BABEŞ-BOLYAI UNIVERSITY
Faculty of Mathematics and Computer Science

Inteligenţă Artificială

2: Optimizare, cautare locala

Camelia Chira

camelia.chira@ubbcluj.ro

Probleme de optimizare

- Scopul este de a determina minimul sau maximul unei anumite functii cu una sau mai multe variabile

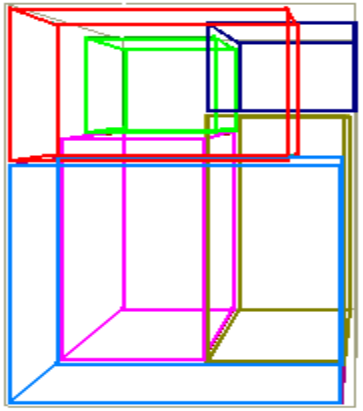
- *Minimizare: Gaseste* x^* in A astfel incat $f(x^*) \leq f(x)$ pentru orice x in A
- Optim global, local

Probleme de optimizare din lumea reala

Logistica, transport

Planificare

Linii de productie, industrie

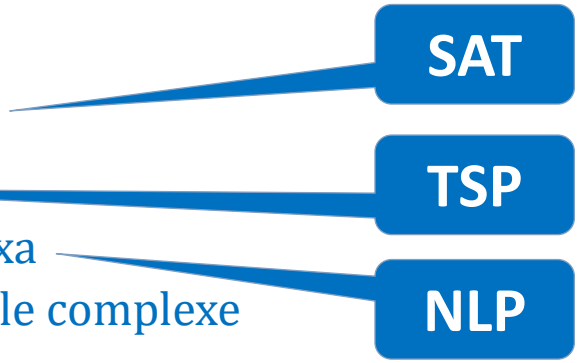


Ambalare, distributie

Retele, telecomunicatii

Optimizare

- Minimizare /maximizare
- Multe aplicatii
- Exemple
  - Gasirea de modele bool pentru propozitii
  - Gasirea celui mai scurt drum
  - Gasirea min/max pentru o functie complexa
  - Detectarea structurii comunitatilor in retele complexe
  - Determinarea structurii proteinelor
- Metode pentru probleme de optimizare
  - **Euristici si metaeuristici**
  - **Tehnici de cautare de tip single-point**
  - **Algoritmi evolutivi (bazati pe populatii)**



The Boolean Satisfiability Problem (SAT)

- O expresie logica trebuie evaluata la TRUE
- Exemplu:
  - Gaseste valoarea fiecarui x_i pentru i=1,…,100 astfel incat F(x)=TRUE

$$F(x) = (x_{17} \vee \bar{x}_{37} \vee x_{73}) \wedge (\bar{x}_{11} \vee \bar{x}_{56}) \wedge \ldots \wedge (x_2 \vee x_{43} \vee \bar{x}_{77} \vee \bar{x}_{89} \vee \bar{x}_{97})$$

  - 2^{100} posibilitati (2 valori pt fiecare din cei 100 x_i)
- Marimea spatiului de cautare

$$|S| = 2^{100} \approx 10^{30}$$

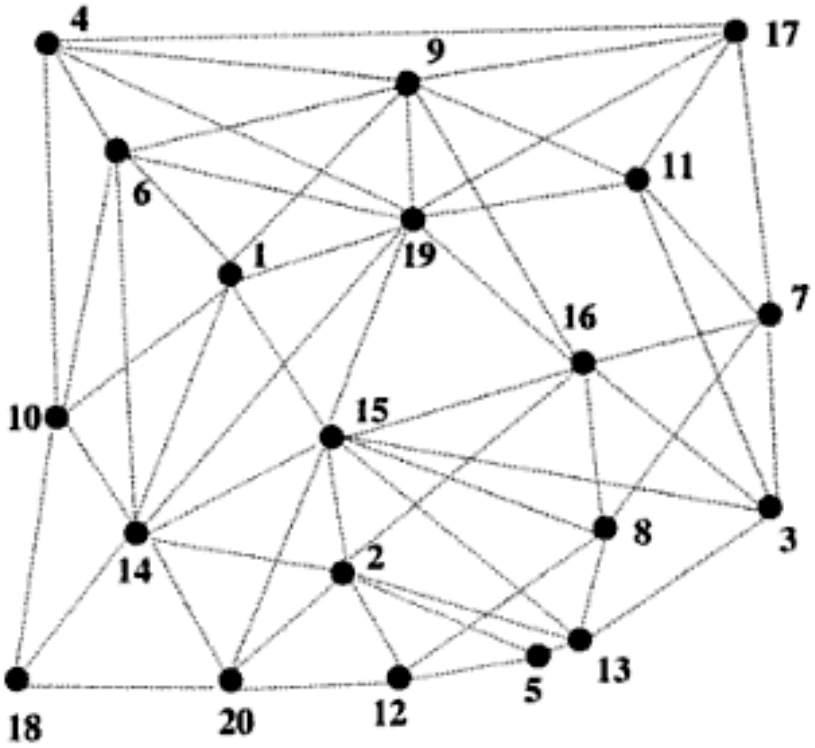
HUGE!!

Daca un calculator ar putea testa 1000 siruri/secunda si am incepe calculele la inceputurile timpului - acum 15 bilioane de ani- am ajunge sa examinam numai 1% din cazuri pana azi!!!!

Traveling Salesman Problem (TSP)
Problema comis-voiajorului

- Parcurgerea traseului pe cel mai scurt drum vizitand fiecare oras o singura data
  - Cost minim: timp, combustibil
- dist(i,j)=dist(j,i)
  - Altfel, TSP asimetrica
- Solutie: orice permutare

de *n* orase



TSP – spatiul de cautare

- *n!* moduri de a permuta *n* orase

- Pentru n>6 numarul solutiilor posibile pentru TSP cu n orase este MAI MARE decat pentru SAT cu n variabile

$$|S| = n!/(2n) = (n-1)!/2$$

HUGE!!

n	TSP	SAT
6	60	64
7	360	128
8	2520	256

Nr orase	Nr solutii
10	181,000
20	10,000,000,000,000,000
50	100,000,000,000,000, 000,000,000,000, 000,000,000,000, 000,000,000,000,

Nonlinear Programming Problem (NLP)

- Problema generala de optimizare: selecteaza n variabile x_1, x_2, ...x_n din domenii fezabile date astfel incat sa fie optimizata o functie $f(x_1, x_2, ...x_n)$ si unele restrictii specificate sa fie respectate

- Ex. Maximizeaza functia:

$$G2(\mathbf{x}) = \left| \frac{\sum_{i=1}^{n} \cos^4(x_i) - 2\prod_{i=1}^{n} \cos^2(x_i)}{\sqrt{\sum_{i=1}^{n} i x_i^2}} \right|,$$

subject to

$$\prod_{i=1}^{n} x_i \geq 0.75, \quad \sum_{i=1}^{n} x_i \leq 7.5n, \text{ and bounds } 0 \leq x_i \leq 10 \text{ for } 1 \leq i \leq n.$$

NLP: maximizare

Maximizeaza f(x), $x = (x_1, x_2, \dots, x_n) \in \mathbb{R}^n$

$$x \in F \subseteq S, \qquad S \subseteq \mathbb{R}^n$$

$$l_i \leq x_i \leq u_i, 1 \leq i \leq n$$

F este definit prin *m* restrictii:

$$g_j(x) \leq 0, j = 1, \dots, q$$
$$h_j(x) = 0, j = q + 1, \dots, m$$

- Spatiul de cautare este n-dimensional
- Nici o metoda de optimizare traditionala nu a dat rezultate satisfacatoare

