

UNIVERSITATEA TEHNICĂ A MOLDOVEI

ANALIZA și PROIECTAREA ALGORITMILOR

Îndrumar pentru lucrări de laborator



Chișinău 2015

UNIVERSITATEA TEHNICĂ A MOLDOVEI
CATEDRA AUTOMATICĂ și TEHNOLOGII INFORMAȚIONALE

ANALIZA și PROIECTAREA ALGORITMILOR

Îndrumar pentru lucrări de laborator



Chișinău
U. T. M.
2015

Îndrumarul este destinat în primul rând studenților de la specialitatea Tehnologii Informaționale și Securitate Informațională, cât și specialităților cu profil ingineresc ce vor să studieze lumea diversă a algoritmilor.

Fiecare lucrare conține scopul lucrării, note din materialul teoretic, exemple și întrebări de control, toate necesare pentru executarea lucrării respective.

La elaborarea îndrumarului dat s-a luat în considerare ca materialul să fie accesibil pentru un cerc cât mai larg de cititori, limbajul este adecvat, iar problemele propuse sunt de o dificultate medie. Acestea pentru ridicarea nivelului de înțelegere și sporirea productivității în elaborarea de noi algoritmi.

Elaborare:	Mariana Catruc, Veronica Andrievschi-Bagrin
Redactor responsabil:	
Recenzent:	
Culegere computerizată:	Mariana Catruc, Veronica Andrievschi-Bagrin

© U.T.M., 2015

LUCRARE DE LABORATOR NR. 1

Tema: Analiza algoritmilor

Scopul lucrării:

1. Analiza empirică a algoritmilor.
2. Analiza teoretică a algoritmilor.
3. Determinarea complexității temporale și asimptotice a algoritmilor.

Note de curs:

1. Timpul de execuție al algoritmilor

De multe ori, pentru rezolvarea unei probleme, trebuie ales un algoritm dintre mai mulți posibili, două criterii principale de alegere fiind contradictorii:

- algoritmul să fie simplu de înțeles, de codificat și de depanat;
- algoritmul să folosească eficient resursele calculatorului, să aibă un timp de execuție redus.

Dacă programul care se scrie trebuie rulat de un număr mic de ori, prima cerință este mai importantă; în această situație, timpul de punere la punct a programului e mai important decât timpul lui de rulare, deci trebuie aleasă varianta cea mai simplă a programului.

Dacă programul urmează a fi rulat de un număr mare de ori, având și un număr mare de date de prelucrat, trebuie ales algoritmul care duce la o execuție mai rapidă. Chiar în această situație, ar trebui implementat mai înainte algoritmul mai simplu și calculată reducerea de timp de execuție pe care ar aduce-o implementarea algoritmului complex.

Timpul de rulare al unui program depinde de următorii factori:

- datele de intrare;
- calitatea codului generat de compilator;

- natura și viteza de execuție a instrucțiunilor programului;
- complexitatea algoritmului care stă la baza programului.

Deci timpul de rulare e o funcție de intrarea sa, de cele mai multe ori, nedepinzând de valorile de la intrare, ci de numărul de date.

În continuare vom nota cu $T(n)$ timpul de execuție al unui algoritm destinat rezolvării unei probleme de dimensiune n . Pentru a estima timpul de execuție trebuie stabilit un *model de calcul* și o unitate de măsură. Vom considera un model de calcul (numit și mașină de calcul cu acces aleator) caracterizat prin:

- Prelucrările se efectuează în mod secvențial.
- Operațiile *elementare* sunt efectuate în timp constant *indiferent* de valoarea operanzilor.
- Timpul de acces la informație nu depinde de poziția acesteia (nu sunt diferențe între prelucrarea primului element și cea a ultimului element al unui tablou).

A stabili o unitate de măsură înseamnă a stabili care sunt operațiile elementare și a considera ca unitate de măsură timpul de execuție a acestora. În acest fel timpul de execuție va fi exprimat prin numărul de operații elementare executate. Operațiile elementare sunt cele aritmetice (adunare, scădere, înmulțire, împărțire), comparațiile și cele logice (negație, conjuncție și disjuncție). Cum scopul calculului timpului de execuție este de a permite compararea algoritmilor, uneori este suficient să se contorizeze doar anumite tipuri de operații elementare, numite *operații de bază* (de exemplu în cazul unui algoritm de căutare sau de sortare se pot contoriza doar operațiile de comparare) și/sau să se considere că timpul de execuție a acestora este unitar (deși operațiile de înmulțire și împărțire sunt mai costisitoare decât cele de adunare și scădere în analiza se poate considera că ele au același cost).

