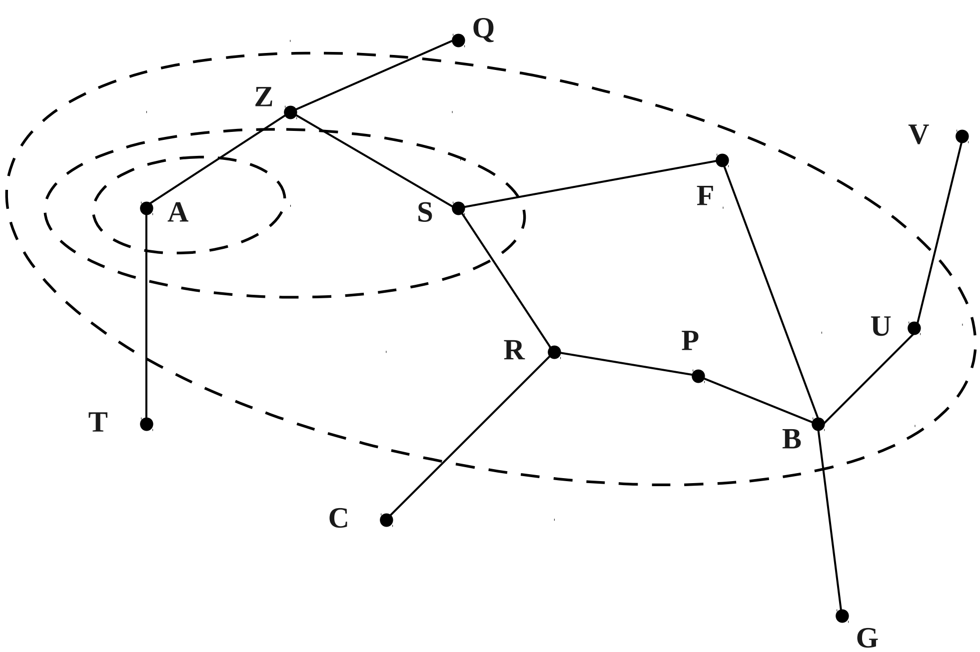
Iterative Deepening A* (IDA*)

Algoritmul IDA* se referă la o căutare iterativă în adâncime de tip A* şi este o extensie logică a lui Iterative Deepening Search care foloseşte, în plus, informaţia euristică.

În cadrul acestui algoritm fiecare iteraţie reprezintă o căutare de tip depth-first, iar căutarea de tip depth-first este modificată astfel încât ea să folosească *o limită a costului* şi nu o limită a adâncimii.

Faptul că în cadrul algoritmului A* *f* nu descreşte niciodată de-a lungul oricărui drum care pleacă din rădăcină ne permite să trasăm, din punct de vedere conceptual, *contururi* în spaţiul stărilor. Astfel, în interiorul unui contur, toate nodurile au valoarea *f(n)* mai mică sau egală cu o aceeaşi valoare. În cazul algoritmului IDA* fiecare iteraţie extinde toate nodurile din interiorul conturului determinat de costul *f* curent, după care se trece la conturul următor. De îndată ce căutarea în interiorul unui contur dat a fost completată, este declanşată o nouă iteraţie, folosind un nou cost *f*, corespunzător următorului contur. Fig. 2.10 prezintă căutări iterative în interiorul câte unui contur.



Algoritmul IDA* este complet şi optim cu aceleaşi amendamente ca şi A*. Deoarece este de tip depth-first *nu necesită decât un spaţiu proporţional cu cel mai lung drum pe care îl explorează.*

Dacă δ este cel mai mic cost de operator, iar *f** este costul soluţiei optime, atunci, în cazul cel mai nefavorabil, IDA* va necesita spaţiu pentru memorarea a $\frac{bf^*}{\delta}$ noduri, unde *b* este acelaşi factor de ramificare.