NLP – spatiul de cautare

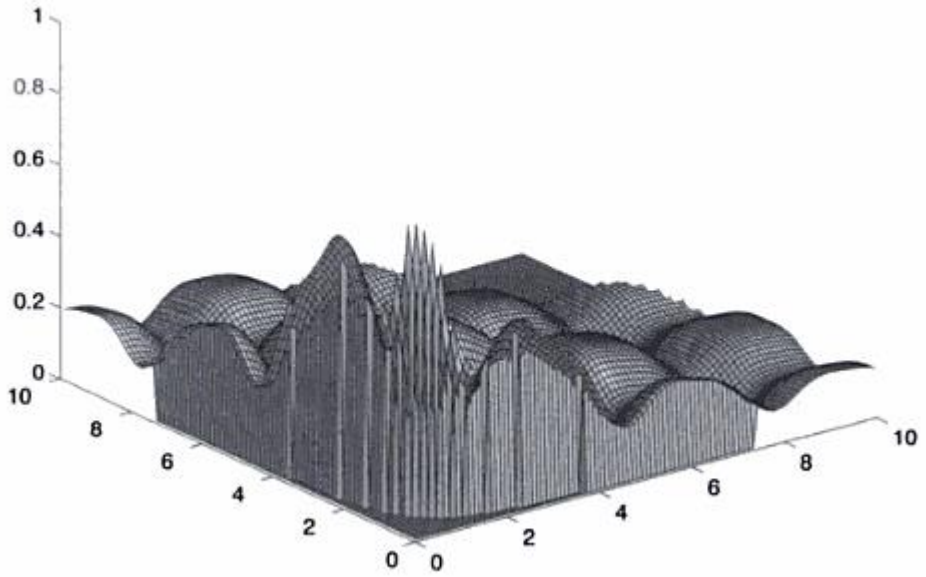
- Depinde de numarul de dimensiuni
- Pur matematic: fiecare dimensiune poate contine un numar INFINIT de valori posibile

Pentru implementarea unui algoritm pe calculator:
-precizia disponibila?
-pentru precizie de 6 puncte zecimale garantata fiecare variabila poate lua 10,000,000 valori

$$|S| = 10{,}000{,}000^{n} = 10^{7n}$$

HUGE!!

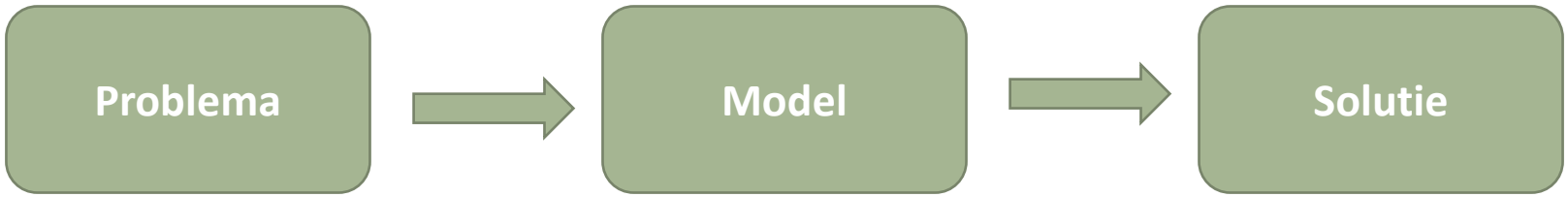
Chiar mai mare decat spatiul de cautare pentru TSP



Modelarea problemei

- Modele – simplificarea lumii reale

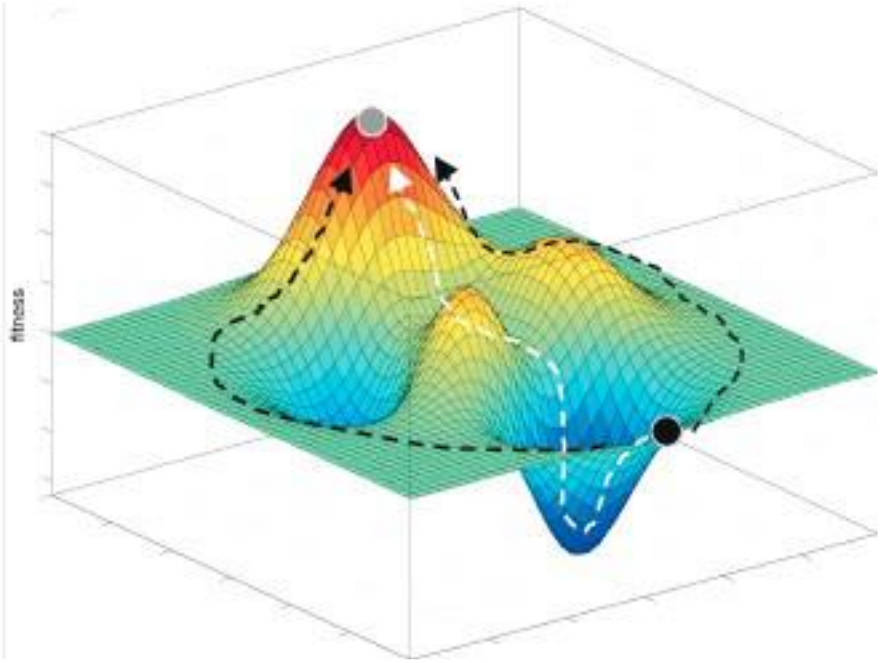
- SAT, TSP, NLP – forme canonice de modele ce pot fi aplicate in multe situatii



De ce sunt unele probleme dificile?

- Numarul solutiilor posibile din spatiul de cautare este prea mare pentru a permite o cautare exhaustiva
- Problema este atat de complicata incat este necesara simplificarea ei pentru a obtine orice raspuns incat orice rezultat devine nefolositor
- Functia de evaluare a calitatii solutiilor gasite se schimba in timp (noisy) – nevoie de o serie de solutii si nu una singura
- Solutiile posibile sunt atat de restrictionate incat chiar si construirea uneia valide este dificila, una optima?

Spatiul de cautare



- Spatiul de cautare este multimea solutiilor candidat ale problemei

Marimea problemei

Exemplu. TSP cu n=10 orase, marimea problemei este 10

Marimea spatiului de cautare

Exemplu: TSP cu 10 orase are aprox. 181,000 solutii posibile

$O(2^n)$

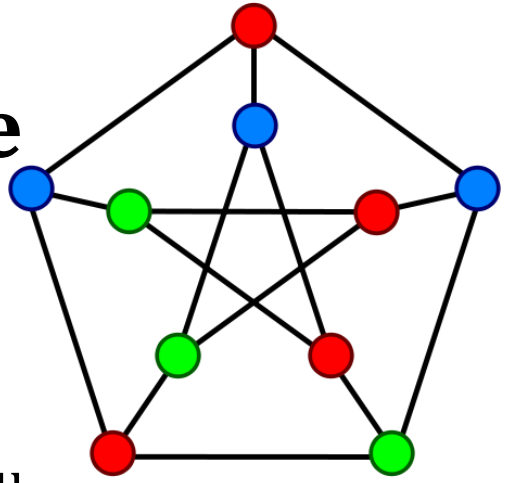
$O(n!)$

Functia de evaluare

- Aceasta functie trebuie sa determine cat de buna este o solutie
- De obicei mapeaza o solutie la o valoare reala (care trebuie minimizata/maximizata)
- Poate fi si o functie de sortare a solutiilor: pentru orice pereche de solutii, determina care este mai buna

- Se mai numeste:
  - Functie obiectiv
  - Functia energie (fizica)
  - Functia de cost (economie)
  - Functia de calitate (inginerie)
  - Functia de fitness (biologie)

