TEHNICI DE CĂUTARE

Căutarea este o traversare *sistematică* a unui *spaţiu de soluţii* posibile ale unei probleme.

Un *spaţiu de căutare* este de obicei un graf (sau, mai exact, un arbore) în care un nod desemnează o soluţie parţială, iar o muchie reprezintă un pas în construirea unei soluţii. Scopul căutării poate fi acela de

- **a găsi *un drum* în graf de la un *nod iniţial* la un *nod-scop* (cu alte cuvinte, de la o situaţie iniţială la una finală);**
- **a găsi un *nod-scop*.**

Agenţii cu care vom lucra vor adopta un *scop* şi vor urmări *satisfacerea* lui.

Rezolvarea problemelor prin intermediul căutării

În procesul de rezolvare a problemelor, *formularea scopului*, bazată pe situaţia curentă, reprezintă primul pas.

Vom considera un *scop* ca fiind o mulţime de stări ale universului, şi anume acele stări în care scopul este satisfăcut. *Acţiunile* pot fi privite ca generând tranziţii între stări ale universului. Agentul va trebui să afle care acţiuni îl vor conduce la o stare în care scopul este satisfăcut. Înainte de a face asta el trebuie să decidă ce tipuri de acţiuni şi de stări să ia în consideraţie.

Procesul decizional cu privire la acţiunile şi stările ce trebuie luate în consideraţie reprezintă *formularea problemei*. Formularea problemei urmează după formularea scopului.

Un agent care va avea la dispoziţie mai multe opţiuni imediate va decide ce să facă examinând mai întâi diferite *secvenţe de acţiuni* posibile, care conduc la stări de valori necunoscute, urmând ca, în urma acestei examinări, să o aleagă pe cea mai bună. Procesul de examinare a unei astfel de succesiuni de acţiuni se numeşte *căutare*. Un *algoritm de căutare* primeşte ca *input* o *problemă* şi întoarce ca *output* o *soluţie* sub forma unei succesiuni de acţiuni.

Odată cu găsirea unei soluţii, acţiunile recomandate de aceasta pot fi duse la îndeplinire. Aceasta este *faza de execuţie*. Prin urmare, agentul *formulează*, *caută* şi *execută*. După formularea unui scop şi a unei probleme de rezolvat, agentul cheamă o

procedură de căutare pentru a o rezolva. El foloseşte apoi soluţia pentru a-l ghida în acţiunile sale, executând ceea ce îi recomandă soluţia ca fiind următoarea acţiune de îndeplinit şi apoi înlătură acest pas din succesiunea de acţiuni. Odată ce soluţia a fost executată, agentul va găsi un nou scop.

Iată o funcţie care implementează un *agent rezolvator de probleme* simplu:

function AGENT_REZOLVATOR_DE_PROBLEME(*p*)
return o acţiune

inputs: *p*, o percepţie

static: *s*, o secvenţă de acţiuni, iniţial vidă

stare, o descriere a stării curente a lumii (universului)

g, un scop, iniţial vid

problemă, o formulare a unei probleme

stare ← ACTUALIZEAZĂ_STARE(*stare*, *p*)
if *s* este vid **then**

g ← FORMULEAZĂ_SCOP(*stare*)

problemă ← FORMULEAZĂ_PROBLEMĂ(*stare*, *g*)

s ← CAUTĂ(*problemă*)

acţiune ← RECOMANDARE(*s*, *stare*)

s ← REST(*s*, *stare*)
return *acţiune*

In cele ce urmeaza, ne vom ocupa de diferite versiuni ale funcţiei CAUTĂ.

Probleme şi soluţii corect definite

➢ *Probleme cu o singură stare*

Elementele de bază ale definirii unei probleme sunt *stările* şi *acţiunile*. Pentru a descrie stările şi acţiunile, din punct de vedere formal, este nevoie de următoarele elemente [9]:

- ***Starea iniţială* în care agentul ştie că se află.**
- **Mulţimea acţiunilor posibile disponibile agentului. Termenul de *operator* este folosit pentru a desemna descrierea unei acţiuni, prin specificarea stării în care se va ajunge ca urmare a îndeplinirii acţiunii respective, atunci când ne aflăm într-o anumită stare. (O formulare alternativă foloseşte o *funcţie succesor S*. Fiind dată o anumită stare *x*, *S(x)* întoarce mulţimea stărilor în care se poate ajunge din *x*, printr-o unică acţiune).**

