O problemă de inteligenţă artificială — planificarea

Raluca Budiu (ralucav+@cs.cmu.edu)

aprilie 1996 ?

Subiect: enunţarea problemei de planificare; un exemplu din lumea blocurilor; un planificator în PROLOG

Cuvinte cheie: planificare, operator, căutare.

Cunoştinţe necesare: cunoştinţe minime legate de tipuri de date şi de căutare; limbajul Prolog pentru program

Cuprins

1 Introducere

În acest articol ne propunem să prezentăm una din problemele centrale ale inteligenţei artificiale — planificarea, împreună cu o soluţie simplă (şi parţială) a ei, implementată în limbajul PROLOG. Pentru înţelegerea articolului nu e necesară cunoaşterea PROLOG-ului. Doar studiul codului din anexă necesită o minimă familiarizare cu acest limbaj.

2 Ce înţelegem prin "planificare"

Înainte de a da o definiţie planificării, aşa cum este ea înţeleasă de inteligenţa artificială, să considerăm următorul exemplu. Să ne imaginăm ca o persoană X, aflată în Bucureşti, vrea să facă o călătorie la Londra. Ca să îşi poată atinge acest ţel, X trebuie să ducă la îndeplinire cîteva acţiuni prealabile: să obţină o viză de la Ambasada Britanică, să-şi cumpere bilet de avion, să-şi facă bagajul, să se ducă la aeroport la data şi ora precizată pe bilet, să ia avionul şi, finalmente, să coboare la Londra. Aceste etape preliminare care trebuie satisfăcute pentru atingerea scopului final — călătoria de la Bucureşti la Londra — se numesc acţiuni, şi ele constituie un plan. Construirea explicită a unei succesiuni de astfel de acţiuni în vederea unui ţel final oarecare este o planificare.

Pînă acum nimic nu pare să se lege de inteligenţa artificială. Să presupunem însă că X este un robot bucureştean, şi că ne-ar place ca acest robot, în momentul în care cineva îi comandă o călătorie la Londra (sau orice altceva), să fie capabil să-şi planifice singur (în sensul de mai sus) această călătorie, fără ca omul să trebuiască să-i precizeze explicit că are nevoie de o viză, de bilet, ş.a.m.d.

Planificarea în sensul inteligenţei artificiale (de altfel singurul sens în care vom folosi acest cuvînt de-acum încolo) înseamnă generarea unei secvenţe de acţiuni pentru un agent (de tipul unui robot) care poate schimba mediul (lumea) în care evoluează. Scopul planificării este atingerea unuia sau mai multor ţeluri exprimate explicit.

O remarcă despre terminologie — termenii "lume" şi mediu se referă în acest articol la universul problemei de rezolvat.

În definiţia de mai sus, am folosit cuvintele "agent care poate schimba mediul în care evoluează". Într-adevăr, dacă robotul X din exemplul dat nu e în stare să interacţioneze cu lumea înconjurătoare altfel decît prin comenzile primite (nu poate să meargă sau să emită la rîndul lui cereri), atunci nici o speranţă să obţină viza sau să cumpere biletul de avion, şi cu atît mai puţin să se deplaseze la aeroport. Pe de altă parte, dacă robotul cu pricina ştie să efectueze doar două acţiuni: să coase şi să taie, oricît de "inteligent" ar fi el, e foarte puţin probabil să găsească o combinaţie de tăieturi şi cusături (adică un plan bazat pe aceste două acţiuni) care să-l poarte la Londra.

Ca să recapitulăm, informaţia de intrare într-o problemă de planificare conţine:

- o stare iniţială a lumii (mediului). In exemplul nostru, starea iniţială este: X se află în Bucureşti, fără viză britanică, fără bilet spre Londra şi fără vreun bagaj făcut.
- o mulţime de acţiuni (operatori) care pot fi folosite în schimbarea lumii. Despre aceste acţiuni n-am spus nimic în exemplul de mai sus, decît că ar fi bine să fie altele decît "a coase" şi "a tăia"; ele se referă la activităţile elementare pe care robotul X ar trebui să le poată îndeplini. Putem presupune, de exemplu, că X ştie să obţină viză, să cînte la pian, să cumpere bilet de avion, să facă bagaje, să meargă dintr-un loc în altul, să ia avionul, să gătească, să coboare din avion (de obicei, însă, operatorii sunt mult mai simpli, după cum veţi vedea).
- o mulţime de scopuri. În exemplul dat, scopul este unul singur — călătoria la Londra.