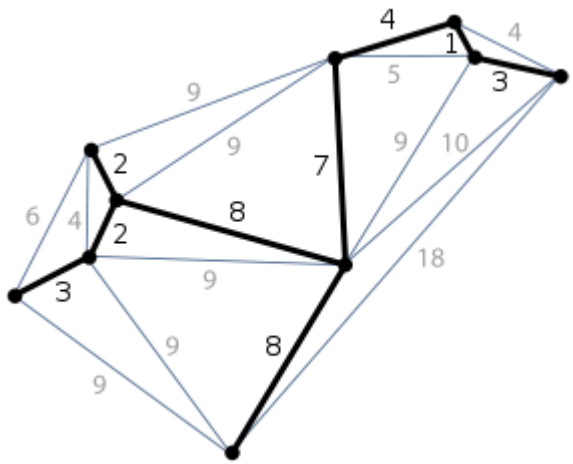
Probleme: optimizare si decizie

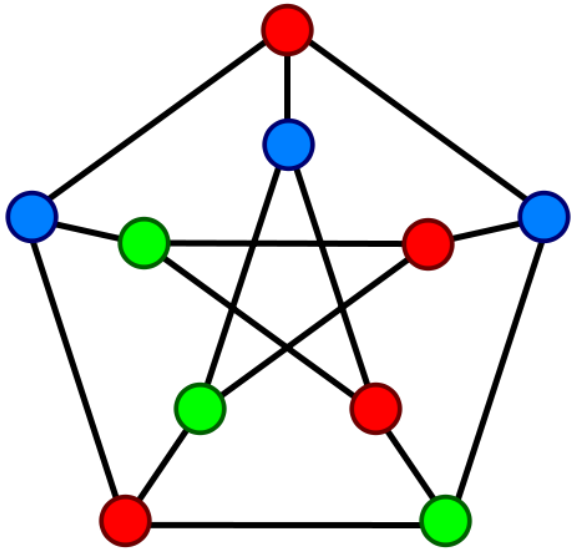


- Problema vs. Instanta a unei probleme
  - O instanta poate avea mai multe solutii
- Probleme de decizie: cele care au un raspuns Da/Nu
  - Ex. Este un graf dat k-colorabil?
  - Daca problema de optimizare poate fi rezolvata usor atunci si problema de decizie corespunzatoare se rezolva usor
- O problema este **undecidable** daca nu exista nici un algoritm care primeste ca input o instanta a problemei si determina raspunsul pentru acea instanta *ex. Turing's Halting problem*
- O problema este **tractabila** daca poate fi rezolvata de un algoritm in timp *polinomial*
- O problema este **intractabila** daca orice algoritm are nevoie de timp superpolinomial

Clasa de complexitate P

- Multimea problemelor de decizie ce pot fi rezolvate in timp **polinomial** *i.e. $O(n^k)$, unde k este o constanta*
- Multe probleme sunt in clasa P *ex. minimum spanning tree*
- Nu toate problemele pot fi rezolvate in timp polinomial





Clasa de complexitate NP (nondeterministic polynomial time)

- **NP –** probleme pentru care o solutie poate fi verificata in timp polinomial

- Probleme intractabile
  - Marimea problemei creste – nu mai pot fi rezolvate in timp rezonabil

T(n)	Name	Problems
O(1)	Constant	Easy-solved
O(logn)	Logarithmic	
O(n)	Linear	
O(nlogn)	Linear-logarithmic	
$O(n^2)$	Quadratic	
$O(n^3)$	Cubic	
$O(2^n)$	Exponential	Hard-solved
O(n!)	Factorial	

- O problema de decizie este in clasa NP daca are un algoritm polinomial nondeterminist:
  - Algoritmul "ghiceste" o solutie (nondeterminist)
  - Verifica determinist in timp polinomial ca solutia este corecta

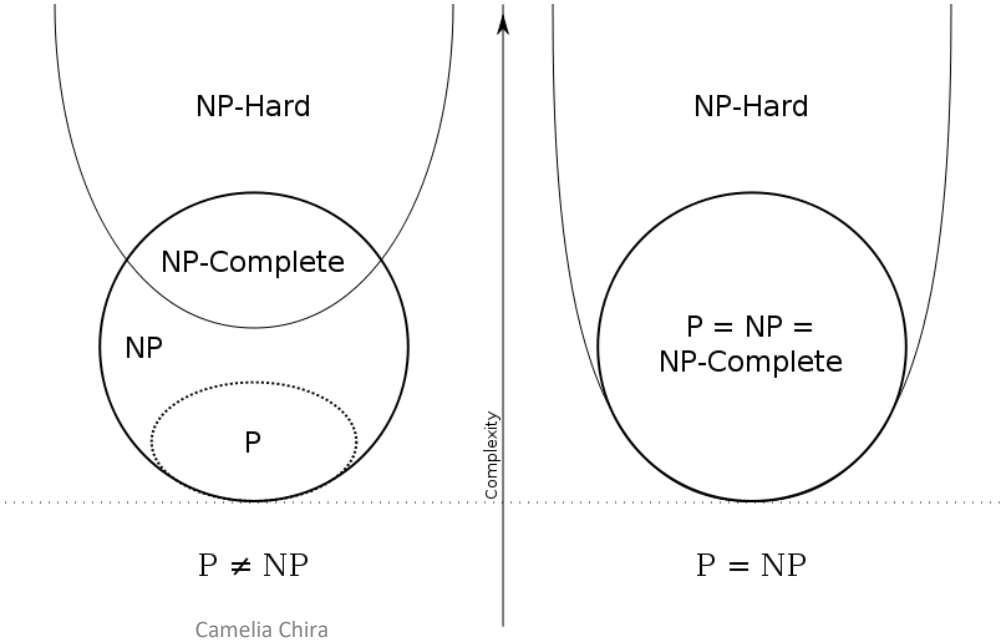
P=NP ?

Probleme NP-complete

- *Toate problemele X din NP pentru care este posibil sa reducem orice alta problema Y din NP la X in timp polinomial*
- **Reducibility**: *O problema P poate fi redusa la alta problema Q daca orice instanta a lui P poate fi transformata intr-o instanta a lui Q a carei solutie ofera o solutie instantei lui P*
  - Daca P se reduce la Q, P nu se rezolva mai greu decat Q
- O problema este **NP-complete** daca:
  - X∈NP
  - $\forall Y \in NP, Y \text{ este reductibila la } X \text{ in timp polinomial}$
- Probleme NP-complete:
  - Intractabile
  - Nu pot fi rezolvate in timp polinomial de nici un algoritm
  - Daca o problema NP-complete ar putea fi rezolvata in timp polinomial atunci orice problema NP-complete are un algoritm polinomial
  - Solutii aproximative
- Ex. SAT

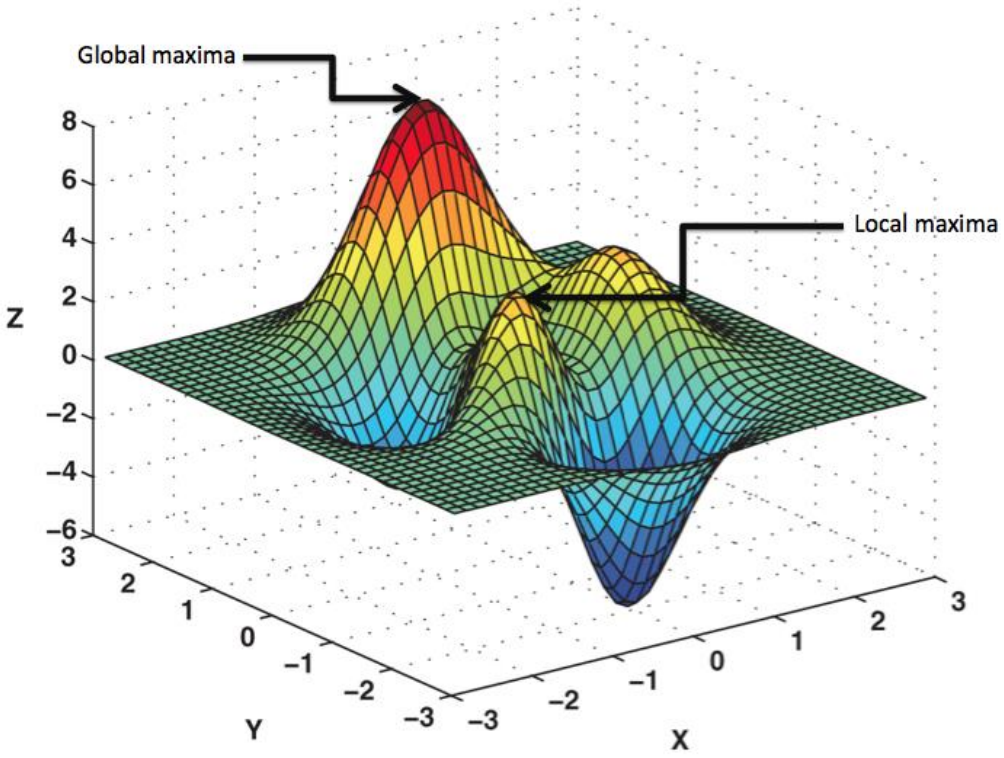
Probleme NP-hard

- Probleme cel putin la fel de grele ca si problemele NP-complete
- O problema X este **NP-hard** daca exista o problema NP-complete Y astfel incat Y este reductibila la X in timp polinomial
  - Daca X∈NP si X este NP-hard atunci X este NP-complete
  - Daca X se poate reduce la Y si X este NP-complete atunci si Y este NP-complete
- Ex. TSP

