Căutarea informată

Căutarea informată **se mai numeşte şi** ***căutare euristică*****. Euristica este o metodă de studiu şi de cercetare bazată pe descoperirea de fapte noi. În acest tip de căutare vom folosi informaţia despre spaţiul de stări. Se folosesc cunoştinţe specifice problemei şi 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 desfăşoară în mod uniform plecând de la nodul iniţial. El înaintează în mod preferenţial de-a lungul unor noduri pe care informaţia euristică, specifică problemei, le indică ca aflându-se pe drumul cel mai bun către un scop. Un asemenea proces de căutare se numeşte *căutare euristică* sau *căutare de tip best-first*.

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

1. Se presupune existenţa unei funcţii euristice de evaluare, $\hat{f}$, cu rolul de a ne ajuta să decidem care nod ar trebui extins la pasul următor. Se va adopta *convenţia* că valori mici ale lui $\hat{f}$ indică nodurile cele mai bune. Această funcţie se bazează pe informaţie specifică domeniului pentru care s-a formulat problema. Este o funcţie 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 toţi 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:

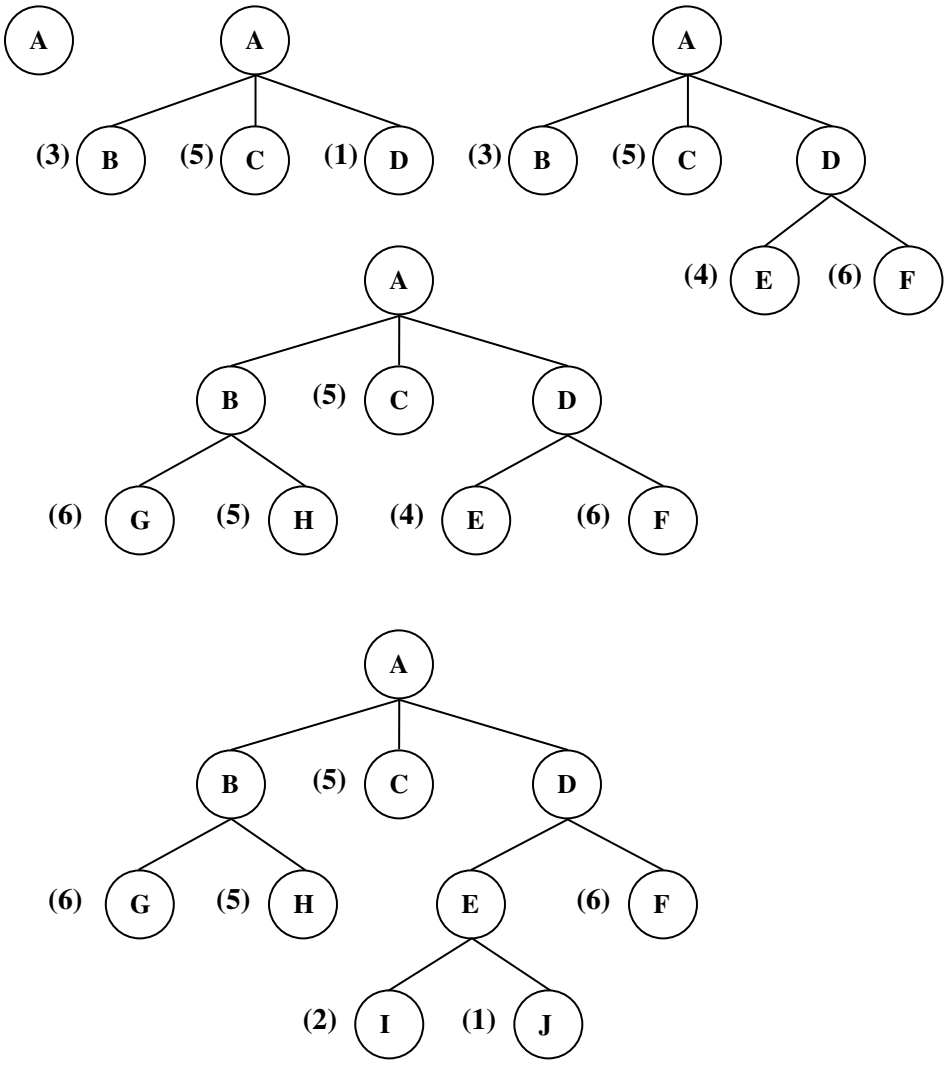

Fig. 2.7

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

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

Pentru a nu fi induşi în eroare de o euristică extrem de optimistă, este necesar să înclinăm căutarea în favoarea posibilităţii 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 estimaţie 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 definiţiei 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 iniţial este vidă.**
3. **Dacă lista OPEN este vidă, EXIT cu eşec.**
4. **Selectează primul nod din OPEN, înlătură-l din lista OPEN şi include-l în lista CLOSED. Numeşte acest nod *n*.**