Înainte de a da un exemplu complet de problemă de planificare, trebuie să spunem că, pentru ca să se poată vorbi de o soluţie la o astfel de problemă, lumea în care planul urmează a fi executat trebuie să fie cel puţin predictibilă, dacă nu chiar deterministă. Aceasta înseamnă, revenind la exemplul nostru, că nu luăm în considerare evenimente neaşteptate de tipul aeroportul e închis, sau avionul se prăbuşeşte la cinci minute după decolare sau avionul e deturnat de pe ruta spre Londra. Este evident că această ipoteză a unei lumi lipsite de neprevăzut este destul de departe de realitate.

3 O problemă tipică

În inteligenţa artificială (şi nu numai), se obişnuieşte ca, atunci cînd o problemă destul de complicată se doreşte a fi rezolvată, să se caute mai întîi soluţii ale ei pentru un caz particular mai simplu. Aşa şi cu planificarea. Ea a fost intensiv studiată pe un exemplu de-acum clasic, şi anume în aşa-numita lume a blocurilor. Toate obiectele în acest univers sunt cuburi de aceeaşi mărime. Ele pot fi aşezate pe o masă (una singură în universul problemei, considerată infinită) sau unul peste celălalt. Agentul care poate modifica această lume este un robot cu un braţ, cu care poate să apuce un singur bloc la un moment dat.

Iată un exemplu de stare iniţială şi de stare finală (de atins) în această lume (Figura 1):

```
                         |                                |
                        _|_                              _|_
    |----|    |----|   /   \          |----|    |----|   /   \
    | b  |    | d  |  /     \         | c  |    | d  |  /     \
    |----|    |----|                  |----|    |----|
    | a  |    | c  |                  | a  |    | b  |
----|----|----|----|-----         -----|----|------|----|------
    Stare initiala                     Scop (Stare finala)


                      Figura 1.
```

Problema este cum putem reprezenta aceste configuraţii cu un limbaj mai apropiat de cel pe care l-ar putea înţelege un calculator. Dacă în locul robotului, cel căruia ar trebui să-i descriem aceste stări ar fi un om, şi dacă am comunica cu acel om la telefon, de pildă, atunci probabil că i-am spune ceva de genul: "starea iniţială e următoarea: blocurile a şi c sunt pe masă, blocul b este deasupra lui a şi blocul d este deasupra lui c. Pe blocurile b şi d nu mai este nici un alt bloc. Robotul nu ţine nimic în mînă. Starea finală este etc.". Ca atare, nu e o idee rea să descriem o stare în lumea blocurilor ca pe o colecţie de "fapte" (adică afirmaţii adevărate în configuraţia respectivă).

4 Reprezentarea stărilor în lumea blocurilor

Prin urmare, pentru a reprezenta o stare vom folosi următoarele literale:

LIBER (x): însemnînd că deasupra blocului x nu se găseşte nici un alt bloc

IN-MINA (x): însemnînd ca braţul robotului ţine blocul x

PE-MASA (x): cu semnificaţia că blocul x se găseşte pe masă

MINA-LIBERA: însemnînd că robotul nu are nimic în mînă

PE (x, y): însemnînd că blocul x este aşezat pe blocul y.

Orice configuraţie din lumea blocurilor poate fi descrisă ca o mulţime sau o conjuncţie (*şi* logic) de astfel de literale. De exemplu, să vedem cum se reprezintă cele două stări din figura 1:

Starea iniţială:

```
PE-MASA (a).
PE (b, a).
LIBER (b).
PE-MASA (c).
PE (d, c).
LIBER (d).
MINA-LIBERA.
```

Starea finală:

```
PE-MASA (a).
PE (c, a).
LIBER (c).
PE-MASA (b).
PE (d, b).
LIBER (d).
MINA-LIBERA.
```

Cu alte cuvinte, o stare se poate exprima printr-o colecţie de fapte care sunt toate adevărate în configuraţia corespunzătoare ei.

5 Reprezentarea operatorilor în lumea blocurilor

Să vedem acum care sunt acţiunile elementare pe care robotul le poate duce la îndeplinire:

RIDICA (x): însemnînd că braţul robotului apucă blocul x de pe masă;

ASEAZA (x): însemnînd că robotul pune blocul x pe masă;

PUNE-PE (x, y): însemnînd că robotul pune blocul x peste blocul y;

IA-DE-PE (x, y): însemnînd că robotul ridică blocul x de pe blocul y.

