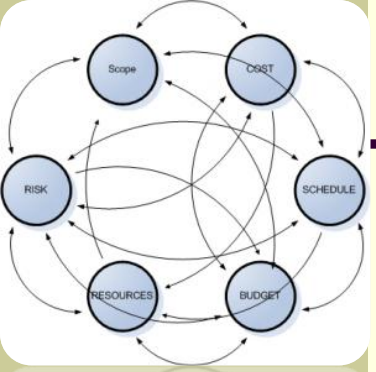


Caracteristici si constrangeri

Catalin Stoean

catalin.stoean@inf.ucv.ro

http://inf.ucv.ro/~cstoean

Caracteristici si stari

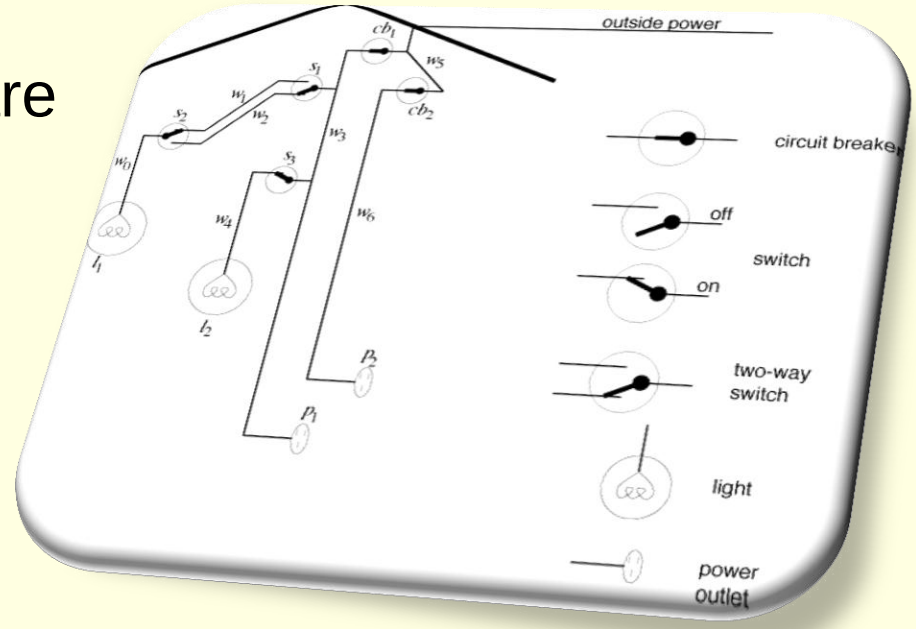
- Nu este mereu posibil ca un agent sa rationeze in functie de **stari** existente.
  - Majoritatea problemelor nu au o lista finita de stari.
  - Starile sunt descrise prin diverse **caracteristici**.
- Adesea este mai simplu sa descriem caracteristicile care construiesc starea decat sa enumeram toate starile.
- Fiecare caracteristica are un **domeniu** care este dat de multimea de valori pe care acea caracteristica le poate lua.

Caracteristici si stari

- Pentru o caracteristica binara, domeniul are doua doua valori.
  - 10 caracteristici binare descriu 2^{10}=1 024 stari.
  - 20 caracteristici binare descriu 2^{20}=1 048 576 stari.
  - 30 caracteristici binare descriu 2^{30}=1 073 741 824 stari.
- Sa lucram cu 20-30 de stari este posibil, dar sa avem de-a face cu peste 1 milion de stari este aproape imposibil.
- Caracteristicile nu sunt intotdeauna independente, exista adesea constrangeri asupra valorilor pentru diferite caracteristici.

Caracteristici si stari

- In circuitul de mai jos avem cate o carcteristica pentru:
  - Pozitia fiecarui switch care ne spune daca acesta este deschis sau inchis.
  - Fiecare lumina ce poate fi deschisa/inchisa.
  - Fiecare componenta care poate fi stricata sau nu.
- O **stare** consta din pozitia fiecarui switch, statusul fiecarei componente etc.



Lumi posibile, variabile si constrangeri

- **Problemele de satisfacere de constrangeri** vor fi descrise in termeni de lumi posibile.
- O **lume posibila** este modul in care o lume (reala sau imaginara) ar putea fi.
- Ex:
  - In reprezentarea unui rebus, lumile posibile corespund modului in care rebusul ar putea fi populat.
  - La circuitul din slide-ul precedent, o lume posibila specifica pozitia fiecarui switch si statusul fiecarei componente.

Lumi posibile, variabile si constrangeri

- **Ex**: Daca avem doua variabile A cu domeniul {0, 1, 2} si B cu domeniul {adevarat, fals}, atunci exista 6 lumi posibile l0, l1, …, l5:
- l0: A = 0 si B = adevarat;
- l1: A = 0 si B = fals;
- l2: A = 1 si B = adevarat;
- l3: A = 1 si B = fals;
- l4: A = 2 si B = adevarat;
- l5: A = 2 si B = fals.

Lumi posibile, variabile si constrangeri

- **Ex**: Un agent de turism trebuie sa planifice o serie de activitati pentru un grup de turisti.
  - Pot fi doua variabile pentru fiecare activitate
    - Una pentru data cu domeniul dat de o multime de zile pentru activitate.
    - Una pentru locatie cu domeniul dat de o multime de orase unde poate avea loc activitatea.
  - O lume posibila poate fi data de stabilirea pentru fiecare activitate a datei si orasului in care sa aiba loc.

Constrangeri

- O constrangere specifica combinatiile permise de asignari de valori pentru variabile.
- O **schema** sau **scop** este o multime de variabile.
- Un **tuplu** pe un scop S este o asociere a cate unei valori pentru fiecare variabila din S.
- O **constrangere** c pe un scop S este o multime de tupluri pe S.
- O lume posibila l satisface o multime de constrangeri daca, pentru fiecare constrangere, valorile asociate variabilelor din schema in l satisfac acea constrangere.
  - Spunem ca lumea l este un **model** pentru constrangeri.
  - Un model este o lume posibila care satisface toate constrangerile.

Constrangeri

- **Ex**:Fie o multime de activitati a, b, c, d si e.
- Acestea ar putea fi:
  - a - proiect de facut la IA
  - b – proiect de facut la BD
  - c – mers la film
  - d – instalare Dev-C++
  - e – reinstalare sistem de operare
- Presupunem ca fiecare actiune are loc la unul din momentele 1, 2, 3 sau 4.

a - proiect de facut la IA
b – proiect de facut la BD
c – mers la film
d – instalare Dev-C++
e – reinstalare SO

Constrangeri

- Putem avea genul urmator de constrangeri:
  - Sistemul de operare trebuie reinstalat inainte de instalare Dev-C++ si de a face proiectele la IA si BD
  - Proiectele la IA si BD trebuie facute in zile diferite
  - Dev-C++ trebuie instalat chiar in aceeasi zi cu realizarea proiectului la IA
  - Intai la tenis, apoi instalarea Dev-C++
  - s.a.m.d.

a - proiect de facut la IA
b – proiect de facut la BD
c – mers la film
d – instalare Dev-C++
e – reinstalare SO

Constrangeri

- Presupunem ca fiecare actiune are loc intr-una din zilele 1, 2, 3 sau 4.
- Fie A variabila care reprezinta timpul la care are loc activitatea a s.a.m.d. pt celelalte activitati.
- Domeniile vor fi: dom(A) = dom(B)=dom(C)=dom(D)=dom(E) = {1, 2, 3, 4}
- Avem urmatoarele constrangeri: {(B≠3), (C≠2), (A≠B), (B≠C), (C<D), (A=D), (E<A), (E<B), (E<C), (E<D), (B≠D)}
- Gasiti un model pentru aceste constrangeri.