Timpul de execuție al întregului algoritm se obține însumând timpii de execuție a prelucrărilor componente.

1.1. Importanța celui mai defavorabil caz

În aprecierea și compararea algoritmilor interesează în special cel mai defavorabil caz deoarece furnizează cel mai mare timp de execuție relativ la *orice* date de intrare de dimensiune fixă. Pe de altă parte pentru anumiți algoritmi cazul cel mai defavorabil este relativ frecvent.

În ceea ce privește analiza celui mai favorabil caz, acesta furnizează o margine inferioară a timpului de execuție și poate fi utilă pentru a identifica algoritmi ineficienți (dacă un algoritm are un cost mare în cel mai favorabil caz, atunci el nu poate fi considerat o soluție acceptabilă).

1.2. Timp mediu de execuție

Uneori, cazurile extreme (cel mai defavorabil și cel mai favorabil) se întâlnesc rar, astfel că analiza acestor cazuri nu furnizează suficientă informație despre algoritm.

În aceste situații este utilă o altă măsură a complexității algoritmilor și anume *timpul mediu de execuție*. Acesta reprezintă o valoare medie a timpilor de execuție calculată în raport cu distribuția de probabilitate corespunzătoare spațiului datelor de intrare.

2. Etapele analizei complexității

În analiza complexității unui algoritm se parcurg următoarele etape:

1. Se stabilește dimensiunea problemei.
2. Se identifică operația de bază.
3. Se verifică dacă numărul de execuții ale operației de bază depinde doar de dimensiunea problemei. Dacă da, se determină acest număr. Dacă nu, se analizează cazul cel mai favorabil, cazul cel mai defavorabil și (dacă este posibil) cazul mediu.
4. Se află clasa de complexitate căruia îi aparține algoritmul.

Câteva reguli generale pentru evaluarea timpului de execuție în funcție de mărimea datelor de intrare:

1. Timpul de execuție a unei instrucțiuni de asignare, citire sau scriere, este constant și se notează cu $O(1)$.

2. Timpul de rulare a unei secvențe de instrucțiuni e determinat de regula de însumare, fiind proporțional cu cel mai lung timp din cei ai instrucțiunilor secvenței.

3. Timpul de execuție a unei instrucțiuni *if - then - else* este suma dintre timpul de evaluare a condiției ($O(1)$) și cel mai mare dintre timpii de execuție ai instrucțiunilor pentru condiția adevărată sau falsă.

4. Timpul de execuție a unei instrucțiuni de ciclare este suma, pentru toate iterațiile, dintre timpul de execuție a corpului instrucțiunii și cel de evaluare a condiției de terminare ($O(1)$).

5. Pentru evaluarea timpului de execuție a unei proceduri recursive, se asociază fiecărei proceduri recursive un timp necunoscut $T(n)$, unde n măsoară argumentele procedurii; se poate obține o relație recurentă pentru $T(n)$, adică o ecuație pentru $T(n)$, în termeni $T(k)$, pentru diferite valori ale lui k .

6. Timpul de execuție poate fi analizat chiar pentru programele scrise în pseudocod; pentru secvențele care cuprind operații asupra unor structuri de date, se pot alege câteva implementări și astfel se poate face comparație între performanțele implementărilor, în contextul aplicației respective.

3. Analiza empirică a complexității algoritmilor

O alternativă la analiza matematică a complexității o reprezintă *analiza empirică*.

Aceasta poate fi utilă pentru:

- a obține informații preliminare privind clasa de complexitate a unui algoritm;

- pentru a compara eficiența a doi (sau mai mulți) algoritmi destinați rezolvării aceleiași probleme;

- pentru a compara eficiența mai multor implementări ale aceluiași algoritm;
- pentru a obține informații privind eficiența implementării unui algoritm pe un anumit calculator.

Etapele analizei empirice. In analiza empirică a unui algoritm se parcurg de regulă următoarele etape:

1. Se stabilește scopul analizei.
2. Se alege metrica de eficiență ce va fi utilizată (număr de execuții ale unei/unor operații sau timp de execuție a întregului algoritm sau a unei porțiuni din algoritm.
3. Se stabilesc proprietățile datelor de intrare în raport cu care se face analiza (dimensiunea datelor sau proprietăți specifice).
4. Se implementează algoritmul într-un limbaj de programare.
5. Se generează mai multe seturi de date de intrare.
6. Se execută programul pentru fiecare set de date de intrare.
7. Se analizează datele obținute.

