

Mirel COȘULSCHI

Algoritmica grafurilor și aplicații

Mirel COȘULSCHI

ALGORITMICA GRAFURILOR ȘI APLICAȚII



**EDITURA UNIVERSITARIA
Craiova, 2014**

Referenți științifici:

Prof.univ.dr. Ion Iancu

Lect.univ.dr. Petre Băzăvan

Copyright © 2014 Universitaria

Toate drepturile sunt rezervate Editurii Universitaria.

Descrierea CIP a Bibliotecii Naționale a României

COȘULSCHI, MIREL

Algoritmica grafurilor și aplicații / Mirel Coșulschi. –

Craiova : Universitaria, 2014

Bibliogr.

ISBN 978-606-14-0859-7

519.178

*Bunicilor mei,
Constanța și Gheorghe Constantinescu.*

Prefață

Cu toată creșterea de popularitate a tutorialelor video, a cursurilor online, a prezentărilor PowerPoint, cărțile tipărite constituie, încă, principalul mijloc de răspândire al cunoștințelor. În acest sens, vine să își susțină cauza și cartea de față, ca modalitate de a deschide un zăvor ferecat pentru cei care bat la porțile cunoașterii.

Lucrarea de față prezintă capitole de algoritmică grafurilor ce abordează aproape toate aspectele importante din domeniu. Primele elemente de teoria grafurilor datează cu mult înaintea apariției calculatoarelor, grafurile devenind o parte importantă a informaticii, odată cu dezvoltarea acesteia ca știință. Un graf poate fi utilizat pentru modelarea unui sistem distribuit, văzut sub forma unui număr de componente de calcul autonome, interconectate prin intermediul unei rețele de comunicații și care cooperează în vederea realizării unor scopuri comune. Rețelele mobile ad-hoc, rețelele de senzori fără fir (wireless), structurile grid sau cloud sunt instanțe ale unui astfel de sistem distribuit, fiind întâlnite foarte des în ultimul timp. În aceeași idee menționăm că Facebook este un exemplu de graf social sau Web-ul este un exemplu de multi-graf orientat.

Pe lângă conceptele teoretice, cititorul interesat de subiect va găsi peste 90 de algoritmi prezentați în lucrarea structurată pe 10 capitole. O parte din algoritmi sunt introduși mai întâi la nivel conceptual și, mai apoi, prezentați în detaliu în pseudo-cod. Mulți algoritmi sunt analizați din punct de vedere al complexității, punându-se astfel la dispoziția unui programator un criteriu de evaluare a unui algoritm în vederea alegerii sale pentru rezolvarea unei probleme sau situații. Bibliografia bogată conține referințe către lucrări importante din domeniu.

Materialul este prezentat într-o înlănțuire *teorie-algoritm- implementare-exemple-aplicații*, abordarea autorului înclinându-se spre metode de rezolvare a unor probleme cu caracter practic, al căror grad de dificultate variază de la simplu la dificil.

Cartea reprezintă un material de referință ce vine în sprijinul studenților care urmează cursul de algoritmică grafurilor la facultățile de informatică, calculatoare, automatică, telecomunicații și informatică economică, însă poate fi consultată în egală măsură de către elevii de liceu la profilul informatică sau de cititorul interesat de diferitele aspecte de implementare ale algoritmilor prezentați. De asemenea, prin topicile avansate prezentate în lucrare, constituie un material util studenților de la cursurile masterale și doctorale.

Mulțumirile autorului se îndreaptă către referenții științifici ai acestei lucrări pentru sugestiile și observațiile pertinente pe marginea manuscrisului.

Capitolul 1

Introducere în algoritmi

Definiția 1.1 *Algoritmul constituie o reprezentare finită a unei metode de calcul ce permite rezolvarea unei anumite probleme. Se poate spune că un algoritm reprezintă o secvență finită de operații, ordonată și complet definită, care, pornind de la datele de intrare, produce rezultate.*

Termenul de *algoritm* îi este atribuit matematicianului persan Abu Ja'far Mohammed ibn Musa al-Khowarizmi (sec. VIII-IX), care a scris o carte de matematică cunoscută în traducere latină sub titlul de "*Algorithmi de numero indorum*", iar apoi ca "*Liber algorithmi*", unde termenul de "*algorithm*" provine de la "*al-Khowarizmi*", ceea ce literal înseamnă "*din orasul Khowarizm*". Matematicienii din Evul Mediu înțelegeau prin algoritm o regulă (sau o mulțime de reguli) pe baza căreia se efectuau calcule aritmetice: de exemplu în secolul al XVI-lea, algoritmii se foloseau la înmulțiri sau înjumătățiri de numere.

