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 şi 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, şi 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 şi de stări să ia în considerație.

Procesul decizional cu privire la acțiunile şi 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 numeşte căutare. Un algoritm de căutare primeşte ca input o problemă şi 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 şi execută. După formularea unui scop şi a unei probleme de rezolvat, agentul cheamă o procedură de căutare pentru a o rezolva. El foloseşte apoi soluția pentru a-l ghida în acțiunile sale, executând ceea ce îi recomandă soluția ca fiind următoarea acțiune de îndeplinit şi apoi înlătură acest pas din succesiunea de acțiuni. Odată ce soluția a fost executată, agentul va găsi un nou scop.

Probleme şi soluții corect definite

➢ Probleme cu o singură stare

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

- Starea inițială în care agentul ştie 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 foloseşte 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 şi testul efectuat nu face decât să verifice dacă s-a ajuns în una dintre ele. Alteori, scopul este specificat printr-o proprietate abstractă şi nu prin enumerarea unei mulțimi de stări. De exemplu, în şah, scopul este să se ajungă la o stare numită "şah 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 şi 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 şi de bărci, care se află pe malul râului. Starea de pornire (inițială) este (3,3,1).
- Operatori: din fiecare stare, posibilii operatori trebuie să ia fie un misionar, fie un canibal, fie doi misionari, fie doi canibali, fie câte unul din fiecare şi să îi transporte cu barca. Prin urmare, există cel mult cinci operatori, deşi majorității stărilor le corespund mai puțini operatori, intrucât 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 şi 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 şi alte stări. Acest lucru se realizează aplicând operatorii asupra stării curente şi, 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 construieşte 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 şi 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 oraş. 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 şi 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 şi părinți, iar stările nu le au. Mai mult, este posibil ca două noduri diferite să conțină aceeaşi stare, dacă acea stare este generată prin intermediul a două secvențe de acțiuni diferite.

Reprezentarea nodurilor in 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 aşteaptă 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 paşi 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 numeşte şi căutarea oarbă.

Căutarea informată

Să considerăm, de pildă, problema găsirii unui drum de la Arad la Bucureşti, având în față o hartă. De la starea inițială, Arad, există trei acțiuni care conduc la trei noi stări: Sibiu, Timişoara şi Zerind. O căutare neinformată nu are nici o preferință între cele trei variante. Un agent mai inteligent va observa însă că scopul, Bucureşti, se află la sud-est de Arad şi 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 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 informația despre spațiul de stări. Se folosesc cunoştințe specifice problemei şi se rezolvă probleme de optim.

Căutarea de tip best-first

Procesul de căutare nu se desfăşoară 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 numeşte 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. Procesul se încheie atunci când următorul nod care ar trebui extins este un nod-scop.
3.

