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*.

- Observaţie: $f(n_0) = h(n_0)$ 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*.

Observaţie: 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_o. Plasează n_o 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.

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*.

- Observaţii:
  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 $\hat{h}$.

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*.

- Observaţie: 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:

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

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*.

- Observatie: 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*.