COMPLEXITATEA CALCULULUI

Teoria complexităţii constituie un domeniu foarte important al teoriei algoritmilor care investighează rezolvabilitatea individuală a problemelor sub anumite restricţii de resurse.

Teoria complexităţii calculului cuprinde:

1) Un studiu analitic al complexităţii unor clase de probleme în raport cu diverse tipuri concrete de maşini matematice care le rezolvă;

2) Un studiu calitativ al complexităţii algoritmilor care rezolvă astfel de probleme.

Mai precis, ***complexitatea calculului*** se bazează pe legăturile existente între ***problemele rezolvabile algoritmic*** şi ***resursele de calcul***.

Înainte de a trece la analiza unui algoritm, trebuie să se cunoască o tehnologie de implementare a acestuia care va include un model pentru resursele sale şi costurile corespunzătoare. Aşadar, ***resursele de calcul*** sunt introduse prin definirea unui model de calcul care, de regulă, reprezintă o maşină. În acest sens pot fi utilizate: ***maşini Turing de diverse tipuri, maşini cu acces aleatoriu (random-acces machine, RAM), circuite booleene*** etc. Resursele tipice sunt: ***timp, spaţiu, benzi, capete de citire/scriere, nivele într-un circuit*** etc.

Pentru tratarea celor două elemente fundamentale, ***problemă şi maşină,*** considerăm trei nivele abstracte în complexitatea calculului.

a) Un nivel înalt care pune în evidenţă *complexitatea abstractă* a calculului, cunoscută sub numele de „***teoria Blum de complexitate a calculului“***, care este independentă atât de problemă cât şi de maşină. O astfel de teorie este construită pe o enumerare efectivă a tuturor funcţiilor parţial recursive, o enumerare a aşa numitelor funcţii „ cost-măsură “ şi două axiome de bază (axiomele lui Blum).

b) Un nivel mediu care pune în evidenţă *complexitatea structurală,* independentă de problemă dar dependentă de maşină. Scopul unei astfel de complexităţi este de a studia capabilitatea unor modele de calcul concrete pentru resurse sau combinaţii de resurse, independent de problema propusă.

c) Un nivel inferior ce indică o *complexitate concretă* care depinde atât de maşină cât şi de problemă. Scopul său este de a detecta o bună evaluare, cu un cost pe cât posibil minim, folosind o resursă (sau combinaţii de resurse) într-un model precizat pentru rezolvarea problemei.

În general, dificultatea unei probleme P se poate măsura prin resursele de calcul limitate asociate algoritmului secvenţial care o rezolvă. Aceste resurse sunt ***timpul şi spaţiul de memorie*** necesare pentru execuţia algoritmului respectiv. O astfel de măsură se poate evalua în diverse moduri. În termenii maşinii Turing, ca model de calcul, complexitatea temporală este stabilită de numărul de deplasări necesare realizării calculului respectiv, iar pentru un computer digital, de numărul de ciclări ale maşinii sau timpul real solicitat de maşină pentru acel calcul. În ceea ce priveşte complexitatea spaţială, pentru maşina Turing, ea constă în numărul de celule (pătrate) ale benzii folosite în calcul, iar pentru computer, în numărul de bytes utilizaţi.

Prin urmare, ideea de ***complexitate a unui algoritm*** poate fi privită sub două aspecte:

a) static, prin structura sa.

b) dinamic, prin durata execuţiei sale.

Complexitatea spaţială se exprimă prin dimensiunea spaţiului de memorie necesar algoritmului, independent de comportarea pe care o are el pe parcursul execuţiei.

Complexitatea temporală se exprimă prin timpul necesar execuţiei sale. Axiomele lui Blum, de exemplu, pun în evidenţă măsuri de complexitate dinamică cu consecinţe ca: teorema corelării măsurilor, teorema accelerării, teorema lacunei etc.

Există numeroase probleme de optimizare în reţele de transport, informatică, electronică, logistica construcţiilor, teoria comunicaţiilor (în codurile corectoare de erori), criptografie etc. Pentru algoritmii de rezolvare a acestor probleme este obligatoriu un studiu al complexităţii lor. Complexitatea algoritmilor relativ la coduri, reprezentarea compactă a mesajelor în vederea transmiterii lor prin diverse canale, permite o analiză asupra eficacităţii metodelor de codificare/decodificare. Relativ la criptografie, studiul complexităţii implică elaborarea unor tehnici care să permită cifrarea mesajelor sub o formă care să le asigure, pe cât posibil, inviolabilitatea.