Complexitatea de timp a algoritmului depinde în mare măsură de numărul valorilor diferite pe care le poate lua funcţia euristică.

Implementarea în Prolog a căutarii de tip best-first

Vom imagina căutarea de tip best-first funcţionând în felul următor: căutarea constă dintr-un număr de subprocese "concurente", fiecare explorând alternativa sa, adică propriul subarbore. Subarborii au subarbori, care vor fi la rândul lor exploraţi de subprocese ale subproceselor, ş.a.m.d.. Dintre toate aceste subprocese doar unul este activ la un moment dat şi anume cel care se ocupă de alternativa cea mai promiţătoare (adică alternativa corespunzătoare celei mai mici $\hat{f}$ - valori). Celelalte procese aşteaptă până când $\hat{f}$ - valorile se schimbă astfel încât o altă alternativă devine mai promiţătoare, caz în care procesul corespunzător acesteia devine activ. Acest mecanism de activare-dezactivare poate fi privit după cum urmează: procesului corespunzător alternativei curente de prioritate maximă i se alocă un buget şi, atâta vreme cât acest buget nu este epuizat, procesul este activ. Pe durata activităţii sale, procesul îşi expandează propriul subarbore, iar în cazul atingerii unei stări-scop este anunţată găsirea unei soluţii. Bugetul acestei funcţionări este determinat de $\hat{f}$ -valoarea corespunzătoare celei mai apropiate alternative concurente.

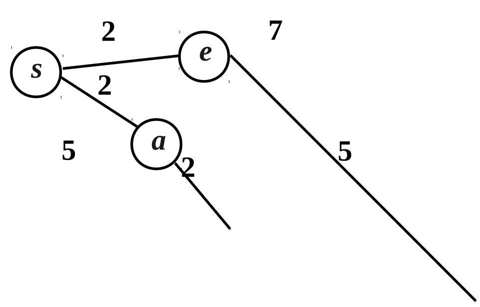
Exemplu

Considerăm oraşele *s, a, b, c, d, e, f, g, t* unite printr-o reţea de drumuri ca în Fig. 2.11. Aici fiecare drum direct între două oraşe este etichetat cu lungimea sa; numărul din căsuţa alăturată unui oraş reprezintă distanţa în linie dreaptă între oraşul respectiv şi oraşul *t*. Ne punem problema determinării celui mai scurt drum între oraşul *s* şi oraşul *t* utilizând strategia best-first. Definim în acest scop funcţia $\hat{h}$ bazându-ne pe distanţa în linie dreaptă între două oraşe. Astfel, pentru un oraş X, definim

$$\hat{f}(X) = \hat{g}(X) + \hat{h}(X) = \hat{g}(X) + dist(X,t)$$

unde $dist(X,t)$ reprezintă distanţa în linie dreaptă între X şi *t*.

În acest exemplu, căutarea de tip best-first este efectuată prin intermediul a două procese, P_1 si P_2, ce explorează fiecare câte una din cele două căi alternative. Calea de la *s* la *t* via nodul *a* corespunde procesului P_1, iar calea prin nodul *e* corespunde procesului P_2.



