

## **Căutarea informată**

***Căutarea informată*** se mai numește și ***căutare euristică***. Euristica este o metodă de studiu și de cercetare bazată pe descoperirea de fapte noi. În acest tip de căutare vom folosi informația despre spațiul de stări. Se folosesc cunoștințe specifice problemei și se rezolvă probleme de optim.

## Căutarea de tip best-first

Tipul de căutare pe care îl discutăm aici se aseamănă cu tehnica breadth-first, numai că procesul nu se desfășoară în mod uniform plecând de la nodul inițial. El înaintează în mod preferențial de-a lungul unor noduri pe care informația euristică, specifică problemei, le indică ca aflându-se pe drumul cel mai bun către un scop. Un asemenea proces de căutare se numește *căutare euristică* sau *căutare de tip best-first*.

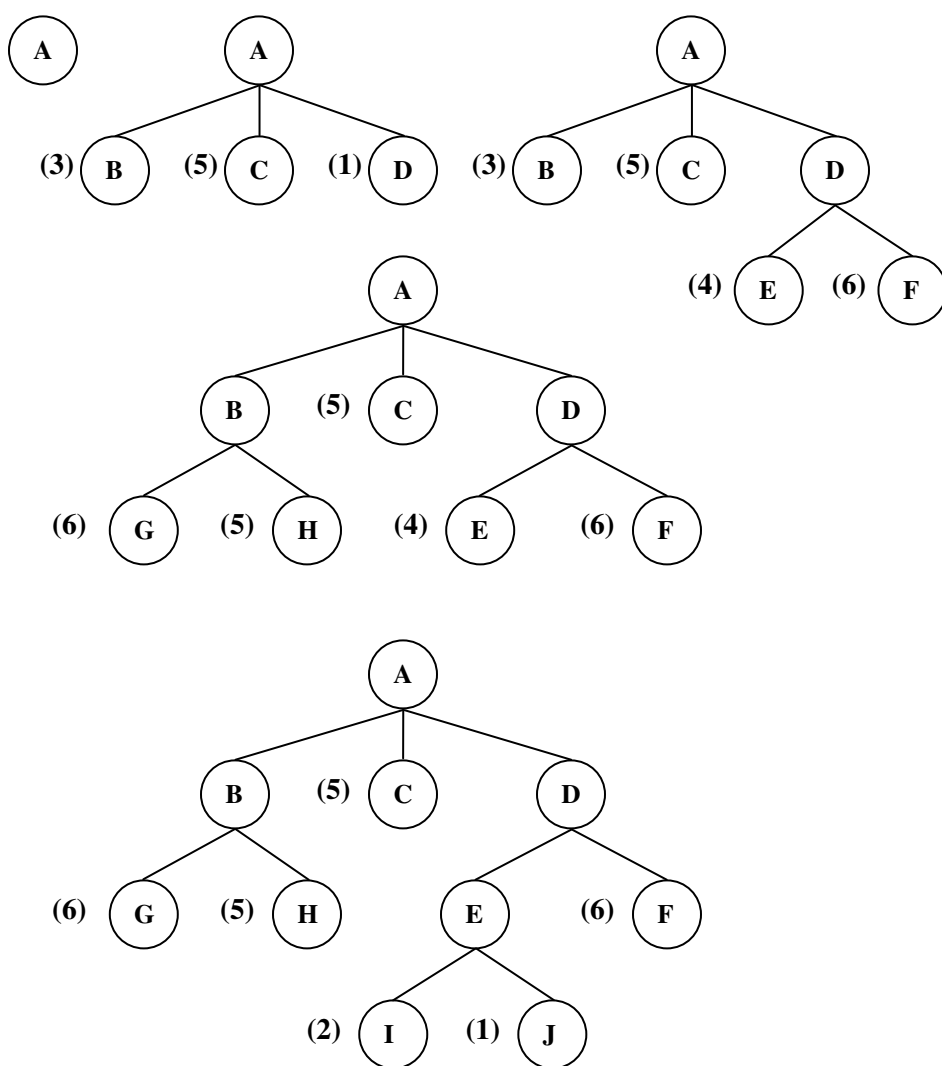
Principiile pe care se bazează căutarea de tip best-first sunt următoarele:

1. Se presupune existența unei funcții euristice de evaluare,  $\hat{f}$ , cu rolul de a ne ajuta să decidem care nod ar trebui extins la pasul următor. Se va adopta *convenția* că valori mici ale lui  $\hat{f}$  indică nodurile cele mai bune. Această funcție se bazează pe informație specifică domeniului pentru care s-a formulat problema. Este o funcție de descriere a stărilor, cu valori reale.

