## 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.