Alegerea măsurii de eficiență depinde de scopul analizei. Dacă, de exemplu, se urmărește obținerea unor informații privind clasa de complexitate sau chiar verificarea acurateții unei estimări teoretice atunci este adecvată utilizarea numărului de operații efectuate. Dacă însă scopul este evaluarea comportării implementării unui algoritm atunci este potrivit timpul de execuție.

Pentru a efectua o analiză empirică nu este suficient un singur set de date de intrare ci mai multe, care să pună în evidență diferitele caracteristici ale algoritmului. În general este bine să se aleagă date de diferite dimensiuni astfel încât să fie acoperită plaja tuturor dimensiunilor care vor apare în practică. Pe de altă parte are importanță și analiza diferitelor valori sau configurații ale datelor de intrare. Dacă se analizează un algoritm care verifică dacă un număr este prim sau nu și testarea se face doar pentru numere ce nu sunt prime sau doar pentru numere care sunt prime atunci nu se va obține un rezultat relevant. Același lucru se poate întâmpla pentru un algoritm a cărui comportare depinde de gradul de sortare a unui tablou.

În vederea analizei empirice la implementarea algoritmului într-un limbaj de programare vor trebui introduse secvențe al căror scop este monitorizarea execuției. Dacă metrica de eficiență este numărul de execuții ale unei operații atunci se utilizează un contor care se incrementează după fiecare execuție a operației respective. Dacă metrica este timpul de execuție atunci trebuie înregistrat momentul intrării în secvența analizată și momentul ieșirii. Majoritatea limbajelor de programare oferă funcții de măsurare a timpului scurs între două momente. Este important, în special în cazul în care pe calculator sunt active mai multe taskuri, să se contorizeze doar timpul afectat de execuția programului analizat. În special dacă este vorba de măsurarea timpului este indicat să se ruleze programul de test de mai multe ori și să se calculeze valoarea medie a timpilor.

După execuția programului pentru datele de test se înregistrează rezultatele iar în scopul analizei fie se calculează mărimi sintetice (de ex. media) fie se reprezintă grafic perechi de puncte de o anumite formă (dimensiune problemă, măsură de eficiență).

4. Ordin de creștere

Pentru a aprecia eficiența unui algoritm nu este necesară cunoașterea expresiei detaliate a timpului de execuție. Mai degrabă este interesant modul în care timpul de execuție crește o dată cu creșterea dimensiunii problemei. O măsură utilă în acest sens este *ordinul de creștere*. Acesta este determinat de *termenul dominant* din expresia timpului de execuție. Când dimensiunea problemei este mare valoarea termenului dominant depășește semnificativ valorile celorlalți termeni astfel că aceștia din urmă pot fi neglijați.

Întrucât problema eficienței devine critică pentru probleme de dimensiuni mari se face analiza complexității pentru cazul când n este mare (teoretic se consideră că $n \rightarrow \infty$), în felul acesta luându-se în considerare doar comportarea termenului dominant. Acest tip de analiză se numește *analiză asimptotică*. În cadrul analizei asimptotice se consideră că un algoritm este mai eficient decât altul

dacă ordinul de creștere al timpului de execuție al primului este mai mic decât al celuiilalt.

Există următoarele cazuri când ordinul de creștere a timpului de execuție, nu e cel mai bun criteriu de apreciere a performanțelor unui algoritm:

- dacă un program se rulează de puține ori, se alege algoritmul cel mai ușor de implementat;

- dacă întreținerea trebuie făcută de o altă persoană decât cea care l-a scris, un algoritm simplu, chiar mai puțin eficient, e de preferat unuia performant, dar foarte complex și greu de înțeles;

- există algoritmi foarte eficienți, dar care necesită un spațiu de memorie foarte mare, astfel încât folosirea memoriei externe le diminuează foarte mult performanțele.

