

# Algoritmi de căutare locală și globală

- Motivație: optimizare locală vs. optimizare globală
- Structura generală a algoritmilor de optimizare locală
- Metaeuristici deterministe pentru căutare locală:
  - alg Pattern Search
  - alg Nelder Mead
- Metaeuristici aleatoare pentru căutare locală:
  - alg Matyas, Solis-Wets
- Metaeuristici pentru căutare globală:
  - Căutare locală cu restartare
  - Căutare locală iterată
  - Simulated Annealing

# Optimizare locală vs. optimizare globală

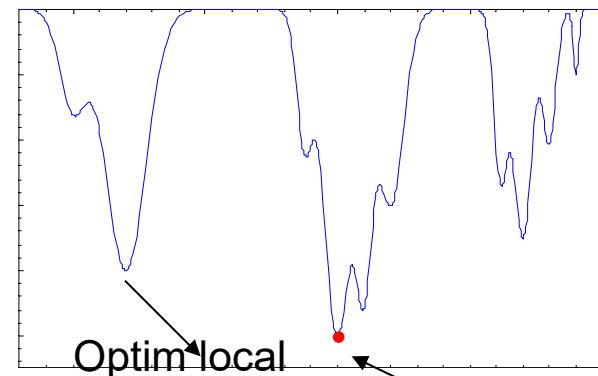
Optimizare locală:  $f(x^*) \leq f(x)$  pentru orice  $x$  în  $V(x^*)$

( $V(x^*)$  = vecinătate a lui  $x$ );

Obs: e necesară cunoașterea unei aproximații inițiale

Optimizare globală:

- identificarea optimului global al unei funcții: pentru o problemă de maximizare se caută  $x^*$  cu proprietatea că  $f(x^*) \leq f(x)$ , pentru orice  $x$
- dacă funcția obiectiv are și optime locale atunci metodele de căutare locală (cum este metoda gradientului) se pot bloca în punctele de optim local



# Optimizare locală

## Spațiu de căutare discret:

- Vecinătatea unui element este o mulțime finită care poate fi explorată exhaustiv

## Caz particular (soluții de tip permutare):

- $s=(s_1, s_2, \dots, s_n)$   $s_i$  din  $\{1, \dots, n\}$
- $V(s)=\{s' | s' \text{ poate fi obținută din } s \text{ prin interschimbarea a două elemente}\}$
- $\text{Card } V(s)=n(n-1)/2$

## Exemplu (n=4)

$$s=(2,4,1,3)$$

$$s'=(1,4,2,3)$$

## Spațiu de căutare continuu:

### a) funcția obiectiv este derivabilă

- Metoda gradientului
- Metode de tip Newton

### b) funcția obiectiv nu este derivabilă

- Metode bazate pe căutare directă (ex: Nelder Mead)
- Metode bazate pe perturbații aleatoare mici

# Căutare locală: structura generală

## Notatii:

S – spațiul de căutare

f – funcție obiectiv

$S_*$  - mulțimea optimelor locale/globale

$s=(s_1, s_2, \dots, s_n)$  : element din S/  
configurație/ soluție candidat

$s_*$  = cel mai bun element descoperit  
până în etapa curentă

$s^*$  = soluție optimă

## Algoritm de căutare locală:

s = aproximație inițială

repeat

$s'$  = perturbare(s)

    if  $f(s') < f(s)$  then

$s = s'$

until <condiție de oprire>

## Observații:

1. Aproximația inițială poate fi selectată aleator sau în baza unei euristici
2. Perturbarea poate fi deterministă (ex: metoda gradientului) sau aleatoare
3. Inlocuirea lui s cu  $s'$  se poate face și când  $f(s') \leq f(s)$
4. Condiția de oprire:
  - (a) nu se mai obține îmbunătățire în estimarea soluției;
  - (b) s-a atins numărul maxim de iterații (sau de evaluări ale funcției obiectiv)

# Căutare locală: variante (I)

Algoritm de căutare locală:

```
s = aproximație inițială
repeat
  s'=perturbare(s)
  if f(s')<f(s) then
    s=s'
until <condiție de oprire>
```

Mai mulți candidați:

```
s = aproximație inițială
repeat
  s'=perturbare(s)
  for k=1:m
    s'' = perturbare(s)
    if f(s'')<f(s) then s'=s''
  end
  if f(s')<f(s) then
    s=s'
until <condiție de oprire>
```

Observații:

1. căutarea e mai „agresivă” – la fiecare iterație se analizează mai mulți candidați
2. fiecare evaluare a funcției obiectiv trebuie contorizată (dacă condiția de oprire folosește nr de evaluări)

