

TEHNICI DE CĂUTARE

Căutarea este o traversare sistematică a unui spațiu de soluții posibile ale unei probleme.

Un spațiu de căutare este de obicei un graf (sau, mai exact, un arbore) în care un nod desemnează o soluție parțială, iar o muchie reprezintă un pas în construirea unei soluții. Scopul căutării poate fi acela de

- a găsi un drum în graf de la o situație inițială la una finală;**
- a ajunge într-un nod care reprezintă situația finală.**

- Programul reprezinta un agent inteligent.
- Agenții cu care vom lucra vor adopta un *scop* și vor urmări *satisfacerea* lui.

Rezolvarea problemelor prin intermediul căutării

În procesul de rezolvare a problemelor, *formularea scopului*, bazată pe situația curentă, reprezintă primul pas.

Vom considera un *scop* ca fiind o mulțime de stări ale universului, și anume acele stări în care scopul este satisfăcut. *Acțiunile* pot fi privite ca generând tranziții între stări ale universului. Agentul va trebui să afle care acțiuni îl vor conduce la o stare în care scopul este satisfăcut. Înainte de a face asta el

trebuie să decidă ce tipuri de acțiuni și de stări să ia în considerație.

Procesul decizional cu privire la acțiunile și stările ce trebuie luate în considerație reprezintă *formularea problemei*. Formularea problemei urmează după formularea scopului.

Un agent care va avea la dispoziție mai multe opțiuni imediate va decide ce să facă examinând mai întâi diferite *secvențe de acțiuni* posibile, care conduc la stări de valori necunoscute, urmând ca, în urma acestei examinări, să o aleagă pe cea mai bună. Procesul de examinare a unei astfel de succesiuni de acțiuni se numește *căutare*. Un *algoritm de căutare* primește ca *input* o *problemă* și întoarce ca *output* o *soluție* sub forma unei succesiuni de acțiuni.

Odată cu găsirea unei soluții, acțiunile recomandate de aceasta pot fi duse la îndeplinire. Aceasta este *faza de execuție*. Prin urmare, agentul *formulează, caută și execută*. După formularea unui scop și a unei probleme de rezolvat, agentul cheamă o

procedură de căutare pentru a o rezolva. El folosește apoi soluția pentru a-l ghida în acțiunile sale, executând ceea ce îi recomandă soluția ca fiind următoarea acțiune de îndeplinit și apoi înlătură acest pas din succesiunea de acțiuni. Odată ce soluția a fost executată, agentul va găsi un nou scop.

Probleme și soluții corect definite

➤ Probleme cu o singură stare

Elementele de bază ale definirii unei probleme sunt *stările* și *acțiunile*. Pentru a descrie stările și acțiunile, din punct de vedere formal, este nevoie de următoarele elemente [9]:

- Starea inițială în care agentul știe că se află.
- Mulțimea acțiunilor posibile disponibile agentului. Termenul de operator este folosit pentru a desemna descrierea unei acțiuni, prin specificarea stării în care se va ajunge ca urmare a îndeplinirii acțiunii respective, atunci când ne aflăm într-o anumită stare. (O formulare alternativă folosește o *funcție succesor* S . Fiind dată o anumită stare x , $S(x)$ întoarce mulțimea stărilor în care se poate ajunge din x , printr-o unică acțiune).

- Spațiul de stări al unei probleme reprezintă mulțimea tuturor stărilor în care se poate ajunge plecând din starea inițială, prin intermediul oricărei secvențe de acțiuni.
- Un drum în spațiul de stări este orice secvență de acțiuni care conduce de la o stare la alta.
- Testul scop este testul pe care un agent îl poate aplica unei singure descrieri de stare pentru a determina dacă ea este o stare de tip scop, adică o stare în care scopul este atins (sau realizat). Uneori există o mulțime explicită de stări scop posibile și testul efectuat nu face decât să verifice dacă s-a ajuns în una dintre ele. Alteori, scopul este specificat printr-o proprietate abstractă și nu prin enumerarea unei mulțimi de stări. De exemplu, în șah, scopul este să se ajungă la o stare numită “șah mat”, în care regele

adversarului poate fi capturat la următoarea mutare, orice ar face adversarul. S-ar putea întâmpla ca o soluție să fie preferabilă alteia, chiar dacă amândouă ating scopul. Spre exemplu, pot fi preferate drumuri cu mai puține acțiuni sau cu acțiuni mai puțin costisitoare.