- ***Spaţiul de stări*** **al unei probleme reprezintă mulţimea tuturor stărilor în care se poate ajunge plecând din starea iniţială, prin intermediul oricărei secvenţe de acţiuni.**
- **Un *drum* în spaţiul de stări este orice secvenţă de acţiuni care conduce de la o stare la alta.**
- ***Testul scop*** **este testul pe care un agent îl poate aplica unei singure descrieri de stare pentru a determina dacă ea este o *stare de tip scop*, adică o stare în care scopul este atins (sau realizat). Uneori există o mulţime explicită de stări scop posibile şi testul efectuat nu face decât să verifice dacă s-a ajuns în una dintre ele. Alteori, scopul este specificat printr-o proprietate abstractă şi nu prin enumerarea unei mulţimi de stări. De exemplu, în şah, scopul este să se ajungă la o stare numită "şah mat", în care regele**

adversarului poate fi capturat la următoarea mutare, orice ar face adversarul. S-ar putea întâmpla ca o soluţie să fie preferabilă alteia, chiar dacă amândouă ating scopul. Spre exemplu, pot fi preferate drumuri cu mai puţine acţiuni sau cu acţiuni mai puţin costisitoare.

- ***Funcţia de cost a unui drum*** **este o funcţie care atribuie un cost unui drum. Ea este adeseori notată prin *g*. Vom considera costul unui drum ca fiind suma costurilor acţiunilor individuale care compun drumul.**

Împreună *starea iniţială*, *mulţimea operatorilor*, *testul scop* şi *funcţia de cost a unui drum* definesc o *problemă*.

➢ ***Probleme cu stări multiple***

Pentru definirea unei astfel de probleme trebuie specificate:

- **o *mulţime* de stări iniţiale;**
- **o mulţime de operatori care indică, în cazul fiecărei acţiuni, mulţimea stărilor în care se ajunge plecând de la orice stare dată;**
- **un test scop (la fel ca la problema cu o singură stare);**
- **funcţia de cost a unui drum (la fel ca la problema cu o singură stare).**

Un operator se aplică unei mulţimi de stări prin reunirea rezultatelor aplicării operatorului fiecărei stări din mulţime. Aici

un *drum* leagă *mulţimi de stări*, iar o *soluţie* este un drum care conduce la o *mulţime de stări*, dintre care *toate sunt stări scop*. Spaţiul de stări este aici înlocuit de *spaţiul mulţimii de stări*.

Un exemplu: Problema misionarilor si a canibalilor

Definiţie formală a problemei:

- ***Stări*: o stare constă dintr-o secvenţă ordonată de trei numere reprezentând numărul de misionari, de canibali şi de bărci, care se află pe malul râului. Starea de pornire (iniţială) este (3,3,1).**
- ***Operatori*: din fiecare stare, posibilii operatori trebuie să ia fie un misionar, fie un canibal, fie doi misionari, fie doi canibali, fie câte unul din fiecare şi să îi transporte cu barca. Prin urmare, există *cel mult cinci operatori*, deşi majorităţii stărilor le corespund mai puţini operatori, intrucât trebuie evitate stările interzise.**

(*Observaţie*: Dacă am fi ales să distingem între indivizi, în loc de cinci operatori ar fi existat 27).

- ***Testul scop*: să se ajungă în starea (0,0,0).**
- ***Costul drumului*: este dat de numărul de traversări.**

Acest spaţiu al stărilor este suficient de mic pentru ca problema să fie una trivială pentru calculator.

Eficacitatea unei căutări poate fi măsurată în cel puţin trei moduri, şi anume conform următoarelor criterii de bază:

- **dacă se găseşte o soluţie;**
- **dacă s-a găsit o soluţie bună (adică o soluţie cu un cost scăzut al drumului);**
- **care este costul căutării asociat timpului calculator şi memoriei necesare pentru a găsi o soluţie.**