4.1. Notății asimptotice.

Fie \mathbf{N} mulțimea numerelor naturale (pozitive sau zero) și \mathbf{R} mulțimea numerelor reale. Notăm prin \mathbf{N}^+ și \mathbf{R}^+ mulțimea numerelor naturale, respectiv reale, strict pozitive, și prin \mathbf{R}^* mulțimea numerelor reale nenegative. Fie $f : \mathbf{N} \rightarrow \mathbf{R}^*$ o funcție arbitrară. Definim mulțimea

$$O(f) = \{t : \mathbf{N} \rightarrow \mathbf{R}^* \mid (\exists c \in \mathbf{R}^+) (\exists n_0 \in \mathbf{N}) (\forall n \geq n_0) [t(n) \leq cf(n)]\}$$

Cu alte cuvinte, $O(f)$ (se citește “ordinul lui f ”) este mulțimea tuturor funcțiilor t mărginite superior de un multiplu real pozitiv al lui f , pentru valori suficient de mari ale argumentului. Vom conveni să spunem că t este în ordinul lui f (sau, echivalent, t este în $O(f)$, sau $t \in O(f)$) chiar și atunci când valoarea $f(n)$ este negativă sau nedefinită pentru anumite valori $n < n_0$. În mod similar, vom vorbi despre ordinul lui f chiar și atunci când valoarea $t(n)$ este negativă sau nedefinită pentru un număr finit de valori ale lui n ; în acest caz, vom alege n_0 suficient de mare, astfel încât, pentru $n \geq n_0$, acest lucru să nu mai apară. În loc de $t \in O(f)$, uneori este mai convenabil să folosim notația $t(n) \in O(f(n))$, subînțelegând aici că $t(n)$ și $f(n)$ sunt funcții.

Notăția asimptotică definește o relație de ordine parțială între funcții și deci, între eficiența relativă a diferiților algoritmi care rezolvă o anumită problemă. Vom da în continuare o interpretare algebrică a notației asimptotice. Pentru oricare două funcții $f, g : \mathbf{N} \rightarrow \mathbf{R}^*$, definim următoarea relație binară: $f \leq g$ dacă $O(f) \subseteq O(g)$. Relația “ \leq ” este o *relație de ordine parțială* în mulțimea funcțiilor definite pe \mathbf{N} și cu valori în \mathbf{R}^* . Definim și o *relație de echivalență*: $f \equiv g$ dacă $O(f) = O(g)$.

În mulțimea $O(f)$ putem înlocui pe f cu orice altă funcție echivalentă cu f . De exemplu, $\lg n \equiv \ln n \equiv \log n$ și avem $O(\lg n) = O(\ln n) = O(\log n)$. Notând cu $O(1)$ ordinul funcțiilor mărginite superior de o constantă, obținem ierarhia:

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \subset O(2^n)$$

Această ierarhie corespunde unei clasificări a algoritmilor după un criteriu al performanței. Pentru o problemă dată, dorim mereu să obținem un algoritm corespunzător unui ordin cât mai mic. Astfel, este o mare realizare dacă în locul unui algoritm exponențial găsim un algoritm polinomial.

Notăția $O(f)$ este folosită pentru a limita superior timpul necesar unui algoritm, măsurând eficiența algoritmului respectiv. Uneori este util să estimăm și o limită inferioară a acestui timp. În acest scop, definim mulțimea

$$\Omega(f) = \{t : \mathbf{N} \rightarrow \mathbf{R}^* \mid (\exists c \in \mathbf{R}^+) (\exists n_0 \in \mathbf{N}) (\forall n \geq n_0) [t(n) \geq cf(n)]\}$$

Există o anumită dualitate între notațiile $O(f)$ și $\Omega(f)$. Și anume, pentru două funcții oarecare $f, g : \mathbf{N} \rightarrow \mathbf{R}^*$, avem: $f \in O(g)$, dacă și numai dacă $g \in \Omega(f)$.

O situație fericită este atunci când timpul de execuție al unui algoritm este limitat, atât inferior cât și superior, de câte un multiplu real pozitiv al aceleiași funcții. Introducem notația

$$\Theta(f) = O(f) \cap \Omega(f)$$

numită *ordinul exact* al lui f . Pentru a compara ordinele a două funcții, notația Θ nu este însă mai puternică decât notația O , în sensul că relația

$O(f) = O(g)$ este echivalentă cu $\Theta(f) = \Theta(g)$.

Se poate întâmpla că timpul de execuție al unui algoritm să depindă simultan de mai mulți parametri. Această situație este tipică pentru anumiți algoritmi care operează cu grafuri și în care timpul depinde atât de numărul de vârfuri, cât și de numărul de muchii. Notația asimptotică se generalizează în mod natural și pentru funcții cu mai multe variabile. Astfel, pentru o funcție arbitrară $f: \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{R}^*$ definim

$$O(f) = \{t: \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{R}^* \mid (\exists c \in \mathbf{R}^+) (\exists m_0, n_0 \in \mathbf{N}) (\forall m \geq m_0) (\forall n \geq n_0) [t(m, n) \leq cf(m, n)]\}$$

Similar, se obțin și celelalte generalizări.