Fiind dată o stare iniţială, robotul poate să o modifice folosind una dintre acţiunile (operatorii) de mai sus. În felul acesta, se ajunge într-o nouă configuraţie de blocuri care poate sau nu să fie identică cu cea finală. Dacă nu este, putem încerca să aplicăm din nou un operator, de data aceasta din noua stare în care tocmai am ajuns, şi aşa mai departe pînă cînd se atinge ţelul urmărit. În plus, dacă ajungem într-o situaţie în care scopul devine de neatins (cu alte cuvinte, am ales o secvenţa proastă de operatori), putem oricînd să ne întoarcem la starea precedentă şi să încercăm un alt operator. (Unii dintre dumneavoastră poate au identificat deja în clipa aceasta planificarea cu o căutare). Deci acţiunile oferă un mod de schimbare a lumii în care robotul evoluează.

Problema care se pune acum este să găsim un mod de a exprima tocmai schimbările pe care fiecare dintre aceste acţiuni le poate induce asupra asupra configuraţiei curente de blocuri. Vom încerca să rezolvăm această problemă bazîndu-ne pe următoarele observaţii:

1. o acţiune nu poate fi satisfăcută oricînd.

   Într-adevăr, nu putem folosi operatorul (acţiunea) IA-DE-PE (c, a) în starea iniţială din Figura 1, căci blocul c nu se află pe blocul a. Deci, pentru ca un operator să poată fi aplicat trebuie ca starea curentă să satisfacă anumite condiţii, pe care le vom numi *precondiţii* ale operatorului cu pricina.

2. în general, două acţiuni modifică starea curentă în feluri diferite.

   Dacă în starea iniţială din Figura 1 aplicăm de pildă acţiunea IA-DE-PE (b, a) , blocul a devine liber ; dacă însă folosim operatorul IA-DE-PE (d, c), blocul c este liber şi blocul a este acoperit în continuare de b. Deci, un operator poate fi descris prin efectele (*post-condiţiile*) sale, adică prin starea lumii după ce el a fost aplicat.

Acestea fiind spuse, un operator este unic reprezentat prin două mulţimi — mulţimea precondiţiilor sale şi mulţimea post-condiţiilor sale. Am spus deunăzi că post-condiţiile unui operator sunt date de starea lumii după ce el a fost aplicat. Dar, în principiu, această stare depinde de starea anterioară aplicării operatorului respectiv. Într-adevăr, dacă în starea iniţială din Figura 1 folosim operatorul IA-DE-PE (b, a), atunci starea în care ajungem devine cea din Figura 2:

```
                                              PE-MASA (a).
                  |        |----|             PE-MASA (c).
                /-|-\      | d  |             PE (d, c).
       |----|   /| b |\    |----|             LIBER (d).
       | a  |    |---|     | c  |             IN-MINA (b).
-------|----|--------------|----|------       LIBER (a).

                Figura 2.
```

În această nouă stare, practic nu s-a schimbat faţă de configuraţia iniţială din Figura 1 decît ceea ce era legat de blocurile a şi b, precum şi de braţul robotului. În rest, blocurile c şi d şi-au păstrat configuraţia din starea anterioară. Deci post-condiţiile unui operator cuprind, pe lîngă o descriere a felului în care operatorul respectiv modifică lumea, şi faptele care au rămas neschimbate în urma aplicării operatorului respectiv. Dar acestea din urmă pot fi foarte multe (imaginaţi-vă că în loc de patru blocuri, am folosi cincizeci sau o mie). Prin urmare nu este convenabil să folosim în reprezentarea unui operator întreaga listă a post-condiţiilor sale. Ce este de făcut?

Ne vom folosi de un mic truc. Am văzut că de fapt ceea ce este cu adevărat relevant pentru un operator este felul în care s-a schimbat starea posterioară aplicării operatorului faţă de starea anterioară aceluiaşi moment. Această schimbare este descrisă de două mulţimi: cea a condiţiilor (faptelor) care devin adevărate în noua stare şi cea a condiţiilor care devin false după aplicarea operatorului. Putem scrie:

```
STARE-POST = STARE-ANTE + COND-ADAUGATE - COND-STERSE,
```

unde:

STARE-POST este starea posterioară aplicării unui operator,

STARE-ANTE este starea anterioară aplicării aceluiaşi operator,

COND-ADAUGATE este mulţimea faptelor care devin adevărate în urma aplicării operatorului respectiv,

COND-STERSE este mulţimea faptelor care devin false în urma aplicării operatorului în cauză.