- Funcția de cost a unui drum este o funcție care atribuie un cost unui drum. Ea este adeseori notată prin g . Vom considera costul unui drum ca fiind suma costurilor acțiunilor individuale care compun drumul.

Împreună starea inițială, mulțimea operatorilor, testul scop și funcția de cost a unui drum definesc o problemă.

➤ Probleme cu stări multiple

Pentru definirea unei astfel de probleme trebuie specificate:

- o mulțime de stări inițiale;
- o mulțime de operatori care indică, în cazul fiecărei acțiuni, mulțimea stărilor în care se ajunge plecând de la orice stare dată;
- un test scop (la fel ca la problema cu o singură stare);
- funcția de cost a unui drum (la fel ca la problema cu o singură stare).

Un operator se aplică unei mulțimi de stări prin reunirea rezultatelor aplicării operatorului fiecărei stări din mulțime. Aici

un *drum* leagă *mulțimi de stări*, iar o *soluție* este un drum care conduce la o *mulțime de stări*, dintre care *toate sunt stări scop*. Spațiul de stări este aici înlocuit de *spațiul mulțimii de stări*.

Un exemplu: Problema misionarilor si a canibalilor

Definiție formală a problemei:

- Stări: o stare constă dintr-o secvență ordonată de trei numere reprezentând numărul de misionari, de canibali și de bărci, care se află pe malul râului. Starea de pornire (inițială) este (3,3,1).
- Operatori: din fiecare stare, posibili operatori trebuie să ia fie un misionar, fie un canibal, fie doi misionari, fie doi canibali, fie câte unul din fiecare și să îi transporte cu barca. Prin urmare, există *cel mult cinci operatori*, deși majorității stărilor le corespund mai puțini operatori, întrucât trebuie evitate stările interzise.

(Observație: Dacă am fi ales să distingem între indivizi, în loc de cinci operatori ar fi existat 27).

- ▣ **Testul scop**: să se ajungă în starea (0,0,0).
- ▣ **Costul drumului**: este dat de numărul de traversări.

Acest spațiu al stărilor este suficient de mic pentru ca problema să fie una trivială pentru calculator.

Căutarea soluțiilor și generarea secvențelor **de acțiuni**

Rezolvarea unei probleme începe cu starea inițială. Primul pas este acela de a testa dacă starea inițială este o stare scop. Dacă nu, se iau în considerație și alte stări. Acest lucru se realizează aplicând operatorii asupra stării curente și, în consecință, generând o mulțime de stări. Procesul poartă denumirea de extinderea stării. Atunci când se generează mai multe posibilități, trebuie făcută o alegere relativ la cea care va fi luată în considerație în continuare, aceasta fiind esența căutării. Alegerea referitoare la care dintre stări trebuie extinsă prima este determinată de strategia de căutare.

Procesul de căutare construiește un arbore de căutare, a cărui rădăcină este un nod de căutare corespunzând stării inițiale. La fiecare pas, algoritmul de căutare alege un nod-frunză pentru a-l extinde.

Este important să facem distincția între spațiul stărilor și arborele de căutare. Spre exemplu, într-o problemă de căutare a unui drum pe o hartă, pot exista doar 20 de stări în spațiul stărilor, câte una pentru fiecare oraș. Dar există un număr infinit de drumuri în acest spațiu de stări. Prin urmare, arborele de căutare are un număr infinit de noduri. Evident, un bun algoritm de căutare trebuie să evite urmarea unor asemenea drumuri.

Este importantă distincția între noduri și stări:

- Un nod este o structură de date folosită pentru a reprezenta arborele de căutare corespunzător unei anumite realizări a unei probleme, generată de un anumit algoritm.
- O stare reprezintă o configurație a lumii înconjurătoare.

De aceea, nodurile au adâncimi și părinți, iar stările nu le au. Mai mult, este posibil ca două noduri diferite să conțină aceeași stare, dacă acea stare este generată prin intermediul a două secvențe de acțiuni diferite.

Reprezentarea nodurilor în program

Există numeroase moduri de a reprezenta nodurile. În general, se consideră că un nod este o structură de date cu cinci componente:

- starea din spațiul de stări căreia îi corespunde nodul;
- nodul din arborele de căutare care a generat acest nod (nodul părinte);
- operatorul care a fost aplicat pentru a se genera nodul;
- numărul de noduri aflate pe drumul de la rădăcină la acest nod (adâncimea nodului);
- costul drumului de la starea inițială la acest nod.

