Căutarea de tip depth-first

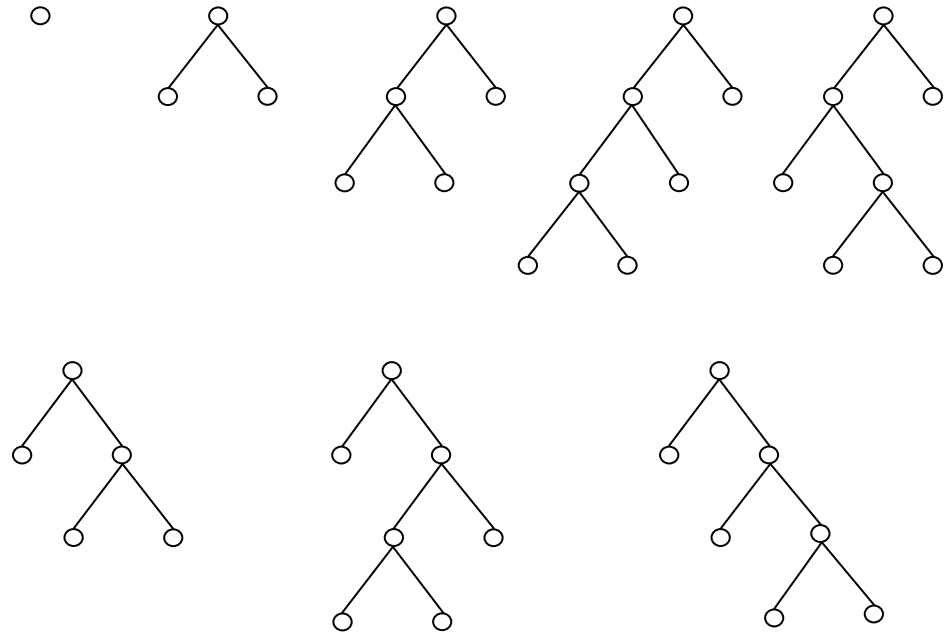
- **Strategia de căutare de tip depth-first extinde întotdeauna unul dintre nodurile aflate la nivelul *cel mai adânc* din arbore. Căutarea se întoarce înapoi şi sunt extinse noduri aflate la adâncimi mai mici numai atunci când a fost atins un nod care nu reprezintă un nod-scop şi care nu mai poate fi extins. Spunem că aceasta este o *căutare în adâncime*.**

- **Implementarea se poate face cu o structură de date de tip *coadă*, care întotdeauna va plasa stările nou generate *în faţa cozii*.**

- ***Necesităţile de memorie* sunt foarte mici la această metodă. Este necesar să se memoreze un singur drum de la rădăcină la un nod-frunză, împreună cu nodurile-frate rămase**

neextinse corespunzător fiecărui nod de pe drum.

- **Modul de a progresa al căutării de tip depth-first este ilustrat în figura urmatoare:**



Pentru un spaţiu al stărilor cu factor de ramificare *b* şi adâncime maximă *m*, trebuie memorate numai *bm* noduri, spre deosebire de cele b^d care ar trebui memorate în cazul strategiei de tip breadth-first (atunci când scopul cel mai puţin adânc se află la adâncimea *d*).

Complexitatea de timp **a strategiei depth-first este de $O(b^m)$. Pentru probleme care au foarte multe soluţii, căutarea de tip depth-first s-ar putea să fie mai rapidă decât cea de tip breadth-first, deoarece sunt şanse mari să fie găsită o soluţie după ce se explorează numai o mică porţiune a întregului spaţiu de căutare. Strategia breadth-first trebuie să investigheze toate drumurile de lungime *d-1* înainte de a lua în consideraţie pe oricare dintre drumurile de lungime *d*. Depth-first este de complexitate $O(b^m)$ şi în cazul cel mai nefavorabil.**

Principalul *dezavantaj* al acestei strategii este acela că o căutare de tip depth-first va continua întotdeauna în jos, chiar dacă o soluţie mai puţin adâncă există. Această strategie

- **poate intra într-un ciclu infinit fără a returna vreodată o soluţie;**
- **poate găsi un drum reprezentând o soluţie, dar care este mai lung decât drumul corespunzător soluţiei optime; din aceste motive căutarea de tip depth-first nu este nici completă şi nici optimală.**

Prin urmare, această strategie trebuie evitată în cazul arborilor de căutare de adâncimi maxime foarte mari sau infinite.