Constrangeri

- In exemplul cu agentia de turism, pot sa existe constrangeri precum:
  - Anumite activitati trebuie sa se desfasoare in zile diferite;
  - Alte activitati trebuie sa aiba loc in acelasi oras si aceeasi zi;
  - Anumite activitati trebuie sa aiba loc inaintea altora;
  - Trebuie sa fie un numar de zile intre doua activitati;
  - Nu pot fi trei activitati in trei zile consecutive, etc.

Probleme de satisfacere de constrangeri (PSC)

- O PSC consta din:
  - o multime de variabile,
  - un domeniu pentru fiecare variabila si
  - o multime de constrangeri.
- Scopul este de a alege o valoare pentru fiecare variabila a.i. lumea posibila rezultata satisface constrangerile.
  - Dorim un model al constrangerilor.
- Determinarea daca exista un model pentru o PSC este o problema NP-hard.
  - Nu exista algoritm care sa nu aiba complexitate exponentiala pentru instante dificile ale PSC.

Algoritmi care genereaza-si-testeaza

- **Spatiul asocierilor** D este multimea tuturor asocierilor de valori pentru toate variabilele.
- Se verifica toate asocierile posibile dintre variabile si valori.
- Daca se gaseste o asociere care satisface constrangerile, aceasta este returnata.
- Pentru exemplul din slide-urile 9-11 spatiul asocierilor este:
- D = { {A = 1, B = 1, C = 1, D = 1, E = 1},

  {A = 1, B = 1, C = 1, D = 1, E = 2}, …,

  {A = 4, B = 4, C = 4, D = 4, E = 4} }.

Algoritmi care genereaza-si-testeaza

- In exemplul anterior avem |D| = 4^5 = 1024 asocieri diferite de testat.
- Daca domeniul fiecarei din cele n variabile are d componente, atunci D are d^n componente.
- Daca avem si c constrangeri, numarul total de constrangeri testate este $O(cd^n)$.
- Daca n creste, algoritmul acesta devine repede imposibil de utilizat.

Rezolvarea PSC folosind cautarea

- Unele constrangeri pot fi testate inainte de a asocia valori pentru toate variabilele.
  - Daca o asociere partiala este inconsistenta cu o constrangere, atunci asocierea completa care o extinde pe cea partiala va fi inconsistenta.
- Cum in exemplul din slide-urile 9-11 asocierile A = 1 si B = 1 sunt inconsistente pentru ca avem constrangerea A ≠ B, nu are rost sa mai asociem valori pentru celelalte variabile.

Rezolvarea PSC folosind cautarea

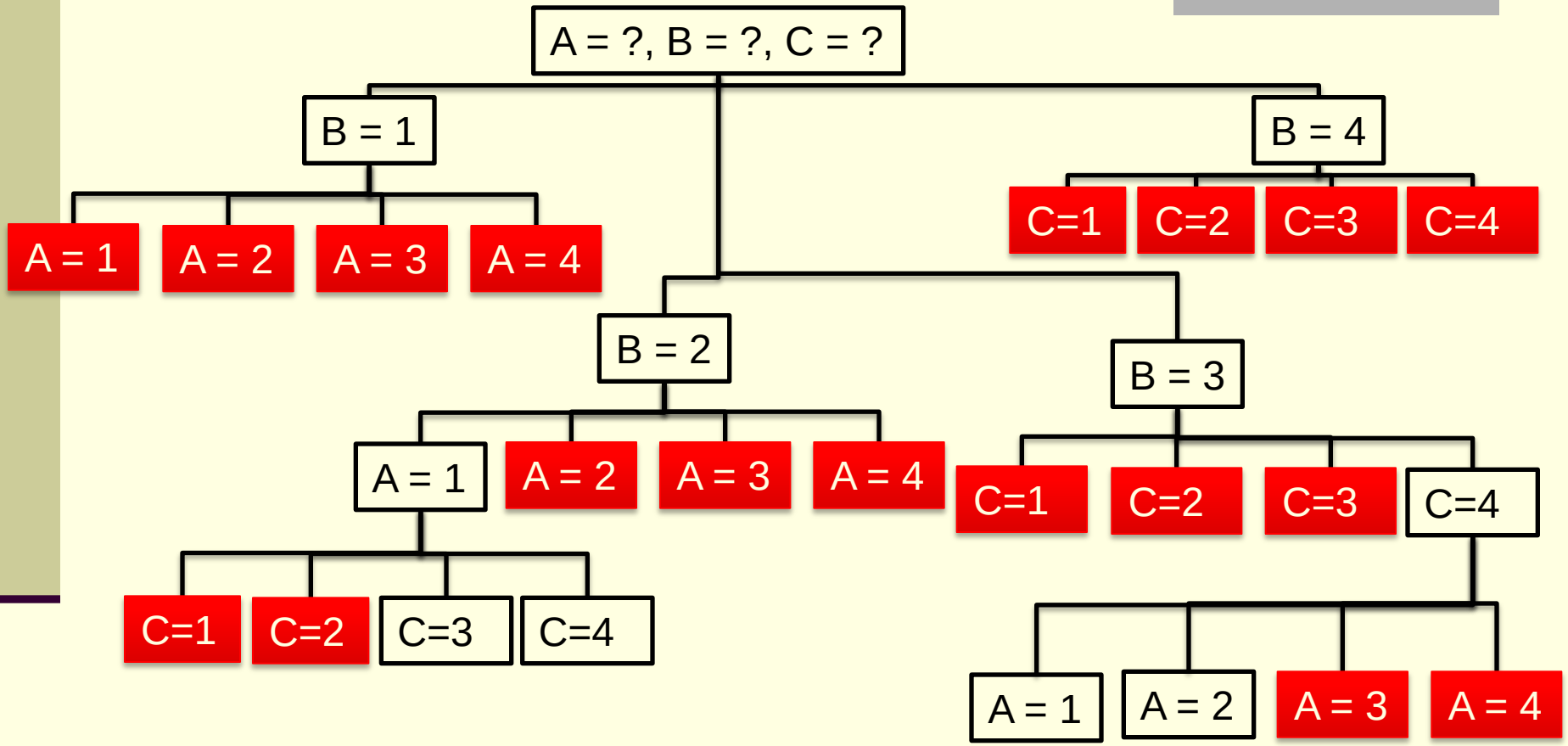
- O alternativa este sa construim un spatiu de cautare in care sa utilizam metode de cautare din cursurile precedente.
  - Nodurile sunt asocieri de valori la subseturi de variabile.
  - Nodul de start este dat doar de variabile, fara vreo asociere.
  - Vecinii unui nod N sunt obtinuti prin selectarea unei variabile V care nu are valori in nodul N si asocierea tuturor valorilor posibile pentru ea care satisfac constrangerile.
  - Nodul tinta este de a asocia valori pentru toate variabilele a.i. constrangerile sa fie respectate.

Rezolvarea PSC folosind cautarea

- Ex: Avem 3 variabile A, B si C, fiecare cu domeniul {1, 2, 3, 4}.
- Constrangeri: A < B si B < C.

- Un nod din cadrul arborelui de cautare corespunde la toate asocierile de la nodul radacina pana la acel nod.
- Nodurile la care cautarea se opreste pentru ca nu este satisfacuta vreo constrangere sunt colorate cu rosu.

Rezolvarea PSC folosind cautarea



- Avem 4 solutii ale problemei.

Rezolvarea PSC folosind cautarea

- Folosind algoritmul genereaza-si-testeaza am fi avut 4^3 = 64 de asocieri posibile.
- Pentru algoritmul de cautare folosit avem numai 22 de asocieri generate.
- Era posibila si obtinerea altui arbore, in functie de variabila cu care continuam.
  - Un arbore care ar fi inceput cu A, apoi continua cu B si C ar fi generat mai multe noduri.
  - Am inceput cu B, variabila care aparea in ambele constrangeri.

