Metode de ABORDARE a problemelor NP

Mihai Stroe

Problemele nedeterminist polinomiale ridică dificultăți destul de mari majorității elevilor care participă la concursurile de programare. În cadrul acestui articol vom prezenta câteva modalități de abordare a acestora.

Problemele care nu se pot rezolva perfect prin algoritmi de complexitate polinomială aparțin clasei *NP \ P*. În cele ce urmează, vom folosi denumirea ***NP*** pentru problemele de acest tip. Majoritatea observațiilor se pot aplica și problemelor pentru care, deși există metode polinomiale de rezolvare, acestea nu sunt cunoscute de către rezolvitor (de exemplu, cazul problemelor mai dificile de la concursuri).

Domeniul problemelor *NP* este deosebit de amplu; ne vom limita prezentarea asupra câtorva aspecte interesante ale acestuia. Expunerea presupune cunoașterea metodei *backtracking*.

Încadrarea problemei în clasa NP \ P

Există câteva probleme standard pentru care nu se cunosc algoritmi de rezolvare polinomiali. În cazul în care întâlniți una dintre acestea, în principiu nu are rost să căutați un algoritm eficient de rezolvare.

Atenție, aparențele pot fi însă înșelătoare și ceea ce poate părea o problemă *NP* clasică poate fi de fapt rezolvabilă, datorită introducerii unor condiții suplimentare.

De exemplu, problema colorării unui graf cu *K* culori, prezentată mai jos, este de fapt *NP*, dar pentru *K*=2 există algoritmi polinomiali. De asemenea, problema submulțimii de sumă dată se rezolvă prin metoda programării dinamice, dacă valorile implicate sunt numere întregi relativ mici (de exemplu, suma este cel mult 10000) și prin alte metode eficiente, dacă renunțăm la condiția determinării exacte a sumei.

Dacă reușim să demonstrăm că, rezolvând problema dată, am rezolva una dintre problemele "clasice" *NP-complete*, atunci problema este *NP-completă*. Pe scurt, o problemă este *NP-completă* dacă nu există un algoritm polinomial pentru rezolvarea ei, dar rezultatul poate fi verificat în timp polinomial.

De exemplu, problema determinării tuturor submulțimilor unei mulțimi nu este *NP*-completă (deși este *NP*) deoarece numărul submulțimilor crește exponențial în funcție de numărul elementelor submulțimii, deci rezultatul nu poate fi verificat în timp polinomial. Câteva dintre cele mai cunoscute probleme *NP* sunt următoarele:

Problema satisfiabilității formulei logice

Se consideră o expresie în care pot apărea variabile logice (valorile pot fi 0 sau 1) și operatorii logici de conjuncție, disjuncție și negație. Să se atribuie valori variabilelor astfel încât rezultatul evaluării expresiei să fie 1.

Problema submulțimii de sumă dată

Se consideră o mulțime *M* cu *n* elemente (numere reale strict pozitive) și un număr *D*. Să se găsească o submulțime *S* a mulțimii *M*, astfel încât suma elementelor lui *S* să fie mai mică sau egală cu *D* și cât mai apropiată de acesta.

Problema acoperirii cu mulțimi (set covering)

Se consideră o mulțime *M* și o mulțime *P* de submulțimi ale lui *M*. Să se selecteze o submulțime *S* a lui *P*, de cardinal minim, astfel încât fiecare element din *M* să se afle în cel puțin una din mulțimile din *P*.

Problema colorării

Se consideră un graf neorientat și un număr *K*. Să se coloreze graful cu *K* culori (sau cu un număr minim de culori), astfel încât oricare două noduri adiacente să fie colorate diferit.

Problema clicii

Se consideră un graf neorientat. Să se aleagă o submulțime de noduri, de cardinal maxim, astfel încât între oricare două noduri să existe muchie.

Problema acoperirii cu vârfuri

Se consideră un graf neorientat cu *N* vârfuri și *M* muchii. Să se determine o submulțime minimală *N'* a vârfurilor grafului astfel încât toate cele *M* muchii să aibă cel puțin o

extremitate care face parte din mulțimea *N'* (cu alte cuvinte, vârfurile din submulțimea aleasă trebuie să "acopere" toate muchiile).