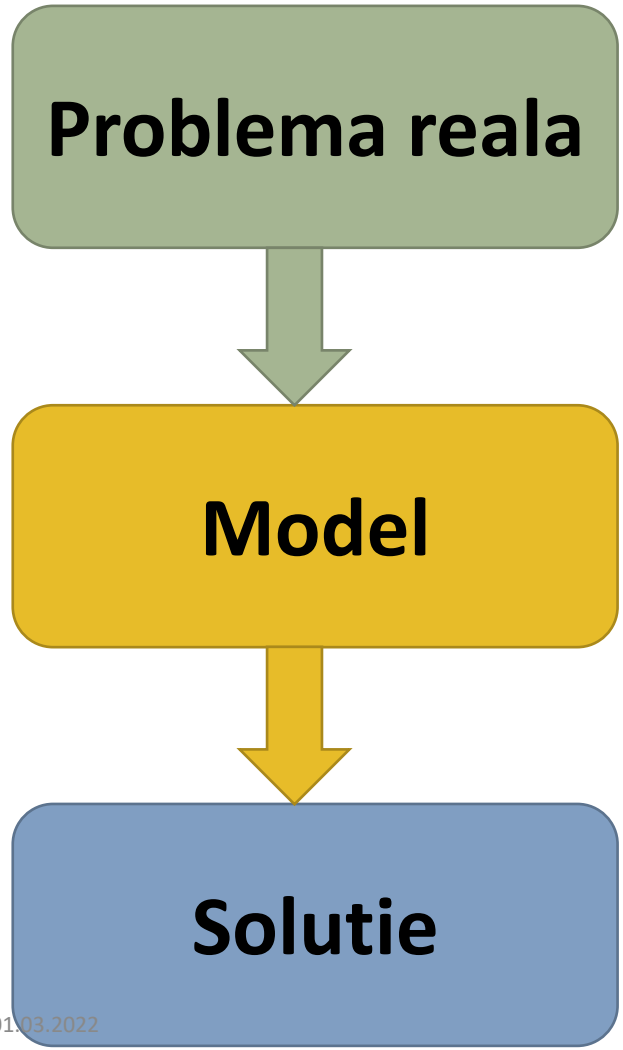


Optimizare

- Intr-o problema de optimizare, cautam cea mai buna solutie care satisface una sau mai multe conditii
- Majoritatea problemelor de optimizare sunt in clasa de complexitate NP
- Spatii de cautare complexe
- Global optima
- Local optima
- Multiple solutions



Optimizare: probleme reale



Model
- Reprezentarea problemei
- Restrictii
- Functia obiectiv

Solutie a modelului
- O solutie candidat fezabila
- Conduce la o valoare optima sau aproape de optim a functiei obiectiv

Algoritm de optimizare
- Exacti
- Euristici, metode aproximative

Concepte de baza

- Reprezentare
  - Cum poate fi reprezentata problema
  - Ce inseamna o solutie potentiala?
- Obiectiv
  - Care este obiectivul problemei?
  - Formuleaza matematic ce se cere
- Functie de evaluare
  - Diferita de obiectiv!
  - Trebuie sa returneze o anumita valoare care sa indice calitatea unei solutii sau care sa permita compararea a doua solutii

SAT: Reprezentare

- n variabile ce pot lua valoarea TRUE sau FALSE
- Sir de lungime n, cu valori posibile 0 sau 1
- Fiecare element din sir corespunde unei variabile
- Spatiul de cautare: $|S| = 2^n$ (orice punct din S este o solutie fezabila)

Exemplu

- F cu 8 clauze si 10 variabile

$$F= (x_1 \vee x_2 \vee \neg x_3) \wedge (x_2 \vee \neg x_{10}) \wedge (\neg x_2) \wedge (x_4 \vee x_{10}) \wedge (x_3 \vee x_5) \wedge (\neg x_4 \vee x_2 \vee \neg x_5) \wedge (\neg x_1 \vee x_6 \vee \neg x_7) \wedge (x_8 \vee x_{10})$$

- Exista 2^{10}=1024 posibile solutii, dintre care numai 32 (3%) sunt valide
- **O posibila solutie: un sir de 10 biti**
- **x=0111000110 inseamna x_1=0, x_2=1, x_3=1, etc.**

TSP: Reprezentare

TSP cu n orase

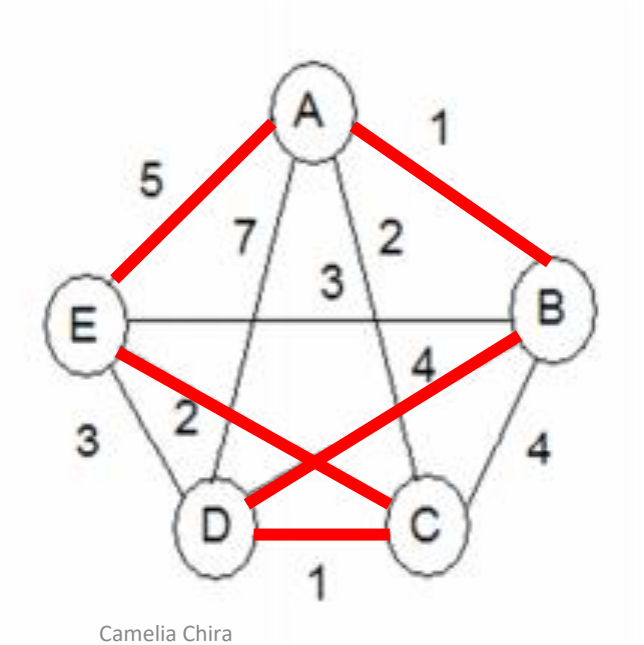
- Permutare de numere naturale 1...n
  - Fiecare numar ii corespunde unui oras
  - Ordinea de vizitare a oraselor este data de permutare
- Spatiul de cautare: $|S| = (n-1)!/2$

Exemplu

5 orase: A,B,C,D,E

A-B-D-C-E-A

1-2-4-3-5-1



NLP: Reprezentare

- Spatiul de cautare este aici format din numere reale in **n** dimensiuni
- Un n-tuplu de n numere reale (vector de numere reale)

$$x = (x_1, x_2, \dots, x_n) \in \mathbb{R}^n$$

- Marimea spatiului de cautare depinde de precizie
  - Pentru 6 pozitii zecimale si numere x cuprinse intre 0 si 10:

$$|S| = 10{,}000{,}000^n = 10^{7n}$$

Reprezentare: observatii

- Pentru fiecare problema, reprezentarea unei solutii si interpretarea ei conduce la **spatiul de cautare** si dimensiunea lui

- Spatiul de cautare nu este determinat de problema!
  - De reprezentare si felul in care solutiile sunt codificate

Obiectivul problemei

- Ce cautam? Care este obiectivul problemei date?
- Mai degraba o expresie decat o functie

- **SAT**

gasirea unui vector de biti astfel incat functia F sa fie evaluata la TRUE

- **TSP**

minimizarea distantei totale parcurse de comis-voiajor cu restrictia de vizita fiecare oras o singura data si de a ajunge inapoi la orasul initial

- **NLP**

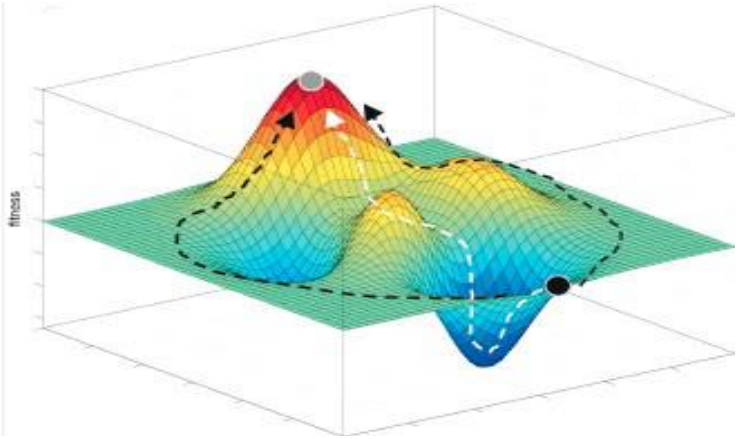
minimize or maximize of functie neliniara care primeste solutiile candidat

Functia de evaluare

- De obicei mapeaza spatiul solutiilor posibile sub reprezentarea aleasa la un set de numere (e.g. reale)
- Fiecare element din spatiul de cautare primeste o valoare numerica care indica calitatea acelei solutii
- Compararea solutiilor (*ordinal eval functions*) vs calitatea solutiilor (*numeric evaluation functions*)

- Cum setam functia de evaluare?
  - Criteriu evident: O solutie care satisface obiectivele problemei in intregime ar trebui sa aiba cea mai buna evaluare.
  - De multe ori, obiectivul problemei sugereaza o functie de evaluare (e.g. TSP, NLP)
  - SAT?