Fiecare propoziție ce face parte din descrierea unui algoritm este, de fapt, o comandă ce trebuie executată de cineva, acesta putând fi o persoană sau o mașină de calcul. De altfel, un algoritm poate fi descris cu ajutorul oricărui limbaj, de la limbajul natural și până la limbajul de asamblare al unui calculator. Denumim *limbaj algoritmic* un limbaj al cărui scop este acela de a descrie algoritmi.

Algoritmul specifică succesiuni posibile de transformări ale datelor. Un *tip de date* poate fi caracterizat printr-o mulțime de valori ce reprezintă domeniul tipului de date și o mulțime de operații definite peste acest domeniu. Tipurile de date pot fi organizate în următoarele categorii:

1. *tipuri de date elementare* (de exemplu tipul *întreg*, tipul *real*) - valorile sunt unități atomice de informație;
2. *tipuri de date structurate* (de exemplu tipul *tablou*, tipul *înregistrare*) - valorile sunt structuri relativ simple rezultate în urma combinației unor valori elementare;
3. *tipuri de date structurate de nivel înalt* (de exemplu *stiva*) - se pot descrie independent de limbaj iar valorile au o structură mai complexă.

Un algoritm trebuie să posede următoarele *trăsături caracteristice*:

1. *claritate* - la fiecare pas trebuie să specifice operația pe care urmează să o efectueze algoritmul asupra datelor de intrare;
2. *corectitudine* - rezultatele trebuie să fie corecte;

3. *generalitate* - algoritmul trebuie să ofere soluția nu numai pentru o singură problemă ci pentru o întreagă clasă de probleme;
4. *finitudine* - algoritmul trebuie să se termine într-un timp finit;
5. *eficiență* - un algoritm poate fi utilizat numai în situația în care resursele de calcul necesare acestuia sunt în cantități rezonabile, și nu depășesc cu mult posibilitățile calculatoarelor la un moment dat.

Un *program* reprezintă implementarea unui algoritm într-un limbaj de programare. Studiul algoritmilor cuprinde mai multe aspecte:

- *elaborare* - activitatea de concepere a unui algoritm are și un caracter creativ, din această cauză nefiind posibilă deprinderea numai pe cale mecanică. Pentru a facilita obținerea unei soluții la o problemă concretă se recomandă folosirea tehnicilor generale de elaborare a algoritmilor la care se adaugă în mod hotărâtor intuiția programatorului;
- *exprimare* - implementarea unui algoritm într-un limbaj de programare se poate face utilizând mai multe stiluri: *programare structurată*, *programare orientată pe obiecte* etc.;
- *validare* - verificarea corectitudinii algoritmului prin metode formale;
- *analiză* - stabilirea unor criterii pentru evaluarea eficienței unui algoritm pentru a-i putea compara și clasifica.

Un model de reprezentare al memoriei unei mașini de calcul este acela al unei structuri liniare compusă din celule, fiecare celulă fiind identificată printr-o adresă și putând păstra o valoare corespunzătoare unui anumit tip de dată. Accesul la celule este facilitat de variabile. O *variabilă* se caracterizează prin:

- *un identificador* - un nume ce referă variabila;
- *o adresă* - desemnează o locație de memorie;
- *un tip de date* - descrie tipul valorilor memorate în celula de memorie asociată.

1.1 Limbajul pseudocod

Limbajul natural nu permite o descriere suficient de riguroasă a algoritmilor, de aceea, pentru reprezentarea acestora se folosesc alte modalități de descriere precum:

- *scheme logice*;
- *limbajul pseudocod*.

În continuare vom prezenta principalele construcții din cadrul limbajului pseudocod.

Intrări/ieșiri

Citirea datelor de intrare se poate realiza prin intermediul enunțului **Input**:

1: **Input** {lista variabile}

Afișarea rezultatelor este reprezentată cu ajutorul instrucțiunii **Output**:

1: **Output** {lista de valori}

Instrucțiunea de atribuire

Este instrucțiunea cel mai des utilizată într-un algoritm și realizează încărcarea unei variabile (locații de memorie) cu o anumită valoare. Are următoarea sintaxă:

1: $\langle \text{variabila} \rangle \leftarrow \langle \text{expresie} \rangle$

unde $\langle \text{expresie} \rangle$ este o expresie aritmetică sau logică.

Se evaluează expresia $\langle \text{expresie} \rangle$ iar rezultatul se atribuie variabilei $\langle \text{variabila} \rangle$, memorându-se în locația de memorie asociată. Această variabilă trebuie să fie de același tip de dată cu expresia sau un tip de dată care să includă și tipul expresiei.