Problema ciclului hamiltonian

Se consideră un graf neorientat (în cazul abordat aici) și o funcție distanță definită pe mulțimea nodurilor grafului, cu valori strict pozitive. Să se găsească un ciclu care trece prin fiecare nod exact o dată și are costul minim (de fapt, se va aborda varianta "și are costul cât mai mic"). Costul ciclului este suma distanțelor date de muchiile pe care le conține.

Problema jocului Perspico

Aceasta nu este o problema *NP* "clasică", dar o vom folosi în expunerea care urmează.

Se consideră o matrice cu patru linii și patru coloane care conține toate numerele cuprinse între 0 și 15.

Să se obțină configurația:

```
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15  0
```

folosind un număr minim de mutări. Mutările permise sunt interschimbări ale elementului 0 cu unul dintre elementele de pe pozițiile vecine pe orizontală și verticală.

Simplificarea problemei

De multe ori, problema poate fi simplificată înaintea aplicării algoritmului de rezolvare. După obținerea soluției pentru varianta simplificată, soluția pentru întreaga problemă se obține foarte rapid, printr-o metodă polinomială.

De exemplu, în cazul problemei colorării nodurilor unui graf cu *K* culori, există o simplificare dată de următoarea propoziție:

"*Problema pentru un graf G în care există un nod P cu gradul mai mic decât K se poate rezolva ușor dacă rezolvăm problema pentru graful G \ {P}.*"

Demonstrația este evidentă. Pentru a obține soluția pentru graful *G* se obține soluția pentru graful *G* \ {*P*}, după care se colorează nodul *P* cu o culoare care nu a fost atribuită nici unui vecin. Datorită faptului că numărul culorilor este mai mare decât gradul nodului *P*, colorarea este posibilă.

Astfel, pașii algoritmului de rezolvare sunt următorii:

- se elimină din graf toate nodurile cu gradul mai mic decât *K*;
- se repetă pasul anterior până când nu mai exista în graf noduri cu gradul mai mic decât *K* (după eliminările de la primul pas, gradele altor noduri pot deveni mai mici decât *K*);
- se rezolvă problema pentru graful rămas (și pentru problema particulară în care gradele tuturor nodurilor sunt cel puțin egale cu *K*);
- se construiește soluția problemei inițiale, colorând nodurile eliminate în ordinea inversă ordinii eliminării.

Alegerea unei modalități de rezolvare

Înaintea abordării unei probleme *NP* este necesară o analiză atentă. Anumite probleme se pretează mai bine anumitor tipuri de rezolvări.

Cele mai importante modalități de abordare pentru probleme *NP* sunt:

- explorarea în lățime a spațiului soluțiilor;
- explorarea în adâncime a spațiului soluțiilor;
- metode euristice;
- metode bazate pe nedeterminism;
- algoritmi genetici;
- algoritmi pseudo-polinomiali;
- scheme de aproximare;
- rețele neuronale.

Metodele de rezolvare pot fi combinate. De exemplu, înaintea aplicării *backtracking*-ului se poate căuta o soluție printr-o metodă euristică; rezultatul obținut va fi folosit pentru evitarea explorării unor cazuri neinteresante.

Câteva dintre avantajele și dezavantajele unor metode sunt descrise în continuare.

Eliminarea situațiilor neinteresante

Majoritatea abordărilor prezentate pot fi îmbunătățite folosindu-se o tehnică de eliminare a unor elemente din spațiul soluțiilor (***pruning***).

Eliminarea poate fi realizată euristic (se estimează, rapid, dacă elementul respectiv "promite" să conducă la o soluție bună și, dacă nu, este eliminat), dar există și probleme pentru care eliminarea este sigură.

De exemplu, în cazul unor probleme de minim, poate există o funcție care aproximează inferior costul transformării configurației curente în soluție a problemei. Această funcție este numită ***euristică optimistă***. Dacă suma dintre costul configurației și funcția aceasta este mai mare decât costul celei mai bune soluții obținute anterior în cadrul algoritmului, configurația curentă poate fi eliminată, deoarece nu va genera o soluție mai bună.

Cu cât funcția aproximează mai bine costul transformării, cu atât algoritmul este mai performant, deoarece elementele neperformante sunt excluse, împreună cu toate elementele (tot neperformante) care ar fi fost introduse de ele.

