

# TEHNICI DE CĂUTARE

Căutarea este o traversare sistematică a unui spațiu de soluții posibile ale unei probleme.

Un spațiu de căutare este de obicei un graf (sau, mai exact, un arbore) în care un nod desemnează o soluție parțială, iar o muchie reprezintă un pas în construirea unei soluții. Scopul căutării poate fi acela de

- a găsi un drum în graf de la un *nod inițial* la un *nod-scop* (cu alte cuvinte, de la o situație inițială la una finală);
- a găsi un nod-scop.

**Agenții cu care vom lucra vor adopta un *scop* și vor urmări *satisfacerea* lui.**

## **Rezolvarea problemelor prin intermediul căutării**

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

**Vom considera un *scop* ca fiind o mulțime de stări ale universului, și anume acele stări în care scopul este satisfăcut. *Acțiunile* pot fi privite ca generând tranziții între stări ale universului. Agentul va trebui să afle care acțiuni îl vor conduce la o stare în care scopul este satisfăcut. Înainte de a face asta el trebuie să decidă ce tipuri de acțiuni și de stări să ia în considerație.**

**Procesul decizional cu privire la acțiunile și stările ce trebuie luate în considerație reprezintă *formularea problemei*. Formularea problemei urmează după formularea scopului.**

Un agent care va avea la dispoziție mai multe opțiuni imediate va decide ce să facă examinând mai întâi diferite *secvențe de acțiuni* 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 acțiuni se numește *căutare*. Un *algoritm de căutare* primește ca *input* o *problemă* și întoarce ca *output* o *soluție* sub forma unei succesiuni de acțiuni.

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

**procedură de căutare pentru a o rezolva. El folosește apoi soluția pentru a-l ghida în acțiunile sale, executând ceea ce îi recomandă soluția ca fiind următoarea acțiune de îndeplinit și apoi înlătură acest pas din succesiunea de acțiuni. Odată ce soluția a fost executată, agentul va găsi un nou scop.**

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

**function**        AGENT\_REZOLVATOR\_DE\_PROBLEME(*p*)  
**return** o acțiune

**inputs:** *p*, o percepție

**static:** *s*, o secvență de acțiuni, inițial vidă

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

*g*, un scop, inițial 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*)

*acțiune* ← RECOMANDARE(*s*, *stare*)

*s* ← REST(*s*, *stare*)

**return** *acțiune*

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

## Probleme și soluții corect definite

### ➤ Probleme cu o singură stare

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

- Starea inițială în care agentul știe că se află.
- Mulțimea acțiunilor posibile disponibile agentului. Termenul de operator este folosit pentru a desemna descrierea unei acțiuni, prin specificarea stării în care se va ajunge ca urmare a îndeplinirii acțiunii respective, atunci când ne aflăm într-o anumită stare. (O formulare alternativă folosește o *funcție succesor*  $S$ . Fiind dată o anumită stare  $x$ ,  $S(x)$  întoarce mulțimea stărilor în care se poate ajunge din  $x$ , printr-o unică acțiune).

- Spațiul de stări al unei probleme reprezintă mulțimea tuturor stărilor în care se poate ajunge plecând din starea inițială, prin intermediul oricărei secvențe de acțiuni.
- Un drum în spațiul de stări este orice secvență de acțiuni 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 mulțime explicită de stări scop posibile și testul efectuat nu face decât să verifice dacă s-a ajuns în una dintre ele. Alteori, scopul este specificat printr-o proprietate abstractă și nu prin enumerarea unei mulțimi de stări. De exemplu, în șah, scopul este să se ajungă la o stare numită “șah mat”, în care regele



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

- Funcția de cost a unui drum este o funcție care atribuie un cost unui drum. Ea este adeseori notată prin  $g$ . Vom considera costul unui drum ca fiind suma costurilor acțiunilor individuale care compun drumul.

Împreună starea inițială, mulțimea operatorilor, testul scop și funcția de cost a unui drum definesc o problemă.

### ➤ Probleme cu stări multiple

Pentru definirea unei astfel de probleme trebuie specificate:

- o mulțime de stări inițiale;
- o mulțime de operatori care indică, în cazul fiecărei acțiuni, mulțimea stărilor în care se ajunge plecând de la orice stare dată;
- un test scop (la fel ca la problema cu o singură stare);
- funcția de cost a unui drum (la fel ca la problema cu o singură stare).

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

un *drum* leagă *mulțimi de stări*, iar o *soluție* este un drum care conduce la o *mulțime de stări*, dintre care *toate sunt stări scop*. Spațiul de stări este aici înlocuit de *spațiul mulțimii de stări*.