Tema – termen o saptamana

- Fie o problema de criptaritmetica

```
   TREI +
    DOI
-----------
   CINCI
```

Exemplu solutie:

```
  9760
   450
---------
 10210
```

- Scrieti un program C care sa rezolve problema folosind un algoritm de cautare studiat anterior(de exemplu, latime).
  - Daca functioneaza pentru orice puzzle, aveti 2 puncte.
  - Daca functioneaza numai pentru un puzzle fixat: 1 punct.

Algoritmi de consistenta

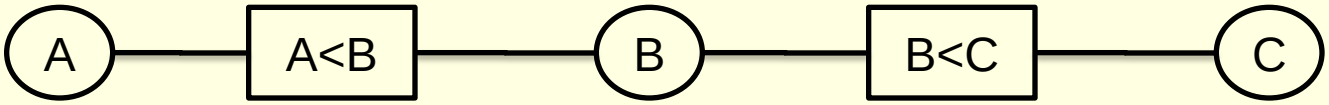
- In ex. precedent, variabilele A si B sunt legate de constrangerea A < B.
  - Asocierea A = 4 este inconsistenta cu fiecare din asocierile posibile pentru B pentru ca dom(B) = {1, 2, 3, 4}.
    - In cautarea precedenta acest lucru era descoperit pentru diverse asocieri ale variabilelor B si C.
    - Aceasta ineficienta poate fi evitata daca stergem 4 din dom(A).
      - Aceasta este ideea de baza de la algoritmii de consistenta.

Algoritmi de consistenta

- Se lucreaza asupra unei retele de constrangeri formata de PSC:
  - Avem un nod pentru fiecare variabila.
    - Aceste noduri se deseneaza intr-o forma rotunda.
  - Avem un nod pentru fiecare constrangere.
    - Desenam aceste noduri ca dreptunghiuri.
  - Asociem pentru fiecare variabila X o multime D_X de valori posibile.
    - Aceasta multime este initial egala cu domeniul variabilei.
  - Pentru fiecare constrangere c si pentru fiecare variabila X din scopul lui c, avem un arc <X, c>.

Algoritmi de consistenta

- Consideram exemplul din slide-ul 18.
  - Avem 3 variabile A, B si C, fiecare cu domeniul {1, 2, 3, 4}.
  - Constrangerile: A < B si B < C.
- Reteaua de constrangeri va fi:



- Cele patru arce vor fi:
  - ⟨ A, A < B ⟩
  - ⟨ B, A < B ⟩
  - ⟨ B, B < C ⟩
  - ⟨ C, B < C ⟩

Algoritmi de consistenta

- Constrangerea X ≠ 3 are un singur arc:
  - <X, X ≠ 3 >
- Constrangerea X + Y = Z are trei arce:
  - <X, X + Y = Z >
  - <Y, X + Y = Z >
  - <Z, X + Y = Z >

Algoritmi de consistenta

- Cel mai simplu caz este cand constrangerea are o singura variabila in scopul sau.
  - In acest caz, arcul este **consistent cu domeniul** daca fiecare valoare a variabilei este consistenta cu domeniul.
- Constrangerea $X \neq 3$ are scopul {X}.
  - Arcul $<X, X \neq 3>$ nu este consistent cu domeniul $D_X = \{1, 2, 3, 4\}$ pentru ca $X = 3$ incalca constrangerea.
  - Daca eliminam 3 din domeniul D_X, atunci arcul $<X, X \neq 3>$ este consistent cu domeniul.

Algoritmi de consistenta

- Fie constrangerea c cu scopul {X, Y1, Y2, …, Yk}.
- Arcul <X, c> este un **arc consistent** daca, pentru fiecare valoare x din D_X, exista valorile y1, …, yk, unde yi $\in$ D_{yi}, a.i. c(X = x, Y1 = y1, …, Yk = yk) este satisfacuta.
- O retea este consistenta daca toate arcele sale sunt **arce consistente**.
- Ex: In ex. din slide-ul 24, niciun arc nu este arc consistent.
  - $\langle$ A, A < B $\rangle$ nu este arc consistent pentru ca pentru A = 4 nu exista valoare corespunzatoare pentru B a.i. A < B.
    - Daca scoatem 4 din Dom(A), atunci obtinem arc consistent.

Algoritmi de consistenta

- Daca un arc <X, c> nu este un arc consistent, exista valori pentru X pentru care nu exista valori pentru Y1, …, Yk a.i. constrangerile sa fie respectate.
  - In acest caz, toate valorile lui X din D_X pentru care nu exista valori corespunzatoare pentru celelalte variabile pot fi sterse din D_X pentru a face arcul <X, c> consistent.

Algoritmi de consistenta

functia consistenta_arce(*V, dom, C*) ***intoarce*** domeniile variabilelor

Pentru fiecare variabila *X*

$D_X = dom\{X\}$

$Arce = \{<X, c> | c \in C$ si $X \in scop(c)\}$

Cat timp $Arce \neq \{\}$

Scoate $<X, c>$ din *Arce*

Noul domeniu

$ND_X = \{x | x \in D_X$ si exista $\{X = x, Y1 = y1, ..., Yk = yk\}$ in c,

unde $yi \in D_{yi}$ pentru orice i de la 1 la $k\}$

Daca $ND_X \neq D_X$ atunci

$Arce = Arce \cup \{<Z, c'> | X \in scop(c'), c' \neq c,$

$Z \in scop(c') \backslash \{X\}\}$

$D_X = ND_X$

Intoarce $\{D_X |$ *X* este variabila$\}$

Algoritmi de consistenta

- Algoritmul face toata reteaua consistenta prin considerarea arcelor din *Arce*.
- Initial *Arce* contine toate arcele grafului.
- Daca arcul curent <*X*, *c*> nu este consistent, el devine consistent prin modificarea domeniului variabilei *X*. (*ND*)
  - Toate arcele **facute anterior consistente** care ar putea prin modificarea domeniului lui *X* sa devina inconsistente sunt adaugate inapoi in *Arce*.
    - Acestea sunt arcele <*Z*, *c*'>, unde *c*' este o constrangere diferita de *c* care implica *X*, iar *Z* este o variabila diferita de *X* care apare in *c*'.

Algoritmi de consistenta

$D_A = \{1, 2, 3, 4\}$
$D_B = \{1, 2, 3, 4\}$
$D_C = \{1, 2, 3, 4\}$

- Fie exemplul din slide-ul 24.
  - Avem 3 variabile *A, B* si *C*, fiecare cu domeniul {1, 2, 3, 4}.
  - Constrangerile: $A < B$ si $B < C$.
  - *Arce* = $\{\langle A, A < B \rangle, \langle B, A < B \rangle, \langle B, B < C \rangle, \langle C, B < C \rangle\}$
- Selectam $\langle A, A < B \rangle$ si il scoatem din *Arce*.
  - Pentru $A = 4$, nu exista valoare pentru B a.i. A < B.
  - Deci $ND_A = \{1, 2, 3\}$.
  - Nu avem ce sa adaugam la *Arce* pentru ca nu am scos anterior niciun alt arc din aceasta multime.

Algoritmi de consistenta

$D_A = \{1, 2, 3\}$
$D_B = \{1, 2, 3, 4\}$
$D_C = \{1, 2, 3, 4\}$

- *Arce* = {⟨ *B, A < B* ⟩, ⟨ *B, B < C* ⟩, ⟨ *C, B < C* ⟩}
- Eliminate = {⟨ *A, A < B* ⟩}
- Selectam ⟨ *B, A < B* ⟩ si il scoatem din *Arce*.
  - Valoarea 1 poate fi scoasa din domeniul lui B.
  - $ND_B = \{2, 3, 4\}$
  - Arcul eliminat anterior nu este afectat de aceasta modificare, deci nu adaugam nimic nou la *Arce*.