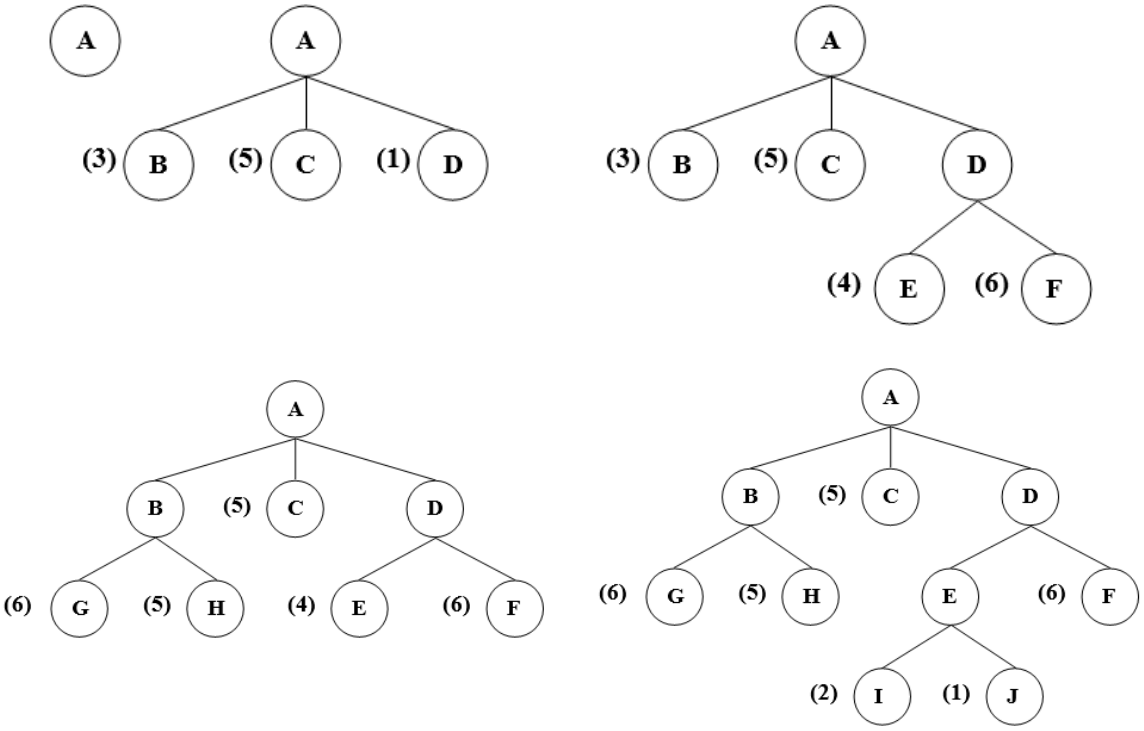


Fig. 2.7 ilustrează începutul unei căutări de tip best-first; există inițial un singur nod, A, astfel încât acesta va fi extins.

Pentru a nu fi induşi 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 estimație 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 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 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 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 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 şi un nod-scop, luând în considerație 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.

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 estimație 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 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 aceeaşi stare a lumii plecând din starea inițială. (Există situații în care fiecare dintre succesorii nodului n îl are pe n ca succesor 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 înlocuieşte pasul 6 prin următorul pas 6":

6". Extinde nodul n, generând o mulțime, 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.

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 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. 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 condiții 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 inițial, succesorii lui ş.a.m.d.. A* menține 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.

Observaţie: 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.

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: $|h(n) - h^*(n)| \leq O(\log h^*(n))$, unde $h^*(n)$ este adevăratul cost de a ajunge de la n la scop.

În afară de timpul mare calculator, algoritmul A* consumă 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.

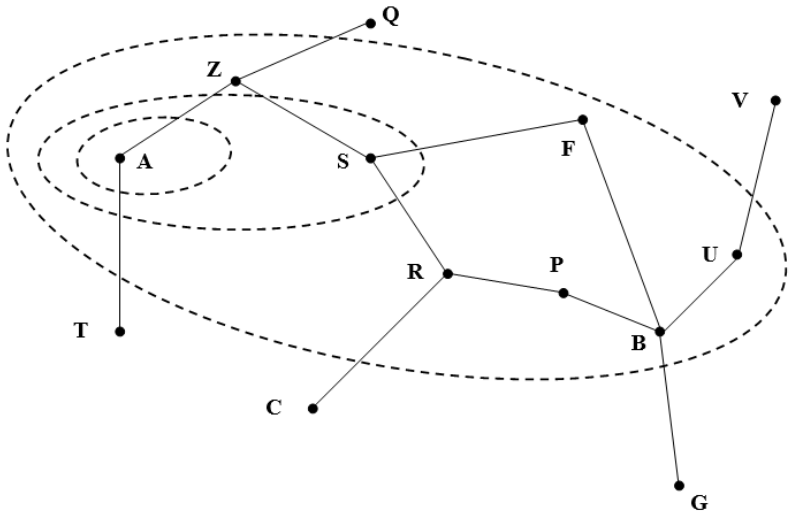


Fig 2.10

Algoritmul IDA* este 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 funcția euristică.

Implementarea căutarii 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 şi anume cel care se ocupă de alternativa cea mai promițătoare (adică alternativa corespunzătoare celei mai mici $\hat{f}$ - valori). Celelalte procese şteaptă 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 şi, atâta vreme cât acest buget nu este epuizat, procesul este activ. Pe durata activității sale, procesul îşi 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 oraşele 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 oraşe este etichetat cu lungimea sa; numărul din căsuța alăturată unui oraş reprezintă distanța 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 funcția $\hat{h}$ bazându-ne pe distanța în linie dreaptă între două oraşe. Astfel, pentru un oraş X, definim $\hat{f}(X) = \breve{g}(X) + \breve{h}(X) = \breve{g}(X) + dist(X,t)$, unde $dist(X,t)$ reprezintă distanța 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.