# Căutare locală: variante (II)

Algoritm de căutare locală:

```
s = aproximație inițială
repeat
  s'=perturbare(s)
  if f(s')<f(s) then
    s=s'
until <condiție de oprire>
```

Mai mulți candidați:

```
s = aproximație inițială
best = s
repeat
  s'=perturbare(s)
  for k=1:m
    s'' = perturbare(s)
    if f(s'')<f(s) then s'=s''
  end
  s=s'
  if f(s)<f(best) then best=s
until <condiție de oprire>
```

Observații:

1. Cea mai bună dintre cele m soluții candidat este acceptată **necondiționat**
2. Cea mai bună soluție candidat obținută până în momentul curent al căutării este reținută (se asigură proprietatea de **elitism** = nu se pierde o soluție candidat bună)

# Căutare locală: variante de perturbare

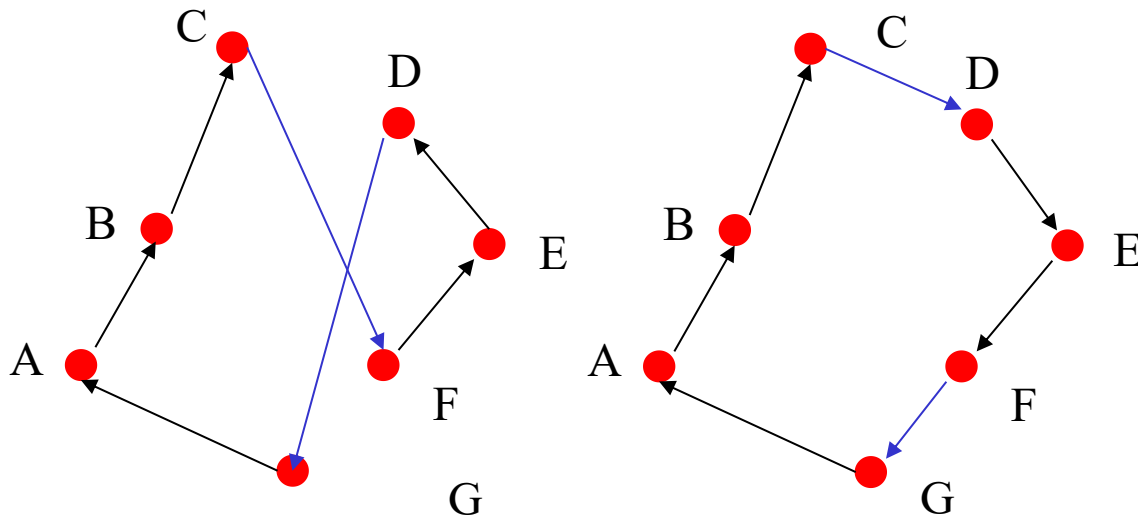
- **Scopul perturbării:** construirea unei noi soluții candidat pornind de la soluția curentă
- **Tipuri de perturbare (în funcție de natura perturbării):**
  - Deterministă
  - Aleatoare
- **Tipuri de perturbare (în funcție de intensitatea perturbării)**
  - Locală
  - Globală
- **Tipuri de perturbare (în funcție de spațiul de căutare)**
  - Specifică spațiilor discrete de căutare (înlocuirea uneia sau a mai multor componente)
  - Specifică spațiilor continue de căutare (adăugarea unui termen perturbator)

# Căutare locală: variante de perturbare

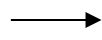
Probleme de optimizare combinatorială: noua configurație se alege în vecinătatea celei curente prin aplicarea unor transformări specifice problemei de rezolvat

Exemplu 1: TSP (Travelling Salesman Problem)

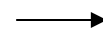
- Generarea unei noi configurații (transformare 2-opt)



ABCFEDG



ABC**FED**G



ABC**DE****F**G

Implementare:

1. Se aleg aleator două poziții
2. Se inversează ordinea elementelor din subtabloul delimitat de cele două poziții



# Căutare locală: variante de perturbare

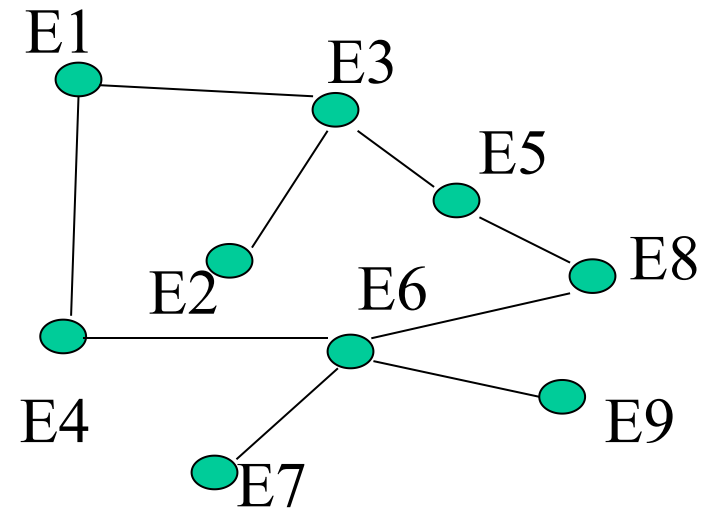
**Probleme de optimizare combinatorială:** noua configurație se alege în vecinătatea celei curente prin aplicarea unor transformări specifice problemei de rezolvat