Problema de optimizare



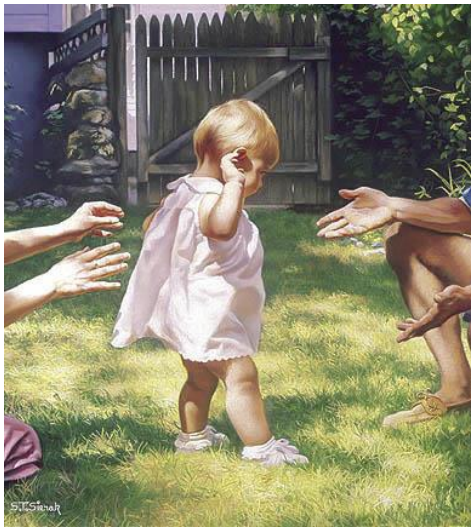
- Problema de cautare
- Spatiu de cautare ***S***
  - Partea fezabila ***F*** a lui ***S***: contine solutii ce satisfac toate restrictiile problemei
  - TSP, NLP: F=S

Definitie (problema de minimizare)

Dat fiind un spatiu de cautare S si partea lui fezabila F, $F \subseteq S$, sa se gaseasca $x \in F$ astfel incat $eval(x) \leq eval(y), \forall y \in F$.

Punctul x care satisface conditia de mai sus se numeste solutie **globala**.

Cei trei pasi spre rezolvare...



- Cautare: o plimbare adaptata spatiului

(fitness landscape)

- Orice problema de optimizare/cautare are

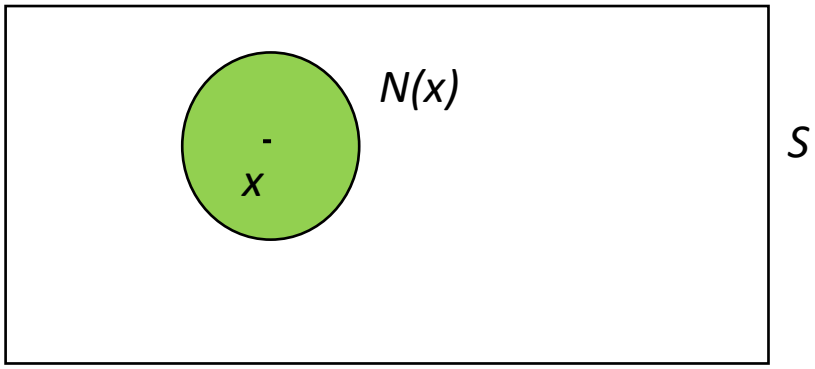
3 componente principale:

L=(S,f, d)

1. **Reprezentare** – multime de solutii posibile – *spatiu de cautare* (S)
2. **Functie de evaluare** (definite pornind de la obiectiv) – functia obiectiv – calitatea solutiilor
3. **Functia de vecinatate** – o metrica de distanta care sa determine solutii din vecinatate

Vecinatate si optime locale

- Vecintatea unui punct x din S este multimea acelor puncte care sunt “aproape” de x intr-un mod masurabil.



- Cand sunt 2 puncte apropiate?

Vecinatate: Cand sunt 2 puncte apropiate?

Putem defini o functie de distanta *dist* pe S

$$dist: S \times S \rightarrow \mathbb{R}$$

si o vecinatate N(x) pentru un ε≥0:

$$N(x) = \{y \in S : dist(x, y) \leq \varepsilon\}$$

- Pentru spatii continue, e.g. NLP, vecinatatea poate fi definita folosind distanta euclidiana:

$$dist(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

- SAT: distanta dintre 2 puncte poate fi **Hamming distance** (= numarul de pozitii care au valori diferite)

Vecinatate: Cand sunt 2 puncte apropiate?

Putem defini o mapare m pe S care defineste o vecinatate pentru fiecare punct x din S:

$$m: S \rightarrow 2^S$$

TSP: 2-swap mapping

O permutare de **n** orase are **n(n-1)/2** vecini
1-2-3-4 -> 1-3-2-4

Ex. n=20, **x= 15-3-11-19-17-2-...-6**

Vecinatate *2-swap* (190 puncte in S):
15-17-11-19-3-2-...-6
2-3-11-19-17-15-...-6
15-3-6-19-17-2-...-11

SAT: 1-flip mapping
(inversarea unui singur bit)
0->1 1->0

Un sir de n biti are n vecini.
Ex. n=20, **x=01101010001000011111**

Vecintate *1-flip* (20 de puncte in S):
11101010001000011111
00101010001000011111
01001010001000011111

Optime locale

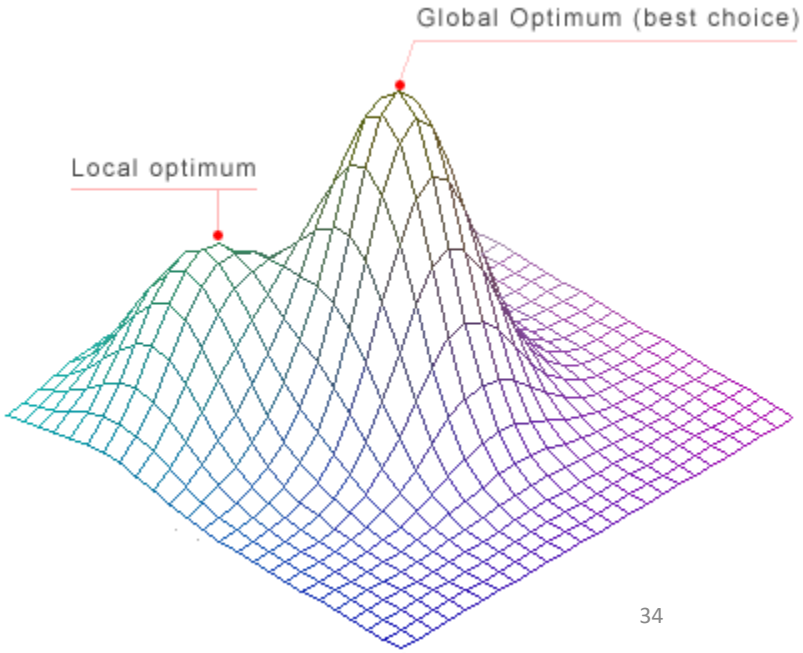
- O solutie potentiala este optim local in relatie cu vecinatatea N daca si numai daca

$$eval(x) \leq eval(y), \forall y \in N(x)$$

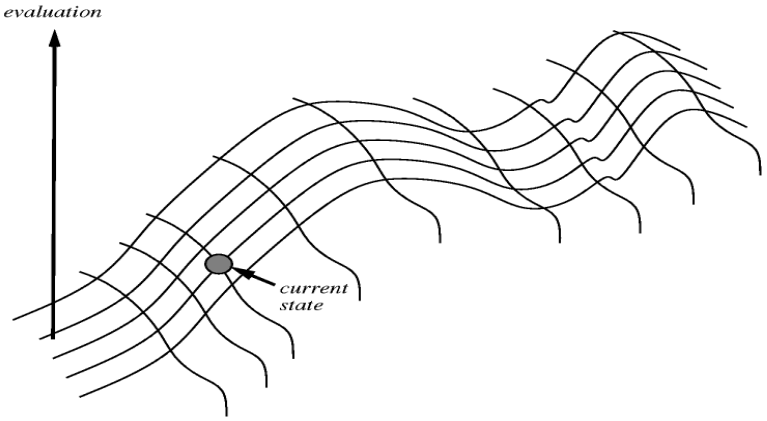
- Usor de verificat pentru vecinatati relativ mici
- Multi algoritmi se bazeaza pe statistica vecinatatii: solutiile generate se bazeaza numai pe *informatie locala* la fiecare pas (*local search strategies*)
- *Exemplu:*