Căutarea soluţiilor şi generarea secvenţelor de acţiuni

Rezolvarea unei probleme începe cu starea iniţială. Primul pas este acela de a testa dacă starea iniţială este o stare scop. Dacă nu, se iau în consideraţie şi alte stări. Acest lucru se realizează aplicând operatorii asupra stării curente şi, în consecinţă, generând o mulţime de stări. Procesul poartă denumirea de *extinderea stării*. Atunci când se generează mai multe posibilităţi, trebuie făcută o alegere relativ la cea care va fi luată în consideraţie în continuare, aceasta fiind esenţa căutării. *Alegerea referitoare la care dintre stări trebuie extinsă prima este determinată de strategia de căutare.*

Procesul de căutare construieşte un *arbore de căutare*, a cărui rădăcină este un *nod de căutare* corespunzând stării iniţiale. La fiecare pas, algoritmul de căutare alege un nod-frunză pentru a-l extinde. *Algoritmul de căutare general* este următorul:

Function CĂUTARE_GENERALĂ (*problemă, strategie*)

return soluţie sau eşec

iniţializează arborele de căutare folosind starea iniţială a lui *problemă*

ciclu do

if nu există candidaţi pentru extindere

then return eşec

alege un nod-frunză pentru extindere conform lui *strategie*

if nodul conţine o stare scop

then return soluţia corespunzătoare

else extinde nodul şi adaugă nodurile rezultate arborelui de căutare

end

Este important să facem *distincţia între spaţiul stărilor şi arborele de căutare*. Spre exemplu, într-o problemă de căutare a unui drum pe o hartă, pot exista doar 20 de stări în spaţiul stărilor, câte una pentru fiecare oraş. Dar există un număr infinit de drumuri în acest spaţiu de stări. Prin urmare, arborele de căutare are un număr infinit de noduri. Evident, un bun algoritm de căutare trebuie să evite urmarea unor asemenea drumuri.

Este importantă distincţia între noduri şi stări. Astfel, un nod este o structură de date folosită pentru a reprezenta arborele de căutare corespunzător unei anumite realizări a unei probleme, generată de un anumit algoritm. O stare reprezintă însă o configuraţie a lumii înconjurătoare. De aceea, nodurile au adâncimi şi părinţi, iar stările nu le au. Mai mult, este posibil ca două noduri diferite să conţină aceeaşi stare, dacă acea stare este generată prin intermediul a două secvenţe de acţiuni diferite.

Există numeroase moduri de a reprezenta nodurile. În general, se consideră că un nod este o structură de date cu cinci componente:

- starea din spaţiul de stări căreia îi corespunde nodul;

- nodul din arborele de căutare care a generat acest nod (nodul părinte);

- operatorul care a fost aplicat pentru a se genera nodul;

- numărul de noduri aflate pe drumul de la rădăcină la acest nod (*adâncimea* nodului);

- costul drumului de la starea iniţială la acest nod.

Este necesară, de asemenea, reprezentarea colecţiei de noduri care aşteaptă pentru a fi extinse. Această colecţie de noduri poartă denumirea de *frontieră*. Cea mai simplă reprezentare ar fi aceea a unei mulţimi de noduri, iar strategia de căutare ar fi o funcţie care selectează, din această mulţime, următorul nod ce trebuie extins. Deşi din punct de vedere conceptual această cale este una directă, din punct de vedere computaţional ea poate fi foarte scumpă, pentru că funcţia strategie ar trebui să se "uite" la fiecare element al mulţimii pentru a-l alege pe cel mai bun. De aceea, *vom presupune că această colecţie de noduri este implementată ca o coadă*.

Operaţiile definite pe o coadă vor fi următoarele:

- CREEAZĂ_COADA(*Elemente*) - creează o coadă cu elementele date;

- GOL?(*Coada*) - întoarce „true” numai dacă nu mai există elemente în coadă;

- ÎNLĂTURĂ_DIN_FAŢĂ(*Coada*) - înlătură elementul din faţă al cozii şi îl transmite înapoi (return);

- COADA_FN(*Elemente, Coada*) - inserează o mulţime de elemente în coadă. Diferite feluri de funcţii de acest tip produc algoritmi de căutare diferiţi.