Analiza unui algoritm conduce la stabilirea eficienţei sale şi implică, în principal, ideea de complexitate pentru că ea prevede resursele solicitate de algoritmul respectiv. Stabilirea complexităţii unui algoritm este necesară, pe de o parte, datorită faptului că mulţi algoritmi sunt eliminaţi de practică deoarece necesită un timp de execuţie şi un spaţiu de memorie foarte mari, iar pe de altă parte, se caută algoritmi cât mai performanţi în

vederea rezolvării cu calculatorul a unor probleme destul de complexe impuse de practică. Un calculator, oricât de performant ar fi, este în esenţă un automat finit, deci el are posibilităţi limitate. Performanţa unui algoritm se stabileşte conform unui criteriu care, de cele mai multe ori, îl constituie timpul de execuţie al algoritmului respectiv.

Având în vedere faptul că pentru seturi de date de intrare diferite acelaşi algoritm foloseşte timpi de execuţie diferiţi este necesar a lua în consideraţie:

• ***timpul în cazul cel mai favorabil*** (durata minimă pentru execuţia algoritmului);

• ***timpul mediu*** (raportul dintre suma timpilor necesari pentru toate seturile de date posibile şi numărul acestor seturi);

• ***timpul în cazul cel mai defavorabil*** (durata maximă pentru execuţia algoritmului).

Acesta din urmă oferă o margine superioară a timpului de execuţie pentru toate intrările de o dimensiune fixă şi reprezintă situaţia cea mai indicată pentru căutarea de informaţii într-o bază de date.

Să considerăm implementarea unui algoritm oarecare care să primească intrări de dimensiuni diferite. La o intrare de dimensiune n vom asocia o măsură $f(n)$ de folosire a resurselor sale care, de regulă, trebuie să se minimizeze. În mod normal, $f(n)$ va reprezenta timpul solicitat de algoritm pentru intrarea de dimensiune n în cazul cel mai defavorabil sau mediu (la alegerea noastră). Există o serie de factori care complică mai mult sau mai puţin calculul exact al lui $f(n)$.

De exemplu:

a) Timpul solicitat pentru execuţia unei instrucţiuni individuale dintr-un program, cu aceeaşi instanţă şi pe aceeaşi maşină, poate varia.

b) Poate exista o mare variaţie de folosire a resurselor pentru intrări de o anume dimensiune constantă n.

c) Cazul cel mai defavorabil poate fi mai rar întâlnit în practică, iar cazul mediu să fie nereprezentativ.

d) Unii algoritmi se pot executa mai bine pe anumite clase de intrări.

În acest sens se impun câteva sugestii, cum ar fi:

- Identificarea operaţiilor abstracte folosite de algoritm. Pentru a obţine o maximizare a independenţei de maşină este necesară o analiză a acestor operaţii abstracte, pentru că resursele solicitate de o mare parte din ele vor fi neimportante. Unele din ele sunt utilizate o singură dată la iniţializare.

- Un algoritm, în general, posedă cel puţin un ciclu. Acesta trebuie identificat pentru că instrucţiunile din interiorul său se execută mult mai des şi ele vor juca un rol important în analiza timpului de execuţie a algoritmului. Prin contorizarea acestor instrucţiuni se determină o margine superioară corespunzătoare pentru valoarea timpului de execuţie în cazul cel mai defavorabil şi, dacă este posibil, o configurare a cazului mediu. Această limită superioară este necesară pentru că, în mod practic, nu poate fi găsită o valoare exactă a timpului de execuţie în cazul cel mai defavorabil.

- Timpul necesar unui algoritm pentru rezolvarea unei probleme cu o instanţă dată, reprezintă numărul operaţiilor elementare (primitive) efectuate de algoritm pentru acea instanţă. Se acceptă o astfel de interpretare deoarece se presupune că execuţia unei astfel de operaţii se produce într-un timp constant şi nu depinde de mărimea operanzilor şi nici de timpul de memorare a rezultatului. De aceea resursa timp asociată se analizează independent de sistemul de calcul pe care se implementează algoritmul respectiv.

- Deşi progresele tehnologice actuale fac ca necesarul de memorie să treacă pe un plan secund, există totuşi situaţii în care spaţiul de memorie utilizat este folosit ca argument în stabilirea unor limite inferioare pentru timpul de execuţie.

- Viteza de calcul ce caracterizează actuala generaţie de calculatoare are drept consecinţă faptul că problema timpului de execuţie se pune, în general, pentru valori foarte mari ale dimensiunii intrării. Aceasta conduce la ideea unei analize a termenului dominant (cel care tinde cel mai repede la infinit) din formula de exprimare a numărului de operaţii elementare executate de către algoritm.

Dimensiunea informaţiilor prelucrate în majoritatea problemelor pe care le întâlnim în practică nu rămâne constantă.