  $max.\ f(x) = -x^2, \varepsilon = 0.1$

  (vecinatate +/- 0.1 pentru un punct x)



Cautare locala



- **Hill-climbing (HC):**

imbunatatire iterativa

- Tehnica se aplica unui singur punct (punct curent) din spatiul de cautare
- La fiecare iteratie, un nou punct este selectat din vecinatatea punctului curent
- Daca noul punct are o evaluare mai buna (functia obiectiv!), atunci el devine punctul curent
- .... pana cand solutia nu se mai imbunatateste sau max iteratii
- **Variante HC**
  - **Random HC**
  - **Steepest-ascent HC**
  - **Next-ascent HC**

Random HC (RHC)

1. Se selecteaza un punct aleator ***c*** (*current*) in spatiul de cautare
2. Se alege un punct ***x*** din vecinatatea lui c: ***N(c).***

   Daca *eval(x)* este mai bun decat *eval(c)* atunci ***c=x***.
3. Repeta pasul 2 pana cand un numar maxim de evaluari se atinge.
4. ***Returneaza c.***

Steepest Ascent HC (SAHC)

1. Se selecteaza un punct aleator ***c*** (*current hilltop*) in spatiul de cautare.
2. Se determina toate punctele x din vecinatatea lui c: $x \in N(c)$
3. Daca oricare $x \in N(c)$ are un fitness mai bun decat c atunci c=x , unde x are cea mai buna valoare eval(x).
4. Daca nici un punct $x \in N(c)$ nu are un fitness mai bun decat **c**, se salveaza **c** si se trece la ***pasul 1.*** Altfel, se trece la ***pasul 2*** cu noul **c**.
5. Dupa un numar maxim de evaluari, se returneaza cel mai bun **c** (hilltop).

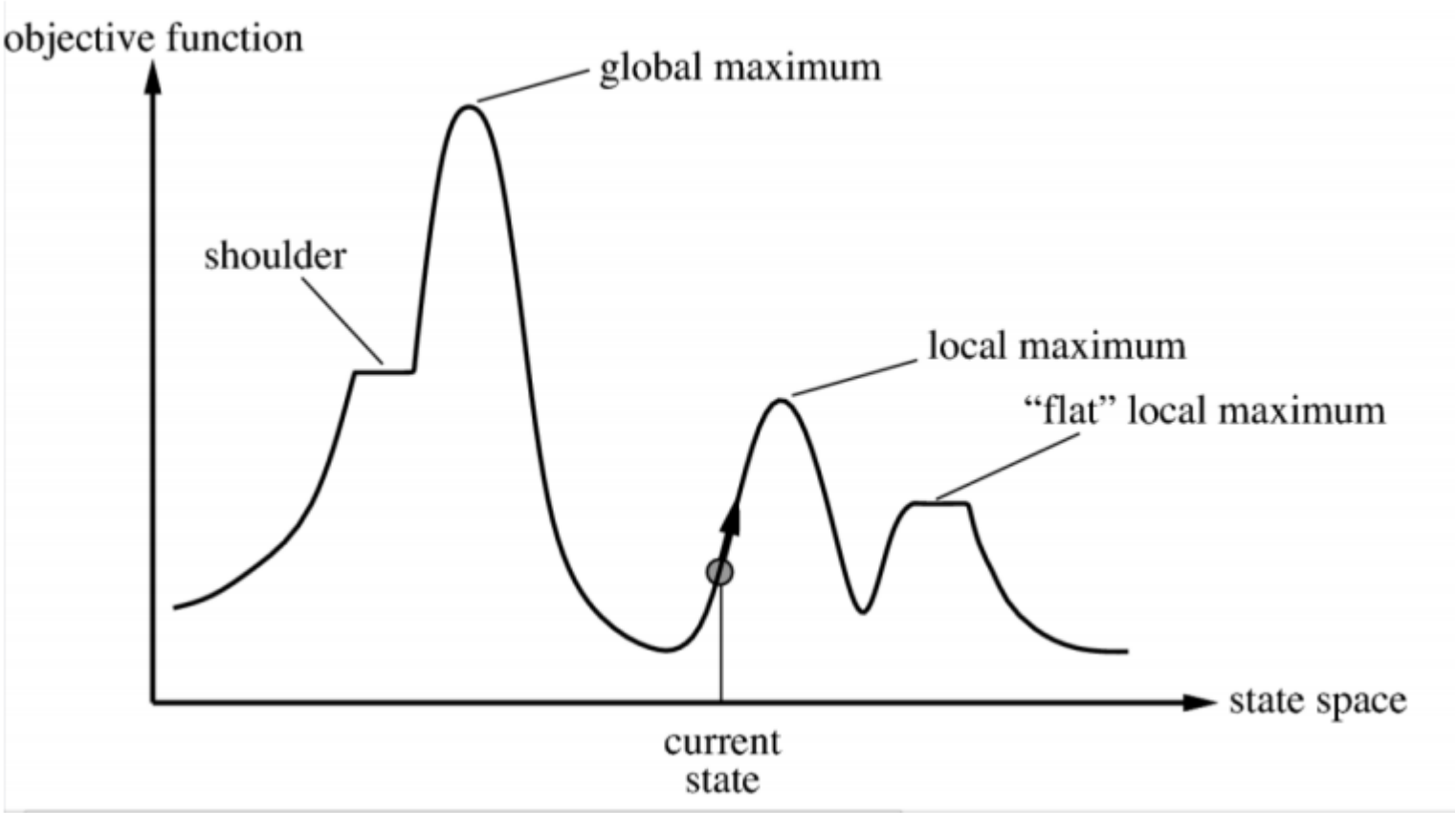
a.k.a.
Best-Improvement HC

Next Ascent HC (NAHC)

1. Se selecteaza un punct aleator ***c*** (*current hilltop*) in spatiul de cautare.
2. Se considera pe rand vecinii x ai punctului c. Daca eval(x) este mai bun decat eval(c), atunci c=x si nu se mai evalueaza restul vecinilor lui c. Se continua pasul 2 cu noul c si se considera vecinii lui c mai departe (pornind din acelasi punct din vecinatate unde s-a ramas cu vechiul c).
3. Daca nici un vecin x al punctului c nu duce la o evaluare mai buna, se salveaza c si se continua procesul de la pasul 1.
4. Dupa un numar maxim de evaluari, se returneaza cel mai bun **c** (hilltop).

a.k.a.
First-Improvement HC

Hill-climbing si S



Hill-climbing: dezavantaje

- De obicei, metodele HC se opresc in optime locale
- Nu se stie cat de mult optimele locale descoperite deviaza de la optimul global, sau chiar de la alte optime locale
- ***Optimul obtinut depinde de configuratia initiala***
- Nu este posibil sa specificam un timp maxim de calcul
- *Explorare - exploatare*

....DAR:

✓ **Usor de implementat si aplicat**

✓ **Rezultate bune ca si metode de cautare locala**

Stuck in local optima?

