BLIA-MAS Laborator 04

Cuprins:

a. Strategii de cautare (partea I-a)
b. Strategii de cautare neinformata
c. Strategii de cautare euristica
d. Strategii de cautare (partea a II-a)

STRATEGII DE CAUTARE (partea I-a)

STRATEGII DE CAUTARE NEINFORMATA

O strategie de cautare poate fi aplicata daca sunt indeplinite urmatoarele conditii:

- Universul problemei poate fi descris simbolic, iar descrierea initiala a problemei si obiectele candidate la solutie care apar in timpul cautarii se pot identifica cu o multime de stari. Multimea starilor investigate din starea initiala pana in momentul ajungerii in starea finala formeaza spatiul de cautare a solutiei.

- Exista o multime de operatori care realizeaza trecerea dintr-o stare in alta.

- Exista o modalitate de alegere a operatorului de aplicat la un moment dat unei stari, numit strategie de control. Acesta trebuie sa asigure gasirea soultiei, daca aceasta exista (completitudine) si sa fie cat mai eficienta, intr-un cuvant trebuie sa fie sistematica.

Strategii de cautare neinformata (de baza) inspecteaza sistematic toate starile spatiului de cautare pana la momentul gasirii starii finale, urmand o ordine dinainte prestabilita in inspectarea starilor. Cele mai importante strategii de acest fel sunt cautarea pe nivel si cautarea in adancime.

Spatiul de cautare poate fi vazut ca un graf, ale carui noduri sunt stari, iar arcele reprezinta operatori de trecere dintr-o stare in alta. In cazul in care fiecare stare are un singur predecesor, spatiul de cautare este un arbore. Adancimea unui nod/stare se defineste recursiv astfel:

1. Adancime(S1)=0, unde S1 este nodul stare initiala,
2. Adancime(S)=Adancime(Sp)+1, unde Sp este nodul predecesor nodului S.

Cautare in adancime

In cazul cautarii in adancime ordinea de parcurgere a starilor este invers proportionala cu adancimea acestora. Astfel, analizam la un moment dat starea cu adancimea cea mai mare, succesorii unei stari fiind considerati de la stanga la dreapta. Vom considera ca

spatiul de cautare este un arbore bina, fiecare nod-stare fiind identificat printr-un intreg pozitiv, iar succesorii starii i sunt starile 2i si 2i+1.

```
(defparameter *FINAL* 18)

(defparameter *STFINALE* '(18 10 14))

(defparameter *LIM* 10)

(defparameter *SOLUTII* nil)

;Recunoasterea starii finale

(defun estescop (x) (equal x *FINAL*))

;Generarea succesorilor (functia succf) se opreste daca o stare depaseste o anumita
;valoare limita. Rezultatul functiei succf este o lista formata din cei doi succesori ai
;nodului primit ca parametru.

(defun succf(x)
        (if (< x *LIM*) (list (* 2 x) (+ 1 (* 2 x)))))

;Cautarea este realizata de functia adanc care primeste ca parametru o lista FRONTIERA
;care contine toate starile vizitate la un moment dat, pentru care nu s-au generat inca
;succesorii. Daca aceasta lista este vida, s-au epuizat toate starile spatiului de ;cautare,
fara a se fi gasit solutia, deci problema nu are solutie. In caz contrar se ;verifica daca
prima stare din lista este o stare finala, caz in care cautarea se ;opreste si se afiseaza
solutia. Daca nu s-a gasit solutia, cautarea continua prin ;apelul recursiv al functiei adanc,
noua lista FRONTIERA fiind obtinuta din cea curenta ;prin eliminarea starii analizate si
adaugarea la inceput a tuturor succesorilor ;acesteia.

(defun adanc (FRONTIERA)
        (format t "~& ADINC: ~a" FRONTIERA)
        (cond ((null FRONTIERA) nil)
       ((estescop (first FRONTIERA)) (first FRONTIERA))
            (t (adanc (append (succf (first FRONTIERA))
                                      (rest FRONTIERA))))))


;Exemplu de apel:
;(adanc (list 1))
```