O expresie este constituită din *operanzi* și *operatori*. *Operanzii* pot fi variabile și valori constante, iar *operatorii* pot fi:

- *operatori aritmetici* - $+$ (adunare), $-$ (scădere), $*$ (înmulțire), $/$ (împărțire), $^$ (ridicare la putere), *div* (câtul împărțirii întregi), *mod* (restul împărțirii întregi);
- *operatori relaționali* - $=$ (egal), $\neq$ (diferit), $<$ (strict mai mic), $\leq$ (mai mic sau egal), $>$ (strict mai mare), $\geq$ (mai mare sau egal);
- *operatori logici* - *OR* sau $\vee$ (disjuncție), *AND* sau $\wedge$ (conjuncție), *NOT* sau $\neg$ (negație).

Cea mai simplă expresie este formată dintr-o *variabilă* sau o *constantă* (operand). Expresiile mai complicate se obțin din operații efectuate între variabile și constante. La scrierea expresiilor trebuie să se țină cont de faptul că, în momentul evaluării lor, în primul rând se vor evalua expresiile din paranteze, iar operațiile se execută în ordinea determinată de prioritățile lor.

Enunțuri de ramificare

```
1: if <expresie-booleana> then
2:   <instrucțiune1>
3:   [
4:     <instrucțiune2>
5:   ]
6: else
7:   <instrucțiune3>
8: end if
```

```
1: if (a mod 2 = 0) then
2:   Output { 'Numarul este par' }
3: else
4:   Output { 'Numarul este impar' }
5: end if
```

Enunțul **if ... then ... else** evaluează mai întâi expresia booleană pentru a determina unul din cele două drumuri pe care le poate lua execuția algoritmului. Ea poate include opțional o clauză **else**.

Dacă $\langle \text{expresie-booleana} \rangle$ se evaluează la valoarea de adevăr **true**, atunci se execută $\langle \text{instrucțiune1} \rangle$ și se continuă cu următoarea instrucțiune după **if**. Dacă $\langle \text{expresie-booleana} \rangle$ are valoarea de adevăr **false**, atunci se execută $\langle \text{instrucțiune2} \rangle$. $\langle \text{instrucțiune1} \rangle$ și $\langle \text{instrucțiune2} \rangle$ sunt instrucțiuni compuse ce pot să conțină, la rândul lor, o altă instrucțiune **if**.

Exemplul 1.1 Un algoritm simplu este cel ce rezolvă ecuația de gradul I, $ax+b=0$, $a, b \in \mathbb{R}$ (algoritmul 1). Acesta se bazează pe rezolvarea matematică a ecuației de gradul I:

1. dacă $a = 0$, atunci ecuația devine $0 \cdot x + b = 0$. Pentru cazul în care $b \neq 0$ avem $0 \cdot x + b = 0$, egalitate ce nu poate fi satisfăcută pentru nicio valoare a lui $x \in \mathbb{R}$ sau $\mathbb{C}$. Spunem, în acest caz, că ecuația este incompatibilă. Dacă $b = 0$, avem $0 \cdot x + 0 = 0$, relația fiind adevărată pentru orice valoare a lui $x \in \mathbb{R}$. Spunem că ecuația este compatibil nedeterminată.

2. dacă $a \neq 0$, ecuația are o singură soluție, $x_1 = -\frac{b}{a} \in \mathbb{R}$.

Algoritm 1 Algoritm pentru rezolvarea ecuației de gradul I

```

1: Input { $a, b$ }
2: if ( $a = 0$ ) then
3:   if ( $b = 0$ ) then
4:     Output { 'Ecuație compatibil nedeterminată' }
5:   else
6:     Output { 'Ecuație incompatibilă' }
7:   end if
8: else
9:    $x \leftarrow -\frac{b}{a}$ 
10:  Output { 'Soluția este:',  $x$  }
11: end if

```

Având acest algoritm drept model să se realizeze un algoritm pentru rezolvarea ecuației generale de gradul al II-lea, $ax^2 + bx + c = 0$, unde $a, b, c \in \mathbb{R}$.

Enunțuri repetitive

Enunțurile repetitive permit descrierea unor prelucrări ce trebuie efectuate în mod repetat, în funcție de poziția condiției de continuare existând două variante de structuri repetitive: *structura repetitivă condiționată anterior* și *structura repetitivă condiționată posterior*.