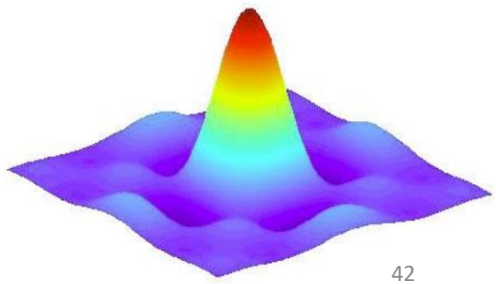
- Nu! Exista metode de cautare inteligenta...
- Metoda aleasa depinde intotdeauna de problema
- Optiuni?
  - Executa algoritmul pentru un numar mai mare de configuratii initiale
  - Foloseste rezultatele intermediare pentru a imbunatati alegerea punctului curent in iteratia urmatoare
  - Mareste vecinatatea
  - Modifica criteriul de acceptare a unor noi puncte astfel incat sa fie permise cateodata si puncte care au o evaluare mai proasta
- Metode?
  - Random restart HC
  - Simulated Annealing
  - Tabu search

Random Restart HC

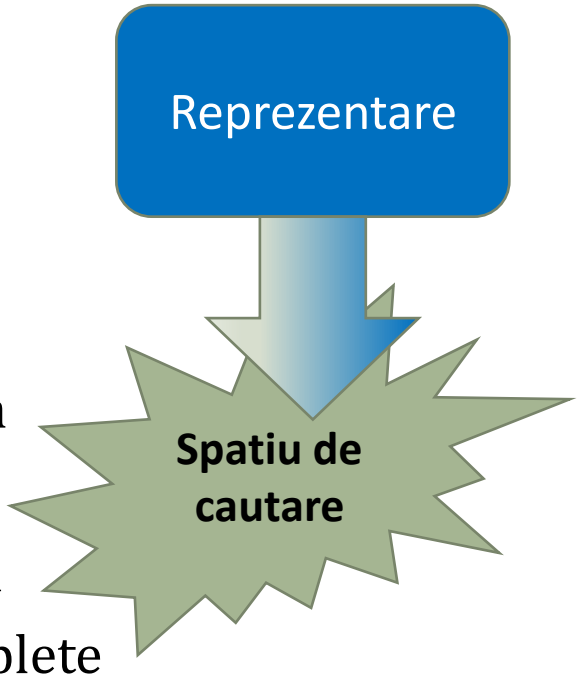
- Cand se ajunge intr-un optim local, se alege un alt punct curent in mod aleator
- HC este rulat de mai multe ori incepand din alte puncte initiale

- Avantaje?
  - Mai multe incercari de a gasi o zona buna a spatiului de cautare
  - Costul computational nu este mai mare

- **Numarul de incercari creste => probabilitatea de a gasi solutia se apropie de 1**
  - **Daca fiecare HC are probabilitatea de success p, numarul de restartari necesare este aproximativ 1/p**



Metode clasice



• Spatiul de cautare determinat de o anumita reprezentare poate fi considerat ca si multimea solutiilor complete => cautare in acest spatiu

• Spatiul de cautare poate fi divizat in subspatii si cautarea sa vizeze solutii partiale sau incomplete

Metode care lucreaza cu solutii complete

- Cautare exhaustiva
- Cautare locala
- Metoda simplex

Metode care lucreaza cu solutii partiale/incomplete

- Algoritmi greedy
- Divide and conquer
- Programare dinamica
- Algoritmul A*

Cautare exhaustiva (Exhaustive search)

- Verifica fiecare solutie din spatiul de cautare pana cand solutia optima globala este gasita.
- Also called: *enumerative algorithms*
- Nu ai cum sa stii daca ai gasit solutia optima decat daca examinezi toate posibilitatile de solutie

- Costisitor! Ex. *TSP cu 50 de orase are 10^{62} rute posibile de examinat!*
- **Totusi...**
  - Usor de implementat
  - Exista metode de a reduce volumul de munca e.g. backtracking
  - Unii algoritmi clasici de optimizare (ex. A*, branch and bound) se bazeaza pe cautare exhaustiva

SAT: cautare exhaustiva

- Trebuie sa generam toate sirurile posibile de n biti
- Cate sunt?
  - 2^n
  - De la $\langle 0 \dots 000 \rangle$ la $\langle 1 \dots 111 \rangle$
  - Fiecare corespunde unui numar intreg
- Cum enumeram fiecare solutie posibila?

1. Generam toate numerele intregi de la 0 la $2^n - 1$
2. Convertim fiecare numar intreg la sir de biti
3. Sirul de biti este evaluat folosind o **functie de evaluare**

   e.g. (**1**, daca sirul de biti satisface toate clauzele din F, **0** altfel)

Obs: Putem opri cautarea mai repede ex. cand o solutie satisface functia de evaluare

000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Am putea face o cautare mai eficienta pentru SAT?

TSP: cautare exhaustiva

- Cum putem genera toate permutarile posibile de n?
- Cate sunt?
  - **n!** permutari de **n** numere
- Probleme
  - Daca nu fiecare oras este conectat cu toate celelalte? => Solutii nefezabile
  - Problema generarii permutarilor

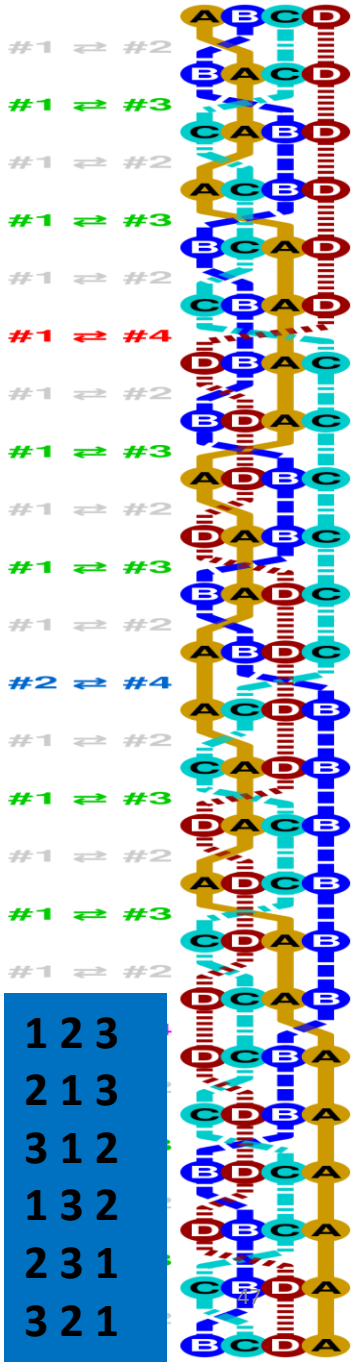
- Cum enumeram fiecare solutie posibila?
  - Definirea unei functii care prioritizeaza permutarile de la 0 la *n!* -1
  - Recursivitate

TSP: cautare exhaustiva

Heap's Algorithm

```
procedure generate(n:int, P:array)
begin
    if n=1 then print P
    else begin
        for i=0; i < n-1; i=i+1 do
            generate(n-1,P)
            if n is even then
                swap(P[i],P[n-1])
            else
                swap(P[0],P[n-1])
        end for
        generate(n-1,P)
    end
end
```

- B. R. Heap, 1963
- Initial P[i]=i
- Fiecare permutare este generata din precedenta prin interschimbarea a 2 elemente si nemodificarea celorlalte n-2
- Metoda sistematica de a alege la fiecare pas cele 2 elemente care se interschimba

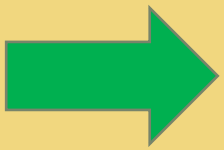


NLP: cautare exhaustiva

- n variabile - domenii continue => o infinitate de valori posibile
- Cum putem enumera toate solutiile posibile?
  - Domeniul fiecarei variabile continue poate fi impartit intr-un numar finit de intervale
  - Se considera produsul cartezian al acestor intervale

max. $f(x_1, x_2)$, $unde\ x_1 \in [-1, 3] si\ x_2 \in [0, 12]$

- Impartim domeniul [-1,3] in 400 intervale de lungime 0.01 fiecare
- Impartim domeniul [0,12] in 600 intervale de lungime 0.02 fiecare



400 x 600 = 240,000 celule de verificat

- Fiecare celula poate fi evaluata considerand un anumit punct din celula e.g. coltul celulei, mijlocul celulei

Cautarea este exhaustiva in sensul ca incercam sa acoperim toate solutiile posibile; totusi, fiecare solutie este acoperita de o celula! (cu cat mai mica cu atat mai bine)