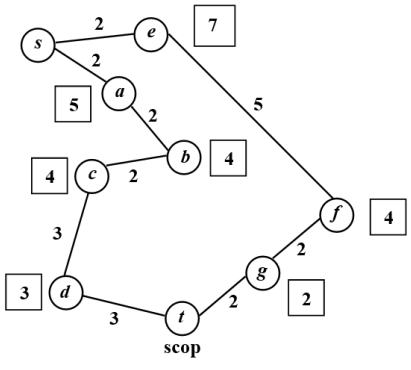


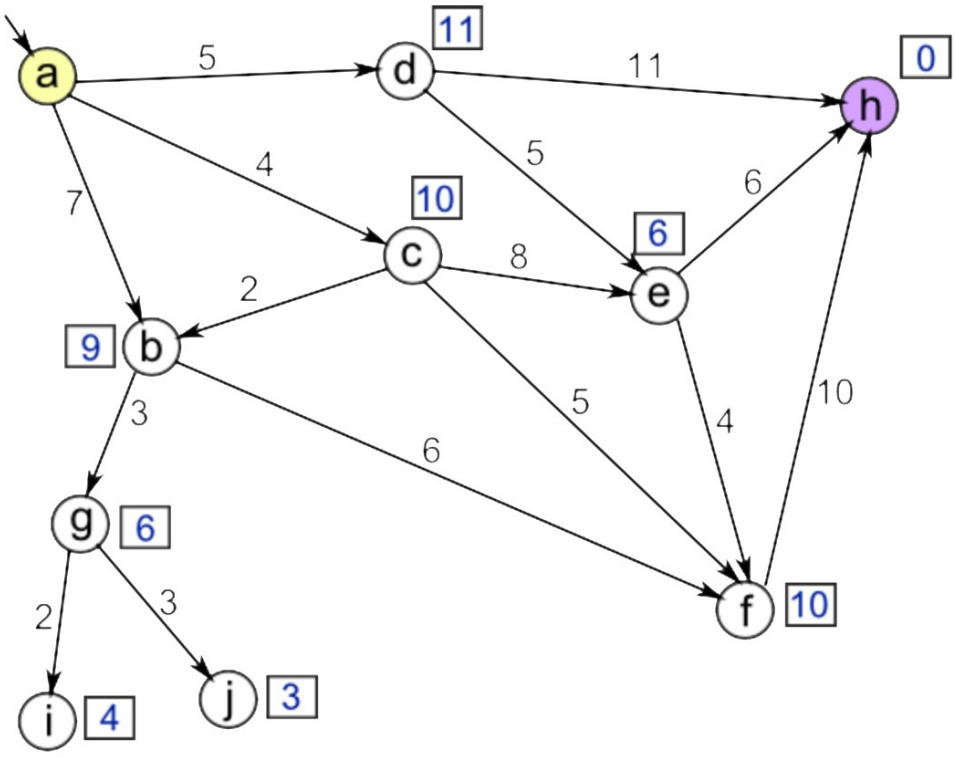
Fig 2.11

În stadiile 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, $\hat{f}(c) = \hat{g}(c) + \hat{h}(c) = 6 + 4 = 10$, $\hat{f}(e) = \hat{g}(e) + \hat{h}(e) = 2 + 7 = 9$ şi deci $\breve{f}(e) < \breve{f}(c)$. În acest moment, situația se schimbă: procesul P_2 devine activ, iar procesul P_1 intră în aşteptare. În continuare, $\breve{f}(c) = 10$, $\breve{f}(f) = 11$, $\breve{f}(c) < \breve{f}(f)$ şi deci P_1 devine activ şi P_2 intră în aşteptare. Pentru că

$\breve{f}(d) = 12 > 11$, procesul P_1 va reintra în aşteptare, iar procesul P_2 va rămâne activ până când se va atinge starea scop t.