4.2. Operații asupra funcțiilor asimptotice

1. Dacă $T1(n)$ și $T2(n)$ sunt timpii de execuție a două secvențe de program $P1$ și $P2$, $T1(n)$ fiind $O(f(n))$, iar $T2(n)$ fiind $O(g(n))$, atunci timpul de execuție $T1(n)+T2(n)$, al secvenței $P1$ urmată de $P2$, va fi $O(\max(f(n),g(n)))$.
2. Dacă $T1(n)$ este $O(f(n))$ și $T2(n)$ este $O(g(n))$, atunci $T1(n)*T2(n)$ este $O(f(n)*g(n))$.

5. Exemple

1. Mai jos este prezentat modul de evaluare a timpului de execuție a procedurii de sortare a elementelor unui tablou de dimensiune n prin metoda "bubble sort":

```

procedure bubble ( a[1..n] ){sortare crescătoare }
1: for i←1 to n - 1 do
2:   for j←n downto i + 1 do
3:     if a[j - 1] > a[j] then
4:       temp←a[j - 1];

```

```

5:          a[j - 1] ← a[j];
6:          a[j] ← temp
return a[1..n] sortat

```

Timpii de rulare pentru instrucțiunile de asignare (4),(5) și (6) sânt $O(1)$, deci pentru secvența (4)-(6), timpul este $O(\max(1,1,1)) = O(1)$.

Pentru instrucțiunea **if - then** (3), timpul este suma dintre cel pentru evaluarea condiției, $O(1)$ și cel al secvenței ce se execută la condiție adevărată, tot $O(1)$, calculat mai sus, deci $O(1)$, acesta fiind și timpul pentru o iterație a instrucțiunii **for** (2).

Pentru cele $n - i$ iterații ale lui **for** (2), timpul de execuție este:

$$O((n-i)*1) = O(n - i).$$

Pentru instrucțiunea **for** (1), având $n-1$ iterații, timpul de execuție este:

$$\sum_{i=1}^{n-1} (n-i) = (n-1) + (n-2) + \dots + (n-n+1) = \frac{n^2}{2} - \frac{n}{2},$$

deci este $O(n^2)$.

2. Modalitatea de evaluare a timpului de execuție al unei proceduri recursive, este ilustrată prin cea a funcției de calcul al factorialului:

```

function factorial(n: integer)
1: if n ≤ 1 then
2:   factorial ← 1
3:   else factorial ← n * factorial(n - 1)
return factorial(n)

```

Dimensiunea intrării este aici n , valoarea numărului al cărui factorial se calculează și se notează cu $T(n)$, timpul de rulare al funcției $factorial(n)$.

Timpul de rulare pentru liniile (1) și (2) este $O(1)$, iar pentru linia (3) este $O(1) + T(n - 1)$, deci cu constantele c și d neprecizate:

$$T(n) = c + T(n - 1), \text{ pentru } n > 1 \text{ sau } T(n) = d, \text{ pentru } n \leq 1.$$

Pentru $n > 2$, expandând pe $T(n - 1)$, se obține $T(n) = 2c + T(n - 2)$, pentru $n > 2$.

Pentru $n > 3$, expandând pe $T(n - 2)$, se obține $T(n) = 3c + T(n - 3)$, pentru $n > 3$.

Deci, în general $T(n) = ic + T(n - i)$, pentru $n > i$.

În final, când $i = n - 1$, se obține $T(n) = (n - 1)c + T(1) = (n - 1)c + d$.

Deci $T(n) = O(n)$.

SARCINA DE BAZĂ:

1. Efectuați analiza empirică a algoritmilor propuși.
2. Determinați relația ce reprezintă complexitatea temporală pentru acești algoritmi.
3. Determinați complexitatea asimptotică a algoritmilor.
4. Faceți o concluzie asupra lucrării efectuate.

Întrebări de control:

1. Enumerați factorii ce influențează timpul de execuție al algoritmului.
2. Când timpul de execuție al algoritmului este dat de o relație de recurență ?
3. Care sunt regulile generale pentru evaluarea timpului de execuție?
4. Care sunt etapele analizei empirice și în care cazuri se face analiza empirică a algoritmilor?
5. Ce proprietăți posedă funcțiile asimptotice?