Algoritmi de consistenta

$D_A = \{1, 2, 3\}$
$D_B = \{2, 3, 4\}$
$D_C = \{1, 2, 3, 4\}$

- *Arce* = {⟨ *B, B* < *C* ⟩, ⟨ *C, B* < *C* ⟩}
- Eliminate = {⟨ *A, A* < *B* ⟩, ⟨ *B, A* < *B* ⟩}
- Selectam ⟨ *B, B* < *C* ⟩ si il scoatem din *Arce*.
  - Valoarea 4 poate fi scoasa din domeniul lui B.
  - $ND_B = \{2, 3\}$
  - Aceasta schimbare afecteaza posibil domeniul lui A, deci readaugam in multimea *Arce* arcul ⟨ *A, A* < *B* ⟩.

Algoritmi de consistenta

$D_A = \{1, 2, 3\}$
$D_B = \{2, 3\}$
$D_C = \{1, 2, 3, 4\}$

- *Arce* = {⟨ *A, A* < *B* ⟩, ⟨ *C, B* < *C* ⟩}
- Eliminate = {⟨ *B, B* < *C* ⟩, ⟨ *B, A* < *B* ⟩}
- Selectam ⟨ *A, A* < *B* ⟩ si il scoatem din *Arce*.
  - Valoarea 3 poate fi scoasa din domeniul lui A.
  - $ND_A = \{1, 2\}$
  - Aceasta schimbare nu afecteaza un alt arc scos anterior, deci in multimea *Arce* nu adaugam nimic nou.

Algoritmi de consistenta

$D_A = \{1, 2\}$
$D_B = \{2, 3\}$
$D_C = \{1, 2, 3, 4\}$

- *Arce* = $\{\langle C, B < C \rangle\}$
- Eliminate = $\{\langle A, A < B \rangle, \langle B, B < C \rangle, \langle B, A < B \rangle\}$
- Ultimul arc din multimea *Arce* este $\langle C, B < C \rangle$.
  - Valoarile 1 si 2 pot fi scoase din domeniul lui C.
  - $ND_C = \{3, 4\}$
  - Aceasta schimbare nu afecteaza un alt arc scos anterior, deci in multimea *Arce* nu adaugam nimic nou.
- Domeniile obtinute vor fi:

$D_A = \{1, 2\}$
$D_B = \{2, 3\}$
$D_C = \{3, 4\}$

Problema nu este complet rezolvata, insa este semnificativ simplificata.

Algoritmi de consistenta

- Exc.: Aplicati algoritmul de consistenta din slide-ul 29 pentru rezolvarea problemei din slide-urile 9-11, reluata mai jos.

- $D_A = D_B = D_C = D_D = D_E = \{1, 2, 3, 4\}$
- Avem urmatoarele constrangeri: {($B \neq 3$), ($C \neq 2$), ($A \neq B$), ($B \neq C$), ($C < D$), ($A = D$), ($E < A$), ($E < B$), ($E < C$), ($E < D$), ($B \neq D$)}

- Solutia este unica (rezolvare la tabla).

A1, D1	D2	D3
A2		
A3		

Algoritmi de consistenta

- Exc.: Fie puzzle-ul de mai sus. Sa se gaseasca 6 cuvinte de cate 3 litere (3 pe orizontala si 3 pe verticala). Fiecare trebuie sa apartina listei de mai jos:
  - add, ado, age, ago, aid, ail, aim, air, and, any, ape, apt, arc, are, ark, arm, art, ash, ask, auk, awe, awl, aye, bad, bag, ban, bat, bee, boa, ear, eel, eft, far, fat, fit, lee, oaf, rat, tar, tie.
- Incercati sa-l rezolvati realizand o retea consistenta.
  - Puteti reprezenta ca variabile pozitiile cuvintelor, A1, A2, A3, D1, D2 si D3, avand multimi de cuvinte ca valori posibile.
    - Constrangerile sunt date de faptul ca litera este aceeasi unde se intersecteaza cuvintele.

Un punct la examenul final!

A1, D1	D2	D3
A2		
A3		

Algoritmi de consistenta

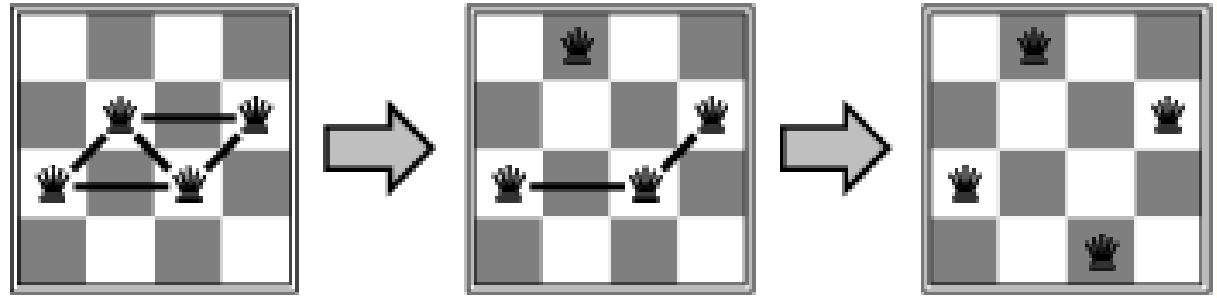
- Exc.: Fie puzzle-ul de mai sus. Sa se gaseasca 6 cuvinte de cate 3 litere (3 pe orizontala si 3 pe verticala). Fiecare trebuie sa apartina listei de mai jos:
  - add, ado, age, ago, aid, ail, aim, air, and, any, ape, apt, arc, are, ark, arm, art, ash, ask, auk, awe, awl, aye, bad, bag, ban, bat, bee, boa, ear, eel, eft, far, fat, fit, lee, oaf, rat, tar, tie.
- Incercati sa-l rezolvati realizand o retea consistenta.
  - Alta posibilitate este data de utilizarea celor 9 patratele ca variabile. Domeniul fiecarei variabile este dat de literele alfabetului.
    - Constrangerile sunt date de faptul ca exista un cuvant in lista care contine literele respective.

Algoritmi de cautare locala

- In multe probleme de optimizare, **drumul** catre tinta este irelevant.
- Spatiul starilor este o multime de configuratii complete, o multime de **potentiale solutii**.
- La unele probleme, trebuie gasite configuratii (solutii) care sa satisfaca contrangeri, de exemplu in problema damelor.
- In astfel de cazuri, putem folosi **algoritmi de cautare locala**.
- In astfel de situatii avem o singura **stare curenta** careia i se aduc **imbunatatiri**.

Exemplu: problema celor *n* dame

- Problema este de a pune *n* dame pe o tabla *n* × *n* in asa fel incat doua dame sa nu se afle pe aceeasi line, pe aceeasi coloana sau pe aceeasi diagonala.



- **Stari**: 4 dame in 4 coloane ($4^4 = 256$ de stari posibile)
- **Actiuni**: muta o dama pe coloana
- **Stare tinta**: sa nu se atace damele reciproc
- ***h*(*n*)** = numarul de perechi de dame care se ataca reciproc

Hill climbing

- Este ca si cand ai urca un munte, este ceata foarte deasa si ai avea amnezie.
- Este vorba de o miscare continua inspre valori mai bune, mai mari (de aici, *urcusul pe munte*).
- Algoritmul nu mentine un arbore de cautare, prin urmare, pentru fiecare nod se retine numai starea pe care o reprezinta si evaluarea sa.