5. **Dacă *n* este un nod scop, algoritmul se încheie cu succes, iar soluţia este cea obţinută 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 mulţime, *M*, de succesori. Include *M* ca succesori ai lui *n* în T_r, prin crearea de arce de la *n* la fiecare membru al mulţimii *M*.**
7. **Reordonează lista OPEN, fie în concordanţă cu un plan arbitrar, fie în mod euristic.**
8. **Mergi la pasul 3.**

❑

- **<u>Observaţie</u>: 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 sfârşitul 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 şi căutare euristică, lista OPEN este reordonată în funcţie 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 funcţie de *valorile crescătoare ale funcţiei* $\hat{f}$. Această versiune a algoritmului GraphSearch se va numi Algoritmul A*.

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

- $h(n) =$**costul *efectiv* al drumului de cost minim dintre nodul *n* şi un nod-scop, luând în consideraţie toate nodurile-scop posibile şi 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*.

- **<u>Observaţie</u>: $f(n_0) = h(n_0)$ reprezintă costul unui drum de cost minim nerestricţionat, de la nodul n_0 la un nod-scop.**

Pentru fiecare nod *n*, fie $\hat{h}(n)$, numit *factor euristic*, o estimaţie a lui $h(n)$ şi 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 funcţia $\hat{f} = \hat{g} + \hat{h}$.

În definirea Algoritmului A* de până acum nu s-a ţinut 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 secvenţă de acţiuni care pot conduce la aceeaşi stare a lumii plecând din starea iniţială. (Există situaţii în care fiecare dintre succesorii nodului *n* îl are pe *n* ca succesor i.e. acţiunile 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 mulţime, *M*, de succesori care nu sunt deja *părinţi* 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 înlocuieşte pasul 6 prin următorul pas 6":

6". Extinde nodul *n*, generând o mulţime, *M*, de succesori care nu sunt deja *strămoşi* 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*.

<u>Observaţie</u>: Pentru a verifica existenţa 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 strămoşii 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 aceeaşi 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 mulţimea *M* se află deja în listele OPEN sau CLOSED. Algoritmul uită deci posibilitatea de a ajunge în aceleaşi noduri urmând drumuri diferite. Acest "acelaşi 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 aceeaşi 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 condiţii suplimentare asupra lui $\hat{f}$, sunt necesare nişte modificări în algoritmul A*, şi anume: deoarece căutarea poate ajunge la acelaşi nod de-a lungul unor drumuri diferite, algoritmul A* generează un *graf de căutare*, notat cu G. G este structura de noduri şi de arce generată de A* pe măsură ce algoritmul extinde nodul iniţial, succesorii lui ş.a.m.d.. A* menţine şi 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 şi în arborele de căutare. Graful de căutare este menţinut deoarece căutări ulterioare pot găsi drumuri mai scurte, care folosesc anumite arce din graful de căutare anterior ce nu se aflau şi în arborele de căutare anterior.

Dăm, în continuare, versiunea algoritmului A* care menţine graful de căutare. *În practică,* această versiune este folosită mai rar deoarece, de obicei, se pot impune *condiţii 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 iniţial n_0. Plasează n_0 într-o listă numită OPEN.

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

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

4. Selectează primul nod din lista OPEN, înlătură-l din OPEN şi plasează-l în lista CLOSED. Numeşte acest nod *n*.

5. Dacă *n* este un nod scop, opreşte execuţia cu succes. Returnează soluţia obţinută urmând un drum de-a lungul pointerilor de la *n* la n_0 în G. (Pointerii definesc un arbore de căutare şi sunt stabiliţi la pasul 7).

6. Extinde nodul *n*, generând o mulţime, *M*, de succesori ai lui care nu sunt deja strămoşi ai lui *n* în G. Instalează aceşti membri ai lui *M* ca succesori ai lui *n* în G.

7. Stabileşte 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 aceşti membri ai lui *M* listei OPEN. Pentru fiecare membru, *m*, al lui *M*, care se afla deja în OPEN sau în CLOSED, redirecţionează 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, redirecţionează pointerii fiecăruia dintre descendenţii săi din G astfel încât aceştia să ţintească înapoi de-a lungul celor mai bune drumuri până la aceşti descendenţi, găsite până în acel moment.

