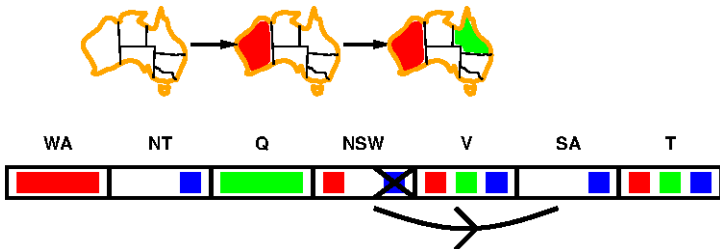
$X \rightarrow Y$ este consistent dacă și numai dacă
pentru *fiecare* valoare x a lui X , există o valoare permisă y pentru Y



Arc consistency

Arc consistency: cea mai simplă formă de propagare; face ca fiecare arc să fie **consistent**.

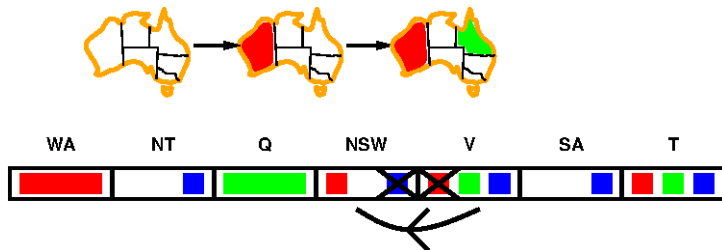
$X \rightarrow Y$ este consistent dacă și numai dacă
pentru *fiecare* valoare x a lui X , există o valoare permisă y pentru Y



Arc consistency

Arc consistency: cea mai simplă formă de propagare; face ca fiecare arc să fie **consistent**.

$X \rightarrow Y$ este consistent dacă și numai dacă
pentru *fiecare* valoare x a lui X , există o valoare permisă y pentru Y



Dacă X pierde o valoare, vecinii lui X trebuie verificați.

Algoritmul Arc consistency

function AC-3(*csp*) **returns** the CSP, possibly with reduced domains

inputs: *csp*, a binary CSP with variables $\{X_1, X_2, \dots, X_n\}$

local variables: *queue*, a queue of arcs, initially all the arcs in *csp*

while *queue* is not empty **do**

$(X_i, X_j) \leftarrow \text{REMOVE-FIRST}(\textit{queue})$

if REMOVE-INCONSISTENT-VALUES(X_i, X_j) **then**

for each X_k **in** NEIGHBORS[X_i] **do**

 add (X_k, X_i) to *queue*

function REMOVE-INCONSISTENT-VALUES(X_i, X_j) **returns** true iff succeeds

removed \leftarrow false

for each x **in** DOMAIN[X_i] **do**

if no value y in DOMAIN[X_j] allows (x, y) to satisfy the constraint $X_i \leftrightarrow X_j$

then delete x from DOMAIN[X_i]; *removed* \leftarrow true

return *removed*

Fiecare arc e adăugat în Q de max d ori.

Complexitate timp: $O(n^2 d^3)$; versiuni $O(n^2 d^2)$.

- ▶ 3-consistență (*path consistency*), k -consistență
 k -consistență: orice asignare consistentă a $k - 1$ variabile poate fi extinsă la o instanțiere de k variabile

Dacă o problemă CSP cu n variabile este n -consistentă, atunci nu mai e necesară căutarea Backtracking.