Backtracking

Ca si cautarea in adancime, backtracking (cautarea cur eveniri) este o strategie de cautare cur evenire (tentativa). Diferenta consta in faptul ca la backtracking sunt memorate numai starile de pe calea dintre starea initiala si cea curenta, fiind posibila intoarcerea la un moment dat numai in acele stari memorate si continuarea cautarii pe alte drumuri. Si in acest caz sunt expandate nodurile cele mai recent generate, cu alte cuvinte cele care au adancime maxima.

```
;generarea caii la o solutie, folosind backtracking

(defparameter *STFINALE* '(18 10 14))

(defparameter *LIM* 10)

(defparameter *SOLUTII* nil)

;Presupunem ca exista mai multe stari finale, astfel functia estescopbkt testeaza
;apartenenta parametrului la lista *STFINALE*. Functiile de cautare bkt,bkt1,bkt2
;primesc ca parametru o lista stari continand starile aflate pe drumul dintre starea ;curenta
si starea initiala. Daca lista de stari este vida, atunci nu a fost specificata ;nici o stare
initiala, deci problema nu are solutie. Daca starea curenta ((first ;stari)) este o stare finala
(a doua ramura a formei cond), atunci lista stari contine ;etapele drumului dintre starea
initiala si cea finala, in ordine inversa. In functia ;bkt se tipareste pe loc solutia, iar in
bkt1 solutia curenta este introdusa in lista ;*SOLUTII*, la inceputul acesteia.
Macrodefinitia de sistem (push <element> <stiva>) are ;ca rezultat inserarea obiectului
<element> la inceputul listei <stiva>, modificand
;lista initiala. Ulterior, elementele listei *SOLUTII* sunt afisate in ordinea gasirii ;lor, in
functia rezbkt.

(defun estescopbkt (x) (member x *STFINALE*))

(defun succf(x)
        (if (< x *LIM*) (list (* 2 x) (+ 1 (* 2 x)))))

;functie ce afiseaza toate solutiile
(defun bkt (stari)
        (format t "~& bkt: ~a" stari)
        (cond ((null stari) nil)
          ((estescopbkt (first stari))
              (format t "~& SOLUTII:   ~a" stari))
             (t (dolist (urm (succf (first stari)))
                    (bkt (cons urm stari))))))
```

```
;functie ce memoreaza toate solutiile in lista *SOLUTII*
(defun bkt1 (stari)
        (cond ((null stari) nil)
            ((estescopbkt (first stari))
             (push stari *SOLUTII*))
            (t (dolist (urm (succf (first stari)))
                  (bkt1 (cons urm stari))))))

;functie ce se opreste dupa ce a gasit o solutie
(defun bkt2 (stari)
        (format t "~& bkt: ~a" stari)
        (cond ((null stari) nil)
                ((estescopbkt (first stari))
                 (format t "~& SOLUTII:   ~a" stari) 1 )
            (t  (do ((indicator nil)
                      (succesor (succf (first stari)) (rest succesor)))
                     ((null succesor) indicator)
                (if (bkt2 (cons (first succesor) stari))
                   (progn
                          (setf succesor '())
                             (setf indicator 1)
; varibila indicator este folosita pentru oprirea executiei ciclului do
; in momentul gasirii primei solutii se intoarce valoarea 1
; cand s-a revenit dintr-un apel recursiv cu valoarea 1, ciclul do curent este parasit ;si rezultatul 1 este transmis apelului superior in lantul recursiv, astfel ca, in final ;se incheie si primul apel al functiei. Pentru iesirea din ciclu, se foloseste forma ;speciala setf care face adevarata conditia de iesire din ciclu
                    1)
                  nil )))))

(defun rezbkt (init)
        (bkt1 (list init))
        (dolist (solutie (reverse *SOLUTII*))
          (format t "~& ~a" solutie)))


;Exemple de apel
(bkt (list 1))
(rezbkt 1)
(bkt2 (list 1))
```

Cautarea pe nivel

Ordinea de parcurgere a starilor este proportionala cu adancimea acestora. Analizam la un moment dat starea cu adancimea cea mai mica dintre starile din FRONTIERA.