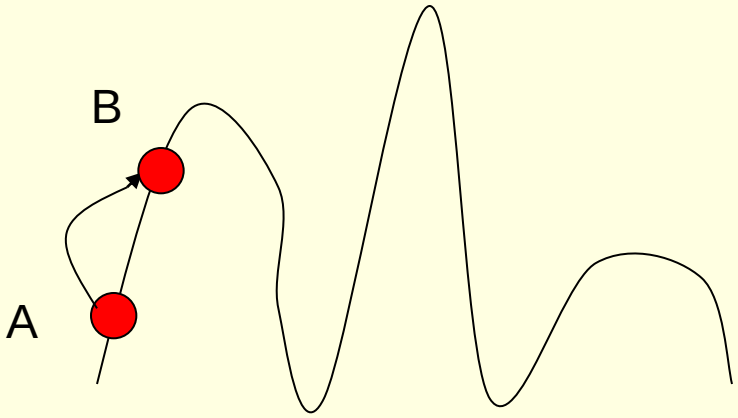
Hill climbing

functia hill_climbing(*problema*) ***intoarce*** o solutie
Se pastreaza la fiecare reapelare: nodul *curent* si nodul *urmator.*

curent = genereaza_nod(stare_initiala[problema])

Cat timp este posibil *executa*
    urmator = succesorul cel mai bun al nodului *curent*
    Daca eval(*urmator*) < eval(*curent*) *atunci*
        intoarce *curent*
    curent = urmator
Sfarsit cat timp

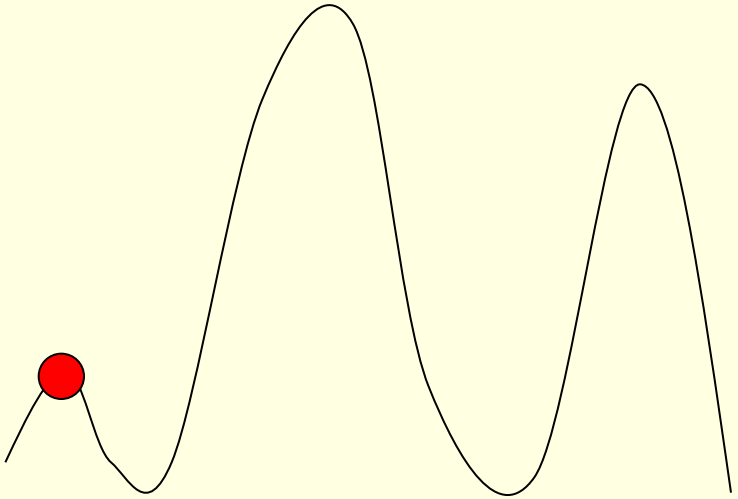
Hill climbing



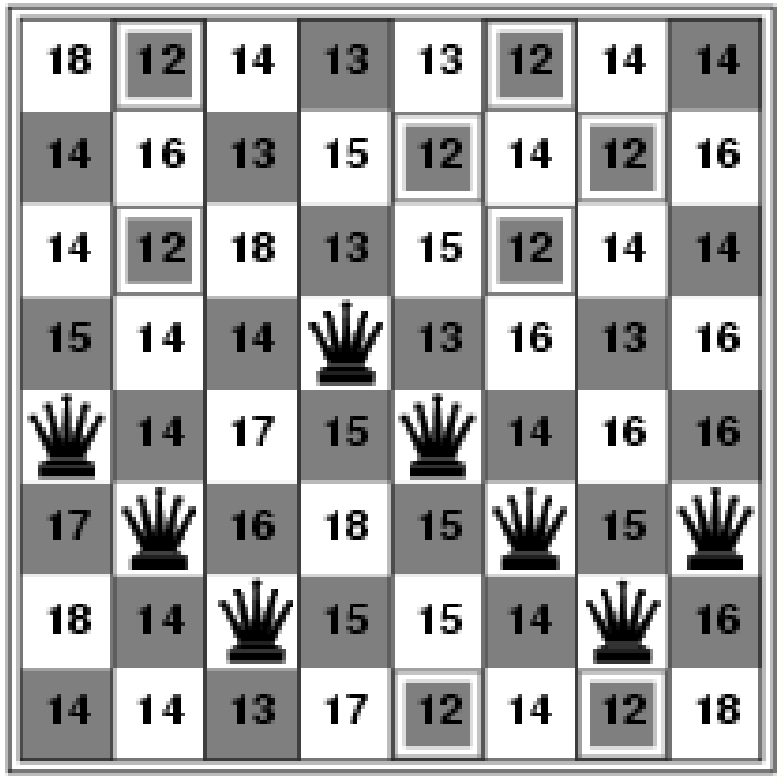
- Din starea curenta A => starea B.
- Atunci cand sunt mai multi succesori care sunt toti la fel de buni, algoritmul il alege pe unul in mod aleator.

Dezavantaje hill climbing

- **Maxime locale**: este vorba de un varf care este mai mic decat cel mai inalt varf din spatiul starilor. Cand se ajunge la maxime locale, algoritmul se opreste pentru ca nu mai poate *cobori* dealul.
  - Solutia gasita poate fi foarte slaba!

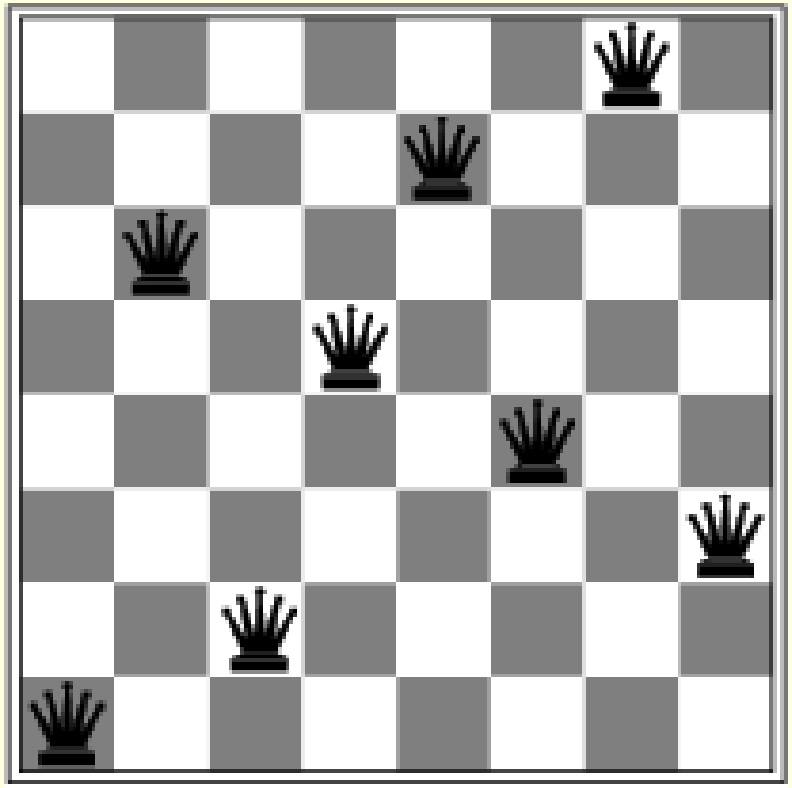


Hill-climbing - problema celor 8 dame



- *h* = numarul de perechi de dame care se ataca reciproc direct sau indirect.
- *h = 17* pentru starea de mai sus.