**2. Se extinde nodul cu cea mai mică valoare a lui  $\hat{f}(n)$ . În cele ce urmează, se va presupune că extinderea unui nod va produce toți succesorii acelui nod.**

**2. Procesul se încheie atunci când următorul nod care ar trebui extins este un nod-scop.**

**Fig. 2.7 ilustrează începutul unei căutări de tip best-first:**



**Fig. 2.7**

**În Fig. 2.7 există inițial un singur nod, A, astfel încât acesta va fi extins.**

**În Fig. 2.7 există inițial un singur nod, A, astfel încât acesta va fi extins. Extinderea lui generează trei noduri noi. Funcția euristică este aplicată fiecăruia dintre acestea. Întrucât nodul D este cel mai promițător, el este următorul nod extins. Extinderea lui va produce două noduri succesori, E și F, cărora li se aplică funcția euristică. În acest moment, un alt drum, și anume acela care trece prin nodul B, pare mai promițător, astfel încât el este urmat, generându-se nodurile G și H. În urma evaluării, aceste noduri par însă mai puțin promițătoare decât un alt drum, astfel încât este ales, în continuare, drumul prin D la E. E este apoi extins producând nodurile I și J. La pasul următor va fi extins nodul J, întrucât acesta este cel mai promițător. Procesul continuă până când este găsită o soluție.**

Pentru a nu fi induși în eroare de o euristică extrem de optimistă, este necesar să înclinăm căutarea în favoarea posibilității de a ne întoarce înapoi, cu scopul de a explora drumuri găsite mai devreme. De aceea, vom adăuga lui  $\hat{f}$  *un factor de adâncime*:

$$\hat{f}(n) = \hat{g}(n) + \hat{h}(n),$$

unde  $\hat{g}(n)$  este o estimatie a adâncimii lui  $n$  în graf, adică reprezintă lungimea celui mai scurt drum de la nodul de start la  $n$ , iar  $\hat{h}(n)$  este o evaluare euristică a nodului  $n$ .

Pentru a studia, în cele ce urmează, *aspectele formale ale căutării de tip best-first*, vom începe prin a prezenta un algoritm de căutare generală bazat pe grafuri. Algoritmul include versiuni ale căutării de tip best-first ca reprezentând cazuri particulare.

### Algoritm de căutare general bazat pe grafuri

Acest algoritm, pe care îl vom numi GraphSearch, este unul general, care permite orice tip de ordonare preferată de utilizator - euristică sau neinformată. Iată o *primă variantă* a definiției sale:

#### GraphSearch

1. Creează un arbore de căutare,  $T_r$ , care constă numai din nodul de start  $n_0$ . Plasează pe  $n_0$  într-o listă ordonată numită OPEN.
2. Creează o listă numită CLOSED, care inițial este vidă.
3. Dacă lista OPEN este vidă, EXIT cu eșec.
4. Selectează primul nod din OPEN, înlătură-l din lista OPEN și include-l în lista CLOSED. Numește acest nod  $n$ .

5. Dacă  $n$  este un nod scop, algoritmul se încheie cu succes, iar soluția este cea obținută prin urmarea în sens invers a unui drum de-a lungul arcelor din arborele  $T_r$ , de la  $n$  la  $n_0$ . (Arcele sunt create la pasul 6).
6. Extinde nodul  $n$ , generând o mulțime,  $M$ , de succesori. Include  $M$  ca succesori ai lui  $n$  în  $T_r$ , prin crearea de arce de la  $n$  la fiecare membru al mulțimii  $M$ .
7. Reordonează lista OPEN, fie în concordanță cu un plan arbitrar, fie în mod euristic.
8. Mergi la pasul 3.

□

- **Observație:** Acest algoritm poate fi folosit pentru a efectua căutări de tip best-first, breadth-first sau depth-first. În cazul algoritmului *breadth-first* noile noduri sunt puse la sfârșitul listei OPEN (organizată ca o coadă), iar nodurile nu sunt reordonate. În cazul căutării de tip *depth-first* noile noduri sunt plasate la începutul listei OPEN (organizată ca o stivă). În cazul căutării de tip *best-*



*first*, numită și căutare euristică, lista OPEN este reordonată în funcție de meritele euristice ale nodurilor.

## Algoritmul A\*

Vom particulariza algoritmul GraphSearch la un algoritm de căutare best-first care reordonează, la pasul 7, nodurile listei OPEN în funcție de *valorile crescătoare ale funcției  $\hat{f}$* . Această versiune a algoritmului GraphSearch se va numi Algoritmul A\*.

Pentru a specifica familia funcțiilor  $\hat{f}$  care vor fi folosite, introducem următoarele notații:

- $h(n)$  = costul *efectiv* al drumului de cost minim dintre nodul  $n$  și un nod-scop, luând în considerație toate nodurile-scop posibile și toate drumurile posibile de la  $n$  la ele;
- $g(n)$  = costul unui drum de cost minim de la nodul de start  $n_0$  la nodul  $n$ .

Atunci,  $f(n) = g(n) + h(n)$  este costul unui drum de cost minim de la  $n_0$  la un nod-scop, drum ales dintre toate drumurile care trebuie să treacă prin nodul  $n$ .