Cautarea pe nivel se poate obtine prin modificarea cautarii in adancime adaugand lista succesorilor starii curente la sfarsitul listei FRONTIERA, in loc de a o adauga la inceput, ca in functia adanc. In acest fel
se expandeaza mai intai starile de adancime minima. Functia append din ultima linie a functiei adanc se va inlocui cu functia la_urma:

```
(defun la_urma (succesori FRONTIERA)
        (append FRONTIERA succesori))
```

Scriem o singura functie de cautare care in functie de parametrul strategie primit, aplica cautarea pe nivel sau in adancime (la-urma, append):

```
(defparameter *FINAL* 18)
(defparameter *STFINALE* '(18 10 14))
(defparameter *LIM* 10)
(defparameter *SOLUTII* nil)

(defun estescop (x) (equal x *FINAL*))

(defun succf(x) (if (< x *LIM*) (list (* 2 x) (+ 1 (* 2 x)))))

(defun caut (stari scop succesori strategie)
; caut este o functie generala care primeste ca parametru strategia dorita.
(format t "~& CAUT: ~a" stari)
(cond ((null stari) nil)
    ((funcall scop (first stari)) (first stari))
    (t (caut (funcall strategie
                        (funcall succesori (first stari)) (rest stari))
        scop succesori strategie))))


(defun cautadanc (init scop succesori)
        (caut (list init) scop succesori #'append))

(defun cautlat (init scop succesori)
        (caut (list init) scop succesori #'la_urma))


;Exemple de apel
(cautadanc 1 #'estescop #'succf)
(cautlat 1 #'estescop #'succf)
```

In cazul in care spatiul de cautare este un graf (deci pot exista cicluri), vom mentine, pe langa starile intalnite dar carora nu li s-au generat toti succesorii, si o lista ce contine starile pentru care toti succesorii au fost generati: TERITORIU. Astfel, se poate determina daca o stare nou generata a mai fost vizitata, caz in care aceasta este neglijata.

Operatia este efectuata de functia expand. Vom folosi functia speciala remove-if, cu sintaxa:

(remove-if <functie-test> <lista>)

care produce o noua lista rezultata din lista initiala, dupa ce au fost eliminate toate elementele pentru care <functie-test> intoarce valoarea adevarat. <lista> nu este distrusa in urma operatiei.

Functia cautgraf este implementata pornind de la functia caut (definita anterior) la care se mai adauga parametrii functiile scop si succesori care testeaza daca o stare este stare finala si, respectiv, genereaza lista de succesori ai unei stari date.

```
; cautare genereala, cand spatiul starilor este un graf
; strategia de cautare, modul de determinare a succesorilor si testul starii
; finale sunt parametri

(defparameter *FINAL* 18)
(defparameter *STFINALE* '(18 10 14))
(defparameter *LIM* 10)
(defparameter *SOLUTII* nil)

(defun estescop (x) (equal x *FINAL*))

(defun succf(x) (if (< x *LIM*) (list (* 2 x) (+ 1 (* 2 x)))))

(defun cautgraf (FRONTIERA scop succesori strategie TERITORIU)
  (format t "~& CAUT: ~a" FRONTIERA)
  (cond ((null FRONTIERA) nil)
     ((funcall scop (first FRONTIERA)) (first FRONTIERA))
     (t (cautgraf(funcall strategie (expand FRONTIERA succesori TERITORIU)
          (rest FRONTIERA)) scop succesori strategie
          (adaug (first FRONTIERA) TERITORIU)))))

(defun expand (FRONTIERA succesori TERITORIU)
   (remove-if
     #'(lambda (stare)
         (or (member stare FRONTIERA)
            (member stare TERITORIU)))
     (funcall succesori (first FRONTIERA))))

(defun adaug (stare TERITORIU)
   (if (member stare TERITORIU)
     TERITORIU
     (cons stare TERITORIU)))
```

```
(defun cautgraf-lat (init scop succesori)
  (cautgraf (list init) scop succesori #'la_urma nil))

(defun cautgraf-adanc (init scop succesori)
  (cautgraf (list init) scop succesori #'append nil))

;Exemple de apel
(cautgraf-lat 1 #'estescop #'succf)
(cautgraf-adanc 1 #'estescop #'succf)
```