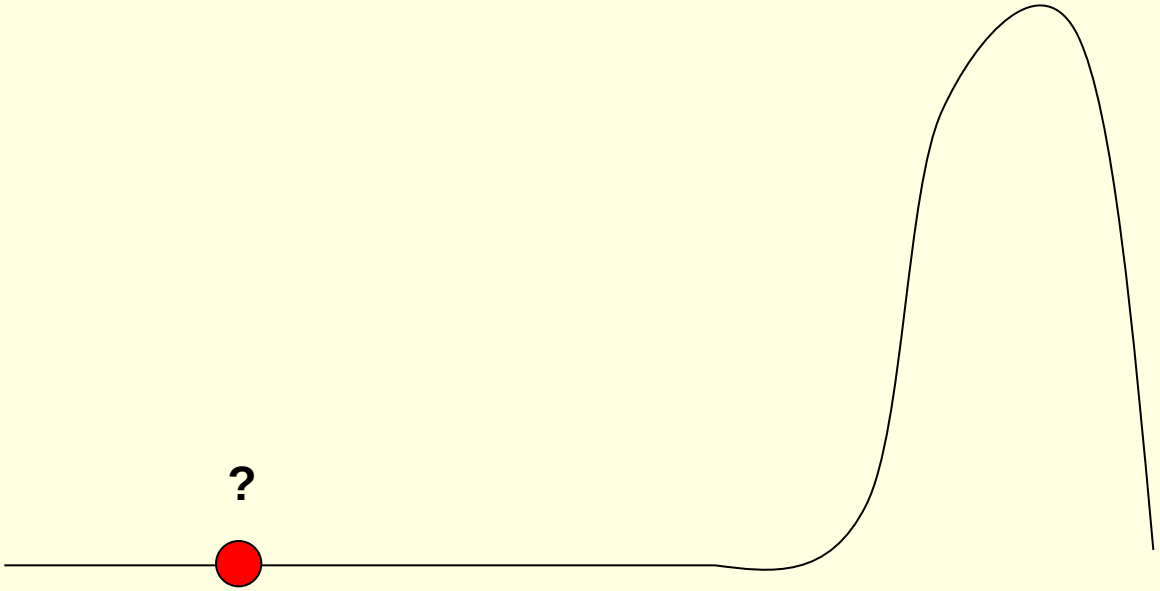
Hill-climbing problema celor 8 dame



- Un minim local unde *h = 1*

Dezavantaje hill climbing

- **Platouri** – o zona din spatiul de cautare in care functia de evaluare are valori constante.
- Cautarea va merge in aceste cazuri la intamplare.



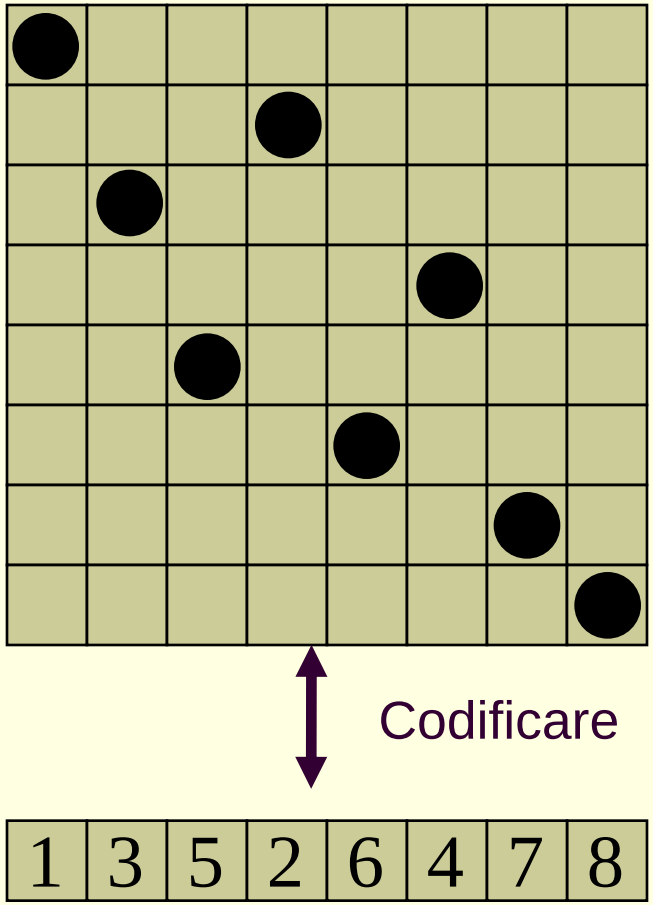
Hill climbing cu restart aleator

- Cand apar astfel de situatii in care cautarea nu realizeaza nici un progres, un lucru bun ar fi sa se reinceapa cautarea de la un alt punct de start.
- **Hill climbing cu restart aleator** face o serie de cautari folosind hill climbing cu porniri din diverse stari aleatoare.
- Fiecare rulare dureaza pana cand cautarea nu mai inregistreaza imbunatatiri sau a trecut un numar de iteratii.
- Cel mai bun rezultat din fiecare cautare este retinut.
- Se poate repeta pentru un numar fixat de iteratii sau se poate continua pana cand cel mai bun rezultat obtinut nu a mai fost imbunatatit de mai multe generatii.

Reprezentarea pentru problema damelor

Potenţială soluţie:
o configuraţie a celor 8 dame

Cromozomul:
o permutare a primelor 8 cifre.



Reprezentarea pentru problema damelor

- Considerăm pentru început o configuraţie pe care o generăm aleator.

Cum?

Initializarea unei posibile solutii

functie initializare() intoarce configuratie

M = {1, 2, 3, 4, 5, 6, 7, 8}

i = 0

Pe scurt:
g = rand() % lungime(M)

Cat timp (lungime(M) > 0) executa

g = genereaza numar aleatoriu intre 0 si lungime(M)-1

configuratie[i++] = M[g]

Scoate elementul M[g] din multimea M

lungime(M)--;

Sfarsit cat timp

intoarce **configuratie**

A se vedea slide-ul urmator pentru C

Cum generam in limbajul *C* un intreg *g* aleator intre 0 si *n*-1

```
include <time.h>
include <stdlib.h>
int main(){
…
time_t t;
srand((unsigned) (time(&t)));
int g = rand() % n;
…
}
```

Evaluarea unei configuratii

```
functie evaluare(configuratie) intoarce numar_erori
    erori = 0;
    Pentru i = 1 pana la 7 executa
        Pentru j = i + 1 pana la 8 executa
            Daca (|configuratie[i] - configuratie[j]| == |i - j|) atunci
                erori++
            Sfarsit daca
        Sfarsit pentru
    Sfarsit pentru
Intoarce erori
```

Schimbarea pentru problema damelor

- O mică variaţie într-o permutare:
  - Se aleg două valori în mod aleator (5 şi 7 în imaginea din stânga).
  - Poziţiile celor două valori sunt interschimbate.

Schimbarea pentru problema damelor

```
functie perturbare(configuratie) intoarce configuratie
    x = rand() % lungime(configuratie)
    y = rand() % lungime(configuratie)
    Cat timp (x == y) executa
        y = rand() % lungime(configuratie)
    Sfarsit cat timp
    temp = configuratie[x]
    configuratie[x] = configuratie[y]
    configuratie[y] = temp
intoarce configuratie
```

Hill climbing pentru problema celor 8 dame

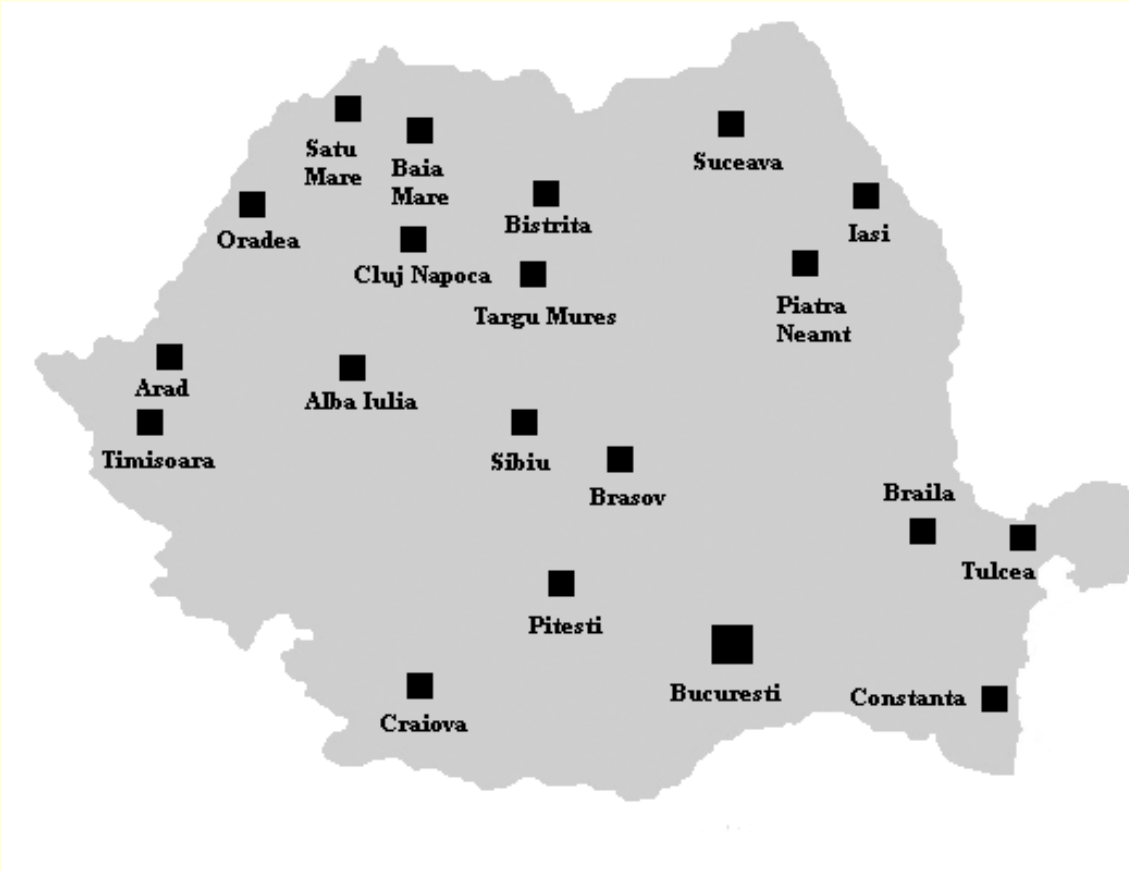
1. configuratie = initializare();
2. Pentru un numar de iteratii
    1. eval_curent = evaluare(configuratie)
    2. configuratie1 = perturbare(configuratie)
    3. if(eval(configuratie1) < eval(configuratie))
        1. configuratie = configuratie1
3. Sfarsit pentru
4. Afisare (configuratie)

