Prolog & Lisp vs. Java. Hill-climbing

Ruxandra Stoean
http://inf.ucv.ro/~rstoean
ruxandra.stoean@inf.ucv.ro

Avantajele programelor PNP

- ✓Sunt orientate catre implementari logice.
- ✓Sunt extrem de potrivite pentru sistemele expert apartinand domeniului inteligentei artificiale.
- ✓Sunt usor de analizat, transformat, verificat, intretinut.
- ✓In general, sunt eficiente si competitive in comparatie cu programele nedeclarative.
- ✓Programul este mai degraba “intrebat” decat executat.

Dezavantajele programelor PNP

- Nu sunt usor implementabile si utilizabile pentru algoritmii matematici de cautare si optimizare din cadrul inteligentei artificiale.
- Nu sunt cele mai potrivite pentru exprimarea algoritmilor folositi in industrie.
- Au mecanisme mai putin directe decat ale celor nedeclarative si sunt considerate de multe ori mai “neprietenoase”.

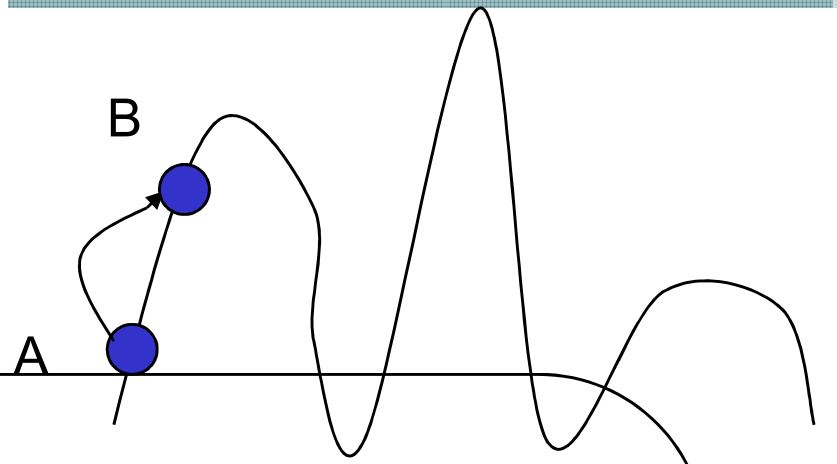
O privire de ansamblu

- Cursul prezent incearca formularea unei concluzii finale asupra:
  - Diferentelor intre programarea procedurala si cea nonprocedurala
  - Avantajelor si dezavantajelor fiecareia
- Se doreste asadar:
  - Rezolvarea unei probleme reale
  - Utilizarea unui algoritm clasic de inteligenta artificiala
  - Comparatia intre implementarile in Prolog, Lisp (programare nonprocedurala) si Java (programare procedurala)

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

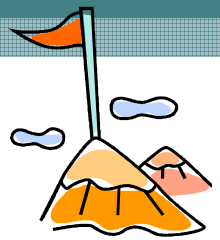


functia hill_climbing(*problema*) ***intoarce*** o solutie
Se pastreaza la fiecare reapelare: nodul *curent* si nodul *urmator.*

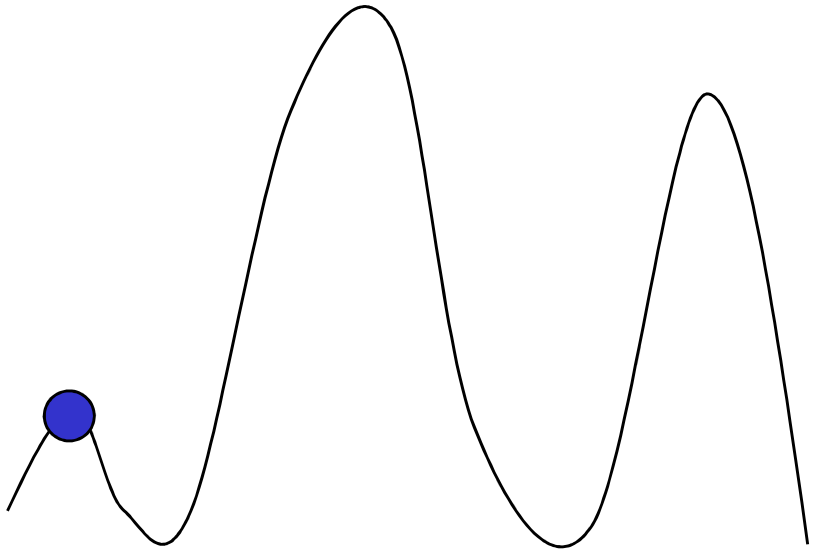
curent = genereaza_nod(stare_initiala[problema])