Căutarea schițată mai sus porneşte din nodul inițial şi 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.

EXEMPLU



Output A*:

Pasul 1) Lista open initiala: [((a, h=inf), parinte=None, f=inf, g=0)]
Pasul 2) Lista closed initiala: []

==========

Pasul 3) Lista open nu e vida, urmeaza pasii repetitivi.
Noduri in lista open: [((a, h=inf), parinte=None, f=inf, g=0)]
Noduri in lista closed: []
Pasul 4) Extragem ((a, h=inf), parinte=None, f=inf, g=0) din lista open si il plasam in lista closed
Pasul 5) Nodul nu este scop deci il expandam.
Pasul 6) Succesorii nodului de expandat ((a, h=inf), parinte=None, f=inf, g=0) sunt
nod: (b, h=9), cost arc:7
nod: (c, h=10), cost arc:4
nod: (d, h=11), cost arc:5
Pasul 7) Procesez succesorii.
Lista open dupa ce au fost adaugati succesorii este: [((b, h=9), parinte=a, f=16, g=7) ((c, h=10), parinte=a, f=14, g=4) ((d, h=11), parinte=a, f=16, g=5)]
Pasul 8) Sortam lista open crescator dupa f:
Lista open dupa sortare: [((c, h=10), parinte=a, f=14, g=4) ((b, h=9), parinte=a, f=16, g=7) ((d, h=11), parinte=a, f=16, g=5)]
Pasul 9) Revenim la pasul 3

==========

Pasul 3) Lista open nu e vida, urmeaza pasii repetitivi.
Noduri in lista open: [((c, h=10), parinte=a, f=14, g=4) ((b, h=9), parinte=a, f=16, g=7) ((d, h=11), parinte=a, f=16, g=5)]
Noduri in lista closed: [((a, h=inf), parinte=None, f=inf, g=0)]
Pasul 4) Extragem ((c, h=10), parinte=a, f=14, g=4) din lista open si il plasam in lista closed

Pasul 5) Nodul nu este scop deci il expandam.
Pasul 6) Succesorii nodului de expandat ((c, h=10), parinte=a, f=14, g=4) sunt
nod: (b, h=9), cost arc:2
nod: (e, h=6), cost arc:8
nod: (f, h=10), cost arc:5
Pasul 7) Procesez succesorii.
Nodul ((b, h=9), parinte=a, f=16, g=7) se afla deja in open cu un g estimat (adancime) mai mare, asa ca il actualizez la (redirectionare pointeri):
((b, h=9), parinte=c, f=15, g=6)
Lista open dupa ce au fost adaugati succesorii este: [((b, h=9), parinte=c, f=15, g=6) ((d, h=11), parinte=a, f=16, g=5) ((e, h=6), parinte=c, f=18, g=12) ((f, h=10), parinte=c, f=19, g=9)]
Pasul 8) Sortam lista open crescator dupa f:
Lista open dupa sortare: [((b, h=9), parinte=c, f=15, g=6) ((d, h=11), parinte=a, f=16, g=5) ((e, h=6), parinte=c, f=18, g=12) ((f, h=10), parinte=c, f=19, g=9)]
Pasul 9) Revenim la pasul 3

==========

Pasul 3) Lista open nu e vida, urmeaza pasii repetitivi.
Noduri in lista open: [((b, h=9), parinte=c, f=15, g=6) ((d, h=11), parinte=a, f=16, g=5) ((e, h=6), parinte=c, f=18, g=12) ((f, h=10), parinte=c, f=19, g=9)]
Noduri in lista closed: [((a, h=inf), parinte=None, f=inf, g=0) ((c, h=10), parinte=a, f=14, g=4)]
Pasul 4) Extragem ((b, h=9), parinte=c, f=15, g=6) din lista open si il plasam in lista closed
Pasul 5) Nodul nu este scop deci il expandam.
Pasul 6) Succesorii nodului de expandat ((b, h=9), parinte=c, f=15, g=6) sunt
nod: (f, h=10), cost arc:6
nod: (g, h=6), cost arc:3
Pasul 7) Procesez succesorii.
Lista open dupa ce au fost adaugati succesorii este: [((d, h=11), parinte=a, f=16, g=5) ((e, h=6), parinte=c, f=18, g=12) ((f, h=10), parinte=c, f=19, g=9) ((g, h=6), parinte=b, f=15, g=9)]
Pasul 8) Sortam lista open crescator dupa f:
Lista open dupa sortare: [((g, h=6), parinte=b, f=15, g=9) ((d, h=11), parinte=a, f=16, g=5) ((e, h=6), parinte=c, f=18, g=12) ((f, h=10), parinte=c, f=19, g=9)]
Pasul 9) Revenim la pasul 3