Spre deosebire de implementarea cautarii backtracking, implementarea cautarilor in adancime si pe nivel nu gaseste si drumul pana la solutie.

```
;cautare parametrizata cu gasirea caii la solutie;
;spatiul de cautare este un graf

(defparameter *FINAL* 18)
(defparameter *STFINALE* '(18 10 14))
(defparameter *LIM* 10)
(defparameter *SOLUTII* nil)

(defun estescop (x) (equal x *FINAL*))

(defun succf(x) (if (< x *LIM*) (list (* 2 x) (+ 1 (* 2 x)))))
```

;Pentru a retine drumul trebuie ca o stare sa pastreze un indicator spre starea ;precedenta. Astfel, un nod al grafului va fir eprezentat ca o structura, numita cale, ;ce contine campurile stare si predecesor (un indicator catre starea ;anterioara).

```
(defstruct cale stare (predecesor nil))
```

;Functia expand-cale este o varianta a lui expand si creaza o lista de structuri cale ce ;contin starile succesori ale starii curente, care nu au mai fost vizitate si indicatori ;catre starea curenta ce a generat succesorii. Parametrul suplimentar test-st este o ;functie ce testeaza daca o stare este continuta intr-una din structurile cale ale unei ;liste. Un exemplu de astfel de functie este estein.

```
(defun expand-cale (FRONTIERA succesori TERITORIU test-st)
(let ((curenta (first FRONTIERA)))
 (mapcar #'(lambda (noua)
            (make-cale :stare noua  :predecesor  curenta))
        (remove-if #'(lambda (st)
                     (or (funcall test-st st FRONTIERA)
```

```
                    (funcall test-st st TERITORIU)))
                (funcall succesori (cale-stare curenta))))))

(defun afis-cale (stare)
(do ((curenta stare (cale-predecesor curenta)))
   ((null (cale-predecesor curenta))
        (format t "~& STARE: ~a" (cale-stare curenta)))
   (format t "~& STARE: ~a" (cale-stare curenta))))

(defun scopcale (x)
(equal (cale-stare x) *FINAL*))

(defun adaug (stare TERITORIU test-st)
(if (funcall test-st stare TERITORIU)
  TERITORIU
  (cons stare TERITORIU)))

;Functia some, cu sintaxa:
;          (some <predicat> <lista<)
;          are ca rezultatul adevarat daca functia <predicat> are rezultatul adevarat pentru
;cel putin unul dintre elementele listei asupra careia este aplicata. In caz contrar
;rezultatul este nil. Listele FRONTIERA si TERITORIU contin acum structuri cale si nu
;simple stari.

(defun estein (elem lista)
"Un exemplu de parametru actual care instantiaza parametrul formal test-st"
(some #'(lambda (x)
             (equal elem (cale-stare x)))
     lista))

(defun cautgraf (FRONTIERA scop succesori strategie TERITORIU test-st)
  (format t "~& CAUT: ~a" (first FRONTIERA))
  (cond
     ((null FRONTIERA) nil)
     ((funcall scop (first FRONTIERA)) (first FRONTIERA))
     (t (cautgraf
          (funcall strategie
                (expand-cale FRONTIERA succesori TERITORIU test-st)
                (rest FRONTIERA))
          scop succesori strategie
          (adaug (first FRONTIERA) TERITORIU test-st)
          test-st))))

(defun la-urma (succesori FRONTIERA)
(append FRONTIERA succesori))
```

```
(defun cautcale-lat (init scop succesori)
(cautgraf
        (list (make-cale :stare init))
        scop succesori #'la-urma nil #'estein))

(defun cautcale-adanc (init scop succesori)
(cautgraf
        (list (make-cale :stare init))
        scop succesori #'append nil #'estein))

;Exemple de apel
(afis-cale (cautcale-lat 1 #'scopcale #'succf))
(afis-cale (cautcale-adanc 1 #'scopcale #'succf))
```

STRATEGII DE CAUTARE EURISTICA

Folosind strategiile de cautare neinformata, in rezolvarea problemelor, numarul de stari investigate inainte de a gasi o solutie poate ajunge prohibitiv de mare, chiar si pentru probleme relativ simple. Spatiul de cautare explorat poate fi insa redus prin aplicarea informatiilor euristice despre problema. In cele din urma se va decide pe baza unei functii de evaluare euristica atasata starilor, care este starea ce urmeaza sa fie expandata. O astfel de comportare corespunde unei strategii de cautare informata, numita si cautare euristica.