Cat timp este posibil *executa*
    urmator = un succesor al nodului *curent*
    Daca eval(*urmator*) < eval(*curent*) *atunci*
        intoarce *curent*
    curent = urmator
Sfarsit cat timp

Dezavantaje hill-climbing

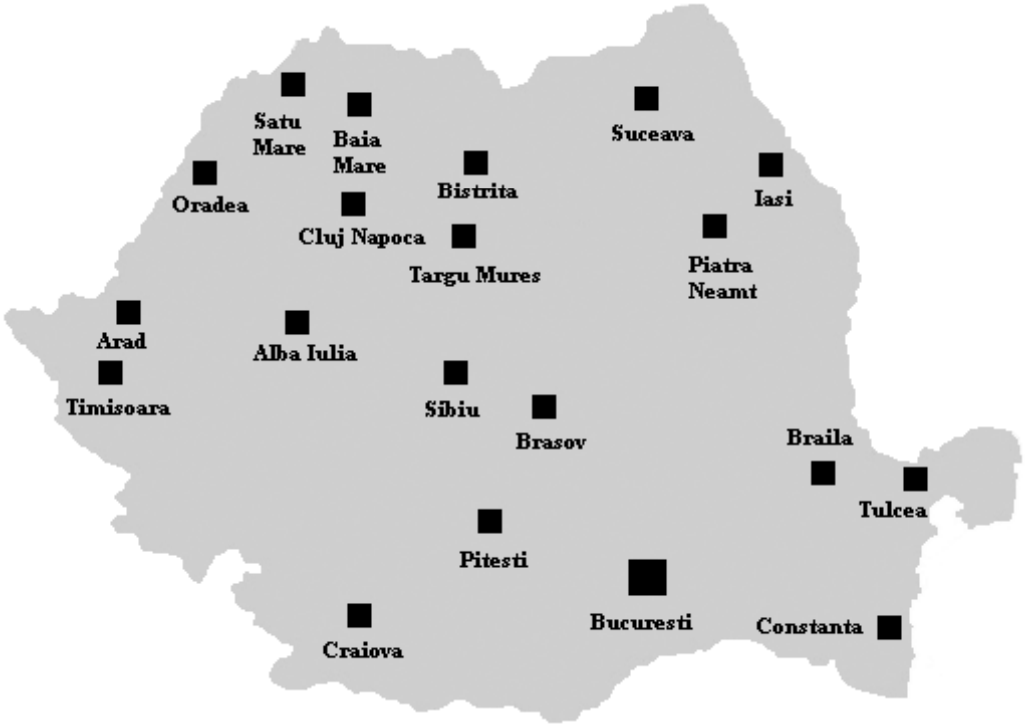


- **Maxime locale**: este vorba de un varf care este mai mic decat cel mai inalt varf din spatiul starilor. Cand se ajunge la maxime locale, algoritmul se opreste pentru ca nu mai poate *cobori* dealul.
  - Solutia gasita poate fi foarte slaba!



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!