Ca să conchidem, un operator poate fi reprezentat prin următorul triplet:

- mulţimea precondiţiilor care fac posibilă aplicarea lui;
- mulţimea condiţiilor care trebuie adăugate la starea curentă după ce operatorul a fost aplicat;
- mulţimea condiţiilor care trebuie şterse din starea curentă pentru a obţine starea următoare.

De exemplu, operatorul IA-DE-PE (x, y) poate fi descris astfel:

```
IA-DE-PE (y, x)
preconditii:
        LIBER (y).
        MINA-LIBERA.
        PE (y, x).
postconditii de adaugat:
        IN-MINA (y).
        LIBER (x).
postconditii de sters:
        LIBER (y).
        MINA-LIBERA.
        PE (y, x).
```

6 Planificarea – o problemă de căutare

Scopul nostru este să construim un planificator, adică un program capabil să construiască un plan pentru o scop şi o stare iniţială date, folosind nişte operatori dependenţi de problemă. Cum putem face acest lucru? Folosindu-ne de nişte algoritmi informaţi de căutare. Vom încerca să explicăm ce sunt aceştia şi în ce fel îi putem folosi în lunga expunere care urmează.

Planificarea este o problemă de căutare într-un spaţiu al stărilor. Să vedem ce înţelegem prin această afirmaţie. Am văzut că în orice problemă de planificare există o stare de pornire (stare iniţială) şi un scop pe care dorim să-l atingem. Pentru ajunge în starea finală care satisface scopul de atins, putem folosi o mulţime de operatori. Aplicarea unuia sau altuia dintre aceşti operatori atrage după sine o modificare de stare. Toate stările noi care pot fi generate din starea iniţială aplicînd oricare din operatorii ale căror precondiţii sunt îndeplinite formează mulţimea succesorilor (stărilor succesoare) stării iniţiale. În acelaşi mod putem defini mulţimea succesorilor unei stări alta decît cea iniţială. Starea iniţială, împreună cu toate stările care sunt succesoare unei alte stări formează un *spaţiu de stări*. Cu alte cuvinte, un spaţiu de stări este mulţimea tuturor stărilor care pot fi obţinute din starea iniţială aplicînd orice secvenţa posibilă de operatori. (Reamintim că un operator poate fi aplicat numai dacă precondiţiile sale sunt îndeplinite). Starea finală este şi ea o stare în acest spaţiu (dacă problema are soluţie). Soluţia problemei de planificare este o succesiune de stări care porneşte din starea iniţială şi ajunge în starea finală, iar secvenţa de operatori folosiţi pentru a trece dintr-o stare în alta (pe drumul între starea iniţială şi cea finală) formează planul căutat. Un spaţiu de stări poate fi reprezentat ca un graf în care nodurile sunt stări, iar muchiile operatori; o soluţie ca o cărare în acest graf (vezi Figura 3).

```
            stare initiala
                o
              / | \
            /   |   \
          o     o     o    --> succesorii starii initiale
        / |   / | \     \
      /   | /   |   \     \
     o    o     o    o     o  ---> succesorii succesorilor starii
   /            |         / \             initiale
 /              |       /     \
o               o      o       o
                     stare
                    finala

            Figura 3.
```

Cum se găseşte soluţia? Există două variante — sau se alege o cărare anume după un criteriu oarecare şi dacă la sfîrşitul cărării nu s-a ajuns în starea finală, atunci se abandonează căutarea (căutare ireversibilă); sau se explorează toate cărările pînă cînd se ajunge la o soluţie (căutare exhaustivă). Ordinea de explorare este specifică fiecărui algoritm; astfel căutarea în adîncime explorează întotdeauna succesorii unei stări înainte de a-i explora "fraţii" (nodurile cu acelaşi părinte); căutarea în lăţime face exact invers — explorează mai întîi fraţii unui nod şi apoi succesorii lui. Backtracking-ul este un caz special de căutare în adîncime.

Pentru că de cele mai multe ori spaţiul stărilor este uriaş, se folosesc anumiţi algoritmi de căutare informaţi. Aceşti algoritmi fac alegerea stării următoare celei curente după un criteriu legat de structura internă a stărilor candidate, care estimează apropierea acestora de soluţie. În felul acesta, e posibil să se economisească timp şi spaţiu; însă, pe de altă, există riscul să nu se găsească o soluţie pentru problema dată.