Reprezentarea colecției de noduri care așteaptă pentru a fi extinse

Această colecție de noduri poartă denumirea de frontieră. Cea mai simplă reprezentare ar fi aceea a unei mulțimi de noduri, iar strategia de căutare ar fi o funcție care selectează, din această mulțime, următorul nod ce trebuie extins. Deși din punct de vedere conceptual această cale este una directă, din punct de vedere computațional ea poate fi foarte scumpă, pentru că funcția strategie ar trebui să se “uite” la fiecare element al mulțimii pentru a-l alege pe cel mai bun. De aceea, *vom presupune că această colecție de noduri este implementată ca o coadă.*

Evaluarea strategiilor de căutare

Strategiile de căutare se evaluează conform următoarelor patru criterii:

- **Completitudine**: dacă, atunci când o soluție există, strategia dată garantează găsirea acesteia;
- **Complexitate a timpului**: durata de timp pentru găsirea unei soluții;
- **Complexitate a spațiului**: necesitățile de memorie pentru efectuarea căutării;
- **Optimalitate**: atunci când există mai multe soluții, strategia dată să o găsească pe cea mai de calitate dintre ele.

Căutarea neinformată

Termenul de **căutare neinformată** desemnează faptul că o strategie de acest tip nu deține nici o informație despre numărul de pași sau despre costul drumului de la starea curentă la scop. Tot ceea ce se poate face este să se distingă o stare-scop de o stare care nu este scop. Căutarea neinformată se mai numește și **căutarea oarbă**.

Căutarea informată

Să considerăm, de pildă, problema găsirii unui drum de la Arad la București, având în față o hartă. De la starea inițială, Arad, există trei acțiuni care conduc la trei noi stări: Sibiu, Timișoara și Zerind. O căutare neinformată nu are nici o preferință între cele trei variante. Un agent mai inteligent va observa însă că scopul, București, se află la sud-est de Arad și că numai *Sibiu* este în această direcție, care reprezintă, probabil, cea mai bună alegere. Strategiile care folosesc asemenea considerații se numesc strategii de căutare informată sau strategii de căutare euristică.

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

Procesul de căutare 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:

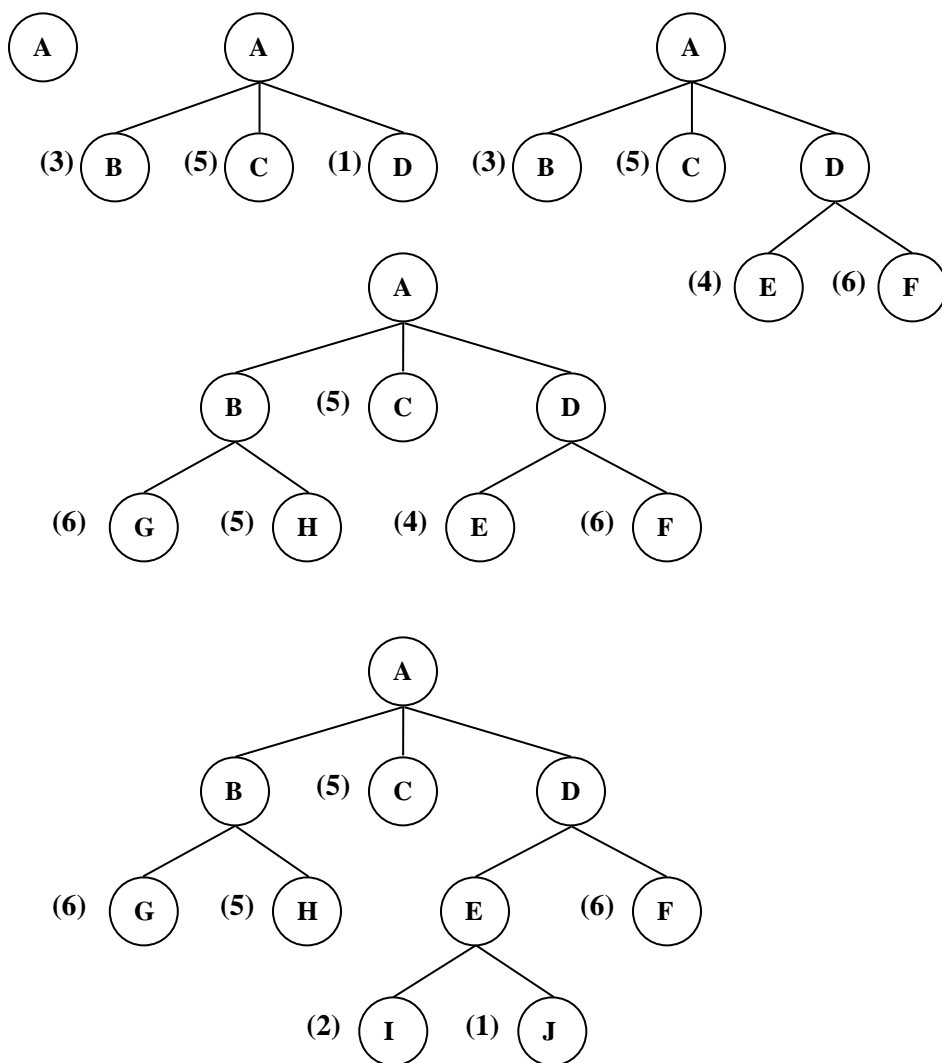


Fig. 2.7

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

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 .

Prezentăm 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. 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. Redirecționarea 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.

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.

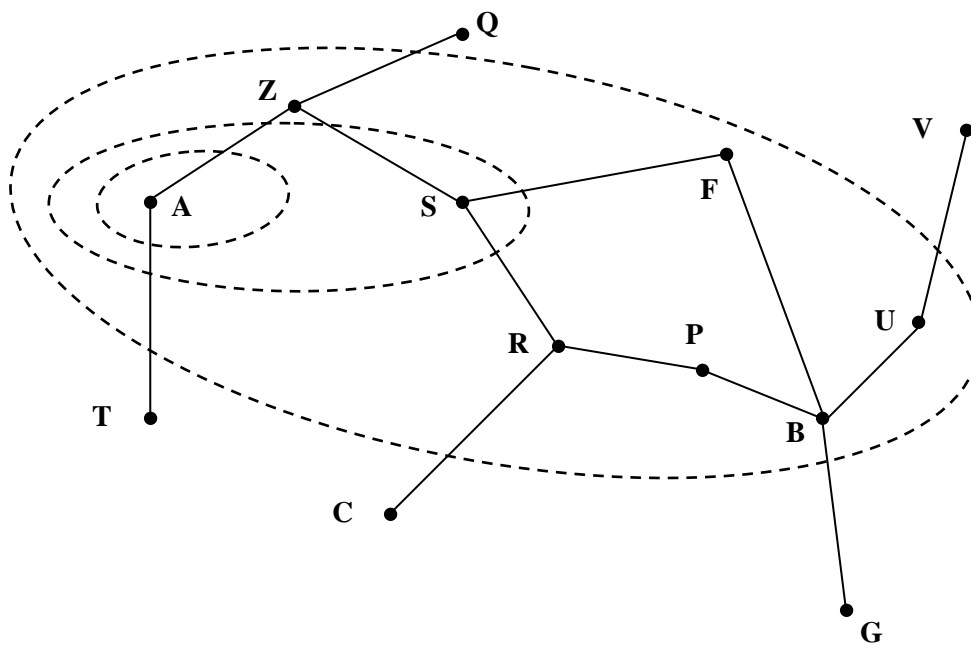


Fig. 2.10

Algoritmul IDA* este 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 δ 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 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 .

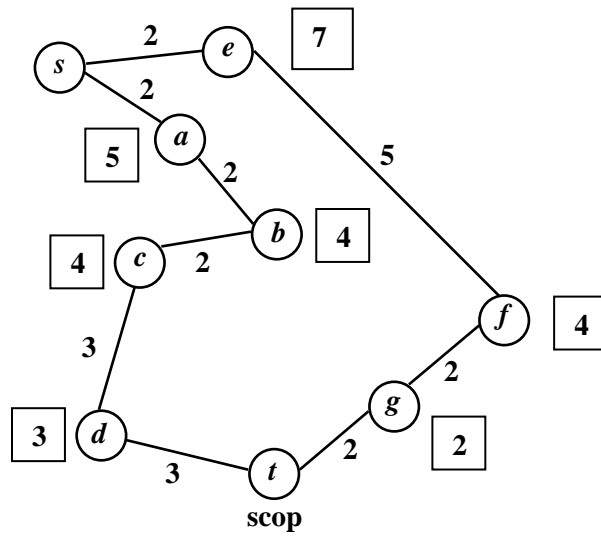


Fig. 2.11

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.**