Cu aceste definiţii, putem da o descriere mai formală a *algoritmului general de căutare*:

```
function CĂUTARE_GENERALĂ(problemă,COADA_FN)  return
            o soluţie sau eşec
  noduri ← CREEAZĂ_COADA(CREEAZĂ_NOD(STARE_INIŢIALĂ
          [problemă]))
  ciclu do
      if noduri este vidă
                then return eşec
      nod  ←  ÎNLĂTURĂ_DIN_FAŢĂ(noduri)
      if     TEST_SCOP[problemă]  aplicat  lui  STARE(nod)  are
    succes
                then return nod
       noduri←COADA_FN(noduri, EXTINDERE(nod, OPERATORI
              [problemă]))
  end
```

Evaluarea strategiilor de căutare

Strategiile de căutare se evaluează conform următoarelor patru criterii:

- **Completitudine: dacă, atunci când o soluţie există, strategia dată garantează găsirea acesteia;**
- **Complexitate a timpului: durata de timp pentru găsirea unei soluţii;**
- **Complexitate a spaţiului: necesităţile de memorie pentru efectuarea căutării;**
- **Optimalitate: atunci când există mai multe soluţii, strategia dată să o găsească pe cea mai de calitate dintre ele.**

Termenul de *căutare neinformată* desemnează faptul că o strategie de acest tip nu deţine nici o informaţie despre numărul de paşi sau despre costul drumului de la starea curentă la scop. Tot ceea ce se poate face este să se distingă o stare-scop de o stare care nu este scop. Căutarea neinformată se mai numeşte şi *căutarea oarbă*.

Să considerăm, de pildă, problema găsirii unui drum de la Arad la Bucureşti, având în faţă o hartă. De la starea iniţială, Arad, există trei acţiuni care conduc la trei noi stări: Sibiu, Timişoara şi Zerind. O căutare neinformată nu are nici o preferinţă între cele trei variante. Un agent mai inteligent va observa însă că scopul, Bucureşti, se află la sud-est de Arad şi că numai *Sibiu* este în această direcţie, care reprezintă, probabil, cea mai bună alegere. Strategiile care folosesc asemenea consideraţii se numesc *strategii de căutare informată* sau *strategii de căutare euristică*.

Căutarea neinformată este mai puţin eficientă decât cea informată. Căutarea

neinformată este însă importantă deoarece există foarte multe probleme pentru care nu este disponibilă nici o informaţie suplimentară.

În cele ce urmează, vom aborda mai multe strategii de căutare neinformată, acestea deosebindu-se între ele prin ordinea în care nodurile sunt extinse.

Căutarea de tip breadth-first

Strategia de căutare de tip breadth-first extinde mai întâi nodul rădăcină. Apoi se extind toate nodurile generate de nodul rădăcină, apoi succesorii lor şi aşa mai departe. În general, toate nodurile aflate la adâncimea *d* în arborele de căutare sunt extinse înaintea nodurilor aflate la adâncimea *d+1*. Spunem ca aceasta este o *căutare în lăţime*.

Căutarea de tip breadth-first poate fi implementată chemând algoritmul general de căutare, CĂUTARE_GENERALĂ, cu o funcţie COADA_FN care plasează stările nou generate *la sfârşitul cozii*, după toate stările generate anterior.

Strategia breadth-first este foarte *sistematică* deoarece ia în consideraţie toate drumurile de lungime 1, apoi pe cele de lungime 2 etc., aşa cum se arată în Fig. 2.1. Dacă există o soluţie, este sigur că această metodă o va găsi, iar dacă există mai multe soluţii, căutarea de tip breadth-first va găsi întotdeauna mai întâi *soluţia cel mai puţin adâncă*.

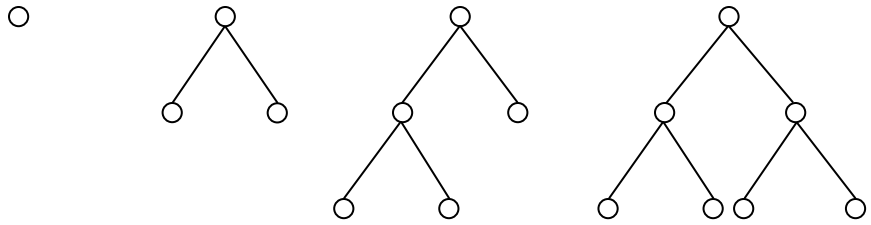

Fig. 2.1

În termenii celor patru criterii de evaluare a strategiilor de căutare, cea de tip breadth-first este *completă* şi este *optimă* cu condiţia ca *costul drumului să fie o funcţie descrescătoare de adâncimea nodului*. Această condiţie este de obicei satisfăcută numai atunci când toţi operatorii au acelaşi cost.