## Un exemplu: Problema misionarilor si a canibalilor

### Definiție formală a problemei:

- Stări: o stare constă dintr-o secvență ordonată de trei numere reprezentând numărul de misionari, de canibali și de bărci, care se află pe malul râului. Starea de pornire (inițială) este (3,3,1).
- Operatori: din fiecare stare, posibili operatori trebuie să ia fie un misionar, fie un canibal, fie doi misionari, fie doi canibali, fie câte unul din fiecare și să îi transporte cu barca. Prin urmare, există *cel mult cinci operatori*, deși majorității stărilor le corespund mai puțini operatori, întrucât trebuie evitate stările interzise.

**(Observație: 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 spațiu al stărilor este suficient de mic pentru ca problema să fie una trivială pentru calculator.**

**Eficacitatea unei căutări poate fi măsurată în cel puțin trei moduri, și anume conform următoarelor criterii de bază:**

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

## **Căutarea soluțiilor și generarea secvențelor** **de acțiuni**

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

Procesul de căutare construiește un arbore de căutare, a cărui rădăcină este un nod de căutare corespunzând stării inițiale. 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** soluție sau eșec

inițializează arborele de căutare folosind starea  
inițială a lui *problemă*

**ciclu do**

**if** nu există candidați pentru extindere

**then return** eșec

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

**if** nodul conține o stare scop

**then return** soluția corespunzătoare

**else** extinde nodul și adaugă nodurile rezultate  
arborelui de căutare

**end**

**Este important să facem distincția între spațiul stărilor și arborele de căutare. Spre exemplu, într-o problemă de căutare a unui drum pe o hartă, pot exista doar 20 de stări în spațiul stărilor, câte una pentru fiecare oraș. Dar există un număr infinit de drumuri în acest spațiu 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 distincția între noduri și 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 configurație a lumii înconjurătoare. De aceea, nodurile au adâncimi și părinți, iar stările nu le au. Mai mult, este posibil ca două noduri diferite să conțină aceeași stare, dacă acea stare este generată prin intermediul a două secvențe de acțiuni 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 spațiul 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 inițială la acest nod.**

Este necesară, de asemenea, reprezentarea colecției de noduri care așteaptă pentru a fi extinse. Această colecție de noduri poartă denumirea de *frontieră*. Cea mai simplă reprezentare ar fi aceea a unei mulțimi de noduri, iar strategia de căutare ar fi o funcție care selectează, din această mulțime, următorul nod ce trebuie extins. Deși din punct de vedere conceptual această cale este una directă, din punct de vedere computațional ea poate fi foarte scumpă, pentru că funcția strategie ar trebui să se “uite” la fiecare element al mulțimii pentru a-l alege pe cel mai bun. De aceea, *vom presupune că această colecție de noduri este implementată ca o coadă.*

**Operațiile definite pe o coadă vor fi**  
**următoarele:**

- **CREEAZĂ\_COADA(*Elemente*)** - creează o coadă cu elementele date;
- **GOL?(*Coadă*)** - întoarce „true” numai dacă nu mai există elemente în coadă;
- **ÎNLĂTURĂ\_DIN\_FAȚĂ(*Coadă*)** - înlătură elementul din față al cozii și îl transmite înapoi (return);
- **COADA\_FN(*Elemente, Coadă*)** - inserează o mulțime de elemente în coadă. **Diferite feluri de funcții de acest tip produc algoritmi de căutare diferiți.**