Cautarea informata de tip "best-first"

Ideea este de a selecta spre expandare cel mai bun nod din spatiul de cautare generat pana la un moment dat, pe baza cunostintelor euristice, deci pe baza unei estimari. Aceasta strategie se foloseste daca ne intereseaza cautarea cat mai rapida a solutiei, deci ajungerea la starea finala in cel mai scurt timp. O astfel de strategie selecteaza urmatoarea stare de expandat ca fiind starea cu cea mai mica functie euristica asociata (w(n) asociata nodului n) dintre toate starile generate pana la un moment dat. Functia euristica trebuie astfel definita incat cu cat este mai mica valoarea acesteia cu atat starea corespunzatoare este mai promitatoare din punct de vedere al avansului spre solutie.

```
; cautare a solutiei si a caii pana la solutie folosind
; strategia "best-first". Spatiul de cautare este un graf. Cele doua functii
; folosite (w si ww) simuleaza cautare pe nivel si respectiv in adancime

(defparameter *STARE-FINALA* 18)
(defparameter *LIM* 10)
(defparameter *SOLUTII* nil)

(defun succf(x) (if (< x *LIM*) (list (* 2 x) (+ 1 (* 2 x)))))
```

```
(defstruct cale stare w (predec nil))

(defun insert (x lista-ord)
  (cond ((null lista-ord) (list x))
        ((<= (cale-w x) (cale-w (first lista-ord)))
           (cons x lista-ord))
      (t (cons (first lista-ord)
   (insert x (rest lista-ord))))))

(defun test-st* (stare FRONTIERA)
  (labels ((test (x lst1 lst2)
            (cond
              ((null lst2)
               (values nil lst1 nil lst2))
              ((equal x (cale-stare (first lst2)))
               (values t lst1 (first lst2) (rest lst2)))
              (t (let (flag l1 e l2)
                   (multiple-value-setq (flag l1 e l2)
                      (test x (append lst1 (list (first lst2)))
                         (rest lst2)))
                   (values flag l1 e l2))))))
    (test (cale-stare stare) nil FRONTIERA)))

(defun succf* (x fct-w)
  (let ((ss (cale-stare x)) (ww (cale-w x)))
    (mapcar #'(lambda (y)
                (make-cale :stare y
                  :w (funcall fct-w x) :predec x))
            (succf ss))))

(defun estescop* (x)
  (equal (cale-stare x) *STARE-FINALA*))

(defun cautcale (FRONTIERA scop succesori TERITORIU test-st fct-w)
  (format t"~& CAUT: ~a" (first FRONTIERA))
  (cond
    ((null FRONTIERA) nil)
    ((funcall scop (first FRONTIERA)) (first FRONTIERA))
    (t
     (let ((succlist
          (funcall succesori (first FRONTIERA) fct-w)))
       (setf TERITORIU
          (insert (first FRONTIERA) TERITORIU))
       (setf FRONTIERA (rest FRONTIERA))
       (do ((crt succlist (rest crt)) )
          ((null crt) nil)
```

```
        (let (flag l1 elem l2)
          (multiple-value-setq (flag l1 elem l2)
             (funcall test-st (first crt) FRONTIERA))
          (if flag
            (if (< (cale-w (first crt)) (cale-w elem))
              (setf FRONTIERA (insert (first crt) (append l1 l2))))
              (progn
               (multiple-value-setq (flag l1 elem l2)
                (funcall test-st (first crt) TERITORIU))
                (if flag
                  (if (< (cale-w (first crt))
                         (cale-w elem))
                    (progn
                      (setf FRONTIERA
                        (insert (first crt) FRONTIERA))
                      (setf TERITORIU (append l1 l2))))
                  (setf FRONTIERA
                    (insert (first crt) FRONTIERA)))))))
      (cautcale FRONTIERA scop succesori TERITORIU test-st fct-w)))))

(defun w (x) (1+ (cale-w x)))

(defun ww (x) (1- (cale-w x)))

;(defun www (x)
;   (+ (cale-w x)
;      (cost-operator
;         (cale-stare (cale-predec x)) (cale-stare x))))
;unde cost-operator nu este definita aici.

;Exemple de apel
(cautcale (list (make-cale :stare 1 :w 0)) #'estescop* #'succf*
  nil #'test-st* #'w)
(cautcale (list (make-cale :stare 1 :w 0)) #'estescop* #'succf*
  nil #'test-st* #'ww)
```