Exemplu 2: **Generarea orarelor**

- Eliminarea conflictelor prin mutare sau interschimbare
- **Perturbarea unei configurații curente:**
  - Transferul unui eveniment ce încalcă o restricție puternică într-o zonă liberă

	S1	S2	S3
T1	E1	E3	E9
T2	E4		E8
T3	E6	E5	
T4	E2		E7

	S1	S2	S3
T1	E1		E9
T2	E4	E3	E8
T3	E6	E5	
T4	E2		E7



Graful conflictelor

# Căutare locală: variante de perturbare

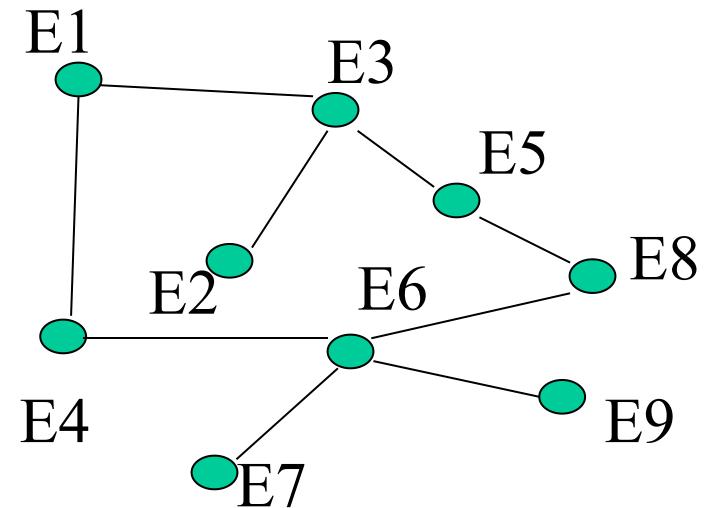
**Probleme de optimizare combinatorială:** noua configurație se alege în vecinătatea celei curente prin aplicarea unor transformări specifice problemei de rezolvat

Exemplu 2: **Generarea orarelor**

- Eliminarea conflictelor prin mutare sau interschimbare
- **Perturbarea unei configurații curente:**
  - Interschimbarea a două evenimente

	S1	S2	S3
T1	E1		E9
T2	E4	E3	E8
T3	E2	E5	
T4	E6		E7

	S1	S2	S3
T1	E1		E9
T2	E4	E3	E8
T3	E6	E5	
T4	E2		E7



Graful conflictelor

# Căutare locală: variante de perturbare

## Optimizare în domenii continue

### Perturbare aleatoare

```
Perturb(s,p,inf,sup,r)
  for i=1:n
    if rand(0,1)<=p then
      repeat
        n=rand(-r,r)
        until inf<=si+n<=sup
        si=si+n
      end
    end
  end
  return s
```

### Notății:

s=soluția candidat ce va fi perturbată

p=probabilitate de perturbare

r=„raza” perturbării

rand(a,b) = valoare aleatoare uniform repartizată în [a,b]

### Perturbare deterministă prin căutare directă (nu se folosesc derivate)

- Pattern Search (Hooke -Jeeves)
- Nelder - Mead

# Căutare locală: pattern search

Idee: modificare succesivă a componentelor soluției curente

PatternSearch(s,r)

s=aproximație inițială

r=valoare inițială

best=s

repeat

s'=s

for i=1:n

if  $f(s+r \cdot e_i) < f(s')$  then  $s'=s+r \cdot e_i$  end

if  $f(s-r \cdot e_i) < f(s')$  then  $s'=s-r \cdot e_i$  end

end

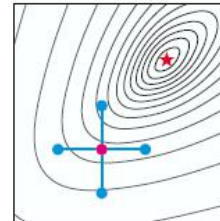
if s==s' then r=r/2

else s=s'