Algoritmul de căutare breadth-first

Presupunând că a fost specificată o mulţime de reguli care descriu acţiunile sau operatorii disponibili, *algoritmul de căutare breadth-first* se defineşte după cum urmează:

Algoritmul 2.1

1. Creează o variabilă numită LISTA_NODURI şi setează-o la starea iniţială.

2. Până când este găsită o stare-scop sau până când LISTA_NODURI devine vidă, execută:

2.1. Înlătură primul element din LISTA_NODURI şi numeşte-l E. Dacă LISTA_NODURI a fost vidă, STOP.

2.2. Pentru fiecare mod în care fiecare regulă se potriveşte cu starea descrisă în E, execută:

2.2.1. Aplică regula pentru a genera o nouă stare.

2.2.2. Dacă noua stare este o stare-scop, întoarce această stare şi STOP.

2.2.3. Altfel, adaugă noua stare la sfârşitul lui LISTA_NODURI.

Implementare în Prolog

Pentru a programa în Prolog strategia de căutare breadth-first, trebuie menţinută în memorie *o mulţime de noduri candidate alternative*. Această mulţime de candidaţi reprezintă marginea de jos a arborelui de căutare, aflată în continuă creştere (frontiera). Totuşi, această mulţime de noduri nu este suficientă dacă se doreşte şi extragerea unui drum-soluţie în urma procesului de căutare. Prin urmare, în loc de a menţine o mulţime de noduri candidate, vom menţine *o mulţime de drumuri candidate*.

Este utilă, pentru programarea în Prolog, o anumită reprezentare a mulţimii de drumuri candidate, şi anume: *mulţimea va fi*

***reprezentată ca o listă de drumuri*, *iar fiecare drum va fi o listă de noduri în ordine inversă*. Capul listei va fi, prin urmare, nodul cel mai recent generat, iar ultimul element al listei va fi nodul de început al căutării.**

Căutarea este începută cu o mulţime de candidaţi având un singur element:

[[NodIniţial]].

Fiind dată o mulţime de drumuri candidate, căutarea de tip breadth-first se desfăşoară astfel:

- **dacă primul drum conţine un nod-scop pe post de cap, atunci acesta este o soluţie a problemei;**
- **altfel, înlătură primul drum din mulţimea de candidaţi şi generează toate extensiile de un pas ale acestui drum, adăugând această mulţime de extensii la sfârşitul mulţimii de candidaţi. Execută apoi căutarea de tip breadth-first asupra mulţimii astfel actualizate.**

Vom considera un exemplu în care nodul *a* este nodul de start, *f* şi *j* sunt nodurile-scop, iar spaţiul stărilor este cel din Fig. 2.2:

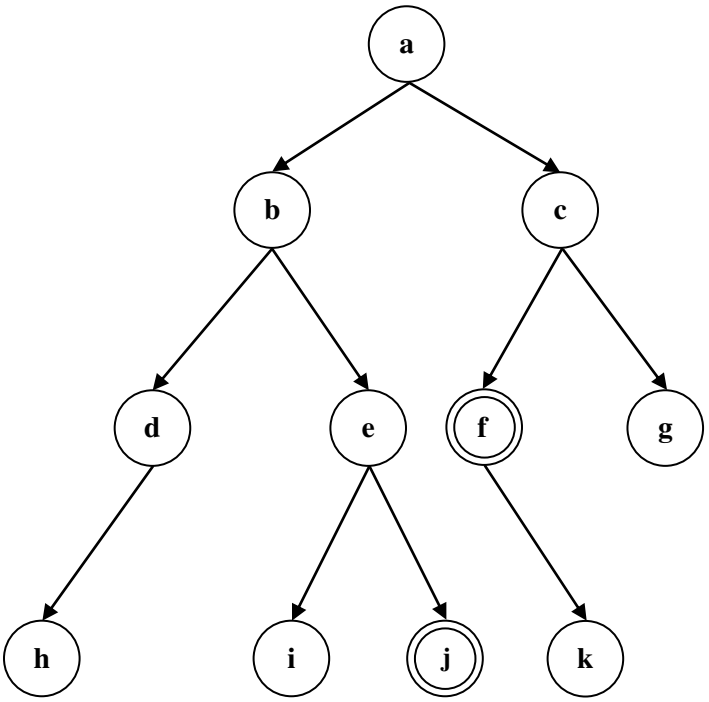

Fig. 2.2

Ordinea în care strategia breadth-first vizitează nodurile din acest spaţiu de stări este: *a, b, c, d, e, f*. Soluţia mai scurtă [a, c, f] va fi găsită înaintea soluţiei mai lungi [a, b, e, j].