Să presupunem că am folosi pentru problema noastră o tehnică de căutare ireversibilă şi informată de tipul hill-climbing ("urcuş de deal" — pe româneşte). Aceasta presupune utilizarea unei funcţii euristice (de merit), care reprezintă tocmai criteriul despre care vorbeam în paragraful precedent. Iată cum arată algoritmul de hill-climbing:

1. fă starea iniţială stare curentă.

2. generează toţi succesorii stării curente (folosind operatorii ale căror precondiţii sunt îndeplinite); dacă vreunul este o stare finală, atunci stop

3. alege ca nouă stare curentă succesorul care are funcţia de merit cea mai "bună" dintre toţi succesorii care nu au fost pînă acum stări curente ; du-te la pasul 1.

Rămîne să vedem cum am putea alege o funcţie de merit (euristică). Am spus că o astfel de funcţie estimează apropierea unei stări de soluţie; am putea deci să folosim ca euristică pentru o stare s funcţia diferenţă dată de numărul de condiţii din starea finală care nu sunt încă satisfăcute în starea s. O stare este cu atît mai bună cu cît funcţia sa de merit este mai mică.

Vom vedea că această euristică este acceptabilă numai cînd condiţiile ce trebuie satifăcute în starea finală sunt independente. Să considerăm un exemplu în care funcţia diferenţă, împreună cu un algoritm hill-climbing, ratează soluţia.

Fie din nou problema expusă în Figura 1. Din starea iniţială putem ajunge în două stări succesoare distincte, corespunzînd aplicării operatorilor IA-DE-PE (b, a) sau IA-DE-PE (d, c). Cei doi succesori sunt descrişi în Figurile 2 şi 4.

```
          |----|     |
          | b  |   /-|-\
          |----|  /| d |\    |----|
          | a  |   |---|     | c  |
   -------|----|-------------|----|------
```

Figura 4.

```
PE-MASA (a).
PE (b, a).
LIBER (b).
PE-MASA (c).
LIBER (c).
IN-MINA (d).
```

Condiţiile satisfăcute în starea scop din Figura 1 şi nesatisfăcute în Figura 2 sunt { `LIBER (c)`, `PE (c, a)`, `PE (d, b)`, `PE-MASA (b)`, `MINA-LIBERA` }. Funcţia diferenţa corespunzătoare este deci 5. În ceea ce priveşte starea din Figura 4, condiţiile încă nesatisfăcute sunt { `PE (c, a)`, `PE (d, b)`, `PE-MASA (b)`, `MINA-LIBERA` }, deci funcţia diferenţa este 4. Algoritmul va alege succesorul cu cea mai mică funcţie de merit, adică starea din Figura 4. Aceasta are o funcţie diferenţa mai bună pentru simplul fapt că lasă blocul c liber, fapt de dorit în starea finală. În această nouă stare (cea din Figura 4), se pot folosi trei operatori: `ASEAZA (d)` sau `PUNE (d, b)` sau `PUNE (d, c)`. Ultimul este exclus din start pentru că ne întoarce înapoi în starea iniţială din Figura 1. Să vedem care sunt succesorii ce se pot obţine prin aplicarea celorlalţi doi operatori

(Figura 5 şi Figura 6).

```
                                 |
       |----|                /-|-\                  PE-MASA (a).
       | d  |               /     \                 PE-MASA (c).
       |----|                                       PE (b, a).
       | b  |                                       PE (d, b).
       |----|     |----|                            LIBER (d).
       | a  |     | c  |                            MINA-LIBERA.
--------|----|-----|----|-----------
                          Figura 5.
```

```
                                 |                            PE-MASA (a).
                               /-|-\                          PE-MASA (d).
             |----|           /     \                         PE-MASA (c).
             | b  |                                           PE (b, a).
             |----|    |----|    |----|                       LIBER (b).
             | a  |    | c  |    | d  |                       LIBER (c).
----------|----|----|----|----|----|-----                     MINA-LIBERA.
                                                              LIBER (d).
                         Figura 6.
```