- ▶ Utilizarea tehnicilor de propagare a constrângerilor implică și o creștere a timpului de execuție
 - ▶ un compromis între propagare și căutare; dacă propagarea durează mai mult decât căutarea, atunci nu se merită

Conflict-Directed Backjumping (CBJ)

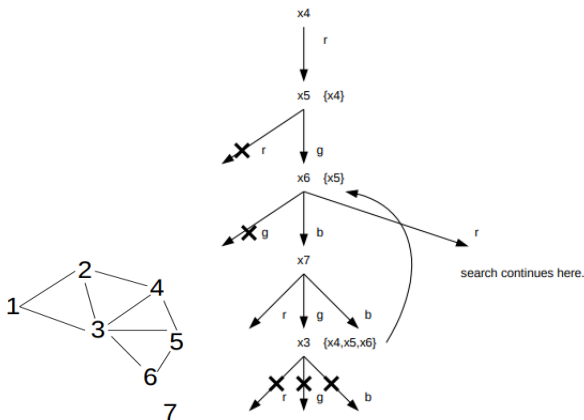
- ▶ Backtracking: ne întoarcem la variabila anterioară pentru a-i asigura o nouă valoare
 - ▶ ne întoarcem UN nivel în arborele de căutare
- ▶ Când ajungem într-un punct din care nu mai putem continua (datorită unei inconsistențe), putem încerca să identificăm cauza problemei
 - ▶ în loc să ne întoarcem un nivel, ne putem întoarce direct la variabila care a cauzat problema

Conflict-Directed Backjumping

- ▶ **Idee:** Menține o mulțime de conflicte `CONFLICT_SET` pentru fiecare variabilă (actualizată pe măsură ce asignăm valori variabilelor)
- ▶ Considerăm variabila curentă X_i . Mulțimea `CONFLICT_SET` a lui X_i este mulțimea de `VARIABLE ASIGNATE ANTERIOR` conectate cu X_i (datorită unei restricții)
- ▶ Dacă nu am identificat o asignare validă pentru variabila curentă X_i , ne ÎNTOARCEM la variabila X_k , din mulțimea de conflicte a lui X_i , cea mai apropiată
- ▶ Actualizăm mulțimea de conflicte a lui X_k
$$CONFLICT_SET(X_k) =$$
$$CONFLICT_SET(X_k) \cup CONFLICT_SET(X_i) \setminus X_k$$

Conflict-Directed Backjumping

Exemplu: colorarea unei hărți



X_7 nu este în mulțimea de conflicte a lui X_3 ; ne întoarcem la cea mai apropiată variabilă din mulțimea de conflicte (X_6)

Conflict-Directed Backjumping

procedure CONFLICT-DIRECTED-BACKJUMPING

Input: A constraint network $\mathcal{R} = (X, D, C)$.

Output: Either a solution, or a decision that the network is inconsistent.

```
 $i \leftarrow 1$                                 (initialize variable counter)
 $D'_i \leftarrow D_i$                       (copy domain)
 $J_i \leftarrow \emptyset$                   (initialize conflict set)
while  $1 \leq i \leq n$ 
    instantiate  $x_i \leftarrow \text{SELECTVALUE-CBJ}$ 
    if  $x_i$  is null                        (no value was returned)
         $i_{prev} \leftarrow i$ 
         $i \leftarrow \text{index of last variable in } J_i$   (backjump)
         $J_i \leftarrow J_i \cup J_{i_{prev}} - \{x_i\}$  (merge conflict sets)
    else
         $i \leftarrow i + 1$                 (step forward)
         $D'_i \leftarrow D_i$               (reset mutable domain)
         $J_i \leftarrow \emptyset$           (reset conflict set)
    end while
    if  $i = 0$ 
        return "inconsistent"
    else
        return instantiated values of  $\{x_1, \dots, x_n\}$ 
end procedure
```

Conflict-Directed Backjumping

subprocedure SELECTVALUE-CBJ

```
  while  $D'_i$  is not empty
    select an arbitrary element  $a \in D'_i$ , and remove  $a$  from  $D'_i$ 
     $consistent \leftarrow true$ 
     $k \leftarrow 1$ 
    while  $k < i$  and  $consistent$ 
      if CONSISTENT( $\vec{a}_k, x_i = a$ )
         $k \leftarrow k + 1$ 
      else
        let  $R_S$  be the earliest constraint causing the conflict
        add the variables in  $R_S$ 's scope  $S$ , but not  $x_i$ , to  $J_i$ 
         $consistent \leftarrow false$ 
    end while
    if  $consistent$ 
      return  $a$ 
    end while
  return null (no consistent value)
end procedure
```