Principalele *avantaje* ale acestui tip de căutare sunt:

- **consumul redus de memorie;**

- **posibilitatea găsirii unei soluţii fără a se explora o mare parte din spaţiul de căutare.**

Implementare in PROLOG

Problema găsirii unui drum-soluţie, Sol, de la un nod dat, N, până la un nod-scop, se rezolvă în felul următor:

- **dacă N este un nod-scop, atunci Sol=[N] sau**
- **dacă există un nod succesor, N1, al lui N, astfel încât să existe un drum, Sol1, de la N1 la un nod-scop, atunci Sol=[N|Sol1].**

Aceasta se traduce în Prolog astfel:

```
rezolva_d(N,[N]):-
        scop(N).
rezolva_d(N,[N|Sol1]):-
        s(N,N1),
        rezolva_d(N1,Sol1).
```

Interogarea Prologului se va face în felul următor:

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

Vom lua din nou în consideraţie spaţiul stărilor din Fig. 2.2, în care *a* este nodul de start, iar *f* şi *j* sunt noduri-scop. Ordinea în care strategia depth-first vizitează nodurile în acest spaţiu de stări este: a, *b, d, h, e, i, j*. Soluţia găsită este [a, b, e, j]. După efectuarea backtracking-ului este găsită şi cealaltă soluţie, [a, c, f].

Programul Prolog complet, corespunzător acestui exemplu:

```
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).

rezolva_d(N,[N]):-scop(N).

rezolva_d(N,[N|Sol1]):-
       s(N,N1),
       rezolva_d(N1,Sol1).
```

Interogarea Prologului se face astfel:

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

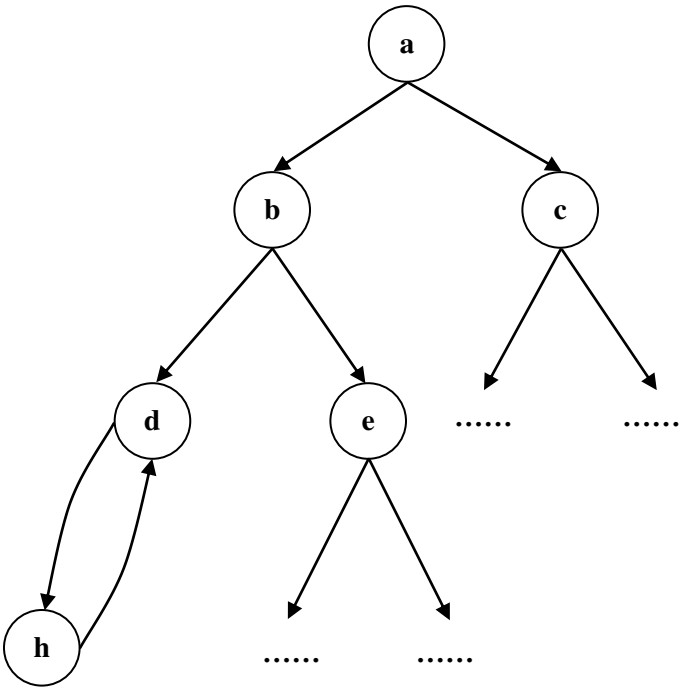
Răspunsul Prologului va fi:

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

<u>Obsevatie</u>: Forma drumurilor-soluţie este cea firească, de la nodul de start la nodul-scop (spre deosebire de cazul strategiei breadth-first, unde nodurile intervin în ordine inversă).

Există însă multe situaţii în care procedura `rezolva_d` poate să nu lucreze "bine", acest fapt depinzând în exclusivitate de spaţiul de stări. Pentru exemplificare, este suficient să adăugăm un arc de la nodul *h* la nodul *d* în spaţiul de stări reprezentat de Fig. 2.2. Corespunzător Fig. 2.4, căutarea se va efectua în felul următor: se pleacă din nodul *a*, apoi se coboară la nodul *h*, pe ramura cea mai din stânga a arborelui. În acest moment, spre deosebire de situaţia anterioară, *h* are un succesor, şi anume pe *d*. Prin urmare, de la *h*, execuţia nu va mai efectua un backtracking, ci se va îndrepta spre *d*. Apoi va fi găsit succesorul lui *d*, adică *h*, ş.a.m.d., rezultând un *ciclu infinit* între *d* şi *h*.

Fig. 2.4:


a
b
c
d
e
.....
.....
h
.....
.....

Pentru îmbunătăţirea programului, trebuie adăugat un mecanism de detectare a ciclurilor. Conform acestuia, orice nod care se află deja pe drumul de la nodul de start la nodul curent nu mai trebuie luat vreodată în consideraţie. Această cerinţă poate fi formulată ca o relaţie:

depthfirst(Drum, Nod, Soluţie).