- **Observație:**  $f(n_0) = h(n_0)$  reprezintă costul unui drum de cost minim nerestricționat, de la nodul  $n_0$  la un nod-scop.

Pentru fiecare nod  $n$ , fie  $\hat{h}(n)$ , numit *factor euristic*, o estimatie a lui  $h(n)$  și fie  $\hat{g}(n)$ , numit *factor de adâncime*, costul drumului de cost minim până la  $n$  găsit de A\* până la pasul curent. Algoritmul A\* va folosi funcția  $\hat{f} = \hat{g} + \hat{h}$ .

În definirea Algoritmului A\* de până acum nu s-a ținut cont de următoarea problemă: ce se întâmplă dacă graful implicit în care se efectuează căutarea nu este un arbore? Cu alte cuvinte, există mai mult decât o unică secvență de acțiuni care pot conduce la aceeași stare a lumii plecând din starea inițială. (Există situații în care fiecare dintre succesorii nodului  $n$  îl are pe  $n$  ca succesori i.e. acțiunile sunt reversibile). Pentru a rezolva astfel de cazuri, pasul 6 al algoritmului GraphSearch trebuie înlocuit cu următorul pas 6' :

6'. Extinde nodul  $n$ , generând o mulțime,  $M$ , de succesori care nu sunt deja *părinți* ai lui  $n$  în  $T_r$ . Instalează  $M$  ca succesori ai lui  $n$  în  $T_r$  prin crearea de arce de la  $n$  la fiecare membru al lui  $M$ .

Pentru a rezolva problema ciclurilor mai lungi, se înlocuiește pasul 6 prin următorul pas 6'':

6''. Extinde nodul  $n$ , generând o mulțime,  $M$ , de succesori care nu sunt deja *strămoși* ai lui  $n$  în  $T_r$ . Instalează  $M$  ca

succesori ai lui  $n$  în  $T_r$  prin crearea de arce de la  $n$  la fiecare membru al lui  $M$ .

**Observație:** Pentru a verifica existența acestor cicluri mai lungi, trebuie văzut dacă structura de date care etichetează fiecare succesor al nodului  $n$  este egală cu structura de date care etichetează pe oricare dintre strămoșii nodului  $n$ . Pentru structuri de date complexe, acest pas poate mări complexitatea algoritmului. Pasul 6 modificat în pasul 6” face însă ca algoritmul să nu se mai învârtă în cerc, în căutarea unui drum la scop.

Există încă posibilitatea de a vizita aceeași stare a lumii via drumuri diferite. O modalitate de a trata această problemă este ignorarea ei. Cu alte cuvinte, algoritmul nu verifică dacă un nod din mulțimea  $M$  se află deja în listele OPEN sau CLOSED. Algoritmul uită deci posibilitatea de a ajunge în aceleași noduri urmând drumuri diferite. Acest “același nod” s-ar putea repeta în  $T_r$  de atâtea ori de câte ori algoritmul descoperă drumuri diferite care duc la el. Dacă două noduri din  $T_r$  sunt etichetate cu aceeași structură de date, vor avea sub ele subarbori identici. Prin urmare, algoritmul va duplica anumite eforturi de căutare.

Pentru a preveni duplicarea efortului de căutare atunci când nu s-au impus condiții suplimentare asupra lui  $\hat{f}$ , sunt necesare niște modificări în algoritmul  $A^*$ , și anume: deoarece căutarea poate ajunge la același nod de-a lungul unor drumuri diferite, algoritmul  $A^*$  generează un *graf de căutare*, notat cu  $G$ .  $G$  este structura de noduri și de arce generată de  $A^*$  pe măsură ce algoritmul extinde nodul inițial, succesorii lui ș.a.m.d..  $A^*$  menține și un arbore de căutare,  $T_r$ .

$T_r$ , un subgraf al lui  $G$ , este arborele cu cele mai bune drumuri (de cost minim) produse până la pasul curent, drumuri până la toate nodurile din graful de căutare. Prin urmare, unele drumuri pot fi în graful de căutare, dar nu și în arborele de căutare. Graful de căutare este menținut deoarece căutări ulterioare pot găsi drumuri mai scurte, care folosesc anumite arce din graful de căutare anterior ce nu se aflau și în arborele de căutare anterior.



**Dăm, în continuare, versiunea algoritmului  $A^*$  care menține graful de căutare. În practică, această versiune este folosită mai rar deoarece, de obicei, se pot impune condiții asupra lui  $\hat{f}$  care garantează faptul că, atunci când algoritmul  $A^*$  extinde un nod, el a găsit deja drumul de cost minim până la acel nod.**

## **Algoritmul A\***

1. Creează un graf de căutare  $G$ , constând numai din nodul inițial  $n_0$ . Plasează  $n_0$  într-o listă numită OPEN.