Nu uitati ca valorile lui *eval* trebuie minimizate!!

Transformati acest algoritm intr-un hill climbing cu restart aleator!

Problema comis voiajorului

- Problema:
  - Se dau *n* oraşe
  - Să se găsească un tur complet de lungime minimală
- Reprezentare:
  - Etichetăm oraşele 1, 2, … , *n*
  - Un tur complet este o permutare (pt. *n* =4: [1,2,3,4], [3,4,2,1])
- Spaţiul de căutare este **imens**: pentru 30 de oraşe sunt 30! ≈ 10^{32} tururi posibile!



Distantele in km dintre orase

n = 20

1 Bucuresti
2 Satu Mare
3 Baia Mare
4 Oradea
5 Arad
6 Timisoara
7 Alba Iulia
8 Cluj Napoca
9 Bistrita
10 Targu Mures
11 Sibiu
12 Brasov
13 Pitesti
14 Craiova
15 Suceava
16 Piatra Neamt
17 Iasi
18 Braila
19 Tulcea
20 Constanta

Distanta de la Bucuresti la Satu Mare

596 550 574 555 538 394 426 419 330 282 161 126 248 436 349 406 213 278 225
67 135 250 304 331 170 216 271 333 434 485 544 369 429 463 660 752 815
183 298 352 303 146 148 219 305 388 457 516 326 387 420 618 710 768
115 169 278 147 263 249 311 412 463 463 478 444 538 671 763 792
52 239 263 378 350 273 415 429 394 593 531 646 674 766 780
217 316 417 327 256 399 406 353 575 509 691 651 743 758
160 200 116 113 232 268 293 358 292 407 490 583 612
119 101 163 264 315 374 334 297 390 523 615 644
89 200 257 352 411 214 247 308 486 578 638
112 168 262 321 261 195 310 426 519 548
142 155 236 358 289 441 401 493 507
149 205 319 228 299 258 350 380
123 468 378 448 318 404 351
524 434 504 434 504 451
122 144 341 433 520
131 254 346 432
271 364 434
92 178
124

Distanta de la Braila la Tulcea

Hill climbing

functia hill_climbing(*problema*) ***intoarce*** o solutie
Se pastreaza la fiecare reapelare: nodul *curent* si nodul *urmator.*

curent = genereaza_nod(stare_initiala[problema])

Cat timp este posibil *executa*
urmator = succesorul cel mai bun al nodului *curent*
Daca eval(*urmator*) < eval(*curent*) *atunci*
intoarce *curent*
curent = urmator
Sfarsit cat timp

- Cum generam nodul initial?
- Generam in mod aleator o permutare a primelor 5 numere.
  - Exemplu:

[1, 3, 5, 4, 2]

- Cat de buna este solutia generata?

dist(1, 3) + dist(3, 5) + dist(5, 4) + dist(4, 2) + dist(2, 1) = 406 + 691 + 316 + 426 + 161 = **2000**

Initializarea unei posibile solutii

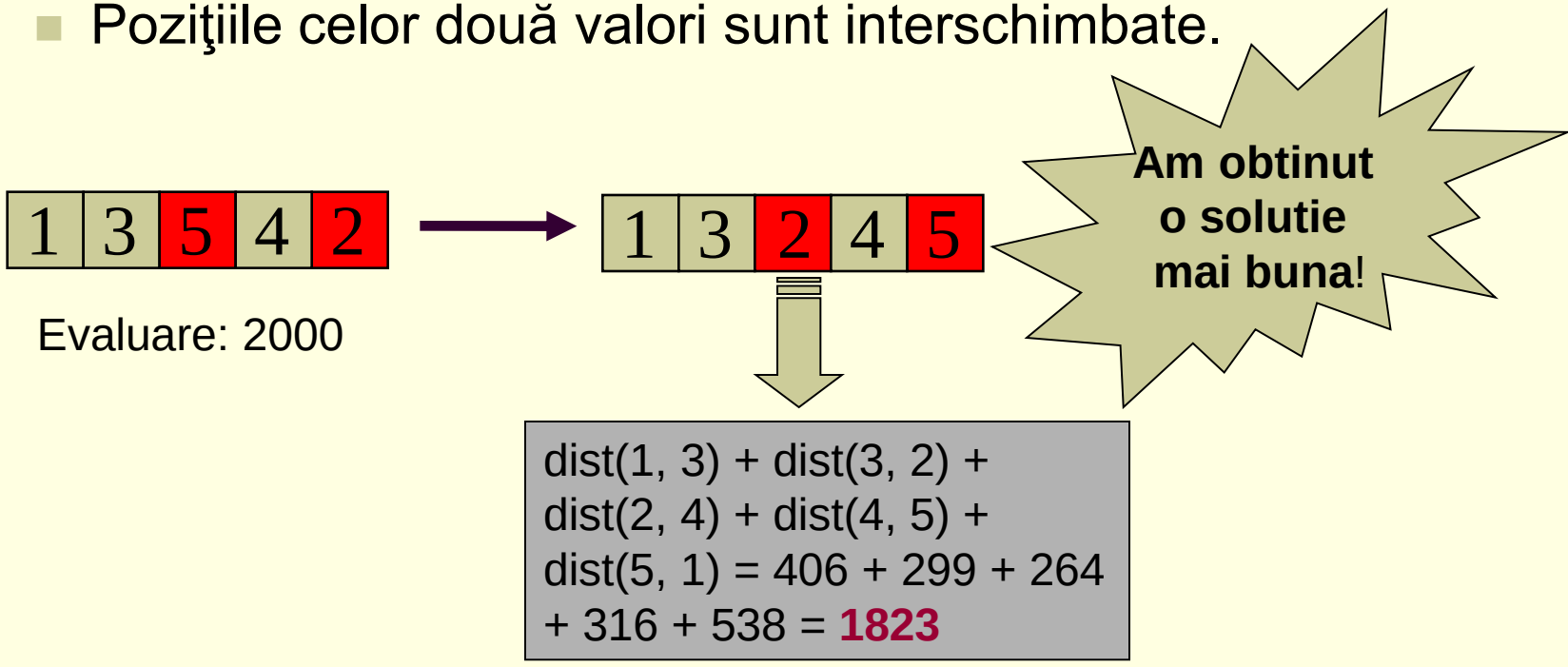
```
functie initializare() intoarce configuratie
    M = {1, 2, 3, 4, 5}
    i = 0
    Cat timp (lungime(M) > 0) executa
        g = rand() % lungime(M)
        configuratie[i++] = M[g]
        Scoate elementul M[g] din multimea M
        lungime(M)--;
    Sfarsit cat timp
intoarce configuratie
```

Functia de evaluare

```
functie eval(configuratie) intoarce distanta_totala
    distanta_totala = 0;
    Pentru i = 1 pana la n-1 executa
                    distanta_totala = distanta_totala +
                                distanta(configuratie[i], configuratie[i+1])
            Sfarsit pentru
            distanta_totala = distanta_totala +
                                distanta(configuratie[n], configuratie[1])
    Sfarsit pentru
intoarce distanta_totala
```

Variatie

- O mică variaţie într-o permutare:
  - Se aleg două valori în mod aleator (5 şi 2 în imaginea din stânga).
  - Poziţiile celor două valori sunt interschimbate.