Pentru problema jocului *Perspico*, o funcție *euristică optimistă* bună este suma *distanțelor Manhattan* de la poziția fiecărui element din matrice până la poziția sa finală (fără a aduna și distanța pentru elementul 0). *Distanța Manhattan* dintre două puncte este suma dintre distanța pe orizontală și distanța pe verticală.

Pentru problema ciclului hamiltonian, cea mai bună *euristică optimistă* cunoscută se bazează pe calculul arborelui minim de acoperire pentru nodurile care încă nu au fost selectate. Se pot alege și alte funcții euristice pentru eliminarea unor configurații, de exemplu soluția dată de un *greedy* rapid (se construiește un drum alegând la fiecare pas muchia de cost minim), dar aceasta nu este *euristică optimistă*.

Alegerea ordinii explorării soluțiilor

Metodele bazate pe explorarea în lățime a grafului configurațiilor și cele bazate pe explorarea în adâncime utilizează deseori o anumită ordine în care sunt explorate soluțiile. Problema pe care se va baza expunerea este cea a jocului *Perspico*.

În cazul explorării în lățime, configurațiile se introduc într-o coadă. O aceeași configurație nu va fi introdusă de două ori. Pentru aceasta se folosesc structuri de date avansate care permit căutarea rapidă, cum ar fi tabelele de dispersie. Dacă numărul configurațiilor este redus, se poate folosi o structură de selecție simplă (o mulțime sau un vector de valori logice). Principalul dezavantaj constă în depășirea rapidă a capacității memoriei disponibile, dacă spațiul configurațiilor este mare.

În cazul explorării în adâncime, metoda cea mai des folosită este *backtracking*. Nu prezentăm pe larg metoda, deoarece o considerăm cunoscută.

Se folosește o stivă. La fiecare pas se introduce în stivă una dintre configurațiile vecine celei din vârful stivei; după ce căutarea pentru aceasta s-a terminat, se va introduce o altă configurație vecină etc. Memoria disponibilă nu este o problemă (structura de bază este o stivă de configurații), în schimb există posibilitatea explorării repetate a aceleiași stări. În plus, soluțiile foarte apropiate de configurația de plecare pot fi pierdute.

Există o metodă care înlătură acest ultim neajuns, depășind și limitarea de memorie dată de explorarea în lățime. Tehnica se numește ***Depth First Search with Iterative Deepening*** (pe scurt ***DFSID***) și constă în parcurgerea în adâncime a regiunilor din graful configurațiilor apropiate de configurația de start. Este explorată inițial mulțimea configurațiilor aflate la distanța 1, apoi la distanța 2, 3 etc. de configurația inițială (aici prin "distanță" înțelegem lungimea drumului minim în graful configurațiilor; pentru jocul *Perspico* aceasta reprezintă numărul de mutări efectuate). Algoritmul se încheie la găsirea unei soluții bune sau la depășirea timpului acordat. Evident că *DFSID* duce la explorarea repetată a unor noduri ale grafului, dar acesta nu este un neajuns foarte mare, comparativ cu avantajele aduse.

De exemplu, dacă la fiecare introducere în stivă ar exista exact trei opțiuni, atunci *DFSID* ar genera, la pasul K, $3^1 + 3^2 + 3^3 + \ldots + 3^K$ configurații. Numărul de configurații generate la toți pașii precedenți este net inferior (demonstrația se face prin inducție și este foarte simplă), deci performanța, din punct de vedere al vitezei, scade de mai puțin de două ori față de cea a explorării în lățime.

Tehnica *DFSID* se referă la modificarea adâncimii stivei în *backtracking*, dar există și alte variante ale acestui stil de abordare. De exemplu, în problema ciclului hamiltonian se poate încerca o abordare în care nu se acceptă soluții cu un cost mai mare decât K. Dacă nu este găsită nici o soluție, se procedează la incrementarea lui K cu un anumit pas și se reia explorarea.

Toate aceste abordări pot fi îmbunătățite prin tehnicile de *pruning* amintite anterior. Alegerea unei metode trebuie să fie, de obicei, completată de găsirea unor tehnici de eliminare a nodurilor neinteresante.

Există o variantă mai bună a explorării în lățime, și anume algoritmul ***A****, folosit pentru problemele de minim. În cazul acestui algoritm, se folosește tehnica de *pruning* bazată pe *euristica optimistă*.