Pentru figura anterioară, căutarea breadth-first se desfăşoară astfel:

(1) Se începe cu mulţimea de candidaţi iniţială:

[[a]]

(2) Se generează extensii ale lui [a]:

[[b, a], [c, a]]

(Se observă reprezentarea drumurilor în ordine inversă).

(3) Se înlătură primul drum candidat, [b, a], din mulţime şi se generează extensii ale acestui drum:

[[d, b, a], [e, b, a]]

Se adaugă lista extensiilor la sfârşitul mulţimii de candidaţi:

[[c, a], [d, b, a], [e, b, a]]

(4) Se înlătură [c, a] şi se adaugă extensiile sale la sfârşitul mulţimii de candidaţi, rezultând următoarea mulţime de drumuri:

[[d, b, a], [e, b, a], [f, c, a], [g, c, a]]

La următorii paşi, [d, b, a] şi [e, b, a] sunt extinse, iar mulţimea de candidaţi modificată devine:

[[f, c, a], [g, c, a], [h, d, b, a], [i, e, b, a], [j, e, b, a]]

Acum procesul de căutare întâlneşte [f, c, a], care conţine un nod scop, *f*. Prin urmare, acest drum este returnat ca soluţie.

Programul Prolog care implementează acest proces de căutare va reprezenta mulţimile de noduri ca pe nişte liste, efectuând şi un test care să prevină generarea unor drumuri ciclice. În cadrul acestui program, toate extensiile de un pas vor fi generate prin utilizarea procedurii încorporate *bagof*:

```
rezolva_b(Start,Sol):-
    breadthfirst([[Start]],Sol).
breadthfirst([[Nod|Drum]|_],[Nod|Drum]):-
    scop(Nod).
breadthfirst([Drum|Drumuri],Sol):-
    extinde(Drum,DrumuriNoi),
    concat(Drumuri,DrumuriNoi,Drumuri1),
    breadthfirst(Drumuri1,Sol).
extinde([Nod|Drum],DrumuriNoi):-
    bagof([NodNou,Nod|Drum],
        (s(Nod,NodNou),
        \+(membru(NodNou,[Nod|Drum]))),
        DrumuriNoi),
        !.
extinde(_,[]).
```

Predicatul `rezolva_b(Start,Sol)` este adevărat dacă Sol este un drum (în ordine inversă) de la nodul inițial Start la o stare-scop, drum obținut folosind căutarea de tip breadth-first.

Predicatul `breadthfirst(Drumuri,Sol)` este adevărat dacă un drum din mulțimea de drumuri candidate numită Drumuri poate fi extins la o stare-scop; un astfel de drum este Sol.

Predicatul `extinde(Drum,DrumuriNoi)` este adevărat dacă prin extinderea mulțimii de noduri Drum obținem mulțimea numită DrumuriNoi, el generând mulțimea tuturor extensiilor acestui drum.

Predicatul `concat(Drumuri,DrumuriNoi,Drumuri1)` este adevărat dacă, atunci când concatenăm lista de

noduri Drumuri cu lista de noduri DrumuriNoi, obţinem lista de noduri Drumuri1.

Predicatul `membru(NodNou,[Nod|Drum])` este adevărat dacă nodul numit NodNou aparţine listei de noduri [Nod|Drum].

Fapta Prolog `scop(Nod)` arată că Nod este un nod-scop.

Funcţia de succesiune este implementată astfel:

`s(Nod,NodNou)` desemnează faptul că NodNou este nodul succesor al nodului Nod.