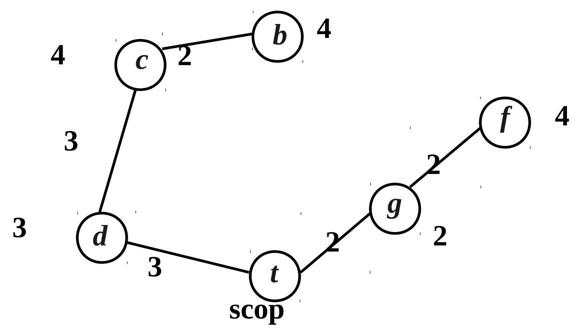


Fig. 2.11

În stadiile iniţiale, procesul P_1 este mai activ, deoarece $\hat{f}$ - valorile de-a lungul căii corespunzătoare lui sunt mai mici decât $\hat{f}$ - valorile de-a lungul celeilalte căi. Atunci când P_1 explorează *c*, iar procesul P_2 este încă la *e*, $\hat{f}(c) = \hat{g}(c) + \hat{h}(c) = 6 + 4 = 10$, $\hat{f}(e) = \hat{g}(e) + \hat{h}(e) = 2 + 7 = 9$ şi deci $\hat{f}(e) < \hat{f}(c)$. În acest moment, situaţia se schimbă: procesul P_2 devine activ, iar procesul P_1 intră în aşteptare. În continuare, $\hat{f}(c) = 10$, $\hat{f}(f) = 11$, $\hat{f}(c) < \hat{f}(f)$ şi deci P_1 devine activ şi P_2 intră în aşteptare. Pentru că $\hat{f}(d) = 12 > 11$, procesul P_1 va reintra în aşteptare, iar procesul P_2 va rămâne activ până când se va atinge starea scop *t*.

Căutarea schiţată mai sus porneşte din nodul iniţial şi este continuată cu generarea unor noduri noi, conform relaţiei de succesiune. În timpul acestui proces, este generat un arbore de căutare, a cărui rădăcină este nodul de start. Acest arbore este expandat în direcţia cea mai promiţătoare conform $\hat{f}$ - valorilor, până la găsirea unei soluţii.

În vederea implementării în Prolog, vom extinde definiţia lui $\hat{f}$, de la noduri în spaţiul stărilor, la arbori, astfel:

- pentru un arbore cu un singur nod N, avem egalitate între $\hat{f}$ - valoarea sa şi $\hat{f}(N)$;
- pentru un arbore T cu rădăcina N şi subarborii S_1, S_2,... definim

$$\hat{f}(T) = \min_i \hat{f}(S_i)$$

În implementarea care urmează, vom reprezenta arborele de căutare prin termeni Prolog de două forme, şi anume:

- $l(N, F/G)$ corespunde unui arbore cu un singur nod N; N este nod în spaţiul stărilor, G este $\hat{g}(N)$ (considerăm $\hat{g}(N)$ ca fiind costul drumului între nodul de start şi nodul N), $F = G + \hat{h}(N)$.
- $t(N, F/G, Subs)$ corespunde unui arbore cu subarbori nevizi; N este rădăcina sa, Subs este lista

subarborilor săi, G este $\breve{g}(N)$, F este $\hat{f}$ - valoarea actualizată a lui N, adică este $\hat{f}$ - valoarea celui mai promiţător succesor al lui N; de asemenea, Subs este ordonată crescător conform $\hat{f}$ - valorilor subarborilor constituenţi.

Recalcularea $\hat{f}$ - valorilor este necesară pentru a permite programului să recunoască cel mai promiţător subarbore, la fiecare nivel al arborelui de căutare (adică arborele care conţine cel mai promiţător nod terminal).

În exemplul anterior, în momentul în care nodul *s* tocmai a fost extins, arborele de căutare va avea 3 noduri: rădăcina *s* şi copiii *a* si *e*. Acest arbore va fi reprezentat, în program, prin intermediul termenului Prolog $t(s,7/0,[l(a,7/2),l(e,9/2)])$. Observăm că $\hat{f}$ - valoarea lui *s* este 7, adică $\hat{f}$ - valoarea celui mai promiţător subarbore al său. În continuare va fi expandat subarborele de rădăcină *a*. Cel mai apropiat competitor al lui *a* este *e*; cum $f(e)=9$, rezultă că subarborele de rădăcină *a* se poate expanda atâta timp cât $\hat{f}$ - valoarea sa nu va depăşi 9. Prin urmare, sunt generate *b* si *c*. Deoarece $f(c)=10$, rezultă că limita de expandare a fost depăşită şi alternativa *a* nu mai poate "creşte". În acest moment, termenul Prolog corespunzător subarborelui de căutare este următorul:

$$t(s,9/0,[l(e,9/2),t(a,10/2,[t(b,10/4,[l(c,10/6)])])])$$

În implementarea care urmează, predicatul cheie va fi predicatul *expandeaza*:

```
expandeaza(Drum,Arb,Limita,Arb1,Rez,Solutie)
```