Implementarea in Prolog

- Inregistrarea oraselor si a distantelor dintre acestea se va realiza prin adaugarea de fapte si formularea comutativitatii.

```
dist(1,2,10).
dist(1,3,25).
dist(1,4,30).
dist(1,5,15).
dist(1,6,30).
dist(1,7,40).
dist(2,3,20).
dist(2,4,10).
dist(2,5,25).
dist(2,6,35).
dist(2,7,45).
dist(3,4,5).
dist(3,5,20).
dist(3,6,25).
dist(3,7,7).
dist(4,5,30).
dist(4,6,15).
dist(4,7,20).
dist(5,6,17).
dist(5,7,10).
dist(6,7,10).
orase(7).
```

```
d(A,B,X):-dist(A,B,X).
d(A,B,X):-dist(B,A,X).
```

Predicatul principal

- Apeleaza algoritmul de hill-climbing si afiseaza drumul optim si costul sau.
- Se specifica de asemenea configuratia principala si numarul de iteratii (considerat drept conditie de oprire).

```
go:-start(Drum), hill_climber(Drum, 0, DrumOpt),
   writeln('Drumul optim este'), writeln(DrumOpt),
   write('Costul sau este '), eval(DrumOpt, FOpt),
   writeln(FOpt).

start([1, 2, 3, 4, 5, 6, 7]).
iteratii(200).
```

Algoritmul de hill-climbing

- Se implementeaza functia definita in algoritm.
- Daca nu s-a ajuns la numarul de iteratii maxim, se incearca “urcarea” intr-o configuratie succesoare.
- Daca se va reusi, nodul curent devine cel nou gasit si se reapeleaza predicatul recursiv.

```
hill_climber(Drum, GenAnt, Drum):-iteratii(No), GenAnt
   == No.
hill_climber(Drum, GenAnt, DrumOpt):-iteratii(No),
   GenAnt \= No, eval(Drum, F), climb(Drum, F,
   DrumNou),
   GenNou is GenAnt + 1, hill_climber(DrumNou, GenNou,
   DrumOpt).
```

Predicatul de urcare

- Se cauta un succesor al nodului curent.
- Daca evaluarea acestuia este mai buna decat a celui curent, nodul curent devine cel nou generat.
- Altfel, nodul curent ramane neschimbat.

```
climb(Drum, F, DrumNou):-mutatie(Drum,
   Drum1), eval(Drum1, FNou), FNou < F,
   DrumNou = Drum1.
climb(Drum, _, Drum).
```

Evaluarea unei configuratii

- Evaluarea presupune costul drumului asociat unei configuratii, adica suma distantelor dintre orasele din circuit.

```
eval([X|Rest], F):-eval1([X|Rest], F1),
   ultim([X|Rest], U), d(U, X, D), F is F1 + D.

eval1([_], 0).
eval1([X,Y|Rest], F):-d(X,Y,C), eval1([Y|Rest], F1),
   F is F1 + C.
ultim([X], X).
ultim([_|Rest], X):-ultim(Rest, X).
```

Mutarea intr-un nod succesor

- Presupune, de fapt, aplicarea unui operator de mutatie asupra configuratiei curente.
- Se genereaza aleator doua pozitii si valorile corespunzatoare sunt interschimbate.

```
mutatie(Drum, Drum1):-orase(N), genereaza(N, Poz1,
   Poz2), cauta(Drum, Poz1, X),
 cauta(Drum, Poz2, Y), schimba(Drum, X, Y, Drum1).

genereaza(N, Poz1, Poz2):- repeat, P1 is random(N) + 1, P2
   is random(N) + 1, P1 \= P2, !, Poz1 = P1, Poz2 = P2.
```

Interschimbare

- Se cauta in configuratia curenta care sunt valorile de pe cele doua pozitii generate aleator.
- Cele doua valori se schimba intre ele.

```
cauta([X|_], 1, X).
cauta([_|Rest], Poz, El):-Poz1 is Poz - 1, cauta(Rest, Poz1,
   El).

schimba([], _, _, []).
schimba([X|Rest], A, B, [B|Rest1]):- X == A, schimba(Rest,
   A, B, Rest1).
schimba([X|Rest], A, B, [A|Rest1]):- X == B, schimba(Rest,
   A, B, Rest1).
schimba([X|Rest], A, B, [X|Rest1]):- X \= A, X \= B,
   schimba(Rest, A, B, Rest1).
```