2. Creează o listă numită CLOSED, care inițial este vidă.

3. Dacă lista OPEN este vidă, EXIT cu eșec.

4. Selectează primul nod din lista OPEN, înlătură-l din OPEN și plasează-l în lista CLOSED. Numește acest nod  $n$ .

5. Dacă  $n$  este un nod scop, oprește execuția cu succes. Returnează soluția obținută urmând un drum de-a lungul pointerilor de la  $n$  la  $n_0$  în  $G$ . (Pointerii definesc un arbore de căutare și sunt stabiliți la pasul 7).

6. Extinde nodul  $n$ , generând o mulțime,  $M$ , de succesori ai lui care nu sunt deja strămoși ai lui  $n$  în  $G$ . Instalează acești membri ai lui  $M$  ca succesori ai lui  $n$  în  $G$ .

7. Stabilește un pointer către  $n$  de la fiecare dintre membrii lui  $M$  care nu se găseau deja în  $G$  (adică nu se aflau deja nici în OPEN, nici în CLOSED). Adaugă acești membri ai lui  $M$  listei OPEN. Pentru fiecare membru,  $m$ , al lui  $M$ , care se afla deja în OPEN sau în CLOSED, redirecționează pointerul său către  $n$ , dacă cel mai bun drum la  $m$  găsit până în acel moment trece prin  $n$ . Pentru fiecare membru al lui  $M$  care se

află deja în lista CLOSED, redirectionează pointerii fiecăruia dintre descendenții săi din G astfel încât aceștia să țintească înapoi de-a lungul celor mai bune drumuri până la acești descendenți, găsite până în acel moment.

8. Reordonează lista OPEN în ordinea valorilor crescătoare ale funcției  $\hat{f}$ . (Eventuale legături între valori minimale ale lui  $\hat{f}$  sunt rezolvate în favoarea nodului din arborele de căutare aflat la cea mai mare adâncime).

9. Mergi la pasul 3.

□

- **Observație:** La pasul 7 sunt redirectionați pointeri de la un nod dacă procesul de căutare descoperă un drum la acel nod care are costul mai mic decât acela indicat de pointerii existenți. Redirectionarea pointerilor descendenților nodurilor care deja se află în lista CLOSED economisește efortul de căutare, dar poate duce la o cantitate exponențială de calcule. De aceea, această parte a pasului 7 de obicei nu este implementată. Unii dintre acești pointeri vor fi până la urmă redirectionați oricum, pe măsură ce căutarea progresează.

## Admisibilitatea Algoritmului A\*

Există anumite condiții asupra grafurilor și a lui  $\hat{h}$  care garantează că algoritmul A\*, aplicat acestor grafuri, găsește întotdeauna drumuri de cost minim.

Condițiile asupra *grafurilor* sunt:

1. Orice nod al grafului, dacă admite succesori, are un număr finit de succesori.

2. Toate arcele din graf au costuri mai mari decât o cantitate pozitivă,  $\epsilon$ .

Condiția asupra lui  $\hat{h}$  este:

3. Pentru toate nodurile  $n$  din graful de căutare,  $\hat{h}(n) \leq h(n)$ . Cu alte cuvinte,  $\hat{h}$  nu supraestimează niciodată valoarea efectivă  $h$ . O asemenea funcție  $\hat{h}$  este uneori numită un *estimator optimist*.

▪ **Observații:**

1. Este relativ ușor să se găsească, în probleme, o funcție  $\hat{h}$  care satisface această *condiție a limitei de jos*. De exemplu, în probleme de găsim a drumurilor în cadrul unor grafuri ale căror noduri sunt orașe, distanța de tip linie dreaptă de la un oraș  $n$  la un oraș-scop constituie o limită inferioară asupra distanței reprezentând un drum optim de la nodul  $n$  la nodul-scop.
2. Cu cele trei condiții formulate anterior, algoritmul  $A^*$  garantează găsirea unui drum optim la un scop, în cazul în care există un drum la scop.

În cele ce urmează, formulăm acest rezultat sub forma unei teoreme:

### **Teorema 2.1**

**Atunci când sunt îndeplinite condițiile asupra grafurilor și asupra lui  $\hat{h}$  enunțate anterior și cu condiția să existe un drum de cost finit de la nodul inițial,  $n_0$ , la un nod-scop, algoritmul A\* garantează găsirea unui drum de cost minim la un scop.**

### **Definiția 2.1**

Orice algoritm care garantează găsirea unui drum optim la scop este un **algoritm admisibil**.

Prin urmare, atunci când cele trei condiții ale Teoremei 2.1 sunt îndeplinite,  $A^*$  este un algoritm admisibil. Prin extensie, vom spune că orice funcție  $\hat{h}$  care nu supraestimează pe  $h$  este *admisibilă*.

În cele ce urmează, atunci când ne vom referi la Algoritmul  $A^*$ , vom presupune că cele trei condiții ale Teoremei 2.1 sunt verificate.