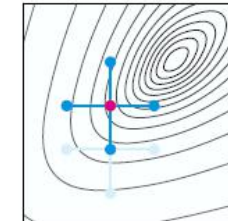
end

if  $f(s) < f(\text{best})$  then best=s

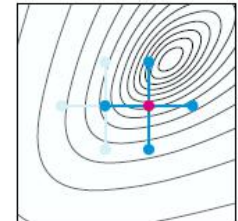
until <condiție de oprire>



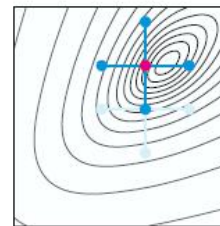
(a) Initial pattern



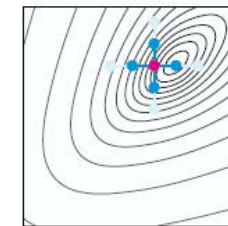
(b) Move North



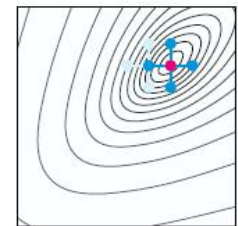
(c) Move West



(d) Move North



(e) Contract



(f) Move West

T.G. Kolda et al., Optimization by direct search: new perspectives on some classical and modern methods, SIAM Review, 45(3), 385-482, 2003

Obs:

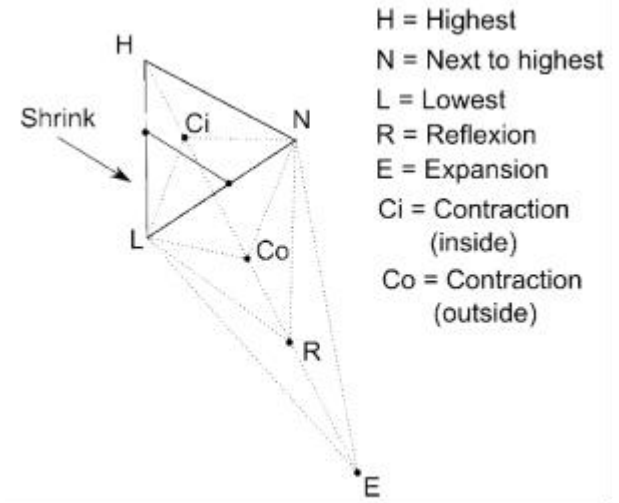
1.  $e_i = (0, 0, \dots, 0, 1, 0, \dots, 0)$  (1 se află pe poziția i)
2. la fiecare iterație se construiesc  $2n$  candidați dintre care se alege cel mai bun

# Căutare locală: algoritmul Nelder-Mead

**Idee:** căutarea se bazează pe utilizarea unui simplex în  $R^n$  (set de  $(n+1)$  puncte din  $R^n$ ) și aplicarea unor transformări asupra simplexului care permite „explorarea” domeniului soluțiilor

**Transformările** se bazează pe:

1. Ordonarea elementelor din simplex crescător după valoarea funcției obiectiv (pentru o problemă de minimizare)
2. Calculul mediei  $M(x_1, \dots, x_n)$  a celor mai bune  $n$  elemente din simplex
3. Construirea succesivă a unor noi elemente prin: **reflexie**, **expandare**, **contractție** (interioară, exterioară), **micșorare simplex**



# Căutare locală: algoritmul Nelder-Mead

Selectează  $(n+1)$  puncte din  $R^n$ :  $(x_1, x_2, \dots, x_{n+1})$

Repeat

calculează valorile funcției  $(f_1, f_2, \dots, f_{n+1})$

sortează  $(x_1, x_2, \dots, x_{n+1})$  astfel încât  $f_1 \leq f_2 \leq \dots \leq f_{n+1}$

$M = (x_1 + x_2 + \dots + x_n) / n$

Pas1 (reflexie - R):

$x_r = M + r(M - x_{n+1})$ ;

if  $f_1 \leq f(x_r) < f_{n+1}$  accept  $x_r$ ; continue;

else goto Pas 2

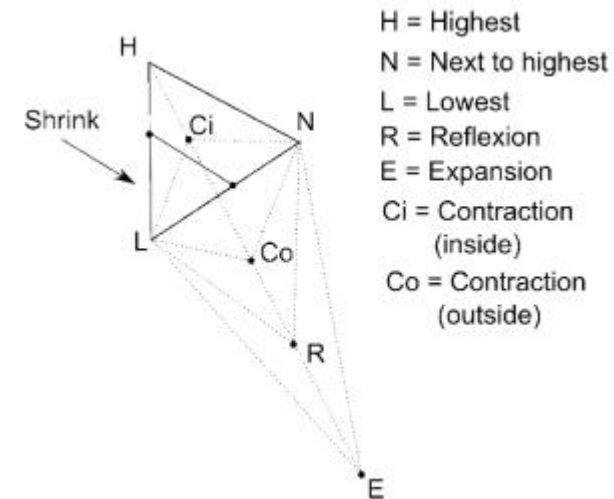
Pas 2 (expandare - E):