Implementarea in Lisp

- Inregistrarea conexiunilor dintre orase si costurilor asociate.

```
(setf (get 'n1 'conectat) '(n2 n3 n4 n5 n6 n7))
(setf (get 'n1 'costuri) '(10 25 30 15 30 40))
(setf (get 'n2 'conectat) '(n1 n3 n4 n5 n6 n7))
(setf (get 'n2 'costuri) '(10 20 10 25 35 45))
(setf (get 'n3 'conectat) '(n1 n2 n4 n5 n6 n7))
(setf (get 'n3 'costuri) '(25 20 5 20 25 7))
(setf (get 'n4 'conectat) '(n1 n2 n3 n5 n6 n7))
(setf (get 'n4 'costuri) '(30 10 5 30 15 20))
(setf (get 'n5 'conectat) '(n1 n2 n3 n4 n6 n7))
(setf (get 'n5 'costuri) '(15 25 20 30 17 10))
(setf (get 'n6 'conectat) '(n1 n2 n3 n4 n5 n7))
(setf (get 'n6 'costuri) '(30 35 25 15 17 10))
(setf (get 'n7 'conectat) '(n1 n2 n3 n4 n5 n6))
(setf (get 'n7 'costuri) '(40 45 7 20 10 10))
```

Functia principala si ciclul de hill-climbing

- Se incepe cu o configuratie initiala si iteratia 0.
- Pana cand se ajunge la numarul de iteratii maxim, se incearca “urcarea” intr-o configuratie mai buna.
- Atunci cand algoritumul se opreste, se intoarce drumul optim si costul sau.

```
(defun porneste ()
(hill_climber '(n1 n2 n3 n4 n5 n6 n7) 0))

(defun hill_climber (l it)
(if (= it 200) (cons (performanta l) l)
(hill_climber (climb l) (+ it 1))))
```

“Urcarea” intr-o configuratie mai buna

- Se genereaza o noua configuratie a drumului prin schimbul valorilor intre doua pozitii aleator selectate.
- Daca evaluarea acesteia este mai buna decat cea a nodului curent, se retine noul drum drept cel curent.

```
(defun climb (l)
(if (< (performanta (mutatie l)) (performanta l)) (get
   'lista 'mut) l))

(defun mutatie (l)
(setf (get 'lista ' mut) (schimba l (cauta l (+ (random
   7) 1)) (cauta l (+ (random 7) 1)))))
```

Evaluarea unei configuratii

- Drumul va fi permanent dat de lista nodurilor.
- Pentru a calcula costul sau, va trebui sa gasim lista costurilor asociata acestei succesiuni de noduri, ce va fi apoi insumata.

```
(defun performanta (l)
(evaluare (append (transforma l) (fl l))))

(defun evaluare (l)
(if (null l) 0
(+ (first l) (evaluare (rest l)))))

(defun fl (l)
(cons (gaseste (first (last l)) (first l)) '()))
```

Lista costurilor asociate drumului

- Pentru doua orase, se cauta pozitia celui de-al doilea in lista de conexiuni a primului.
- Apoi se cauta costul corespunzator acestei pozitii in lista costurilor asociat primului oras.

```
(defun transforma (l)
(if (null (rest l)) '() (cons (gaseste (first l) (second l))
   (transforma (rest l)))))

(defun gaseste (x y)
(cauta (get x 'costuri) (cauta2 (get x 'conectat) y 1))
)
```

Doua functii de cautare

```
; intoarce elementul de pe pozitia poz din lista l
(defun cauta (l poz)
(if (= poz 1) (first l) (cauta (rest l) (- poz 1))
)
)

; intoarce pozitia elementului el din lista l
(defun cauta2 (l el p)
(if (eql el (first l)) p (cauta2 (rest l) el (+ p 1))
)
)
```