**Dacă două versiuni ale lui  $A^*$ ,  $A^*_1$  și  $A^*_2$ , diferă între ele numai prin aceea că  $\hat{h}_1 < \hat{h}_2$  pentru toate nodurile care nu sunt noduri-scop, vom spune că  $A^*_2$  este mai informat decât  $A^*_1$ . Referitor la această situație, formulăm următoarea teoremă (fără demonstrație):**

### **Teorema 2.2**

**Dacă algoritmul  $A^*_2$  este mai informat decât  $A^*_1$ , atunci la terminarea căutării pe care cei doi algoritmi o efectuează asupra oricărui graf având un drum de la  $n_0$  la un nod-scop, fiecare nod extins de către  $A^*_2$  este extins și de către  $A^*_1$ .**

**Rezultă de aici că  $A^*_1$  extinde cel puțin tot atâtea noduri câte extinde  $A^*_2$  și, prin urmare, algoritmul mai informat  $A^*_2$  este și mai eficient. În concluzie, se caută o funcție  $\hat{h}$  ale cărei valori sunt cât se poate de apropiate de cele ale funcției  $h$  (pentru o cât mai mare eficiență a căutării), dar fără să le depășească pe acestea (pentru admisibilitate). Pentru a evalua eficiența totală**



**a căutării, trebuie luat în considerație și costul calculării  
lui  $\hat{h}$ .**

- **Observatie:** Atunci când  $\hat{f}(n) = \hat{g}(n) = \text{adâncime}(n)$ , se obține căutarea de tip *breadth-first*. Algoritmul breadth-first reprezintă un caz particular al lui A\* (cu  $\hat{h} \equiv 0$ ), prin urmare el este un algoritm *admisibil*.

## Condiția de consistență

Fie o pereche de noduri  $(n_i, n_j)$  astfel încât  $n_j$  este un succesor al lui  $n_i$ .

### Definiția 2.2

Se spune că  $\hat{h}$  *îndeplinește condiția de consistență* dacă, pentru orice astfel de pereche  $(n_i, n_j)$  de noduri din graful de căutare,

$$\hat{h}(n_i) - \hat{h}(n_j) \leq c(n_i, n_j),$$

unde  $c(n_i, n_j)$  este costul arcului de la  $n_i$  la  $n_j$ .

Condiția de consistență mai poate fi formulată și sub una din formele următoare:

$$\hat{h}(n_i) \leq \hat{h}(n_j) + c(n_i, n_j)$$

sau

$$\hat{h}(n_j) \geq \hat{h}(n_i) - c(n_i, n_j),$$

ceea ce conduce la următoarea *interpretare* a ei: de-a lungul oricărui drum din graful de căutare, estimația

făcută asupra costului optim rămas pentru a atinge scopul nu poate descrește cu o cantitate mai mare decât costul arcului de-a lungul acelui drum. Se spune că funcția euristică este local consistentă atunci când se ia în considerație costul cunoscut al unui arc.

**Condiția de consistență:**

$$\hat{h}(n_i) \leq c(n_i, n_j) + \hat{h}(n_j)$$

Condiția de consistență implică faptul că valorile funcției  $\hat{f}$  corespunzătoare nodurilor din arborele de căutare descresc monoton pe măsură ce ne îndepărtăm de nodul de start.

➤ Fie  $n_i$  și  $n_j$  două noduri în *arborele de căutare* generat de algoritmul  $A^*$ , cu  $n_j$  succesor al lui  $n_i$ . Atunci, dacă condiția de consistență este satisfăcută, avem:

$$\hat{f}(n_j) \geq \hat{f}(n_i).$$

Din această cauză, condiția de consistență asupra lui  $\hat{h}$  este adesea numită condiție de monotonie asupra lui  $\hat{f}$ .

**Demonstrație:**

Pentru a demonstra acest fapt, se începe cu condiția de consistență:

$$\hat{h}(n_j) \geq \hat{h}(n_i) - c(n_i, n_j)$$

Se adună apoi  $\hat{g}(n_j)$  în ambii membri ai inegalității anterioare ( $\hat{g}$  este factor de adâncime, adică o estimatie a adâncimii nodului):

$$\hat{h}(n_j) + \hat{g}(n_j) \geq \hat{h}(n_i) + \hat{g}(n_j) - c(n_i, n_j)$$

Dar  $\hat{g}(n_j) = \hat{g}(n_i) + c(n_i, n_j)$ , adică adâncimea nodului  $n_j$  este adâncimea lui  $n_i$  plus costul arcului de la  $n_i$  la  $n_j$ .

Dacă egalitatea nu ar avea loc,  $n_j$  nu ar fi un succesor al lui  $n_i$  în *arborele* de căutare. Atunci:

$$\hat{h}(n_j) + \hat{g}(n_j) \geq \hat{h}(n_i) + \hat{g}(n_i) + c(n_i, n_j) - c(n_i, n_j),$$

deci  $\hat{f}(n_j) \geq \hat{f}(n_i)$ .