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



```

function CĂUTARE_GENERALĂ(problemă, COADA_FN) return
    o soluție sau eșec
noduri ← CREEAZĂ_COADA(CREEAZĂ_NOD(STARE_INIȚIALĂ
    [problemă]))
ciclu do
    if noduri este vidă
        then return eșec
    nod ← ÎNLĂTURĂ_DIN_FATĂ(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 soluție există, strategia dată garantează găsirea acesteia;**
- Complexitate a timpului: durata de timp pentru găsirea unei soluții;**
- Complexitate a spațiului: necesitățile de memorie pentru efectuarea căutării;**
- Optimalitate: atunci când există mai multe soluții, 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 deține nici o informație despre numărul de pași 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 numește și căutarea oarbă.**

Să considerăm, de pildă, problema găsirii unui drum de la Arad la București, având în față o hartă. De la starea inițială, Arad, există trei acțiuni care conduc la trei noi stări: Sibiu, Timișoara și Zerind. O căutare neinformată nu are nici o preferință între cele trei variante. Un agent mai inteligent va observa însă că scopul, București, se află la sud-est de Arad și că numai *Sibiu* este în această direcție, care reprezintă, probabil, cea mai bună alegere. Strategiile care folosesc asemenea considerații se numesc strategii de căutare informată sau strategii de căutare euristică.

Căutarea neinformată este mai puțin eficientă decât cea informată. Căutarea

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

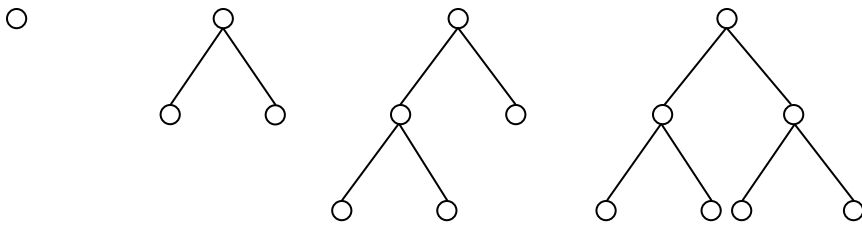
**În cele ce urmează, vom aborda mai multe strategii de căutare neinformată, acestea deosebindu-se între ele prin ordinea în care nodurile sunt extinse.**

## 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 și așa 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 lățime.

Căutarea de tip breadth-first poate fi implementată chemând algoritmul general de căutare, CĂUTARE\_GENERALĂ, cu o funcție COADA\_FN care plasează stările nou generate *la sfârșitul cozii*, după toate stările generate anterior.

Strategia breadth-first este foarte sistematică deoarece ia în considerație toate drumurile de lungime 1, apoi pe cele de lungime 2 etc., așa cum se arată în Fig. 2.1. Dacă există o soluție, este sigur că această metodă o va găsi, iar dacă există mai multe soluții, căutarea de tip breadth-first va găsi întotdeauna mai întâi soluția cel mai puțin adâncă.



**Fig. 2.1**

În termenii celor patru criterii de evaluare a strategiilor de căutare, cea de tip breadth-first este completă și este optimă cu condiția ca costul drumului să fie o funcție descrescătoare de adâncimea nodului. Această condiție este de obicei satisfăcută numai atunci când toți operatorii au același cost.



## Algoritmul de căutare breadth-first

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

### Algoritmul 2.1

1. Creează o variabilă numită LISTA\_NODURI și setează-o la starea inițială.

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

2.1. Înlătură primul element din LISTA\_NODURI și numește-l E. Dacă LISTA\_NODURI a fost vidă, STOP.

2.2. Pentru fiecare mod în care fiecare regulă se potrivește 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 și STOP.**

**2.2.3. Altfel, adaugă noua stare la sfârșitul lui LISTA\_NODURI.**

## Implementare în Prolog

Pentru a programa în Prolog strategia de căutare breadth-first, trebuie menținută în memorie o mulțime de noduri candidate alternative. Această mulțime de candidați reprezintă marginea de jos a arborelui de căutare, aflată în continuă creștere (frontiera). Totuși, această mulțime de noduri nu este suficientă dacă se dorește și extragerea unui drum-soluție în urma procesului de căutare. Prin urmare, în loc de a menține o mulțime de noduri candidate, vom menține o mulțime de drumuri candidate.

Este utilă, pentru programarea în Prolog, o anumită reprezentare a mulțimii de drumuri candidate, și anume: mulțimea 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.**

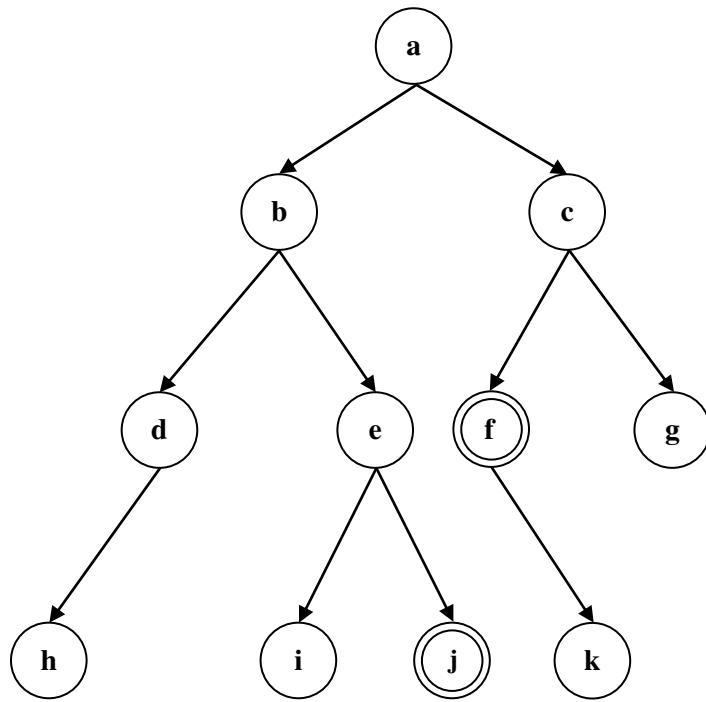
**Căutarea este începută cu o mulțime de candidați având un singur element:**

**[[NodInițial]].**

**Fiind dată o mulțime de drumuri candidate, căutarea de tip breadth-first se desfășoară astfel:**

- dacă primul drum conține un nod-scop pe post de cap, atunci acesta este o soluție a problemei;**
- altfel, înlătură primul drum din mulțimea de candidați și generează toate extensiile de un pas ale acestui drum, adăugând această mulțime de extensii la sfârșitul mulțimii de candidați. Execută apoi căutarea de tip breadth-first asupra mulțimii astfel actualizate.**

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



**Fig. 2.2**

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

**Pentru figura anterioară, căutarea breadth-first se desfășoară astfel:**

**(1) Se începe cu mulțimea de candidați inițială:**

**[[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 mulțime și se generează extensii ale acestui drum:**

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

**Se adaugă lista extensiilor la sfârșitul mulțimii de candidați:**

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

**(4) Se înlătură [c, a] și se adaugă extensiile sale la sfârșitul mulțimii de candidați, rezultând următoarea mulțime de drumuri:**



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

**La următorii pași, [d, b, a] și [e, b, a] sunt extinse, iar mulțimea de candidați modificată devine:**

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

**Acum procesul de căutare întâlnește [f, c, a], care conține un nod scop, *f*. Prin urmare, acest drum este returnat ca soluție.**

**Programul Prolog care implementează acest proces de căutare va reprezenta mulțimile de noduri ca pe niște liste, efectuând și 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, obținem lista de noduri Drumuri1.**

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

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

**Funcția de succesiune este implementată astfel:**

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

**Programul Prolog complet corespunzător exemplului din Fig. 2.2. Programul implementează strategia de căutare breadth-first pentru a găsi soluțiile, cele două drumuri [f, c, a] și respectiv [j, e, b, a]:**

### **Programul 2.1**