==========

Pasul 3) Lista open nu e vida, urmeaza pasii repetitivi.
Noduri in lista open: [((g, h=6), parinte=b, f=15, g=9) ((d, h=11), parinte=a, f=16, g=5) ((e, h=6), parinte=c, f=18, g=12) ((f, h=10), parinte=c, f=19, g=9)]
Noduri in lista closed: [((a, h=inf), parinte=None, f=inf, g=0) ((c, h=10), parinte=a, f=14, g=4) ((b, h=9), parinte=c, f=15, g=6)]
Pasul 4) Extragem ((g, h=6), parinte=b, f=15, g=9) din lista open si il plasam in lista closed
Pasul 5) Nodul nu este scop deci il expandam.
Pasul 6) Succesorii nodului de expandat ((g, h=6), parinte=b, f=15, g=9) sunt
nod: (i, h=4), cost arc:2
nod: (j, h=3), cost arc:3
Pasul 7) Procesez succesorii.
Lista open dupa ce au fost adaugati succesorii este: [((d, h=11), parinte=a, f=16, g=5) ((e, h=6), parinte=c, f=18, g=12) ((f, h=10), parinte=c, f=19, g=9) ((i, h=4), parinte=g, f=15, g=11) ((j, h=3), parinte=g, f=15, g=12)]
Pasul 8) Sortam lista open crescator dupa f:
Lista open dupa sortare: [((j, h=3), parinte=g, f=15, g=12) ((i, h=4), parinte=g, f=15, g=11) ((d, h=11), parinte=a, f=16, g=5) ((e, h=6), parinte=c, f=18, g=12) ((f, h=10), parinte=c, f=19, g=9)]
Pasul 9) Revenim la pasul 3

==========

Pasul 3) Lista open nu e vida, urmeaza pasii repetitivi.
Noduri in lista open: [((j, h=3), parinte=g, f=15, g=12) ((i, h=4), parinte=g, f=15, g=11) ((d, h=11), parinte=a, f=16, g=5) ((e, h=6), parinte=c, f=18, g=12) ((f, h=10), parinte=c, f=19, g=9)]
Noduri in lista closed: [((a, h=inf), parinte=None, f=inf, g=0) ((c, h=10), parinte=a, f=14, g=4) ((b, h=9), parinte=c, f=15, g=6) ((g, h=6), parinte=b, f=15, g=9)]
Pasul 4) Extragem ((j, h=3), parinte=g, f=15, g=12) din lista open si il plasam in lista closed
Pasul 5) Nodul nu este scop deci il expandam.
Pasul 6) Succesorii nodului de expandat ((j, h=3), parinte=g, f=15, g=12) sunt
Pasul 7) Procesez succesorii.
Lista open dupa ce au fost adaugati succesorii este: [((i, h=4), parinte=g, f=15, g=11) ((d, h=11), parinte=a, f=16, g=5) ((e, h=6), parinte=c, f=18, g=12) ((f, h=10), parinte=c, f=19, g=9)]
Pasul 8) Sortam lista open crescator dupa f:
Lista open dupa sortare: [((i, h=4), parinte=g, f=15, g=11) ((d, h=11), parinte=a, f=16, g=5) ((e, h=6), parinte=c, f=18, g=12) ((f, h=10), parinte=c, f=19, g=9)]
Pasul 9) Revenim la pasul 3