**Există următoarea teoremă referitoare la condiția de consistență:**

**Teorema 2.3**

**Dacă este satisfăcută condiția de consistență asupra lui  $\hat{h}$  , atunci, în momentul în care algoritmul  $A^*$  extinde un nod  $n$ , el a găsit deja un drum optim până la  $n$ .**

- **Observație:** Condiția de consistență este extrem de importantă deoarece, atunci când este satisfăcută, algoritmul  $A^*$  nu trebuie să redirecționeze niciodată pointeri la pasul 7. Căutarea într-un graf nu diferă atunci prin nimic de căutarea în cadrul unui arbore.



## **Optimalitatea Algoritmului A\***

Fie  $G$  o stare-scop optimă cu un cost al drumului notat  $f^*$ . Fie  $G_2$  o a doua stare-scop, suboptimală, care este o stare-scop cu un cost al drumului

$$g(G_2) > f^*$$

Presupunem că  $A^*$  selectează din coadă, pentru extindere, pe  $G_2$ . Întrucât  $G_2$  este o stare-scop, această alegere ar încheia căutarea cu o soluție suboptimală. Vom arăta că acest lucru nu este posibil.

Fie un nod  $n$ , care este, la pasul curent, un nod frunză pe un drum optim la  $G$ . (Un asemenea nod trebuie să existe, în afara cazului în care drumul a fost complet extins, caz în care algoritmul ar fi returnat  $G$ ). Pentru acest nod  $n$ , întrucât  $h$  este admisibilă, trebuie să avem:

$$f^* \geq f(n) \quad (1)$$

Mai mult, dacă  $n$  nu este ales pentru extindere în favoarea lui  $G_2$ , trebuie să avem:

$$f(n) \geq f(G_2) \quad (2)$$

Combinând (1) cu (2) obținem:

$$f^* \geq f(G_2) \quad (3)$$

Dar, deoarece  $G_2$  este o stare-scop, avem  $h(G_2) = 0$ . Prin urmare,

$$f(G_2) = g(G_2) \quad (4)$$

Cu presupunerile făcute, conform (3) și (4) am arătat că

$$f^* \geq g(G_2)$$

Această concluzie contrazice faptul că  $G_2$  este suboptimal. Ea arată că  $A^*$  nu selectează niciodată pentru extindere un scop suboptimal.  $A^*$  întoarce

**o soluție numai după ce a selectat-o pentru extindere, de aici rezultând faptul că  $A^*$  este un *algorithm optim*.**

## Completitudinea Algoritmului A\*

Întrucât  $A^*$  extinde noduri în ordinea valorilor crescătoare ale lui  $f$ , în final va exista o extindere care conduce la o stare-scop. Acest lucru este adevărat, în afara cazului în care există un număr foarte mare de noduri, număr care tinde la infinit, cu

$$f(n) < f^*.$$

Singura modalitate în care ar putea exista un număr infinit de noduri ar fi aceea în care:

- există *un nod* cu *factor de ramificare infinit*;
- există *un drum* cu un *cost finit*, dar care are un *număr infinit de noduri*. (Acest lucru ar fi posibil conform paradoxului lui Zeno, care vrea să arate că o piatră aruncată spre un copac nu va ajunge niciodată la acesta. Astfel, se imaginează că traiectoria pietrei este împărțită într-un șir de faze, fiecare dintre acestea acoperind jumătate din distanța rămasă până la copac. Aceasta conduce la un număr infinit de pași cu un cost total finit).

Prin urmare, *exprimarea corectă* este aceea că  $A^*$  este complet relativ la *grafuri local finite*, adică grafuri cu un factor de ramificare finit, cu condiția să existe o constantă pozitivă  $\delta$  astfel încât fiecare operator să coste cel puțin  $\delta$ .

## Complexitatea Algoritmului A\*

S-a arătat că o creștere exponențială va interveni, în afara cazului în care eroarea în funcția euristică nu crește mai repede decât logaritmul costului efectiv al drumului. Cu alte cuvinte, *condiția pentru o creștere subexponențială* este:

$$|h(n) - h^*(n)| \leq O(\log h^*(n)),$$

unde  $h^*(n)$  este *adevăratul* cost de a ajunge de la  $n$  la scop.

În afară de timpul mare calculator, algoritmul A\* consumă și mult spațiu de memorie deoarece *păstrează în memorie toate nodurile generate*.

Algoritmi de căutare mai noi, de tip “memory-bounded” (cu limitare a memoriei), au reușit să înlăture neajunsul legat de problema spațiului de memorie folosit, fără a sacrifica optimalitatea sau completitudinea. Unul dintre aceștia este algoritmul IDA\*.