```
scop(f). % specificare noduri-scop
scop(j).
s(a,b). % descrierea funcției 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 soluții au fost obținute ca liste de noduri în ordine inversă, plecându-se de la nodul de start *a*.**

**CONTINUARE PE FOLIA 333**



**Algoritmul 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),**

```
breadthfirst(Drumuri1,Solutie).  
  
extinde([Nod|Drum],DrumNoi):-  
    bagof([NodNou,Nod|Drum],  
        (s(Nod, NodNou),  
         \+ (membru(NodNou,[Nod|Drum]))),DrumNoi),!.  
extinde(_,[]).
```

```
membru(X,[X|_]).  
membru(X,[_|Y]):-membru(X,Y).
```

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

---

### Interogarea Prologului:

?- bf(a,Solutie).

Solutie = [f,c,a] ?;

no

Observatie: Se obtine numai prima solutie, cea mai scurta.

## Timpul și memoria cerute de strategia breadth-first

Pentru a vedea cantitatea de timp și de memorie necesare completării unei căutări, vom lua în considerație un spațiu 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 (și 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 ș.a.m.d.. Să presupunem că soluția acestei probleme este un drum de lungime  $d$ . Atunci numărul maxim de noduri extinse înainte de găsirea unei soluții este:

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

Prin urmare, algoritmul are o complexitate exponențială de  $O(b^d)$ . Complexitatea spațiului

**este aceeași cu complexitatea timpului deoarece toate nodurile frunză ale arborelui trebuie să fie menținute în memorie în același timp.**

Iată câteva exemple de execuții 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 cerințele de memorie sunt o problemă mai mare, pentru căutarea de tip breadth-first, decât timpul de execuție. (Este posibil să putem aștepta 18 minute pentru a se efectua o căutare de adâncime 6, dar să nu dispunem de 111 megabytes de memorie). La rândul lor, cerințele de timp sunt majore. (Dacă problema are o soluție la adâncimea 12, o căutare neinformată de acest tip o găsește în 35 de ani). În

general, problemele de căutare de complexitate  
exponențială nu pot fi rezolvate decât pe mici porțiuni.