Aici Nod este starea pornind de la care trebuie găsit un drum la o stare- scop; Drum este un drum, adică o listă de noduri, între nodul de start şi Nod; Soluţie este Drum, extins via Nod, la un nod-scop. Reprezentarea relaţiei este cea din Fig. 2.5:

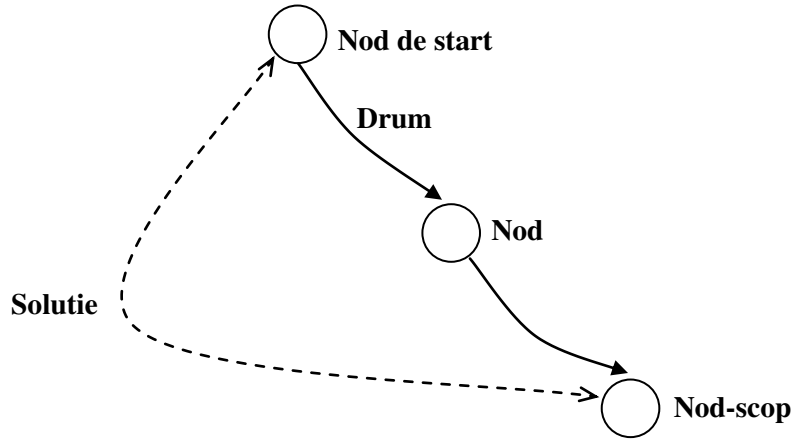


Fig.2.5

Argumentul Drum poate fi folosit cu două scopuri:

- **să împiedice algoritmul să ia în consideraţie acei succesori ai lui Nod care au fost deja întâlniţi, adică *să detecteze ciclurile*;**
- **să construiască un drum-soluţie numit Soluţie.**

Predicatul corespunzător,

```
depthfirst(Drum,Nod,Sol)
```

este adevărat dacă, extinzând calea Drum via Nod, se ajunge la un nod-scop.

În cele ce urmează, prezentăm implementarea în Prolog a strategiei depth-first cu detectare a ciclurilor:

```
rezolva1_d(N,Sol):-
    depthfirst([ ],N,Sol).
depthfirst(Drum,Nod,[Nod|Drum]):-
    scop(Nod).
depthfirst(Drum,Nod,Sol):
        -s(Nod,Nod1),
        \+(membru(Nod1,Drum)),
        depthfirst([Nod|Drum],Nod1,Sol).
```

Se observă că fragmentul de program anterior verifică dacă anumite noduri au fost luate deja în consideraţie până la momentul curent, punând condiţia de nonapartenenţă:

```
\+(membru(Nod1,Drum))
```

Drumurile-soluţie sunt date ca liste de noduri în ordine inversă, aşa cum se va vedea şi în exemplul care urmează.

Programul Prolog complet corespunzător exemplului din Fig. 2.4.

Programul implementează strategia de căutare depth-first cu detectare a ciclurilor, pentru găsirea celor două soluţii posibile:

```
scop(f).  % specificare noduri-scop
scop(j).

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

rezolva1_d(N,Sol):-
       depthfirst([ ],N,Sol).

depthfirst(Drum,Nod,[Nod|Drum]):-
```

```
        scop(Nod).
depthfirst(Drum,Nod,Sol):-
        s(Nod,Nod1),
        \+(membru(Nod1,Drum)),
        depthfirst([Nod|Drum],Nod1,Sol).

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

Interogarea Prologului se face astfel:

```
?- rezolva1_d(a,Sol).
```

Răspunsul Prologului va fi:

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

Un program de acest tip nu va lucra totuşi corect atunci când *spaţiul stărilor este infinit*. Într-un astfel de spaţiu algoritmul depth-first poate omite un nod-scop deplasându-se de-a lungul unei ramuri infinite a grafului. Programul poate atunci să exploreze în mod nedefinit această parte infinită a spaţiului, fără a se apropia vreodată de un scop. Pentru a evita astfel de *ramuri aciclice infinite*, procedura de căutare depth-first de bază poate fi, în continuare, rafinată, prin *limitarea adâncimii căutării*. În acest caz, procedura de căutare depth-first va avea următoarele argumente:

depthfirst1(Nod, Soluţie, Maxdepth),

unde Maxdepth este adâncimea maximă până la care se poate efectua căutarea.

Această constrângere poate fi programată prin micşorarea limitei de adâncime la fiecare apelare recursivă, fără a permite ca această limită să devină negativă. Tipul acesta de căutare se numeşte Depth-limited search ("căutare cu adâncime limitată"). Predicatul Prolog corespunzător,