if  $f(x_r) < f_1$  then

$x_e = M + e(x_r - M)$

if  $f(x_e) < f(x_r)$  then accept  $x_e$ ; continue

else goto Pas 3



# Căutare locală: algoritmul Nelder-Mead

Pas 4 (contracție exterioară/interioară – Co/Ci):

if  $f_n \leq f(xr) < f_{n+1}$  then

$$xc = M + c(xr - M)$$

if  $f(xc) < f(xr)$  accept  $xc$ ; continue

else goto Pas 5

if  $f(xr) \geq f_{n+1}$  then

$$xcc = M - c(M - x_{n+1})$$

if  $f(xcc) < f_{n+1}$  then accept  $xcc$ ; continue

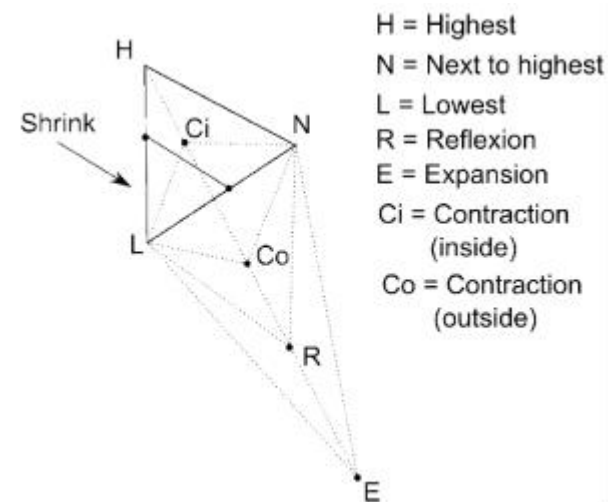
else goto Pas 5

Pas 5 (Micșorare - Shrink):

construiește un nou simplex:

$$x_1, v_2, \dots, v_{n+1} \quad \text{unde } v_i = x_i + s(x_i - x_1)$$

Parametrii:  $r=1$ ,  $e=2$ ,  $c=1/2$ ,  $s=1/2$



# De la optimizare locală la optimizare globală

- Perturbare:** se introduc (ocazional) perturbații aleatoare mari  
Exemplu: utilizare distribuție de probabilitate cu suport infinit (de exemplu repartiția normală sau repartiția Cauchy – **algoritm Matyas, Solis-Wets**)
- Restartarea algoritmului:** se reia procesul de căutare locală pornind de la altă configurație aleatoare (aleasă aleator)  
Exemplu: **căutare locală cu restartare** (local search with random restarts)
- Explorarea optimele locale:** optimul local curent e perturbat și folosit ca punct de pornire pentru căutarea altui optim  
Exemplu: **căutare locală iterată** (iterated local search)
- Selecție:** se acceptă (ocazional) și configurații de calitate mai slabă  
Exemplu: **căutare bazată pe simularea procesului de călire** (simulated annealing)



# Exemplu: algoritmul Matyas (1960)

```
s(0) = configurație inițială
k=0 // contor iteratie
e=0 // numar cazuri de esec
repeat
  generează un vector aleator cu
  componente normal repartizate ( $z_1, \dots, z_n$ )
  IF  $f(s(k)+z) < f(s(k))$  THEN  $s(k+1)=s(k)+z$ 
    e=0
  ELSE  $s(k+1)=s(k)$ 
    e=e+1
  k=k+1
UNTIL (k==kmax) OR (e==emax)
```

**Obs.** Perturbația aleatoare se aplică de regulă asupra unei singure componente (vectorul  $z$  are o singură componentă nenulă)

**Problema:** cum se aleg parametrii repartiției folosite pentru perturbarea valorii curente ?

**Exemplu:**  $N(0, \sigma)$

# Reminder: generare valori aleatoare cu repartiție normală

Pentru generarea valorilor aleatoare cu repartiție normală ( $N(0,1)$ ) se poate folosi [algoritmul Box-Muller](#)

```
u=rand(0,1) // valoare aleatoare uniform repartizată in (0,1)
v=rand(0,1)
r=sqrt(-2*ln(u));
z1=r*cos(2*PI*v)
z2=r*sin(2*PI*v)
RETURN z1,z2
// valorile z1 si z2 pot fi considerate ca fiind realizări a două variabile
// aleatoare independente
```

# Reminder: generare valori aleatoare cu repartiție normală

Alta variantă a [algoritmului Box-Muller](#):

```
repeat
```

```
  u=rand(0,1) // valoare aleatoare uniform repartizată in (0,1)
```

```
  v=rand(0,1)
```

```
  w=u2+v2
```

```
until 0<w<1
```

```
y=sqrt(-2ln(w)/w)
```

```
z1=u*y
```

```
z2=v*y
```

```
RETURN z1,z2
```

```
// valorile z1 si z2 pot fi considerate ca fiind realizări a două variabile
```

```
// aleatoare independente
```