Pentru a înţelege noţiunea de ”procedeu mecanic de calcul“ au fost propuse numeroase modele de calcul formal. Conform celebrei teze a lui Church - tot ceea ce este intuitiv calculabil cu ajutorul unui algoritm este calculabil indiferent de modelul formal de calcul - aceste modele sunt echivalente. În mod clasic, în teoria complexităţii, se consideră drept model maşina Turing care prezintă dublul avantaj de a constitui un model de program pentru calculator, dar şi un cronometru pentru timpul de execuţie al său. Conform acestui model,

complexitatea unui algoritm **A**, pentru orice dimensiune n a intrării, este o funcţie $f(n)$ ce exprimă timpul maxim de execuţie a algoritmului.

În descrierea algoritmilor, cel mai convenabil model este modelul RAM având în vedere şi cele mai importante clase de complexitate studiate (timp şi spaţiu polinomial).

Există câteva funcţii tipice de exprimare a timpului de execuţie a unui algoritm. Putem obţine, în mod obişnuit, o astfel de funcţie printr-o relaţie de recurenţă.

Cel mai adesea se va întâlni funcţia de forma

f*(*n*) = *cg* (*n*) + *tm unde $c > 0$ este o constantă, n reprezintă dimensiunea intrării, iar *tm* este un "termen mic" care este semnificativ doar pentru valori mici ale lui n sau pentru algoritmi sofisticaţi.

Funcţia $g(n)$ poate fi:

• constantă (algoritmul se execută în timp constant);
• log n (algoritmul se execută în timp log);
• n^m ($m = 0, 1, 2\ldots$)(algoritmul se execută în timp polinomial);

Cazuri particulare:

a) n (algoritmul se execută în timp liniar);
b) nlog n (algoritmul se execută în timp liniar log);
c) n^2 (algoritmul se execută în timp pătratic);
d) n^3 (algoritmul se execută în timp cubic);
• $n!$ (algoritmul se execută în timp factorial);
• k^n ($k > 1$, constantă) (algoritmul se execută în timp exponenţial).

Exemplul 1.

Să considerăm un algoritm care efectuează operaţii de punere/extragere a unei date de 32 biţi efectuate pe o stivă cu elemente întregi. Astfel de operaţii vor necesita un timp constant. Un algoritm care se execută într-un timp constant mic se zice că este ***"perfect".***

Exemplul 2.

Să presupunem un algoritm a cărui intrare constă dintr-un şir de n caractere. El poate prelucra intrarea sa caracter cu caracter, solicitând acelaşi timp pentru fiecare astfel de caracter. În acest caz, $f(n) = n$. Un astfel de algoritm care se execută în timp liniar se zice că este ***"excelent".***

Exemplul 3.

Un algoritm poate cicla direct intrarea sa formată din numere întregi sub forma

$$f(n) = n + f(n - 1).$$

Iterând această relaţie se obţine $f(n) = n + f(n - 1) = n + ((n-1) + f(n - 2)) =$.....

$= n + (n - 1) + (n - 2) + ... + 2 + f(1) = n^2/2 + n/2 + k$, unde k este o constantă. Pentru un n suficient de mare, k şi $n/2$ sunt mici faţă de n^2. Un astfel de algoritm se execută într-un timp pătratic.

Un algoritm care solicită o prelucrare a celor n^2 perechi de caractere ce corespund unei intrări care constă dintr-un şir de n caractere, solicită tot un timp pătratic de execuţie.

Pentru intrări de dimensiuni mari algoritmii care se execută în timp pătratic vor evolua mai lent.

Exemplul 4.

Un algoritm poate prelucra, la fiecare pas, jumătate din intrarea sa. Aşadar,

$$f(n) = f(n/2) + 1$$

(cu aproximare în cazul în care n nu este o putere naturală a lui 2).

Fie $n = 2^x$. Atunci, $f(n) = f(2^x) = 1 + f(2^{x-1}) = 1 + (1 + f(2^{x-2})) = ... = x + f(2^0) = x + k$, unde k este o constantă.

În acest caz, $f(2^x)$ este aproximativ x, iar $f(n)$ este aproximativ $\log_2 n$. Un astfel de algoritm se execută în timp ***log.*** Aceşti algoritmi au o creştere foarte lentă şi sunt foarte buni în practică.

Este foarte cunoscut algoritmul "Divide et Impera" folosit în Quicksort, Mergesort etc. Pentru acest algoritm, $f(n) = n + 2f(n/2)$.

Pentru $n = 2^x$, folosind un mic artificiu, se obţine:

$f(2^x)/2^x = 2^x/2^x + 2*f(2^x/2)/2^x = 1 + f(2^{x-1})/2^{x-1} = 1 + (1 + f(2^{x-2})/2^{x-2}) = ...= x + f(2^0)/2^0 =$

$= x+k$, unde k este o constantă.