Cautarea solutiei optime: algoritmul A*

Se urmareste determinarea caii de cost minim intre nodul asociat starii initiale si cel asociat starii finale, in conditiile in care problema are costuri asociate tranzitiilor intre stari. Se gaseste astfel si calea spre solutie care contine un numar minim de arce, adica care are un numar minim de operatori intre starea initiala si starea finala. Acest lucru se obtine considerand costul unei tranzitii implicit egal cu 1 si se poate aplica si in cazul problemelor care nu au costuri atasate tranzitiilor de stari. In acelasi timp algoritmul A* este o strategie informata de tip "best-first" care urmareste gasireasolutiei optime cu un efort de cautare cat mai mic. De aceea algoritmul trebuie sa includa in evaluarea starilor

atat o componenta care sa surprinda distanta intre starea initiala si starea curenta (pentrua ghida cautarea pe drumul de cost minim) cat si o componenta care sa estimeze meritul starii curente din punct de vedere al avansului spre solutie.

Functia de evaluare euristica, f(S), este formata din doua componente: g(S) care estimeaza costul real, g(S), al caii dintre starea initiala si S si h(S) care estimeaza costul real, h(S), al caii dintre S si starea finala.
F(S)=g(S)+h(S)

```
; gasirea solutiei si a caii pana la solutie, folosind
; algoritmul A*. Spatiul de cautare este un graf
(defparameter *FINAL* 18)
(defparameter *LIM* 10)
(defparameter *SOLUTII* nil)
(defparameter *INITIAL* 1)

(defun succf(x) (if (< x *LIM*) (list (* 2 x) (+ 1 (* 2 x)))))

(defstruct cale stare g h f (predec nil))

(defun insert (x lista-ord)
  (setf (cale-f x) (+ (cale-g x) (cale-h x)))
  (cond ((null lista-ord) (list x))
        ((<= (cale-f x) (cale-f (first lista-ord)))
           (cons x lista-ord))
      (t (cons (first lista-ord) (insert x (rest lista-ord))))))

(defun succf* (x fct-h fct-g)
  (let ((xx (cale-stare x)))
      (mapcar #'(lambda (y) (make-cale :stare y :g (funcall fct-g y x)
                                       :h (funcall fct-h y) :predec x))
          (succf xx))))

(defun estescop* (x) (equal (cale-stare x) *FINAL*))

(defun test-st* (stare FRONTIERA)
  (labels ((test (x lst1 lst2)
             (cond ((null lst2) (values nil lst1 nil lst2))
                   ((equal x (cale-stare (first lst2)))
                      (values t lst1 (first lst2) (rest lst2)))
                  (t (let (flag l1 e l2)
                      (multiple-value-setq (flag l1 e l2)
                            (test x (append lst1 (list (first lst2)))
                                  (rest lst2)))
                      (values flag l1 e l2))))))
    (test (cale-stare stare) nil FRONTIERA)))
```

```
(defun cautcale (FRONTIERA scop succesori TERITORIU test-st fct-h fct-g)
(format t"~& CAUT: ~a" (first FRONTIERA))
(cond
 ((null FRONTIERA) nil)
 ((funcall scop (first FRONTIERA)) (print (first FRONTIERA)))
 (t
  (let ((succlist (funcall succesori (first FRONTIERA) fct-h fct-g)))
   ;;(print (cale-g (first FRONTIERA)))
   (setf TERITORIU (insert (first FRONTIERA) TERITORIU))
   (setf FRONTIERA (rest FRONTIERA))
   (do ((crt succlist (rest crt)) )
       ((null crt) nil)
     (let (flag l1 elem l2)
       (multiple-value-setq (flag l1 elem l2)
                    (funcall test-st (first crt) FRONTIERA))
       (if flag
         (if (< (cale-g (first crt)) (cale-g elem))
           (setf FRONTIERA (insert (first crt)
                        (append l1 l2))))
         (progn
          (multiple-value-setq (flag l1 elem l2)
                        (funcall test-st (first crt) TERITORIU))
          (if flag
             (if (< (cale-g (first crt)) (cale-g elem))
               (progn
                 (setf FRONTIERA (insert (first crt) FRONTIERA))
                 (setf TERITORIU (append l1 l2))))
             (setf FRONTIERA (insert (first crt) FRONTIERA)))))))
    (cautcale FRONTIERA scop succesori TERITORIU test-st fct-h fct-g)))))

(defun g (stare-noua cale-veche)
   (+ 1 (cale-g cale-veche)))
;Pentru exemplul considerat, nu are sens utilizarea unei euristici.
 (defun h (stare) 0)
;Exemplu de apel
(cautcale (list (make-cale :stare *INITIAL* :g 0 :h 0 :f 0))
  #'estescop* #'succf* '() #'test-st* #'h #'g)
```