**Obs:** pentru a obține valori corespunzătoare repartiției  $N(m,\sigma)$  se aplică transformarea:  $m+z*\sigma$

# Exemplu: algoritmul Solis-Wets (1981)

$s(0)$  = configurație inițială

$k=0$ ;  $m(0)=0$  // media vectorului perturbatie este ajustata in timp

repeat

    generează un vector aleator  $(z_1, \dots, z_n)$  cu componente avand repartitia  $N(m(k), 1)$

    IF  $f(s(k)+z) < f(s(k))$  THEN  $s(k+1)=s(k)+z$ ;

$m(k+1)=0.4*z+0.2*m(k)$

    IF  $f(s(k)-z) < \min\{f(s(k)), f(s(k)+z)\}$  THEN  $s(k+1)=s(k)-z$ ;

$m(k+1)=m(k)-0.4*z$

    IF  $f(s(k)-z) > f(s(k))$  AND  $f(s(k)+z) > f(s(k))$  THEN

$s(k+1):=s(k)$

$m(k+1):=0.5*m(k)$

$k:=k+1$

UNTIL  $(k==kmax)$

# Căutare locală cu restartare

## Idee:

- procesul de căutare se repetă de mai multe ori pornind de la configurații inițiale diferite
- se alege configurația cea mai bună

## Observații:

- Condiția de oprire a căutării locale se poate baza pe o decizie aleatoare (ex: intervalul de timp alocat poate fi aleator)
- Etapele de căutare aleatoare sunt independente (noua configurație inițială este aleasă aleator) – nu se folosește informația colectată la etapa anterioară

## Random Restart

s=configurație inițială

best=s

Repeat

  repeat

    r=perturb(s)

    if  $f(r) \leq f(s)$  then s=r

  until <conditie oprire căutare locală>

  if  $f(s) < f(\text{best})$  then best =s

  s=altă configurație inițială

until <conditie oprire căutare>

return best

# Căutare locală iterată

## Idee:

- Configurația inițială de la următoarea etapă se alege în vecinătatea optimului local identificat la etapa anterioară (eventual doar dacă acesta este mai bun decât cel identificat anterior)

## Observații:

- Pentru generarea configurației inițiale corespunzătoare fiecărei iterații se utilizează o perturbare mai „agresivă” decât cea utilizată în etapa de căutare locală

## Iterated Local Search (ILS)

s=configurație inițială

s0=s; best=s

Repeat

  repeat

    r=perturbSmall(s)

    if  $f(r) \leq f(s)$  then  $s=r$

  until <condiție oprire căutare locală>

  if  $f(s) < f(\text{best})$  then  $\text{best} = s$

  s0=alege(s0,s)

  s=perturbLarge(s0)

until <conditie oprire căutare>

return best

# Simulated Annealing

## Idee:

- se acceptă, cu o anumită probabilitate, și ajustări ale configurației curente care conduc la creșterea funcției obiectiv

## Sursa de inspirație:

- procesul de reorganizare a structurii unui solid supus unui tratament termic:
  - Solidul este încălzit (topit): particulele sunt distribuite într-o manieră aleatoare
  - Solidul este răcit lent: particulele se reorganizează pentru a se ajunge la configurații de energie din ce în ce mai mică

## Terminologie:

simulated annealing = tratament termic simulat = călire simulată

Istoric: Metropolis(1953), Kirkpatrick, Gelatt, Vecchi (1983), Cerny (1985)

# Simulated Annealing

Analogie:

Proces fizic:

- Energia sistemului → Funcție obiectiv
- Starea sistemului → Configurație (soluție candidat)
- Modificarea stării sistemului → Perturbarea configurației curente
- Temperatura → Parametru de control a procesului de optimizare

SA= metoda euristică inspirată de procese fizice



# Simulated Annealing

## Puțină fizică:

- fiecare stare a unui sistem este caracterizată de o anumită probabilitate de apariție
- probabilitatea asociată unei stări depinde de energia stării și de temperatura sistemului (ex: distribuția Boltzmann)

$$P_T(s) = \frac{1}{Z(T)} \exp\left(-\frac{E(s)}{k_B T}\right)$$

$$Z(T) = \sum_{s \in S} \exp\left(-\frac{E(s)}{k_B T}\right)$$

$E(s)$  = energia stării  $s$   
 $T$  = temperatura sistemului  
 $k_B$  = constanta Boltzmann  
 $Z(T)$  = funcția de partiție  
(factor de normalizare)

# Simulated Annealing

Puțină fizică:

- **Valori mari ale lui T** (T tinde la infinit): argumentul lui exp este aproape 0 => stările sunt echiprobabile
- **Valori mici ale lui T** (T tinde la 0): vor avea probabilitate nenulă doar stările cu energia nulă

$$P_T(s) = \frac{1}{Z(T)} \exp\left(-\frac{E(s)}{k_B T}\right)$$

$$Z(T) = \sum_{s \in S} \exp\left(-\frac{E(s)}{k_B T}\right)$$

$E(s)$  = energia stării  $s$   
 $T$  = temperatura sistemului  
 $k_B$  = constanta Boltzmann  
 $Z(T)$  = funcția de partiție  
(factor de normalizare)

# Simulated Annealing

Cum folosim acest lucru pentru rezolvarea unei probleme de optimizare ?

- Ar fi suficient să generăm configurații în conformitate cu distribuția Boltzmann pentru valori din ce în ce mai mici ale temperaturii
- **Problema:** dificil de calculat  $Z(T)$  (presupune calculul unei sume pentru toate stările posibile, adică generarea tuturor configurațiilor spațiului de căutare - IMPOSIBIL de realizat practic dacă spațiul configurațiilor este mare)
- **Soluție:** se aproximează distribuția prin simularea evoluției unui proces stohastic (lanț Markov) a cărei distribuție staționară coincide cu distribuția Boltzmann => **algoritmul Metropolis**

# Simulated Annealing

Algoritmul Metropolis (1953)

$s(0)$ =aproximație inițială

$k=0$

REPEAT

$s'$ =perturb( $s(k)$ )

IF  $f(s') < f(s(k))$  THEN  $s(k+1)=s'$  (necondiționat)

ELSE  $s(k+1)=s'$

cu probabilitatea  $\min\{1, \exp(-(f(s')-f(s(k)))/T)\}$

$k=k+1$

UNTIL “este indeplinita o conditie de terminare”

# Simulated Annealing

## Algoritmul Metropolis – proprietăți

- Alta probabilitate de acceptare:  
$$P(s(k+1)=s') = 1/(1+\exp((f(s')-f(s(k)))/T))$$
- Implementarea unei reguli de atribuire cu o anumită probabilitate  
 $u=\text{rand}(0,1)$   
IF  $u < P(s(k+1)=s')$  THEN  $s(k+1)=s'$   
ELSE  $s(k+1)=s(k)$
- **Valori mari pentru T** -> probabilitate mare de acceptare a oricarei configurații (similar cu **căutarea pur aleatoare**)  
**Valori mici pentru T** -> probabilitate mare de acceptare doar pentru configurațiile care conduc la micșorarea funcției obiectiv (similar cu o **metodă de descreștere** sau căutare de tip greedy)

# Simulated Annealing

## Algoritmul Metropolis – proprietăți

- Generarea unor noi configurații depinde de problema de rezolvat

## Optimizare în domenii continue

$$s' = s + z$$

$$s = (z_1, \dots, z_n)$$

$z_i$  : generata in cf. cu repartitia

- $N(0, T)$
- Cauchy(T) (Fast SA)
- altele

## Optimizare combinatorială

Noua configurație se selectează determinist/aleator din **vecinătatea** configurației curente

**Exemplu:** TSP – transformare de tip 2-opt

# Simulated Annealing

Simulated Annealing = aplicare repetată a algoritmului Metropolis pentru valori din ce în ce mai mici ale temperaturii

## Structura generală

Init  $s(0)$ ,  $T(0)$

$i=0$

REPEAT

    aplică Metropolis (pentru una sau mai multe iteratii)

    calcul  $T(i+1)$

$i=i+1$

UNTIL  $T(i) < \text{eps}$

**Problema:** alegerea schemei de modificare a temperaturii (“cooling scheme”)

# Simulated Annealing

Scheme de răcire:

$$T(k) = T(0)/(k+1)$$

$$T(k) = T(0)/\ln(k+c)$$

$$T(k) = aT(k-1) \quad (a < 1, \text{ ex: } a = 0.995)$$

- Obs.** 1.  $T(0)$  se alege astfel încât la primele iterații să fie acceptate aproape toate configurațiile generate (pentru a asigura o bună explorare a spațiului soluțiilor)
2. Pe parcursul procesului iterativ este indicat să se rețină de fiecare dată cea mai bună valoare întâlnită



# Simulated Annealing

Proprietăți de convergență:

Dacă sunt satisfăcute proprietățile:

- $P_g(s(k+1)=s'|s(k)=s) > 0$  pentru orice  $s$  și  $s'$  (probabilitatea de trecere între oricare două configurații este nenulă)
- $P_a(s(k+1)=s'|s(k)=s) = \min\{1, \exp(-(f(s')-f(s))/T)\}$  (probabilitate de acceptare de tip Metropolis)
- $T(k) = C/\lg(k+c)$  (schema logaritmică de răcire)