Introducere

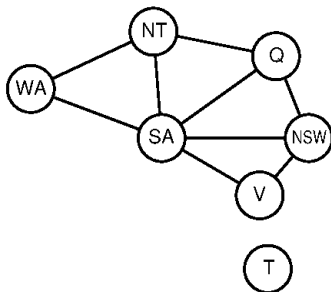
Algoritmi de căutare

Îmbunătățirea algoritmului de căutare

Structura grafului de restricții

Căutare locală

Structura problemei



Tasmania și continentul sunt **subprobleme independente**

Identificabile ca și **componente conexe** ale grafului constrângerilor.

Structura problemei

Presupunem că fiecare subproblemă are c variabile, dintr-un total de n .

Complexitatea în cazul cel mai nefavorabil este $n/c \cdot d^c$, *liniar* în n

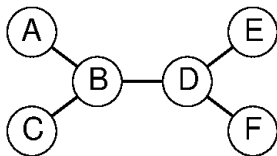
Exemplu: $n = 80$, $d = 2$, $c = 20$

$2^{80} = 4$ miliarde de ani la 10 milioane noduri/sec vs.

$4 \cdot 2^{20} = 0.4$ secunde la 10 milioane noduri/sec

Sunt rare situațiile acestea.

Probleme CSP cu structură arborescentă



Teoremă: dacă graful de constrângeri nu are cicluri, problema CSP poate fi rezolvată în $O(nd^2)$ timp.

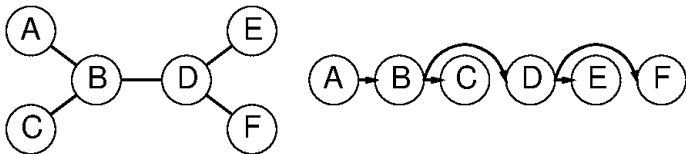
(Reamintim: pentru problemele generale CSP, complexitatea timp în cazul cel mai nefavorabil este $O(d^n)$)

Probleme CSP cu structură arborescentă

O problemă CSP este *directed arc-consistent* pentru o ordonare X_1, X_2, \dots, X_n a variabilelor \iff fiecare X_i este arc-consistent cu $X_j, \forall j > i$.

Metodă:

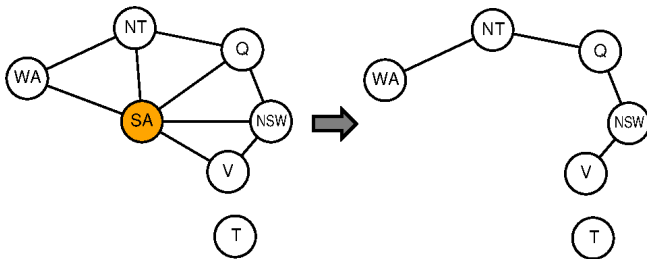
1. Alege o variabilă drept rădăcină, ordonează variabilele de la rădăcină la frunze a.î. părintele fiecărui nod îl precede în ordonare



2. For $j = n$ downto 2, aplică $Make\text{-}Arc\text{-}Consistent(Parent(X_j), X_j)$
3. For $j = 1$ to n , asignează X_j (o valoare consistentă din domeniu)

Probleme CSP cu o structură "aproape" arbore

Cutset conditioning



- ▶ Alege o submulțime S de variabile a.î. graful de constrangeri devine arbore după ștergerea lui S (S cutset)
- ▶ Pentru fiecare asignare a variabilelor din S , șterge din domeniul celorlalte variabile valori care sunt inconsistente cu asignarea; returnează cele două soluții.