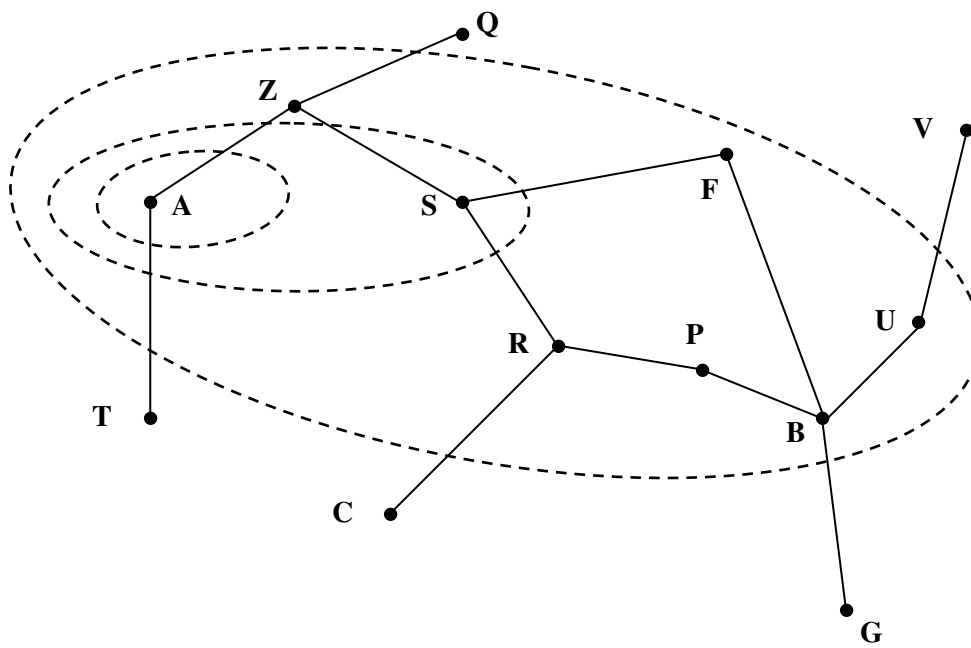
## **Iterative Deepening A\* (IDA\*)**

Algoritmul IDA\* se referă la o căutare iterativă în adâncime de tip A\* și este o extensie logică a lui Iterative Deepening Search care folosește, în plus, informația euristică.

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

Faptul că în cadrul algoritmului  $A^*$   $f$  nu descrește niciodată de-a lungul oricărui drum care pleacă din rădăcină ne permite să trasăm, din punct de vedere conceptual, *contururi* în spațiul stărilor. Astfel, în interiorul unui contur, toate nodurile au valoarea  $f(n)$  mai mică sau egală cu o aceeași valoare. În cazul algoritmului IDA\* fiecare iterație 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 declanșată o nouă iterație, folosind un nou cost  $f$ , corespunzător următorului contur. Fig. 2.10 prezintă căutări iterative în interiorul câte unui contur.





**Fig. 2.10**

**Algoritmul IDA\* este complet și optim cu aceleași amendamente ca și A\*. Deoarece este de tip depth-first nu necesită decât un spațiu proporțional cu cel mai lung drum pe care îl explorează.**

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

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

## **Implementarea în Prolog a căutării de tip best-first**

Vom imagina căutarea de tip best-first funcționând 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 explorați de subprocese ale subproceselor, ș.a.m.d.. Dintre toate aceste subprocese doar unul este activ la un moment dat și anume cel care se ocupă de alternativa cea mai promițătoare (adică alternativa corespunzătoare celei mai mici  $\hat{f}$  - valori). Celelalte procese așteaptă până când  $\hat{f}$  - valorile se schimbă astfel încât o altă alternativă devine mai promițătoare, 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 și, atâta vreme cât acest buget nu este epuizat, procesul este activ. Pe durata activității sale, procesul își

**expandează propriul subarbore, iar în cazul atingerii unei stări-scop este anunțată găsirea unei soluții. Bugetul acestei funcționări este determinat de  $\hat{f}$  - valoarea corespunzătoare celei mai apropiate alternative concurente.**