Mutatia asupra unei configuratii

- Se genereaza aleator doua pozitii in vector si se interschimba elementele de la aceste locatii.
- Se retine noua configuratie, pentru cazul in care este mai performanta.

```
(defun mutatie (l)
(setf (get 'lista ' mut) (schimba l (cauta l (+ (random 7) 1))
   (cauta l (+ (random 7) 1)))))

(defun schimba (l p1 p2)
(cond ((null l) '())
((eql (first l) p1) (cons p2 (schimba (rest l) p1 p2)))
((eql (first l) p2) (cons p1 (schimba (rest l) p1 p2)))
(t (cons (first l) (schimba (rest l) p1 p2)))))
```

Implementarea in Java

- Inregistrarea conexiunilor intre orase si a costurilor asociate, plus comutativitatea.

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        d[j][i] = d[i][j];
```

```
d[0][1] = 10;
d[0][2] = 25;
d[0][3] = 30;
d[0][4] = 15;
d[0][5] = 30;
d[0][6] = 40;
d[1][2] = 20;
d[1][3] = 10;
d[1][4] = 25;
d[1][5] = 35;
d[1][6] = 45;
```

```
d[2][3] = 5;
d[2][4] = 20;
d[2][5] = 25;
d[2][6] = 7;
d[3][4] = 30;
d[3][5] = 15;
d[3][6] = 20;
d[4][5] = 17;
d[4][6] = 10;
d[5][6] = 10;
```

Functia principala

```
public void go()
    {
     int t = 1;

     // initializarea drumului - primei solutii potentiale
     for (int i = 0; i < n; i++)
            drum[i] = i;
     evalD = eval(drum);

     while (t < it)
     {
            for (int i = 0; i < n; i++)   // se lucreaza cu un drumTemp pentru modificari
                   drumTemp[i] = drum[i];
            climb();
            t++;
     }

     // afisam solutia finala si costul acesteia
     for (int i = 0; i < n; i++)
            System.out.print(drum[i] + "  ");
     System.out.println(" cu evaluarea " + evalD);
    }
```

Urcarea intr-o noua solutie

```
public void climb()
   {
    mutatie();
    double evalNou = eval(drumTemp);
    if (evalNou < evalD)
    {
         for (int i = 0; i < n; i++)
              drum[i] = drumTemp[i];
         evalD = evalNou;
    }
   }
```

Evaluarea unei solutii

```
public double eval(int[] sol)
   {
    double evaluare = 0;

    for (int i = 0; i < n - 1; i++)
          evaluare += d[sol[i]][sol[i + 1]];
    evaluare += d[sol[n - 1]][sol[0]];

    return evaluare;
   }
```

Mutatia asupra unei solutii

```
public void mutatie()
   {
    int p1 = -1, p2 = -1;

    // se genereaza doua pozitii aleatoare in vectorul drumTemp,
    // adica se aleg doua orase care se vor schimba in traseul posibil
    while (p1 == p2)
    {
          p1 = (int)Math.round((n - 1)* Math.random());
          p2 = (int)Math.round((n - 1)* Math.random());
    }

    // schimba pozitiile p1 si p2 in drumTemp
    int temp;
    temp = drumTemp[p1];
    drumTemp[p1] = drumTemp[p2];
    drumTemp[p2] = temp;
   }
```

Ce sa alegem?

Programare procedurala	Programare nonprocedurala
➢ Implementare directa/clasica a algoritmilor ➢ Eficienta ➢ Portabilitate ➢ Implementare dificila a cunostintelor despre problema	<u>Prolog</u> ❖ Inregistrare directa a cunostintelor despre problema ❖ Recursivitate ❖ Gandire speciala <u>Lisp</u> ✓ Apropiere de gandirea matematica ✓ Recursivitate ✓ Sintaxa dificila

Pana la examen…