Argumentele sale au următoarele semnificaţii:

- <u>Drum</u> reprezintă calea între nodul de start al căutării şi <u>Arb</u>
- <u>Arb</u> este arborele (subarborele) curent de căutare
- <u>Limita</u> este $\hat{f}$ - limita pentru expandarea lui <u>Arb</u>
- <u>Rez</u> este un indicator a cărui valoare poate fi “da”, “nu”, “imposibil”
- <u>Solutie</u> este o cale de la nodul de start ("prin <u>Arb1</u>") către un nod-scop (în limita <u>Limita</u>), dacă un astfel de nod-scop există.

<u>Drum</u>, <u>Arb</u> şi <u>Limita</u> sunt parametrii de intrare pentru *expandeaza* (în sensul că ei sunt deja instanţiati atunci când *expandeaza* este folosit). Prin utilizarea predicatului *expandeaza* se pot obţine trei feluri de rezultate, rezultate indicate prin valoarea argumentului <u>Rez</u>, după cum urmează:

- Rez=da, caz în care <u>Solutie</u> va unifica cu o cale soluţie găsită expandând <u>Arb</u> în limita <u>Limita</u> (adică fără ca $\hat{f}$ - valoarea să depăşească limita <u>Limita</u>); <u>Arb1</u> va rămâne neinstanţiat;
- Rez=nu, caz în care <u>Arb1</u> va fi, de fapt, <u>Arb</u> expandat până când $\hat{f}$ - valoarea sa a depăşit

Limita; Solutie va rămâne neinstanţiat;

- Rez=imposibil, caz în care argumentele Arb1 şi Solutie vor rămâne neinstanţiate; acest ultim caz indică faptul că explorarea lui Arb este o alternativă “moartă”, deci nu trebuie să i se mai dea o şansă pentru reexplorare în viitor; acest caz apare atunci când $\hat{f}$ - valoarea lui Arb este mai mică sau egală decât Limita, dar arborele nu mai poate fi expandat, fie pentru că nici o frunză a sa nu mai are succesori, fie pentru că un astfel de succesor ar crea un ciclu.

Urmează o implementare a unei variante a metodei best-first în SICStus Prolog, implementare care foloseşte considerentele anterioare.

Strategia best-first

```
%Predicatul bestfirst(Nod_initial,Solutie) este adevarat daca
%Solutie este un drum (obtinut folosind strategia best-first) de %la
nodul Nod_initial la o stare-scop.

bestfirst(Nod_initial,Solutie):-
          expandeaza([],l(Nod_initial,0/0),9999999,_,
          da,Solutie).

expandeaza(Drum,l(N,_),_,_, da,[N|Drum]):-scop(N).

%Caz 1: daca N este nod-scop, atunci construim o cale-solutie.

expandeaza(Drum,l(N,F/G),Limita,Arb1,Rez,Sol):-
  F=<Limita,
  (bagof(M/C,(s(N,M,C), \+ (membru(M,Drum))),Succ),!,
  listasucc(G,Succ,As),
  cea_mai_buna_f(As,F1),
  expandeaza(Drum,t(N,F1/G,As),Limita,Arb1, Rez,Sol);
  Rez=imposibil).
```

%Caz 2: Daca N este nod-frunza a carui $\hat{f}$-valoare este mai mica %decat Limita,atunci ii generez succesorii si ii expandez in %limita Limita.

```
expandeaza(Drum,t(N,F/G,[A|As]),Limita,Arb1,Rez,
Sol):-
     F=<Limita,
     cea_mai_buna_f(As,BF),
```

```
     min(Limita,BF,Limita1),
     expandeaza([N|Drum],A,Limita1,A1,Rez1,Sol),
     continua(Drum,t(N,F/G,[A1|As]),Limita,Arb1,
      Rez1,Rez,Sol).
```