LUCRARE DE LABORATOR NR. 2

Tema: Metoda divide et impera

Scopul lucrării:

1. Studiarea metodei divide et impera.
2. Analiza și implementarea algoritmilor bazați pe metoda divide et impera.

Note de curs:

1. Tehnica divide et impera

Divide et impera este o tehnica de elaborare a algoritmilor care constă în:

1. Descompunerea cazului ce trebuie rezolvat într-un număr de subcazuri mai mici ale aceleiași probleme.
2. Rezolvarea succesivă și independentă a fiecăruia din aceste subcazuri.
3. Combinarea subsoluțiilor astfel obținute pentru a găsi soluția cazului inițial.

Algoritmul formal al metodei divide et impera:

```
function divimp( $x$ )  
  {returnează o soluție pentru cazul  $x$ }  
1: if  $x$  este suficient de mic  
2:   then return adhoc( $x$ )  
3:   {descompune  $x$  în subcazurile  $x_1, x_2, \dots, x_k$ }  
4:   for  $i \leftarrow 1$  to  $k$  do  $y_i \leftarrow \text{divimp}(x_i)$   
5:   {recompune  $y_1, y_2, \dots, y_k$  în scopul obținerii soluției  $y$  pentru  $x$ }  
6:   return  $y$ 
```

unde *adhoc* este subalgoritmul de bază folosit pentru rezolvarea micilor subcazuri ale problemei în cauză.

Un algoritm divide et impera trebuie să evite descompunerea recursivă a subcazurilor “suficient de mici”, deoarece, pentru acestea, este mai eficientă aplicarea directă a subalgoritmului de bază.

Observăm că metoda divide et impera este prin definiție recursivă. Uneori este posibil să eliminăm recursivitatea printr-un ciclu iterativ. Implementată pe o mașina convențională, versiunea iterativă *poate fi* ceva mai rapidă (în limitele unei constante multiplicative). Un alt avantaj al versiunii iterative ar fi faptul că economisește spațiul de memorie. Versiunea recursivă folosește o stivă necesară memorării apelurilor recursive. Pentru un caz de mărime n , numărul apelurilor recursive este de multe ori în $\Omega(\log n)$, uneori chiar în $\Omega(n)$.

2. Algoritmi de sortare „divide et impera”

2.1. Mergesort (sortarea prin interclasare)

Fie $T[1..n]$ un tablou pe care dorim să-l sortăm crescător. Prin tehnica divide et impera putem proceda astfel: separăm tabloul T în două părți de mărimi cât mai apropiate, sortăm aceste părți prin apeluri recursive, apoi interclasăm soluțiile pentru fiecare parte, fiind atenți să păstrăm ordonarea crescătoare a elementelor. Obținem următorul algoritm pentru rezolvarea unei subprobleme:

```
procedure mergesort ( $T, p, r$ )  
1: if  $p < r$   
2: then  $q \leftarrow \lfloor (p+r)/2 \rfloor$   
3:      $\text{mergesort}(T, p, q)$   
4:      $\text{mergesort}(T, q+1, r)$   
5:      $\text{merge}(T, p, q, r)$ 
```

Acum putem sorta întreg tabloul T apelând procedura $\text{mergesort}(T, 1, n)$

Etapa cea mai importantă a algoritmului este interclasarea *merge* (T, p, q, r). Scopul acestei prelucrări este construirea unui tablou ordonat pornind de la două tablouri ordonate. Ideea prelucrării constă în a parcurge în paralel cele două tablouri și a compara elementele curente. În tabloul final se transferă elementul mai mic dintre cele două elemente curente, iar contorul utilizat pentru parcurgerea tabloului din care s-a transferat un element este incrementat. Procesul continuă până când unul din tablouri este transferat în întregime. Elementele celuiilalt tablou sunt transferate direct în tabloul final.

Algoritmul *mergesort* ilustrează perfect principiul *divide et impera*:

- divide problema în subprobleme;
- stăpânește subproblemele prin rezolvare;
- combină soluțiile subproblemelor.

În algoritmul *mergesort*, suma mărimilor subcazurilor este egală cu mărimea cazului inițial. Această proprietate nu este în mod necesar valabilă pentru algoritmi *divide et impera*. Este esențial ca subcazurile să fie de mărimi cât mai apropiate (sau, altfel spus, subcazurile să fie cât mai *echilibrate*).

2.2. Quicksort (sortarea rapida)

Algoritmul de sortare *quicksort*, inventat de Hoare în 1962, se bazează de asemenea pe principiul *divide et impera*. Spre deosebire de *mergesort*, partea nerecursivă a algoritmului este dedicată construirii subcazurilor și nu combinării soluțiilor lor.