8. Reordonează lista OPEN în ordinea valorilor crescătoare ale funcţiei $\hat{f}$. (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.

❑

- **<u>Observaţie:</u> La pasul 7 sunt redirecţionaţi 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 existenţi. Redirecţionarea pointerilor descendenţilor nodurilor care deja se află în lista CLOSED economiseşte efortul de căutare, dar poate duce la o cantitate exponenţială de calcule. De aceea, această parte a pasului 7 de obicei nu este implementată. Unii dintre aceşti pointeri vor fi până la urmă redirecţionaţi oricum, pe măsură ce căutarea progresează.**

Admisibilitatea Algoritmului A*

Există anumite condiţii asupra grafurilor şi a lui $\hat{h}$ care garantează că algoritmul A*, aplicat acestor grafuri, găseşte întotdeauna drumuri de cost minim. Condiţiile 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, ε.

Condiţia 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 funcţie $\hat{h}$ este uneori numită un *estimator optimist*.

- **<u>Observaţii:</u>**
  1. **Este relativ uşor să se găsească, în probleme, o funcţie $\hat{h}$ care satisface această *condiţie a limitei de jos*. De exemplu, în probleme de găsire a drumurilor în cadrul unor grafuri ale căror noduri sunt oraşe, distanţa de tip linie dreaptă de la un oraş *n* la un oraş-scop constituie o limită inferioară asupra distanţei reprezentând un drum optim de la nodul *n* la nodul-scop.**
  2. **Cu cele trei condiţii 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 condiţiile asupra grafurilor şi asupra lui $\hat{h}$ enunţate anterior şi cu condiţia să existe un drum de cost finit de la nodul iniţial, n_0, la un nod-scop, algoritmul A* garantează găsirea unui drum de cost minim la un scop.

Definiţia 2.1

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

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

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

Dacă *două versiuni ale lui A, A*$_1$ şi 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 situaţie, formulăm următoarea teoremă (fără demonstraţie):**

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 şi de către A*$_1$.

Rezultă de aici că A*$_1$ extinde cel puţin tot atâtea noduri câte extinde A*$_2$ şi, prin urmare, algoritmul mai informat A*$_2$ este şi mai eficient. În concluzie, se caută o funcţie $\hat{h}$ ale cărei valori sunt cât se poate de apropiate de cele ale funcţiei h (pentru o cât mai mare eficienţă a căutării), dar fără să le depăşească pe acestea (pentru admisibilitate). Pentru a evalua eficienţa *totală*

a căutării, trebuie luat în consideraţie şi costul calculării lui $\hat{h}$.

- **<u>Observatie:</u> Atunci când $\hat{f}(n) = \hat{g}(n) =$ adâncime(n), se obţine 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*.**

Condiţia de consistenţă

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

Definiţia 2.2

Se spune că $\hat{h}$ *îndeplineşte condiţia de consistenţă* 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.

Condiţia de consistenţă mai poate fi formulată şi 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, estimaţia

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

Condiţia de consistenţă:

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

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

- **Fie n_i şi n_j două noduri în *arborele de căutare* generat de algoritmul A*, cu n_j succesor al lui n_i. Atunci, dacă condiţia de consistenţă este satisfăcută, avem:**

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

Din această cauză, *condiţia de consistenţă asupra lui $\hat{h}$* este adesea numită *condiţie de monotonie asupra lui $\hat{f}$*.

Demonstraţie:

Pentru a demonstra acest fapt, se începe cu condiţia de consistenţă:

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

Se adună apoi $\hat{g}(n_j)$ **în ambii membri ai inegalităţii anterioare (**$\hat{g}$ **este factor de adâncime, adică o estimaţie 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 condiţia de consistenţă:

Teorema 2.3

Dacă este satisfăcută condiţia de consistenţă 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*.

- **<u>Observaţie:</u> Condiţia de consistenţă este extrem de importantă deoarece, atunci când este satisfăcută, algoritmul A* nu trebuie să redirecţioneze 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 optimală 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 soluţie 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:

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

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

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

Combinând (1) cu (2) obţinem:

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

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

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

Cu presupunerile făcute, conform (3) şi (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 soluţie numai după ce a selectat-o pentru extindere, de aici rezultând faptul că *A* este un algoritm 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 împărţită într-un şir de faze, fiecare dintre acestea acoperind jumătate din distanţa rămasă până la copac. Aceasta conduce la un număr infinit de paşi 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 condiţia să existe o constantă pozitivă δ astfel încât fiecare operator să coste cel puţin δ.