Deşi Figura 5 diferă prin mai puţine condiţii de starea finală din Figura 1 (funcţie de merit 2) decît Figura 6 (funcţie de merit 3), pentru un om este evident că Figura 6 este o alegere mai bună decît Figura 5 pentru atingerea stării finale din Figura 1. Algoritmul însă va prefera succesorul din Figura 5 pentru că el satisface în plus condiţia `PE (d, b)`, care apare în starea finală. El ignoră astfel faptul că această satisfacerea reală a acestei condiţii depinde de fapt de condiţia `PE-MASA (b)`. Cu alte cuvinte, condiţiile de îndeplinit pentru a ajunge în starea finală (numite şi sub-scopuri) nu sunt independente unele de altele. În continuare, ţinînd cont că algoritmul va evita ciclarea (adică trecerea printr-o stare în care a mai fost) şi că el nu se va întoarce niciodată înapoi, finalmente se va ajunge în starea în care blocurile a, b, d, c formează un turn şi programul se va opri fără să găsească soluţia problemei.

O problemă se numeşte de planificare *liniară* dacă sub-scopurile cu care ea are de a face sunt independente. În general, foarte puţine probleme intră în această categorie.

Am văzut că nu putem aplica hill-climbing-ul nici măcar pentru o problemă simplă din lumea blocurilor. Nu vom abandona însă funcţia de merit pe care am ales-o (căci ea este de multe ori utilă, îmbunătăţind considerabil timpul de căutare), ci vom face o căutare exhaustivă în ghidată de euristica diferenţă. În principal, aceasta diferă de o hill-climbing prin faptul că, dacă se ajunge la o stare blocantă (în care nu mai e nimic de făcut), algoritmul se întoarce înapoi în starea anterioară şi încearcă un alt operator, chiar dacă acesta are o funcţie de merit mai proastă decît cel folosit iniţial. De exemplu, într-o situaţie în care are de ales între stările din Figura 5 şi Figura 6, acest algoritm o va alege întîi pe cea din Figura 5, apoi se va întoarce înapoi şi va lua în considerare cealaltă alternativă.

În secţiunea care urmează vom prezenta un algoritm (nu singurul posibil) pentru un planificator. El va folosi o căutare în adîncime ghidată de funcţia de merit diferenţă.

7 Algoritm şi implementare

Ideea algoritmului este să aleagă cel mai bun posibil succesor al stării curente S şi să-l facă următoarea stare curentă Succ. Dacă Succ nu se găseşte pe o cărare spre starea finală, atunci programul se întoarce în starea S şi selectează al doilea cel mai bun succesor, şi aşa mai departe.

Algoritmul general este următorul:

```
function plan (StareInitiala, StareFinala)
Pasul 1. Creaza o Stiva goala cu operatorii aplicati pina acum:
              StivaOp = {}
         si o stiva a starilor de pe cararea in explorare:
              StivaStari = { StareInitiala }.
         Seteaza StareCurenta = StareInitiala.
Pasul 2. Daca StareCurenta = StareFinala, atunci
              afiseaza operatorii din StivaOp in ordine inversa
              (ei formeaza planul cautat);
              intoarce succes.
Pasul 3. Genereaza toti succesorii posibili pentru StareCurenta
         aplicind  toti acei operatori ale caror preconditii sunt
         indeplinite.  Pastreaza intr-o coada Succs[StareCurenta]
         toti acei succesori care nu apartin stivei StivaStari.
Pasul 4. Daca coada Succs[StareCurenta] nu este vida, sorteaz-o
         dupa valorile ascendente ale functiei diferenta a starilor
         din aceasta coada.
Pasul 5. Daca coada Succs[StareCurenta] este vida, atunci
              StareCurenta ia valoarea din virful stivei
                     StivaStari; aceasta valoare se sterge de pe
                     StivaStari;
              sterge virful stivei StivaOp;
              du-te la pasul 5.
Pasul 6. Alege prima stare din coada Succs[StareCurenta] si fa-o
         StareCurenta.  Pune StareCurenta pe StivaStari si operatorul care a
         generat-o in stiva StivaOp.   Du-te la pasul 2.
```

Am folosit notaţia `Succs[StareCurenta]` pentru a desemna faptul că fiecare stare are coada ei proprie de succesori.