La fiecare pas, sunt introduși în coadă succesorii celei mai promițătoare configurații. Astfel, pentru fiecare configurație se calculează suma dintre costul ei și *euristica optimistă*. Configurația cu această sumă minimă va fi expandată. Algoritmul garantează obținerea soluției optime; sunt necesare structuri de date care permit găsirea eficientă a minimului.

Pentru problema jocului *Perspico* se va folosi *euristica optimistă* descrisă anterior. Se observă că ea poate fi calculată foarte ușor la trecerea de la o configurație la alta (practic se schimbă distanța unui singur element din matrice față de poziția sa finală).

În general, un program care implementează *A** este mult mai complex decât unul care implementează *DFSID*. În plus, memoria necesară este exponențială pe cazurile defavorabile. De aceea, cu toate că explorează mai multe configurații, *DFSID* combinat cu *pruning* este mai indicat în multe cazuri. Avantajele *DFSID* au fost tratate pe larg în articolul ***Branch&Bound și backtracking***, publicat de *Ovidiu Gheorghioiu* în *GInfo* 9/2 (februarie 1999).

Particularitățile problemelor

În multe cazuri, particularitățile problemelor pot permite optimizări interesante. Vom examina modalități de abordare pentru două probleme *NP*.

Recomandăm cititorului ca, după lecturarea enunțului problemelor, să întrerupă citirea acestui articol și să se gândească puțin la modalitățile de rezolvare și la posibilele optimizări. Observațiile pot fi apoi comparate cu cele ale autorului. Deși nu toate observațiile autorului sunt implementabile în timp de concurs, ele funcționează bine pentru seturi de date de intrare mari.

Probleme

Se consideră un graf neorientat cu cel mult 100 de noduri și 1000 de muchii. Graful se citește dintr-un fișier; pe prima linie se află numărul de noduri N și numărul de muchii M, după care urmează cele M muchii, câte una pe linie. Toate nodurile au gradul cel puțin 1.

1) Să se selecteze o submulțime S a mulțimii nodurilor, de cardinal minim, astfel încât fiecare nod care nu este în S să aibă cel puțin un vecin în S (din mulțimea de noduri S să se "vadă" toate nodurile).
2) Să se selecteze o submulțime S a mulțimii nodurilor, de cardinal minim, astfel încât fiecare muchie din graf să aibă cel puțin unul din nodurile incidente în S (din mulțimea de noduri S să se "vadă" toate muchiile).

Fiecare mulțime S se va afișa prin numerele de ordine ale elementelor. Timpul de execuție nu trebuie să depășească o secundă.

În continuare vom prezenta modul de rezolvare al celor două cerințe.

În majoritatea cazurilor, problema 2) este mai simplă decât 1). Se observă că, dacă un nod *X* nu se află în *S*, toți vecinii lui se vor afla în mod obligatoriu în *S*. În caz contrar, o parte din muchiile incidente lui *X* nu vor fi "văzute" din *S*.

Această observație conduce la o soluție *backtracking*. Nodurile sunt procesate în ordinea descrescătoare a gradelor; la o iterație este introdus în stivă fie nodul curent (ceea ce elimină multe muchii introducând un singur nod în *S*), fie toți vecinii săi (operație care micșorează mult graful, dar introduce noduri suplimentare în *S*). Dacă numărul de noduri din *S* este mai mare decât un optim găsit anterior, căutarea pe varianta curentă este abandonată.

Această abordare poate fi mult îmbunătățită. De exemplu, nodurile de grad 1 nu vor fi selectate în soluția optimă (sau, dacă ar fi, o soluție la fel de bună se obține înlocuindu-le cu vecinii lor), deci pot fi eliminate din graf înainte de începerea căutării și nodurile adiacente lor vor fi introduse în soluție (ceea ce duce la eliminarea unor muchii "văzute" de aceste noduri și eventual la apariția unor alte noduri de grad 1 etc.). Excepție fac perechile de noduri de grad 1, caz în care un nod din fiecare pereche este eliminat, iar celălalt este selectat automat.