Ca prim pas, algoritmul alege un element *pivot* din tabloul care trebuie sortat. Tabloul este apoi partiționat în două subtablouri, alcătuite de-o parte și de alta a acestui pivot în următorul mod: elementele mai mari decât pivotul sunt mutate în dreapta pivotului, iar celelalte elemente sunt mutate în stânga pivotului. Acest mod de partiționare este numit *pivotare*. În continuare, cele două subtablouri sunt sortate în mod independent prin apeluri recursive ale algoritmului. Rezultatul este tabloul complet sortat; nu mai este

necesară nici o interclasare. Pentru a echilibra mărimea celor două subtablouri care se obțin la fiecare partiționare, ar fi ideal să alegem ca pivot elementul median. Intuitiv, *mediana* unui tablou T este elementul m din T , astfel încât numărul elementelor din T mai mici decât m este egal cu numărul celor mai mari decât m . Din păcate, găsirea medianei necesita mai mult timp decât merită. De aceea, putem pur și simplu să folosim ca pivot primul element al tabloului. Iată cum arată acest algoritm:

```
procedure quicksort ( $T, p, r$ )
1: if  $p < r$ 
2:   then  $q \leftarrow \text{Partition}(T, p, r)$ 
3:     quicksort ( $T, p, q$ )
4:     quicksort ( $T, q+1, r$ )
```

Mai rămâne să concepem un algoritm de partiționare (pivotare) cu timp liniar, care să parcurgă tabloul T o singură dată. Putem folosi următoarea procedură

```
procedure Partition ( $T, p, r$ )
1:  $x \leftarrow T[p]$ 
2:  $i \leftarrow p - 1$ ;
3:  $j \leftarrow r + 1$ 
4: while TRUE do
5:   repeat  $j \leftarrow j - 1$  until  $T[j] \leq x$ 
6:   repeat  $i \leftarrow i + 1$  until  $T[i] > x$ 
7:   if  $i < j$ 
8:     then interschimbă  $T[i] \leftrightarrow T[j]$ 
9:   else return  $j$ 
```

Intuitiv, ne dăm seama că algoritmul *quicksort* este inefficient, dacă se întâmplă în mod sistematic ca subcazurile să fie puternic neechilibrate. Operația de pivotare necesită un timp în $\Theta(n)$.

Dacă elementele lui T sunt distincte, cazul cel mai nefavorabil este atunci când inițial tabloul este ordonat crescător sau descrescător, fiecare partiționare fiind total neechilibrată.

SARCINA DE BAZĂ:

1. Studiați noțiunile teoretice despre metoda divide et impera.
2. Implementați algoritmi MergeSort și QuickSort.
3. Efectuați analiza empirică a algoritmilor MergeSort și QuickSort.
4. Faceți o concluzie asupra lucrării efectuate.

Întrebări de control:

1. De ce această metodă se numește divide et impera ?
2. Explicați noțiunea de algoritm recursiv.
3. Descrieți etapele necesare pentru rezolvarea unei probleme prin metoda divide et impera.
4. Ce tip de funcție descrie timpul de execuție al unui algoritm de tipul divide et impera?
5. Care sunt avantajele și dezavantajele algoritmilor divide et impera?

LUCRAREA DE LABORATOR NR.3

Tema: Algoritmi greedy

Scopul lucrării:

1. Studiarea tehnicii greedy.
2. Analiza și implementarea algoritmilor greedy.

Note de curs:

1. Tehnica greedy

Algoritmii *greedy* (greedy = lacom) sunt în general simpli și sunt folosiți la rezolvarea problemelor de optimizare, cum ar fi: să se găsească cea mai bună ordine de executare a unor lucrări pe calculator, să se găsească cel mai scurt drum într-un graf etc. În cele mai multe situații de acest fel avem:

- mulțime de *candidați* (lucrări de executat, vârfuri ale grafului etc);
- o funcție care verifică dacă o anumită mulțime de candidați constituie o *soluție posibilă*, nu neapărat optimă, a problemei;
- o funcție care verifică dacă o mulțime de candidați este *fezabilă*, adică dacă este posibil să completăm această mulțime astfel încât să obținem o soluție posibilă, nu neapărat optimă, a problemei;
- o *funcție de selecție* care indică la orice moment care este cel mai promițător dintre candidații încă nefolosiți;
- o *funcție obiectiv* care dă valoarea unei soluții (timpul necesar executării tuturor lucrărilor într-o anumită ordine, lungimea drumului pe care l-am găsit etc); aceasta este funcția pe care urmărim să o optimizăm (minimizăm/maximizăm).