NLP: cautare exhaustiva

- Dezavantaje
  - Daca folosim celule mai mici (*granularitate fina*), numarul total de celule creste semnificativ. *Ex. Daca domeniul x_1 este impartit in 4000 intervale si cel a lui x_2 in 6000, atunci numarul de celule creste de la **240,000** la **24,000,000***
  - Cu o granularitate mica creste probabilitatea de a nu gasi cea mai buna solutie (celula care o contine nu este neaparat evaluata foarte pozitiv)
  - Cand numarul de variabile este mare, aceasta cautare exhaustiva nu mai este practica pentru ca numarul de celule este mult prea mare. *Ex. O problema cu 50 variabile, fiecare cu 100 intervale, ar insemna $\mathbf{10^{100}}$ celule*
- **Concluzie**: cautarea exhaustiva poate fi ok pentru probleme mici *(garanteaza gasirea celei mai bune solutii*) dar este nepractica pentru probleme mari unde nu este posibila enumerarea tuturor solutiilor posibile

Algoritmi greedy (Greedy algorithms)

- Construiesc solutia completa intr-o serie de pasi
- La fiecare pas, se alege cea mai buna decizie disponibila (ne trebuie o euristica pentru asta)

- Alegerea de la fiecare pas este cea ***optima la nivel local***, in speranta ca vom ajunge la ***optimul global***
- La fiecare pas se alege cel mai mare "profit" **– Greedy**

- In multe probleme, o strategie greedy nu produce in general solutia optima dar poate conduce la solutii optime locale care aproximeaza solutia optima globala intr-un timp rezonabil

SAT: Algoritmi greedy

- Luam fiecare variabila pe rand si o setam la TRUE sau la FALSE in functie de o anumita euristica
- Ce euristica ar putea ghida decizia in SAT?

Pentru fiecare variabila de la 1 la n, in orice ordine, setam valoarea de adevar care rezulta in satisfacerea celui mai mare numar de clauze care acum nu sunt satisfacute. Daca acest numar este acelasi, se alege aleator valoarea.

- ✓ Greedy
- ✓ Performanta slaba chiar si pentru probleme simple

$$\overline{x_1} \wedge (x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (x_1 \vee x_4)$$

- Pentru x_1 am alege TRUE cu 3 clauze satisfacute.
- Dar asa prima clauza nu este si nu va fi niciodata satisfacuta indiferent de valorile lui x_2, x_3 si x_4 !
- **Too greedy**

SAT: Algoritmi greedy

- Abordarea precedenta este prea greedy – ordinea in care sunt considerate variabilele nu conteaza
- Am putea incepe cu acele variabile care apar in mai putine clauze si lasa la sfarsit acele variabile des intalnite (e.g. x_1)

- Sortam variabilele in ordinea frecventei cu care apar in clauze (de la cele mai rare la cele mai frecvente)
- Pentru fiecare variabila, in ordinea de mai sus, setam valoarea de adevar care rezulta in satisfacerea celui mai mare numar de clauze care acum nu sunt satisfacute. Daca acest numar este acelasi, se alege aleator valoarea.

$$(x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (\overline{x_1} \vee x_4) \wedge (\overline{x_2} \vee \overline{x_4}) \wedge (x_2 \vee x_5) \wedge (x_2 \vee x_6) \wedge \cdots$$

Aici nu mai apar x_1 si x_2 dar apar restul de multe ori

- ✓ x_1 este prezent in 3 clauze
- ✓ x_2 este prezent in 4 clauze
- ✓ Toate celelelalte variabile apar de mai multe ori

- x_1 = TRUE (2 clauze OK)
- x_2 = TRUE (+ inca 2 clauze OK)
- Dar a treia si a patra clauza nu vor putea niciodata fi satisfacute in acelasi timp: $(\overline{x_1} \vee \boldsymbol{x_4}) \wedge (\overline{x_2} \vee \overline{\boldsymbol{x_4}})$

SAT: Algoritmi greedy

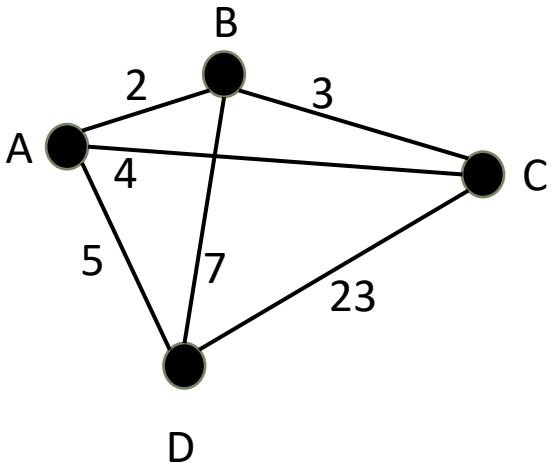
- Euristica poate fi modificata astfel incat sa adreseze problemele identificate
- *De exemplu:*
  - O regula ar putea interzice asignarea unei valori unei variabile daca oricare clauza ar deveni FALSE
  - Am putea considera cat de des apar variabile in clauzele ramase (inca nesatisfacute)
  - Sortarea variabilelor sa nu se bazeze numai pe numarul de aparitii in clauze ci si pe lungimea clauzelor in care apar (o variabila care apare intr-o clauza scurta are o alta semnificatie decat una ce apare intr-o clauza lunga cu multe alte variabile)
- Nici o alternativa nu ar insemna un algoritm care sa mearga pentru toate problemele SAT
- **Nu exista un astfel de algoritm greedy pentru SAT!**

TSP: Algoritmi greedy

- Cel mai intuitiv algoritm greedy pentru TSP se bazeaza pe euristica **nearest-neighbor** (cel mai apropiat vecin)

Incepand de la un oras oarecare, se alege cel mai apropiat oras nevizitat inca si se continua procesul pana cand toate orasele au fost vizitate (de la ultimul oras ne intoarcem in primul pentru o ruta completa).

- ✓ Greedy, dar ruta poate fi departe de una buna/perfecta
- ✓ De multe ori, pretul de platit este mare pentru alegeri greedy la inceput



- Start in A => ruta greedy: **A-B-C-D-A**
- cost (A-B-C-D-A)=2+3+23+5=33

DAR

- **cost (A-C-B-D-A)=4+3+7+5=19**

TSP: Algoritmi greedy

- Alte posibilitati?

Alegem urmatorul oras cel mai apropiat dar evitam situatia in care in care un oras este in mai mult de doua conexiuni (muchii pe graf).

- Incepem cu perechea de orase care are cel mai mic cost asociat.
- Continuam cu urmatoarea pereche de orase in ordinea data de cost.
- Daca am selectat (B,E) si (A,B) inseamna ca nu mai sunt permise perechi in care sa apara B.

Consideram posibilitatea de a “merge” inapoi la un oras deja vizitat => pentru fiecare pereche (i,j) de orase se alege cea mai buna optiune (e.g. sa se mearga direct de la i la j sau sa se treaca prin orasul vizitat).

“For every common sense heuristic you can invent, you can find a pathological case that will make it look very silly.” (Michalewicz, Fogel, 2004)

NLP: Algoritmi greedy

- Nu exista abordari greedy eficiente pentru NLP

Un algoritm cu unele caracteristici greedy ar fi (Line search):

- Sa presupunem ca functia de optimizat are 2 variabile x_1 si x_2
- Pastram x_1 constanta si variem x_2 pana cand ajungem la un optim
- Facem apoi noua valoare x_2 gasita constanta si variem x_1 pana cand un nou optim este gasit

- ✓ Abordarea nu este chiar greedy pentru ca sunt evaluate solutii complete
- ✓ Alegerea celei mai bune valori pentru o dimensiune este insa o caracteristica greedy
- ✓ Probleme: performanta slabe pentru ca nu sunt considerate interactiunile dintre variabile

Concluzii

- Metodele traditionale sunt construite pentru tipuri particulare de probleme
- Pot fi eficiente in anumite cazuri pe tipul particular de problema pentru care au fost dezvoltate
- Daca putem descompune o problema in mai multe subprobleme sau daca putem organiza spatiul de cautare ca un arbore – unele metode traditionale pot fi eficiente pe probleme de dimensiuni mici
- Dar daca asamblarea solutiilor subproblemelor este imposibila? Ex. TSP cu 100 orase descompus in 20 TSP cu 5 orase
- Pentru probleme reale care sunt complexe, intractabile, cu multe optime locale – avem nevoie de metode care sa mearga dincolo de ce pot face algoritmii clasici!

Cursul urmator...

Simulated Annealing

Tabu Search