Timpul de execuție $O(d^c \cdot (n - c)d^2)$, c dimensiune cutset (rapid pentru valori mici ale lui c)

Conținut

Introducere

Algoritmi de căutare

Îmbunătățirea algoritmului de căutare

Structura grafului de restricții

Căutare locală

Algoritmi euristici pentru CSP

Hill-climbing, Simulated annealing lucrează cu stări "complete" (toate variabilele asignate)

Pentru a aplica pe probleme CSP:

- permetem stări cu restricții nesatisfacute
- operatori care *reassignează* valori variabilelor

Selectarea variabilei: alege aleator o variabilă conflictuală

Selectarea valorii utilizând euristica **min-conflicts**:

- alege valoarea care încalcă cele mai puține restricții

$\min h(s) = \text{numărul de restricții violate}$

Min-conflicts

```
function MIN-CONFLICTS(csp, max_steps) returns a solution or failure
inputs: csp, a constraint satisfaction problem
         max_steps, the number of steps allowed before giving up

current  $\leftarrow$  an initial complete assignment for csp
for i = 1 to max_steps do
    if current is a solution for csp then return current
    var  $\leftarrow$  a randomly chosen conflicted variable from csp.VARIABLES
    value  $\leftarrow$  the value v for var that minimizes CONFLICTS(var, v, current, csp)
    set var = value in current
return failure
```

Exemplu: 4-Regine

Variabile: Q_1, Q_2, Q_3, Q_4 (câte o regină pentru fiecare coloană)

Domenii: $D_i = \{1, 2, 3, 4\}$ (pe ce linie se află fiecare regină)

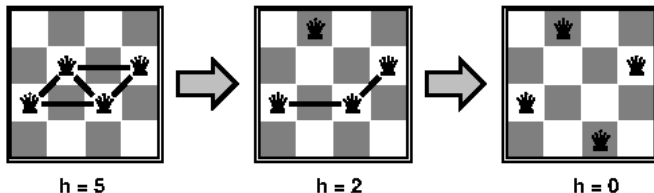
Constrângeri:

$Q_i \neq Q_j$ (nu pot fi pe aceeași linie)

$|Q_i - Q_j| \neq |i - j|$ (sau pe aceeași diagonală)

Exemplu: 4-Regine

- ▶ **Stare:** 4 regine pe 4 coloane ($4^4 = 256$ stări)
- ▶ **Operatori:** mută regina (pe coloană)
- ▶ **Testarea obiectivului:** nu există atacuri
- ▶ **Evaluare:** $h(s) = \text{numărul de atacuri}$



Compararea algoritmilor pentru CSP

Numărul mediu de verificări a consistenței necesare pentru a rezolva problema

Problem	Backtracking	BT+MRV	Forward Checking	FC+MRV	Min-Conflicts
USA	(> 1,000K)	(> 1,000K)	2K	60	64
<i>n</i> -Queens	(> 40,000K)	13,500K	(> 40,000K)	817K	4K
Zebra	3,859K	1K	35K	0.5K	2K
Random 1	415K	3K	26K	2K	
Random 2	942K	27K	77K	15K	

- ▶ Probleme de satisfacere a restricțiilor stări: asignări ale variabilelor restricții între variabile
- ▶ Backtracking
- ▶ Euristici de ordonare a variabilelor și selectare a valorilor
- ▶ *Forward-checking* previne asignări care garantează eșecul ulterior. Propagarea constrangerilor (*Arc consistency*) reduce domeniile.
- ▶ Problemele cu structură arborescentă pot fi rezolvate în timp liniar.
- ▶ Metodele de căutare locală (*Iterative min-conflicts*) sunt de obicei eficiente în practică.

- ▶ S. Russell, P. Norvig. Artificial Intelligence: A Modern Approach (3rd Edition). Prentice-Hall, Englewood Cliffs, NJ, 2010 (6. Constraint Satisfaction Problems)
- ▶ R. Bartak. Constraint Programming
- ▶ Solvers: CP Optimizer, OR-Tools