Algoritmul de mai sus a fost implementat în limbajul PROLOG. Listing-ul programului poate fi găsit în anexa care urmează. Programul a fost testat pentru probleme din lumea blocurilor cu 3, 4, 5 blocuri şi pentru problema maimuţei şi a bananei, pe care o vom prezenta pe scurt. O maimuţa înfometată se află închisă într-o cameră. În cameră se găseşte un scaun lîngă o fereastră şi un ciorchine de banane atîrnă de centrul tavanului. Maimuţa este prea scundă ca să atingă bananele. Ea este capabilă de următoarele acţiuni: să se plimbe prin cameră, să împingă scaunul, să se urce pe scaun, să ia bananele (dacă ajunge la ele). Problema este să se elaboreze un plan pentru maimuţa ca să mănînce bananele.

8 ANEXĂ

```
/*
 * file plan.ari: planning algorithm
 */

% satisfy_conditions(Conditions, State):
%========================================
%       check whether the State has among its preconditions all the
%       Conditions. Both arguments should be bound.

satisfy_conditions(Conditions, State):-
        subset(Conditions, State).

% succ_state(State, Effects, NextState)
% =====================================
%       compute the new state NextState in which the last applied operator
%       brought us ; the NextState is the current State
%               - to which we have added the effects in the add-sublist of
%                       Effects
%               - and from which we have deleted the effects in the delete-
%                       sublist of Effects

succ_state(State, [], State).
succ_state(State, [add(Condition) | RestEffects], NextState) :-
        succ_state(State, RestEffects, Temp),
        insert_unique(Condition, Temp, NextState).
succ_state(State, [del(Condition) | RestEffects], NextState) :-
        succ_state(State, RestEffects, Temp),
        delete(Condition, Temp, NextState).

% plan(CurrentState, StateStack, OpStack):
%===========================================
%       print out which operators for a given problem universe should be
%       applied in order to reach the GoalState from a CurrentState;
%       the StateStack contains the sequence of states that were already
%       met; the OpStack is the sequence of operators that were applied
%       (in reverse order).

plan(State, StateStack, OpStack) :-
        goal(GoalState),
        equal(State, GoalState),                % check State = GoalState
        write('Operators that were applied:'), nl,
        display_stack(OpStack).                % print the sequence of
```

```
                                                % operators applied
plan(State, StateStack, OpStack) :-
        choose_best_next(State, NextState, StateStack, Operator),
        push(NextState, StateStack, NewStateStack),
        push(Operator, OpStack, NewOpStack),
        plan(NextState, NewStateStack, NewOpStack).

% choose_best_next(State, GoalState, NextState, StateStack, Operator):
%====================================================
% compute all the possible successors of the State, and chooses the
% NextState that is the closest to the GoalState; the operator applied
% to generate the best next state is returned in Operator.

choose_best_next(State, NextState, StateStack, Operator) :-
        bagof(NState / Op, succ(State, StateStack, NState, Op)
                                                         , Succs), !,
        quicksort(Succs, OrdSuccs),
        member(NextState/Operator, OrdSuccs).


% greater(X, Y):
% ==============
% some order relation between two successor states

greater(X / OpX, Y / OpY):-
        goal(Goal), !,
        distance(X, Goal, Dx),!,
        distance(Y, Goal, Dy),!,
        Dx > Dy, !.

% succ(State, StateStack, NState, Op):
% ====================================
% compute the successor of a State in NState; the operator applied in order
% to obtain NState is returned in Op; a check is made to avoid generating
% a state already in StateStack.

succ(State, StateStack, NState, Op) :-
        move(Op, Preconditions, Effects),
        satisfy_conditions(Preconditions, State),
        stack(X, X) \= Op,
        succ_state(State, Effects, NState),
        not(list_member_stack(NState, StateStack)).

% distance(State, Goal, d) :
```

```prolog
% ==========================
% computes the distance (heuristic fun) between the State and the Goal;
% returns it in d.

distance(State, Goal, D) :- dist(State, Goal, 0, D).

dist(State, [], Acc, Acc):- !.
dist(State, [Cond | RestCond], Acc, D) :-
        member(Cond, State), !,
        dist(State, RestCond, Acc, D).
dist(State, [Cond | RestCond], Acc, D) :-
        Acc1 is Acc + 1,