Structura repetitivă condiționată anterior **while** (sau instrucțiune de ciclare cu test inițial) are următoarea sintaxă:

	1: $sum \leftarrow 0$
	2: $i \leftarrow 1$
1: while <expresie-booleana> do	3: while ($i \leq n$) do
2: <instrucțiune>	4: $sum \leftarrow sum + i$
3: end while	5: $i \leftarrow i + 1$
	6: end while

Cât timp <expresie-booleana> este adevărată, se execută <instrucțiune>; dacă <expresie-booleana> este falsă chiar la prima evaluare, atunci <instrucțiune> nu ajunge să fie realizată niciodată. Acest comportament este opus celui corespunzător structurii repetitive condiționată posterior **repeat**, unde <instrucțiune> este executată cel puțin o dată. (Dacă o expresie are valoarea de adevăr **true** spunem atunci că expresia este *adevărată*; dacă o expresie are valoarea de adevăr **false** spunem atunci că expresia nu este adevărată - este falsă).

Exemplul 1.2 Vom realiza un algoritm pentru calculul câtului și restului împărțirii a două numere întregi, prin scăderi succesive (a se vedea algoritmul 2).

Mai întâi, se inițializează câtul cu valoarea zero (linia 3) și restul cu valoarea deîmpărțitului (linia 4). Apoi, atâta timp cât restul este mai mare decât valoarea împărțitorului (liniile 5 - 8), vom incrementa câtul cu o unitate, și decrementa restul cu valoarea împărțitorului. La final, sunt afișate valorile câtului și restului.

Un caz particular de structură repetitivă condiționată anterior este **for**, utilizată în cazul în care o instrucțiune sau un grup de instrucțiuni trebuie să se repete de 0 sau mai multe ori - numărul de repetiții fiind cunoscut înainte de începerea sa. *Enunțurile repetitive bazate pe instrucțiunile **while** și **repeat** sunt mult mai potrivite în cazul în care condiția de terminare trebuie reevaluată în timpul ciclării (atunci când numărul de repetiții nu este cunoscut apriori).*

Algoritm 2 Algoritm pentru calculul împărțirii întregi

```

1: Input  $\{a, b\}$ 
2: if  $((a > 0) \wedge (b > 0))$  then
3:    $cat \leftarrow 0$ 
4:    $rest \leftarrow a$ 
5:   while  $(rest \geq b)$  do
6:      $rest \leftarrow rest - b$ 
7:      $cat \leftarrow cat + 1$ 
8:   end while
9:   Output  $\{cat, rest\}$ 
10: else
11:   Output  $\{\text{'Numerele trebuie sa fie strict pozitive'}\}$ 
12: end if

```

Instrucțiunea **for** are următoarea sintaxă:

```

1: for  $\langle Var \rangle \leftarrow \langle Expresie1 \rangle, \langle Expresie2 \rangle$ , 1:  $sum \leftarrow 0$ 
    $Step < p >$  do                                2: for  $i \leftarrow 1, n$  do
2:    $\langle instructiune \rangle$                                 3:    $sum \leftarrow sum + i$ 
3: end for                                           4: end for

```

Comportamentul enunțului repetitiv **for** se descrie cel mai bine prin comparație cu enunțul repetitiv **while**. Astfel secvența următoare de limbaj pseudocod

```

1: for  $v \leftarrow e_1, e_2$  STEP  $p$  do
2:    $\langle instructiune \rangle$ 
3: end for

```

este echivalentă cu secvența ce conține enunțul repetitiv **while**:

```

1:  $v \leftarrow e_1$ 
2: while  $(v \leq e_2)$  do
3:    $\langle instructiune \rangle$ 
4:    $v \leftarrow v + p$ 
5: end while

```

e_1 reprezintă valoarea inițială, e_2 limita finală, iar p pasul de lucru: la fiecare pas, valoarea variabilei v este incrementată cu valoarea variabilei p , valoarea ce poate fi atât pozitivă cât și negativă. Dacă $p \geq 0$ atunci v va lua valori în ordine crescătoare, iar dacă $p < 0$ atunci v va primi valori în ordine descrescătoare.

În cazul în care pasul de incrementare este 1, atunci el se poate omite din enunțul repetitiv **for**, variabila contor fiind incrementată automat la fiecare repetiție cu valoarea 1. Dacă pasul de incrementare p are valoarea 1 atunci avem următoarele situații posibile pentru construcția **for**:

- $Expresie1 > Expresie2$: $\langle instructiune \rangle$ nu se execută niciodată;
- $Expresie1 = Expresie2$: $\langle instructiune \rangle$ se execută exact o dată;
- $Expresie1 < Expresie2$: $\langle instructiune \rangle$ se execută de $Expresie2 - Expresie1 + 1$ ori.

Exemplul 1.3 Un caz destul de frecvent întâlnit în practică este cel în care se cere determinarea elementului de valoare maximă, respectiv minimă dintr-un șir de elemente. Acest șir de elemente poate fi păstrat într-o structură de date elementară, cunoscută sub numele de vector. Un vector este un caz particular de matrice având o singură linie și n coloane.