%Caz 3: Daca arborele de radacina N are subarbori nevizi si $\hat{f}$-%valoarea este mai mica decat Limita, atunci expandam cel mai %"promitator" subarbore al sau; in functie de rezultatul obtinut, %Rez, vom decide cum anume vom continua cautarea prin intermediul %procedurii (predicatului) continua.

```
expandeaza(_,t(_,_,[]),_,_,imposibil,_):-!.
```

%Caz 4: pe aceasta varianta nu o sa obtinem niciodata o solutie.

```
expandeaza(_,Arb,Limita,Arb,nu,_):-
           f(Arb,F),
           F>Limita.
```

%Caz 5: In cazul unor $\hat{f}$-valori mai mari decat Bound, arborele nu %mai poate fi extins.

```
continua(_,_,_,_,da,da,Sol).
continua(P,t(N,F/G,[A1|As]),Limita,Arb1,nu,Rez,Sol):-
     insereaza(A1,As,NAs),
     cea_mai_buna_f(NAs,F1),
     expandeaza(P,t(N,F1/G,NAs),Limita,Arb1,Rez,Sol).
continua(P,t(N,F/G,[_|As]),Limita,Arb1,imposibil,Rez,Sol):-
cea_mai_buna_f(As,F1),
      expandeaza(P,t(N,F1/G,As),Limita,Arb1,Rez,Sol).

listasucc(_,[],[]).
listasucc(G0,[N/C|NCs],Ts):-
          G is G0+C,
          h(N,H),
          F is G+H,
          listasucc(G0,NCs,Ts1),
          insereaza(l(N,F/G),Ts1,Ts).
```

%Predicatul insereaza(A,As,As1) este utilizat pentru inserarea %unui arbore A intr-o lista de arbori As, mentinand ordinea %impusa de $\hat{f}$-valorile lor.

```
insereaza(A,As,[A|As]):-
          f(A,F),
          cea_mai_buna_f(As,F1),
          F=<F1,!.
insereaza(A,[A1|As],[A1|As1]):-insereaza(A,As,As1).

min(X,Y,X):-X=<Y,!.
min(_,Y,Y).

f(l(_,F/_),F).     % f-val unei frunze
f(t(_,F/_,_),F).   % f-val unui arbore
```

%Predicatul cea_mai_buna_f(As,F) este utilizat pentru a determina %cea mai buna $\hat{f}$-valoare a unui arbore din lista de arbori As, %daca aceasta lista este nevida; lista As este ordonata dupa $\hat{f}$-%valorile subarborilor constituenti.

```
cea_mai_buna_f([A|_],F):-f(A,F).
cea_mai_buna_f([],999999).
```

%In cazul unei liste de arbori vide, $\hat{f}$-valoarea determinata este %foarte mare.

Pentru aplicarea programului anterior la o problemă particulară, trebuie adăugate anumite relaţii specifice problemei. Aceste relaţii definesc de fapt problema particulară („regulile jocului”) şi, de asemenea, adaugă o anumită informaţie euristică despre cum anume s-ar putea rezolva aceasta.

Predicatele specifice problemei sunt:

- `s(Nod,Nod1,Cost)`

  % acest predicat este adevărat dacă există un arc între Nod1 şi Nod în spaţiul stărilor

- `scop(Nod)`

  % acest predicat este adevărat dacă Nod este stare-scop în spaţiul stărilor

- `h(Nod,H)`

  % H este o estimaţie euristică a costului celui mai ieftin drum între Nod şi o stare-scop.

<u>Exemple:</u> Problema misionarilor şi canibalilor, Problema “8-puzzle”