        dist(State, RestCond, Acc1, D).


% go(InitialState, GoalState):
% ============================
%       ask for getting from an InitialState to a GoalState.
%       Both arguments should be bound.

go(InitialState, GoalState):-
        asserta(goal(GoalState)),
        empty(StateStack),
        empty(OpStack),
        push(InitialState, StateStack, NewStateStack),
        plan(InitialState, NewStateStack, OpStack),
        retract(goal(X)).

testblock31:-
        go([handempty, ontable(b), ontable(c), on(a, b), clear(a), clear(c)],
           [handempty, ontable(a), ontable(b), on(c, b), clear(a), clear(c)]).

testblock32 :-
        go([handempty, ontable(b), ontable(c), on(a,b), clear(a), clear(c)],
           [handempty, ontable(a), on(c,a), on(b,c), clear(b)]).

testblock33 :- /* Sussman anomaly */
        go([handempty, on(c, a), ontable(b), ontable(a), clear(c), clear(b)],
           [handempty, ontable(c), on(b, c), on(a, b), clear(a)]).

testblock41 :-
        go([handempty, ontable(a), ontable(c), on(b, a), clear(b), on(d, c),
            clear(d)],
           [handempty, ontable(a), ontable(b), on(c, a), clear(c), on(d, b),
```

```
            clear(d)]).

testblock42 :-
        go([handempty, ontable(a), ontable(c), ontable(d), on(b, a),
      clear(b), clear(c), clear(d)],
           [handempty, on(c, a), ontable(a), on(b, d),
            ontable(d), clear(c), clear(b)]).
testblock43 :-
        go([handempty, ontable(a), ontable(c), on(b, a), on(d, b), clear(c),
            clear(d)],
           [handempty, ontable(b), on(d, b), on(a, d), on(c, a), clear(c)]).
testblock5:-
        go([handempty, ontable(a), ontable(c), ontable(d),
            on(b, a), clear(b), clear(c), on(e,d), clear(e)],
           [handempty, ontable(c), on(b, a), on(a,c), clear(b),
            ontable(e), on(d,e), clear(d)]).

testape :-
        go([ape(door), on_the_earth, box(window), has(nothing)],
           [ape(middle), on_the_box, box(middle), has(bannana)]).

testrich :-
        go([poor, unknown], [rich, famous]).


/*
 * file blocks.ari: operators for the planning problem in the blocks world
 */

move(
        putdown(X),
        [holding(X)],
        [del(holding(X)),
         add(ontable(X)), add(handempty)]
                ).

move(
        pickup(X),
        [handempty, clear(X), on(X, Y)],
        [del(handempty), del(on(X, Y)),
         add(clear(Y)), add(holding(X))]
                ).

move(
```

```
        pickup(X),
        [handempty, clear(X), ontable(X)],
        [del(handempty), del(ontable(X)),
         add(holding(X))]
                ).

move(
        stack(X, Y),
        [holding(X), clear(Y)],
        [del(holding(X)), del(clear(Y)),
         add(handempty), add(on(X,Y))]
     ).

/*
 * file ape.ari: the problem of the ape and the banana
 */

move(
        HOLDING,
        [ape(middle), on_the_box, box(middle), has(nothing)],
        [add(has(bannana)),del(has(nothing))]).

move(
        leaping(X),
        [ape(X), on_the_earth, box(X)],
        [add(on_the_box), del(on_the_earth)]).

% pushing
% ========
move(
        pushing(window, corner),
        [ape(window), on_the_earth, box(window)],
        [add(ape(corner)), add(box(corner)),
         del(ape(window)), del(box(window))]).

move(
        pushing(corner, middle),
        [ape(corner), on_the_earth, box(corner)],
        [add(ape(middle)), add(box(middle)),
         del(ape(corner)), del(box(corner))]).

move(
        pushing(window, door),
        [ape(window), on_the_earth, box(window)],
```

```
        [add(ape(door)), add(box(door)),
         del(ape(window)), del(box(window))]).

move(
        pushing(window, middle),
        [ape(window), on_the_earth, box(window)],
        [add(ape(middle)), add(box(middle)),
         del(ape(window)), del(box(window))]).


% walking
% =======
move(
        walking(door, window),
        [ape(door), on_the_earth],
        [add(ape(window)), del(ape(door))]).

move(
        walking(window, middle),
        [ape(window), on_the_earth],
        [add(ape(middle)), del(ape(window))]).

move(
        walking(corner, window),
        [ape(corner), on_the_earth],
        [add(ape(window)), del(ape(corner))]).

move(
        walking(door, middle),
        [ape(door), on_the_earth],
        [add(ape(middle)), del(ape(door))]).

move(
        walking(door, corner),
        [ape(door), on_the_earth],
        [add(ape(corner)), del(ape(door))]).
move(
        walking(middle, window),
        [ape(middle), on_the_earth],
        [add(ape(window)), del(ape(middle))]).
```