Atunci  $P(f(s(k))=f(s^*)) \rightarrow 1$  ( $s(k)$  tinde în probabilitate la minimul global  $s^*$  când  $k$  tinde la infinit)

# Simulated Annealing

Variante: alte probabilități de acceptare (Tsallis)

$$P_a(s') = \begin{cases} 1, & \Delta f \leq 0 \\ (1 - (1 - q)\Delta f / T)^{1/(1-q)}, & \Delta f > 0, (1-q)\Delta f \leq 1 \\ 0, & \Delta f > 0, (1-q)\Delta f > 1 \end{cases}$$

$$\Delta f = f(s') - f(s)$$

$$q \in (0,1)$$

# Simulated Annealing

Exemplu: problema comis voiajorului

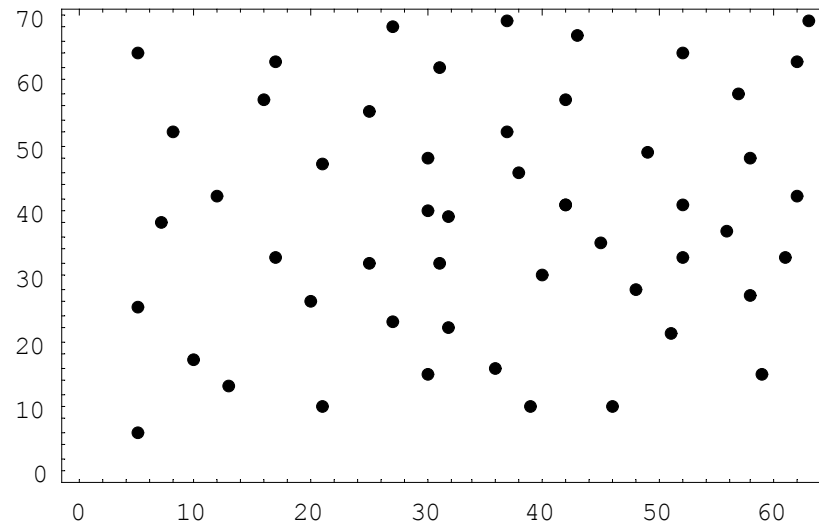
(TSPLib: <http://comopt.ifl.uni-heidelberg.de/software/TSPLIB95>)

Instanta test: eil51 – 51 orase

Parametrii:

- 5000 iterații cu modificarea parametrului T la fiecare 100 de iterații
- $T(k) = T(0) / (1 + \log(k))$

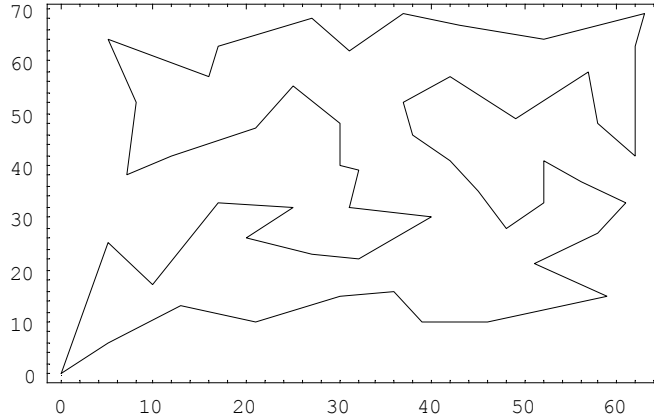
Distribuția oraselor



# Simulated Annealing

Exemplu: problema comis voiajorului

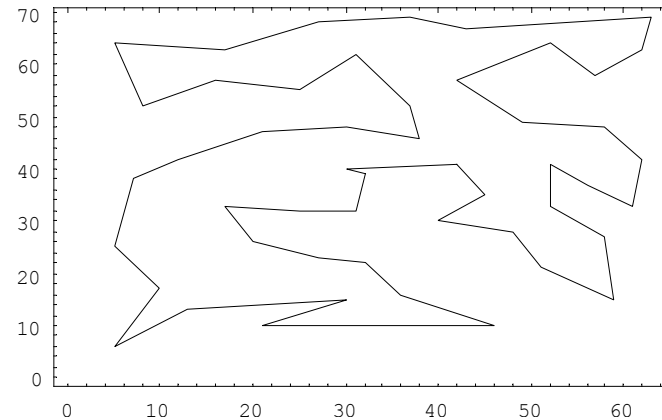
Instanta test: eil51 (TSPLib)



$T(0)=10, \text{cost}=478.384$

Cost minim: 426

$T(0)=5, \text{cost}=474.178$



$T(0)=1, \text{cost}=481.32$