Alte posibile optimizări:

- la un moment dat, un nod care are toți vecinii în *S* (introduși anterior) nu va fi introdus în *S* pentru că nu aduce nimic în plus, în schimb conduce la o soluție mai slabă;
- dacă graful nu este conex, problema se va rezolva separat pentru fiecare componentă conexă; optimizarea se poate aplica și pe parcurs (în momentul deconectării grafului prin eliminarea unor noduri și muchii);
- se poate folosi o *euristică optimistă* pentru eliminarea unor variante. De exemplu, dacă în graf au mai rămas N_1 noduri și M_1 muchii, este necesar ca suma gradelor nodurilor care vor fi introduse în soluție să fie cel puțin M_1. Se aleg din cele N_1 noduri rămase nodurile cu gradele cele mai mari (relativ la cele M_1 muchii), până când suma gradelor ajunge să fie cel puțin M_1 (chiar dacă aceste noduri nu "văd" cele M_1 muchii). Dacă numărul de noduri introduse, plus numărul de noduri existente, depășesc optimul obținut anterior, varianta curentă trebuie abandonată.

Presupunând că nu se mai introduc optimizări în căutare, pasul următor îl constituie scrierea unui program care să execute operațiile descrise. Optimizările de cod contează foarte mult, deoarece vor fi examinate mai multe variante și șansele de a se obține o soluție mai bună cresc.

Problema 1) este un caz particular al problemei acoperirii cu mulțimi (*set covering*), cunoscută ca fiind *NP-completă*. Astfel, o mulțime este formată dintr-un nod și din vecinii lui. Acest fapt nu este suficient pentru a demonstra că 1) este *NP*, dar în cazul de față problema 1) este într-adevăr *NP*. Mai mult, particularitățile problemei 1) fac ca anumite optimizări cunoscute pentru *set covering* să funcționeze foarte bine. Din aceste motive putem trata 1) printr-o abordare care rezolvă problema mai generală. Este de reținut faptul că o astfel de abordate (rezolvarea unei probleme mai generale) nu este recomandată în majoritatea cazurilor, deoarece se pierde informația specifică problemei.

În continuare vom discuta problema acoperirii cu mulțimi. Începem cu două observații fundamentale:

- dacă o mulțime este inclusă în alta, ea nu va intra în soluția finală;
- dacă un element din mulțimea {1, 2, …, *N*} este conținut într-o singură mulțime, aceasta va face parte din soluția finală.

În plus, intuitiv este avantajos să dăm șanse mai mari mulțimilor cu multe elemente să facă parte din soluția finală. Din nefericire, algoritmul *greedy* care ar rezulta din această ultimă observație nu dă întotdeauna soluția optimă.

Observațiile conduc la o rezolvare *backtracking*, care poate fi implementată recursiv. Aceasta funcționează astfel:

- pasul "elimină": se elimină toate mulțimile care sunt incluse în alte mulțimi (atenție la cazul cu mulțimi egale);
- pasul "găsește": se caută un element conținut într-o singură mulțime; dacă un astfel de element este găsit, mulțimea respectivă este selectată, atunci se continuă cu apelarea *back(k+1)* și se iese din procedura recursivă;
- dacă pasul "găsește" eșuează, se va apela *back(k+1)* selectând, pe rând, fiecare mulțime rămasă, în ordinea descrescătoare a numărului de elemente;
- la apelul *back(k+1)* se va lucra cu mulțimile rămase, din care se elimină toate elementele care se găsesc în mulțimi deja selectate (ceea ce poate duce la succese noi ale pașilor "elimină" și "găsește").

Observație

Utilitatea abordării problemelor *NP* nu poate fi pusă la îndoială. O soluție bună pentru problema 1) poate duce, de exemplu, la scăderea costurilor necesare pentru deschiderea unei rețele de *fast food*-uri care să acopere integral piața într-o anumită zonă etc.

Concluzii

Problemele *NP* apar în numeroase domenii. Deși se știe că timpul necesar pentru rezolvare este exponențial, aceasta nu înseamnă că studiul lor ar trebui abandonat. Alternativa este de a găsi soluții practice cât mai eficiente.

Pentru abordarea cu succes a unei probleme *NP* este necesară o analiză atentă, cunoașterea tuturor opțiunilor și alegerea celor mai bune soluții. Aceasta poate implica testări complexe ale mai multor programe, folosind date de test cât mai apropiate de cele care vor apărea pe parcursul utilizării soluției finale.

Mihai Stroe este student în anul V la Universitatea Politehnica din București și poate fi contactat prin e-mail la adresa mihai_stroe@yahoo.com.