Strategii de cautare (partea a II-a)

function GENERAL-SEARCH(*problem,* QUEUING-FN) **returns** a solution, or failure

nodes – MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[*problem*]))
loop do
if *nodes* is empty **then return** failure
node – REMOVE-FRONT(*nodes*)
if GOAL-TEST[*problem*] applied to STATE(*node*) succeeds **then return** *node*
nodes – QUEUING-FN(*nodes,* EXPAND(*node,* OPERATORS[*problem*]))
end
(*Russel, J., Norvik, P. - Artificial Intelligence A modern Approach*)

MAKE-QUEUE(*Elements*)	creaza o coada cu niste elemente specificate
EMPTY?(*Queue*)	intoarce true doar daca nu mai sunt elemente in coada
REMOVE-FRONT(*Queue*)	scoate elementul din varful cozii si il intoarce
QUEUING-FN(*Elements, Queue*)	introduce un set de elemente in coada
EXPAND	este responsabila pentru calcularea fiecaruia dintre componentele nodurile generate

datatype node
components: STATE, PARENT-NODE, OPERATOR, DEPTH, PATH-COST

STATE	starea in spatiul de stari in care nodul se afla
PARENT-NODE	nodul din arborele de cautare care genereaza acest nod
OPERATOR	operatorul care a fost aplicat pentru generarea acestui nod
DEPTH	numarul de noduri din drumul de la radacina pana la acest nod
PATH-COST	costul drumului din starea initiala pana la acest nod

Breadth-first search (cautarea in nivel) BFS

function BREADTH-FIRST-SEARCH (*problem*) **returns** a solution or failure
return GENERAL-SEARCH(*problem,* ENQUEUE-AT-END)
(*Russel, J., Norvik, P. - Artificial Intelligence A modern Approach*)

Depth-First search (cautarea in adancime) DFS

function DEPTH-FIRST-SEARCH (*problem*) **returns** a solution or failure
return GENERAL-SEARCH(*problem,* ENQUEUE-AT-FRONT)
(*Russel, J., Norvik, P. - Artificial Intelligence A modern Approach*)

Uniform cost search UCS

Cautarea de cost uniform mofica strategia cautarii pe nivel prin expandarea de fiecare data a nodului de cost cel mai scazut. Cand anumite conditii sunt indeplinite, prima solutie gasita este garantata ca fiind cea mai ieftina solutie, deoarece daca a fost un drum cu cel mai scazut cost care a dus la solutie, atunci a fost expandat anterior.

Depth-limited search DLS

Asemanator cu DFS-ul doar ca stabileste o limita cat de adanc poate un DFS sa mearga.

Iterative deepening search IDS

Apeleaza DLS cu incrementarea limitei pana cand scopul este gasit

function ITERATIVE-DEEPENING-SEARCH (*problem*) **returns** a solution sequence
    inputs: *problem*, a problem
    for *depth* – 0 **to** *x* **do**
        if DEPTH-LIMITED-SEARCH(*problem*, *depth*) succeeds
            then return its result
    end
    return failure
(*Russel, J., Norvik, P. - Artificial Intelligence A modern Approach*)

Bidirectional search BDS

Cautarea se incepe in paralel din starea initiala si din starea scop. Cand cele doua cautari

se intalnesc la mijloc atunci algoritmul se termina.