Variatie

pm este un parametru din intervalul (0, 1) care da probabilitatea de a modifica un element din configuratie.

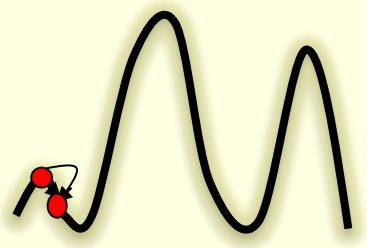
```
functie variatie() intoarce configuratie
        Pentru j = 0 pana la lungime(configuratie) executa
            p = random(1)
            Daca (p < pm) atunci
                x = rand() % lungime(configuratie)
                Cat timp (x == j) executa
                    x = rand() % lungime(configuratie)
                Sfarsit cat timp
                temp = configuratie[x]
                configuratie[x] = configuratie[j]
                configuratie[j] = temp
            Sfarsit daca
        Sfarsit pentru
intoarce configuratie
```

Hill climbing pentru problema comis-voiajorului

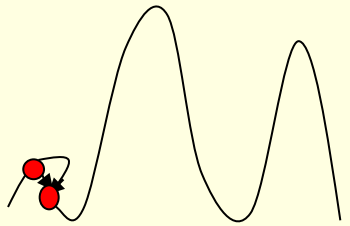
1. configuratie = initializare();
2. Executa
   1. eval_curent = eval(configuratie)
   2. configuratie1 = variatie(configuratie)
3. Cat timp (eval(configuratie1) < eval(configuratie))
4. Afisare (configuratie)

Nu uitati ca valorile lui *eval* trebuie minimizate!!

Simulated annealing

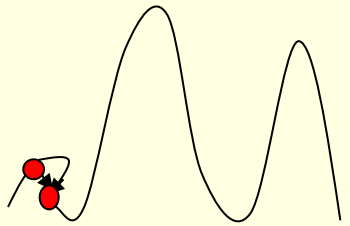


Simulated annealing



- In traducere *decalire simulata*.
  - Termenul vine din metalurgie, unde metalele sunt racite usor pentru a le face sa ajunga la o stare cand sunt foarte solide.
- In cadrul IA, mutarile aleatorii corespund temperaturilor ridicate.
  - La temperaturi joase, aleatoriul este scazut.
- Simulated annealing este un proces in care temperatura se reduce usor, incepand de la o cautare aleatorie la temperaturi ridicate si ajungand un algoritm greedy cand temperatura se apropie de 0.
- La temperaturi ridicate, pasi spre mai rau sunt mai probabili pentru a evita optime locale.

Simulated annealing



- In loc de a reincepe cautarea dintr-o noua stare generata aleator, procesul de cautare permite si *coboarea de pe munte* pentru a scapa de optime locale – acest lucru este permis de catre **simulated annealing**.
- In loc sa fie alese cele mai bune actiuni, sunt alese si miscari in mod aleator.
- Daca actiunea imbunatateste situatia, atunci este intotdeauna executata.
  - Altfel, algoritmul alege o actiune cu o anumita probabilitate mai mica decat 1.
  - Aceasta probabilitate descreste exponential cu cat de rea este actiunea (cu ΔE, care spune cu cat este inrautatita solutia).

Simulated annealing

- *T* este un parametru *temperatura* folosit pentru a determina probabilitatea de a selecta actiunea mai rea.
- Cu cat *T* este mai mare, cu atat actiunile proaste au sanse mai mari sa fie selectate.
- Cand *T* tinde la 0, algoritmul devine din ce in ce mai mult precum hill climbing.
- Apare un parametru de intrare, *planificare,* cu rolul de a determina valoarea lui *T* in raport cu cate iteratii au avut loc.

Simulated annealing

functia simulated_annealing(*problema, planificare*) ***intoarce*** o solutie
Se pastreaza la fiecare reapelare: nodul *curent* si nodul *urmator.*

curent = genereaza_nod(stare_initiala[problema])

Pentru t = 1 *pana la* ∞ *executa*
T = planificare[t]
urmator = un succesor al nodului *curent* ales aleator
ΔE = *eval*(*urmator*) - eval(*curent*)
Daca $\Delta E > 0$ *atunci curent = urmator*
Altfel curent = urmator cu probabilitatea $e^{\Delta E/T}$
Sfarsit pentru

Am presupus ca evaluare mai mare inseamna mai buna.

Simulated annealing

- Cand *T* tinde la 0, exponentul din $e^{\Delta E/T}$ tinde catre $-\infty$ si probabilitatea catre 0.
- Tabelul de mai jos arata probabilitati de a accepta pasi spre solutii mai slabe pentru diverse valori ale parametrului T.

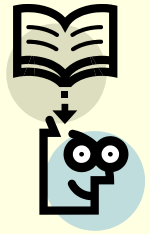
T	**Probabilitatea de a accepta**		
	1 x mai rau	2 x mai rau	3 x mai rau
10	0.9	0.82	0.74
1	0.37	0.14	0.05
0.25	0.018	0.0003	0.000006
0.1	0.00005	2×10^{-9}	9×10^{-14}

O posibilitate de a calcula valorile lui T este sa pornim cu el 10 si, la fiecare iteratie, sa ii inmultim valoarea cu 0.97. Dupa 100 de pasi obtinem 0.48.

Aplicatii la probleme cu constrangeri

- La problema celor 8 dame, o stare initiala are toate cele 8 dame pe tabla, iar un operator muta o dama dintr-un patrat in altul.
- Algoritmii care rezolva astfel de probleme se numesc metode **euristice de reparare** pentru ca repara inconsistentele din configuratia curenta.
- In alegerea unei noi valori pentru o variabila, o **euristica cu conflicte minime** este cea mai potrivita – se urmareste selectarea unei valori care sa duca la minimizarea numarului de conflicte cu celelalte variabile.
- Aceste euristici sunt capabile sa rezolve problema celor un milion de dame in medie in mai putin de 50 de pasi.

Recapitulare 1/2



- O problema de satisfacere de constrangeri consta din:
  - o multime de variabile,
  - un domeniu pentru fiecare variabila si
  - o multime de constrangeri.
- Scopul este de a alege o valoare pentru fiecare variabila a.i. lumea posibila rezultata satisface constrangerile.
- Algoritmi care genereaza-si-testeaza verificia toate asocierile posibile dintre variabile si valori.
- Algoritmii de consistenta reduc domeniile pentru variabilele din cadrul problemei in vederea simplificarii problemei.

Recapitulare 2/2

- Algoritmii bazati pe imbunatatiri iterative tin in memorie o singura stare, dar se pot bloca in maxime locale.

- Simulated annealing ofera o cale de a scapa de maxime locale si este complet si optimal daca variabila *planificare* are un proces lung si gradual de scadere.

- Pentru probleme cu satisfacere de constrangeri, euristicile care ordoneaza variabilele pot aduce imbunatatiri foarte mari ale performantei.