```
depthfirst1(Nod,Sol,Max)
```

va fi adevărat dacă, pornind din nodul Nod, obţinem un drum-soluţie numit Sol, prin efectuarea unei căutări de tip depth-first până la adâncimea maximă notată Max.

<u>Exerciţiu:</u> scrierea programului Prolog care va explora cu succes un spaţiu aciclic infinit, prin stabilirea unei limite de adâncime.

Numărul de extinderi într-o căutare depth-limited, până la o adâncime *d*, cu factor de ramificare *b*, este:

$$1+b+b^2+\cdots+b^{d-1}+b^d$$

Neajunsul **în cazul acestui tip de căutare este acela că trebuie găsită dinainte o limită convenabilă a adâncimii căutarii. Dacă această limită este prea mică, căutarea va eşua. Dacă limita este mare, atunci căutarea va deveni prea scumpă. Pentru a evita aceste neajunsuri, putem executa căutarea de tip depth-limited *în mod iterativ*, variind limita pentru adâncime. Se începe cu o limită a adâncimii foarte mică şi se măreşte această limită în mod gradat, până când se găseşte o soluţie. Această tehnică de căutare se numeşte Iterative Deepening Search ("căutare în adâncime iterativă").**

Căutarea în adâncime iterativă

Căutarea în adâncime iterativă este o strategie care evită chestiunea stabilirii unei adâncimi optime la care trebuie căutată soluţia, prin testarea tuturor limitelor de adâncime posibile: mai întâi adâncimea 0, apoi 1, apoi 2, ş.a.m.d.. Acest tip de căutare combină beneficiile căutării breadth-first şi depth-first, după cum urmează:

- **este optimă şi completă ca şi căutarea breadth-first;**
- **consumă numai cantitatea mică de memorie necesară căutarii depth-first (cerinţa de memorie este liniară).**

Ordinea extinderii stărilor este similară cu cea de la căutarea de tip breadth-first, numai că anumite stări sunt extinse de mai multe ori. Această strategie de căutare garantează găsirea

nodului-scop de la adâncimea minimă, *dacă* un scop poate fi găsit. Deşi anumite noduri sunt extinse de mai multe ori, *numărul total de noduri extinse* nu este mult mai mare decât cel dintr-o căutare de tip breadth-first (vezi in cursul scris acest calcul).

- **Strategia de căutare în adâncime iterativă are tot complexitatea de timp $O(b^d)$, iar complexitatea sa de spatiu este $O(bd)$ (vezi cursul scris). În general, căutarea în adâncime iterativă este metoda de căutare preferată atunci când există un spaţiu al căutării foarte mare, iar adâncimea soluţiei nu este cunoscută.**

Implementare în Prolog

Pentru implementarea în Prolog a căutării în adâncime iterative vom folosi predicatul `cale` de forma

```
cale(Nod1,Nod2,Drum)
```

care este adevărat dacă Drum reprezintă o cale aciclică între nodurile Nod1 şi Nod2 în spaţiul stărilor. Această cale va fi reprezentată ca o listă de noduri date în ordine inversă. Corespunzător nodului de start dat, predicatul cale generează toate drumurile aciclice posibile de lungime care creşte cu câte o unitate. Drumurile sunt generate până când se generează o cale care se termină cu un nod-scop.

Implementarea în Prolog a căutării în adâncime iterative este următoarea:

```
cale(Nod,Nod,[Nod]).
cale(PrimNod,UltimNod,[UltimNod|Drum]):-
```

```
    cale(PrimNod,PenultimNod,Drum),
    s(PenultimNod,UltimNod),
    \+(membru(UltimNod,Drum)).
depth_first_iterative_deepening(Nod,Sol):-
    cale(Nod,NodScop,Sol),
    scop(NodScop), !.
```

Programul Prolog complet corespunzător aceluiaşi exemplu dat de Fig. 2.2:

```
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).

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

cale(Nod,Nod,[Nod]).
cale(PrimNod,UltimNod,[UltimNod|Drum]):-
    cale(PrimNod,PenultimNod,Drum),
    s(PenultimNod,UltimNod),
    \+(membru(UltimNod,Drum)).

depth_first_iterative_deepening(Nod,Sol):-
    cale(Nod,NodScop,Sol),
    scop(NodScop),!.
```

Interogarea Prologului se face astfel:

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

Răspunsul Prologului va fi:

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

Programul găseşte soluţia cel mai puţin adâncă, sub forma unui drum scris în ordine inversă, după care opreşte căutarea. El va funcţiona la fel de bine şi într-un spaţiu al stărilor conţinând cicluri, datorită mecanismului de verificare `\+(membru(UltimNod,Drum))`, care evită luarea în consideraţie a nodurilor deja vizitate.

- ***Principalul avantaj*** **al acestei metode este acela că ea necesită puţină memorie. La orice moment al execuţiei, necesităţile de spaţiu se reduc la *un singur drum*, acela dintre nodul de început al căutării şi nodul curent.**

- ***Dezavantajul*** **metodei este acela că, la fiecare iteraţie, drumurile calculate anterior sunt recalculate, fiind extinse până la o nouă limită de adâncime. Timpul calculator nu este însă foarte afectat, deoarece nu se extind cu mult mai multe noduri.**