==========

Pasul 3) Lista open nu e vida, urmeaza pasii repetitivi.
Noduri in lista open: [((i, h=4), parinte=g, f=15, g=11) ((d, h=11), parinte=a, f=16, g=5) ((e, h=6), parinte=c, f=18, g=12) ((f, h=10), parinte=c, f=19, g=9)]
Noduri in lista closed: [((a, h=inf), parinte=None, f=inf, g=0) ((c, h=10), parinte=a, f=14, g=4) ((b, h=9), parinte=c, f=15, g=6) ((g, h=6), parinte=b, f=15, g=9) ((j, h=3), parinte=g, f=15, g=12)]
Pasul 4) Extragem ((i, h=4), parinte=g, f=15, g=11) din lista open si il plasam in lista closed
Pasul 5) Nodul nu este scop deci il expandam.
Pasul 6) Succesorii nodului de expandat ((i, h=4), parinte=g, f=15, g=11) sunt
Pasul 7) Procesez succesorii.
Lista open dupa ce au fost adaugati succesorii este: [((d, h=11), parinte=a, f=16, g=5) ((e, h=6), parinte=c, f=18, g=12) ((f, h=10), parinte=c, f=19, g=9)]
Pasul 8) Sortam lista open crescator dupa f:
Lista open dupa sortare: [((d, h=11), parinte=a, f=16, g=5) ((e, h=6), parinte=c, f=18, g=12) ((f, h=10), parinte=c, f=19, g=9)]
Pasul 9) Revenim la pasul 3

==========

Pasul 3) Lista open nu e vida, urmeaza pasii repetitivi.
Noduri in lista open: [((d, h=11), parinte=a, f=16, g=5) ((e, h=6), parinte=c, f=18, g=12) ((f, h=10), parinte=c, f=19, g=9)]
Noduri in lista closed: [((a, h=inf), parinte=None, f=inf, g=0) ((c, h=10), parinte=a, f=14, g=4) ((b, h=9), parinte=c, f=15, g=6) ((g, h=6), parinte=b, f=15, g=9) ((j, h=3), parinte=g, f=15, g=12) ((i, h=4), parinte=g, f=15, g=11)]
Pasul 4) Extragem ((d, h=11), parinte=a, f=16, g=5) din lista open si il plasam in lista closed
Pasul 5) Nodul nu este scop deci il expandam.
Pasul 6) Succesorii nodului de expandat ((d, h=11), parinte=a, f=16, g=5) sunt
nod: (h, h=0), cost arc:11
nod: (e, h=6), cost arc:5
Pasul 7) Procesez succesorii.
Nodul ((e, h=6), parinte=c, f=18, g=12) se afla deja in open cu un g estimat (adancime) mai mare, asa ca il actualizez la (redirectionare pointeri):
((e, h=6), parinte=d, f=16, g=10)
Lista open dupa ce au fost adaugati succesorii este: [((e, h=6), parinte=d, f=16, g=10) ((f, h=10), parinte=c, f=19, g=9) ((h, h=0), parinte=d, f=16, g=16)]
Pasul 8) Sortam lista open crescator dupa f:
Lista open dupa sortare: [((h, h=0), parinte=d, f=16, g=16) ((e, h=6), parinte=d, f=16, g=10) ((f, h=10), parinte=c, f=19, g=9)]

Pasul 9) Revenim la pasul 3

==========

Pasul 3) Lista open nu e vida, urmeaza pasii repetitivi.
Noduri in lista open: [((h, h=0), parinte=d, f=16, g=16) ((e, h=6), parinte=d, f=16, g=10) ((f, h=10), parinte=c, f=19, g=9)]
Noduri in lista closed: [((a, h=inf), parinte=None, f=inf, g=0) ((c, h=10), parinte=a, f=14, g=4) ((b, h=9), parinte=c, f=15, g=6) ((g, h=6), parinte=b, f=15, g=9) ((j, h=3), parinte=g, f=15, g=12) ((i, h=4), parinte=g, f=15, g=11) ((d, h=11), parinte=a, f=16, g=5)]
Pasul 4) Extragem ((h, h=0), parinte=d, f=16, g=16) din lista open si il plasam in lista closed
Nodul extras din open este nod scop

---------- **Concluzie** ----------

Drum de cost minim: [((a, h=inf), parinte=None, f=inf, g=0) ((d, h=11), parinte=a, f=16, g=5) ((h, h=0), parinte=d, f=16, g=16)]