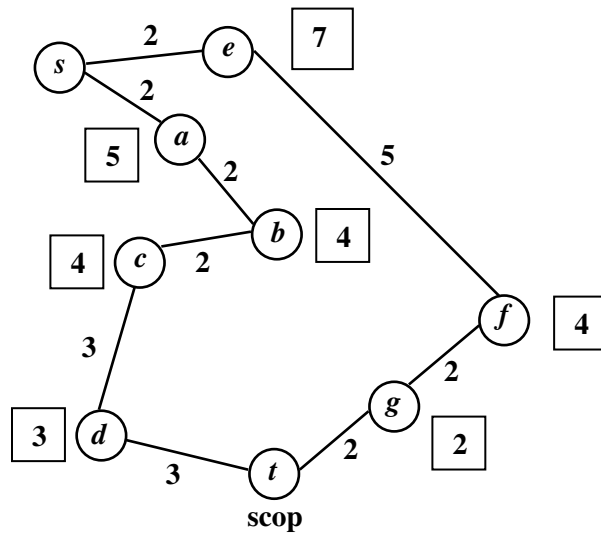
## Exemplu

Considerăm orașele  $s, a, b, c, d, e, f, g, t$  unite printr-o rețea de drumuri ca în Fig. 2.11. Aici fiecare drum direct între două orașe este etichetat cu lungimea sa; numărul din căsuța alăturată unui oraș reprezintă distanța în linie dreaptă între orașul respectiv și orașul  $t$ . Ne punem problema determinării celui mai scurt drum între orașul  $s$  și orașul  $t$  utilizând strategia best-first. Definim în acest scop funcția  $\hat{h}$  bazându-ne pe distanța în linie dreaptă între două orașe. Astfel, pentru un oraș  $X$ , definim

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

unde  $\text{dist}(X, t)$  reprezintă distanța în linie dreaptă între  $X$  și  $t$ .

În acest exemplu, căutarea de tip best-first este efectuată prin intermediul a două procese,  $P_1$  și  $P_2$ , 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 inițiale, 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  $e$ ,  $\hat{f}(c) = \hat{g}(c) + \hat{h}(c) = 6 + 4 = 10$ ,  $\hat{f}(e) = \hat{g}(e) + \hat{h}(e) = 2 + 7 = 9$  și deci  $\hat{f}(e) < \hat{f}(c)$ . În acest moment, situația se schimbă: procesul  $P_2$  devine activ, iar procesul  $P_1$  intră în așteptare. În continuare,  $\hat{f}(c) = 10$ ,  $\hat{f}(f) = 11$ ,  $\hat{f}(c) < \hat{f}(f)$  și deci  $P_1$  devine activ și  $P_2$  intră în așteptare. Pentru că  $\hat{f}(d) = 12 > 11$ , procesul  $P_1$  va reentra în așteptare, iar procesul  $P_2$  va rămâne activ până când se va atinge starea scop  $t$ .

Căutarea schițată mai sus pornește din nodul inițial și este continuată cu generarea unor noduri noi, conform relației de succesiune. În timpul acestui proces, este generat un arbore de căutare, a cărui rădăcină este nodul de start. Acest arbore este expandat în direcția

**cea mai promițătoare conform  $\hat{f}$  - valorilor, până la  
găsirea unei soluții.**



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

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

- $l(N, F/G)$  corespunde unui arbore cu un singur nod  $N$ ;  $N$  este nod în spațiul stărilor,  $G$  este  $\hat{g}(N)$  (considerăm  $\hat{g}(N)$  ca fiind costul drumului între nodul de start și 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  $\hat{g}(N)$ ,  $F$  este  $\hat{f}$  - valoarea

actualizată a lui  $N$ , adică este  $\hat{f}$  - valoarea celui mai promițător succesor al lui  $N$ ; de asemenea,  $Subs$  este ordonată crescător conform  $\hat{f}$  - valorilor subarborilor constituenți.

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

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

*expandea* (*Drum*, *Arb*, *Limita*, *Arb1*, *Rez*, *Solutie*)

Argumentele sale au următoarele semnificații:

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

Drum, Arb și Limita sunt parametrii de intrare pentru *expandea* (în sensul că ei sunt deja instanțiați atunci când *expandea* este folosit). Prin utilizarea predicatului *expandea* se pot obține trei feluri de rezultate, rezultate indicate prin valoarea argumentului Rez, după cum urmează:

- Rez=da, caz în care Solutie va unifica cu o cale soluție găsită expandând Arb în limita Limita (adică fără ca  $\hat{f}$  - valoarea să depășească limita Limita ); Arb1 va rămâne neinstanțiat;
- Rez=nu, caz în care Arb1 va fi, de fapt, Arb expandat până când  $\hat{f}$  - valoarea sa a depășit Limita; Solutie va rămâne neinstanțiat;
- Rez=imposibil, caz în care argumentele Arb1 și Solutie vor rămâne neinstanțiate; acest ultim caz indică faptul că explorarea lui Arb este o alternativă “moartă”, deci nu trebuie să i se mai dea o șansă 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 succesori ar crea un ciclu.

Urmează o implementare a unei variante a metodei best-first în SICStus Prolog, implementare care folosește 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([],1(Nod\_initial,0/0),9999999,\_,**  
    **da,Solutie).**

**expandeaza(Drum,1(N,\_) ,\_,\_, da,[N|Drum]):-scop(N).**

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

**expandeaza(Drum,1(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 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 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 relații specifice problemei. Aceste relații definesc de fapt problema particulară („regulile jocului”) și, de asemenea, adaugă o anumită informație 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 și Nod în spațiul stărilor

- **scop (Nod)**

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

- **h (Nod, H)**

% H este o estimatie euristica a costului celui mai ieftin drum între Nod și o stare-scop.

**Exemple:** Problema misionarilor și canibalilor, Problema “8-puzzle”