Complexitatea Algoritmului A*

S-a arătat că o creştere exponenţială va interveni, în afara cazului în care eroarea în funcţia euristică nu creşte mai repede decât logaritmul costului efectiv al drumului. Cu alte cuvinte, *condiţia pentru o creştere subexponenţială* este:

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

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

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

Algoritmi de căutare mai noi, de tip "memory-bounded" (cu limitare a memoriei), au reuşit să înlăture neajunsul legat de problema spaţiului de memorie folosit, fără a sacrifica optimalitatea sau completitudinea. Unul dintre aceştia este algoritmul IDA*.

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.

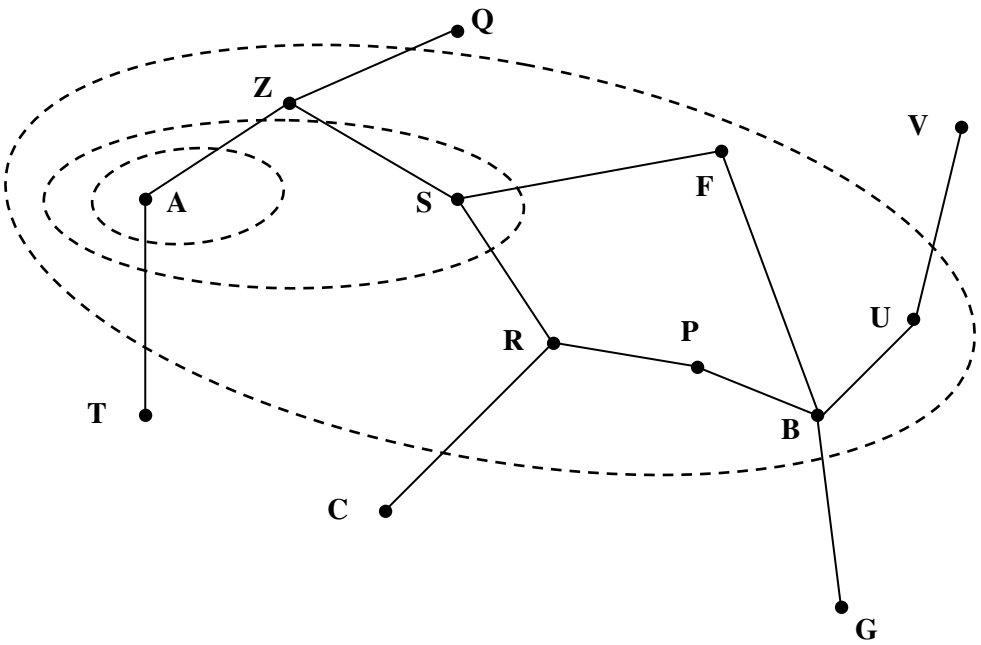

Fig. 2.10

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.

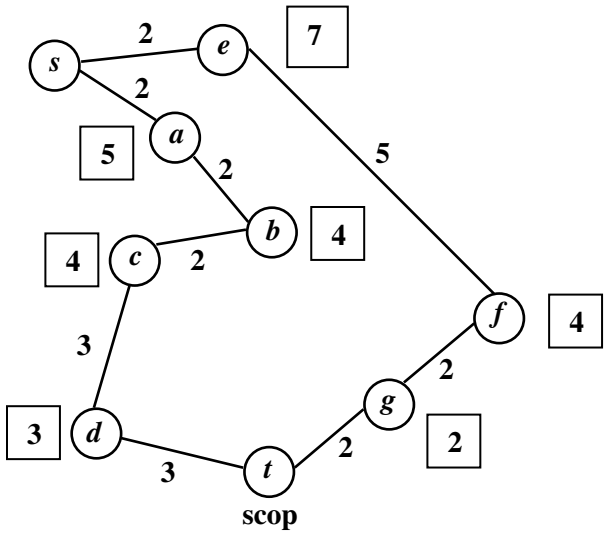

Fig. 2.11

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 $\hat{g}(N)$, 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** $\hat{f}(e)=9$, **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** $\hat{f}(c)=10$, **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:

- **Drum reprezintă calea între nodul de start al căutării şi 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 şi Limita 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 Rez, după cum urmează:

- **Rez=da, caz în care Solutie va unifica cu o cale soluţie găsită expandând Arb în limita Limita (adică fără ca $\hat{f}$ - valoarea să depăşească limita Limita); Arb1 va rămâne neinstanţiat;**
- **Rez=nu, caz în care Arb1 va fi, de fapt, Arb 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 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 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 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 f^-valoare a unui arbore din lista de arbori As,
%daca aceasta lista este nevida; lista As este ordonata dupa 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, 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”