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*.

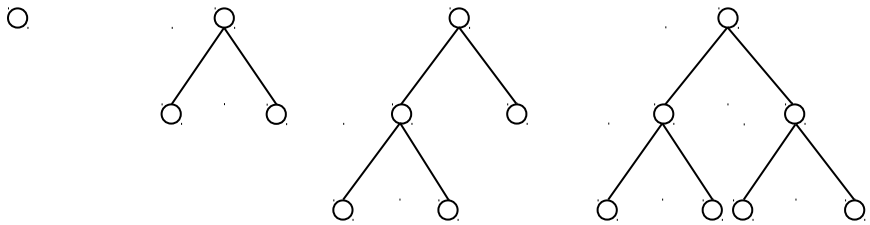

Fig. 2.1

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:

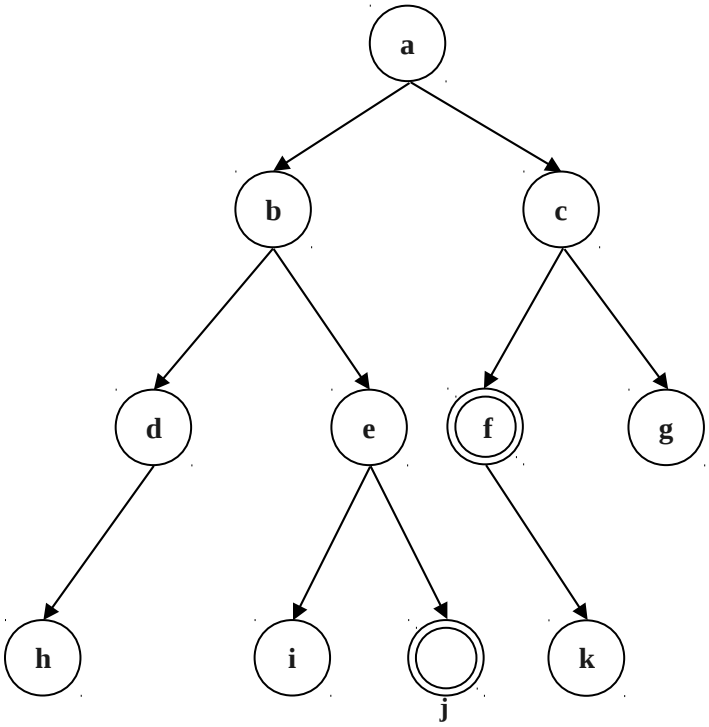

Fig. 2.2

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 iniţial Start la o stare-scop, drum obţinut folosind căutarea de tip breadth-first.

Predicatul `breadthfirst(Drumuri,Sol)` este adevărat dacă un drum din mulţimea 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 mulţimii de noduri Drum obţinem mulţimea numită DrumuriNoi, el generând mulţimea 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 complet 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*.

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 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.