Programul Prolog <u>complet</u> corespunzător exemplului din Fig. 2.2. Programul implementează strategia de căutare breadth-first pentru a găsi soluţiile, cele două drumuri [f, c, a] şi respectiv [j, e, b, a]:

Programul 2.1

```
scop(f). % specificare noduri-scop
scop(j).
s(a,b).  % descrierea funcţiei succesor
s(a,c).
s(b,d).
s(d,h).
s(b,e).
s(e,i).
s(e,j).
s(c,f).
s(c,g).
s(f,k).

concat([],L,L).
concat([H|T],L,[H|T1]):-concat(T,L,T1).

membru(H,[H|T]).
membru(X,[H|T]):-membru(X,T).

rezolva_b(Start,Sol):-
    breadthfirst([[Start]],Sol).
```

```
breadthfirst([[Nod|Drum]|_],[Nod|Drum]):-
    scop(Nod).
breadthfirst([Drum|Drumuri],Sol):-
       extinde(Drum,DrumuriNoi),
       concat(Drumuri,DrumuriNoi,Drumuri1),
       breadthfirst(Drumuri1,Sol).

extinde([Nod|Drum],DrumuriNoi):-
bagof([NodNou,Nod|Drum],(s(Nod,NodNou),
 \+(membru(NodNou,[Nod|Drum]))),DrumuriNoi),
!.
extinde(_,[]).
```

Interogarea Prologului se face astfel:

```
?- rezolva_b(a,Sol).
```

Răspunsul Prologului va fi:

```
Sol=[f, c, a] ? ;
Sol=[j, e, b, a] ? ;
no
```

Cele două soluţii au fost obţinute ca liste de noduri în ordine inversă, plecându-se de la nodul de start *a*.

CONTINUARE PE FOLIA 333

Timpul şi memoria cerute de strategia breadth-first

Pentru a vedea cantitatea de timp şi de memorie necesare completării unei căutări, vom lua în consideraţie un spaţiu al stărilor ipotetic, în care fiecare stare poate fi extinsă pentru a genera *b* stări noi. Se spune că *factorul de ramificare* al acestor stări (şi al arborelui de căutare) este *b*. Rădăcina generează *b* noduri la primul nivel, fiecare dintre acestea generează încă *b* noduri, rezultând un total de b^2 noduri la al doilea nivel ş.a.m.d.. Să presupunem că soluţia acestei probleme este un drum de lungime *d*. Atunci numărul maxim de noduri extinse înainte de găsirea unei soluţii este:

$$1+b+b^2+\cdots+b^d.$$

Prin urmare, algoritmul are o *complexitate exponenţială* de $O(b^d)$. Complexitatea spaţiului

este aceeaşi cu complexitatea timpului deoarece toate nodurile frunză ale arborelui trebuie să fie menţinute în memorie în acelaşi timp.

Iată câteva exemple de execuţii cu factor de ramificare b=10:

Adânci-me	Noduri	Timp	Memorie
2	111	0.1 sec.	11 kilobytes
6	10^6	18 min.	111 megabytes
8	10^8	31 ore	11 gigabytes
12	10^{12}	35 ani	111 terabytes

Se observă că cerinţele de memorie sunt o problemă mai mare, pentru căutarea de tip breadth-first, decât timpul de execuţie. (Este posibil să putem aştepta 18 minute pentru a se efectua o căutare de adâncime 6, dar să nu dispunem de 111 megabytes de memorie). La rândul lor, cerinţele de timp sunt majore. (Dacă problema are o soluţie la adâncimea 12, o căutare neinformată de acest tip o găseşte în 35 de ani). În

general, problemele de căutare de *complexitate exponenţială* nu pot fi rezolvate decât pe mici porţiuni.

Alboritmul breadth-first in varianta in care nu se mai face extinderea drumurilor [d, b, a] si [e, b, a] (vezi exemplul anterior).

```
s(a,b).
s(a,c).
s(b,d).
s(b,e).
s(c,f).
s(c,g).
s(d,h).
s(e,i).
s(e,j).
s(f,k).

scop(f).
scop(j).

bf(NodInitial,Solutie):-
breadthfirst([[NodInitial]],Solutie).

breadthfirst(D,S):-terminare(D,S),!.

terminare([[Nod|Drum]|_],[Nod|Drum]):-scop(Nod).
terminare([_|T],S):-terminare(T,S).

breadthfirst([Drum|Drumuri],Solutie):-
        extinde(Drum,DrumNoi),
        concat(Drumuri,DrumNoi,Drumuri1),
```