Rezultă că $f(2^x) = 2^x(x + k) = 2^x x + tm$. Prin urmare, $f(n) = n\log_2 n + tm$. În concluzie, algoritmul respectiv se execută în timp ***liniar log.***

Exemplul 5.

Să presupunem că intrarea unui algoritm constă într-o mulţime de n numere întregi. Se cere să se găsească o submulţime a acestei mulţimi cu proprietatea că suma elementelor sale este nulă.

Printr-o cercetare exhaustivă, în cazul cel mai defavorabil, se vor verifica toate cele 2^n submulţimi posibile. Un astfel de algoritm se va executa într-un timp exponenţial.

Exemplul 6.

Un algoritm, prin care se cere obţinerea tuturor anagramelor corespunzătoare unui cuvânt format din n caractere la intrare, se va executa într-un timp factorial, pentru că există $n!$ astfel de posibilităţi.

Funcţia $n!$ creşte aproximativ cu aceeaşi viteză ca n^n, dar mai repede decât 2^n.

În 1964, Cobham a introdus clasa P a ***problemelor rezolvabile algoritmic în timp polinomial***, adică o problemă ce se rezolvă printr-un algoritm **A** care pentru orice număr natural n, funcţia sa de complexitate este mărginită superior de un polinom cunoscut $p(n)$ în variabila n, adică $f(n) \leq kp(n)$, unde k este o constantă nenulă. În această clasă se întâlnesc aşa numitele ***probleme "facile***". Se mai spune despre un algoritm de complexitate polinomială că reprezintă rezultatul unei cunoaşteri detaliate a problemei pe care o rezolvă.

Despre un algoritm care nu este de complexitate polinomială se zice că se comportă "***exponenţial".*** În aceeaşi clasă a algoritmilor exponenţiali sunt încadraţi şi algoritmii de complexitate $n\log n$ deşi nu corespund în sensul strict matematic al noţiunii respective. Având în vedere viteza de creştere sau ordinul de creştere a timpului de execuţie în raport cu n a funcţiilor exponenţiale faţă de cele polinomiale, în general, algoritmii exponenţiali sunt inutilizabili în practică. Despre un algoritm de complexitate exponenţială se spune că este o variantă a unei enumerări totale a căilor de identificare a soluţiilor problemei respective.

Există unii algoritmi cu comportare exponenţială care, pentru valori relativ mici ale lui n, sunt mai eficienţi decât cei alternativi cu comportare polinomială. De asemenea, unii algoritmi exponenţiali se comportă acceptabil pentru unele probleme particulare (vezi metoda simplex de rezolvare a problemelor de programare liniară).

În general, despre o problemă pentru care s-a dovedit că nu există un algoritm de complexitate polinomială pentru rezolvarea ei, adică nu este ***tractabilă,*** se spune că este ***intractabilă.*** Dificultatea unei astfel de probleme trebuie remarcată prin faptul că

timpul necesar pentru găsirea unei soluţii este de ordin exponenţial, adică funcţia nu se poate exprima printr-o expresie care să poată fi mărginită superior de un polinom în variabila n.

Dacă se are în vedere ca model, maşina de calcul în paralel (capabilă de a efectua simultan un număr relativ mare de calcule independente), ***clasa problemelor dificile*** (cele care nu sunt facile) se poate împărţi în două subclase:

a) subclasa problemelor pentru care nu există algoritmi nedeterminişti de complexitate polinomială pentru rezolvarea lor;

b) subclasa problemelor pentru care există algoritmi nedeterminişti de complexitate polinomială pentru rezolvarea lor (***NP***).

Trebuie observat faptul că majoritatea problemelor care par a fi dificile admit algoritmi nedeterminişti de complexitate polinomială pentru rezolvarea lor. Noţiunea de ***"algoritm determinist"*** trebuie înţeleasă în sensul că structura sa nu permite o alegere între mai multe căi posibile în vederea obţinerii rezultatului dorit. Conceptul de ***"algoritm nedeterminist"*** trebuie înţeles în sensul că el se poate afla în mai multe stări independente care nu se pot alege după anumite criterii şi nici genera aleator. Un algoritm se numeşte ***nedeterminist polinomial*** dacă complexitatea calculelor, efectuate pe orice cale a arborelui ce descrie ramificările procesului de căutare a soluţiei, este polinomială.

Având în vedere aceste interpretări, conceptul de ***algoritm determinist polinomial*** este evident.

Clasa problemelor ***"dificile în sensul cel mai tare"*** este formată din acele probleme pentru care s-a demonstrat că nu există algoritmi pentru rezolvarea lor (de exemplu, problema a zecea a lui Hilbert cu privire la rezolvabilitatea în numere întregi a ecuaţiilor polinomiale).