Pentru a rezolva problema de optimizare, se caută o soluție posibilă care să optimizeze valoarea funcției obiectiv. Un algoritm greedy construiește soluția pas cu pas. Inițial, mulțimea candidaților selectați este vidă. La fiecare pas, se adaugă acestei mulțimi cel

mai promițător candidat, conform funcției de selecție. Dacă, după o astfel de adăugare, mulțimea de candidați selectați nu mai este fezabilă, se elimină ultimul candidat adăugat; acesta nu va mai fi niciodată considerat. Dacă, după adăugare, mulțimea de candidați selectați este fezabilă, ultimul candidat adăugat va rămâne de acum încolo în ea. De fiecare dată când se lărgeste mulțimea candidaților selectați, se verifică dacă această mulțime nu constituie o soluție posibilă a problemei. Dacă algoritmul greedy funcționează corect, prima soluție găsită va fi totodată o soluție optimă a problemei. Soluția optimă nu este în mod necesar unică: se poate că funcția obiectiv să aibă aceeași valoare optimă pentru mai multe soluții posibile. Funcția de selecție este de obicei derivată din funcția obiectiv; uneori aceste două funcții sunt chiar identice.

2. Arbori de acoperire cu cost minim

Fie $G = \langle V, M \rangle$ un graf neorientat conex, unde V este mulțimea vârfurilor și M este mulțimea muchiilor. Fiecare muchie are un *cost* nenegativ w (sau o *lungime* nenegativă). Problema este să găsim o submulțime $A \subseteq M$, astfel încât toate vârfurile din V să rămână conectate atunci când sunt folosite doar muchii din A , iar suma lungimilor muchiilor din A să fie cât mai mică. Căutăm deci o submulțime A de cost total minim. Această problemă se mai numește și *problema conectării orașelor cu cost minim*, având numeroase aplicații.

Graful parțial $\langle V, A \rangle$ este un arbore și este numit *arborele de acoperire minim* al grafului G (*minimal spanning tree* (MST)). Un graf poate avea mai mulți arbori de acoperire de cost minim. Vom prezenta doi algoritmi greedy care determină arborele de acoperire minim al unui graf. În terminologia metodei greedy, vom spune că o mulțime de muchii este o *soluție*, dacă constituie un arbore de acoperire al grafului G , și este *fezabilă*, dacă nu conține cicluri. O mulțime fezabilă de muchii este *promițătoare*, dacă poate fi completată pentru a forma soluția optimă. O muchie *atinge* o

mulțime dată de vârfuri, dacă exact un capăt al muchiei este în mulțime.

Mulțimea inițială a candidaților este M . Cei doi algoritmi greedy aleg muchiile una câte una într-o anumită ordine, această ordine fiind specifică fiecărui algoritm.

2.1. Algoritmul lui Kruskal

Arborele de acoperire minim poate fi construit muchie, cu muchie, după următoarea metoda a lui Kruskal (1956): se alege întâi muchia de cost minim, iar apoi se adaugă repetat muchia de cost minim nealeasă anterior și care nu formează cu precedentele un ciclu. Alegem astfel $V-1$ muchii. Este ușor de dedus că obținem în final un arbore.

În algoritmul lui Kruskal, la fiecare pas, graful parțial $\langle V, A \rangle$ formează o pădure de componente conexe, în care fiecare componentă conexă este la rândul ei un arbore de acoperire minim pentru vârfurile pe care le conectează. În final, se obține arborele parțial de cost minim al grafului G .

Pentru a implementa algoritmul, trebuie să putem manipula submulțimile formate din vârfurile componentelor conexe. Folosim pentru aceasta o structură de date pentru mulțimi disjuncte pentru prezentarea mai multor mulțimi de elemente disjuncte [Cormen]. Fiecare mulțime conține vârfurile unui arbore din pădurea curentă. Funcția **Find-Set** (u) returnează un element reprezentativ din mulțimea care îl conține pe u . Astfel, putem determina dacă două vârfuri u și v aparțin aceluiași arbore testând dacă **Find-Set** (u) este egal cu **Find-Set** (v). Combinarea arborilor este realizată de procedura **Union**. În acest caz, este preferabil să reprezentăm graful ca o listă de muchii cu costul asociat lor, astfel încât să putem ordona această listă în funcție de cost. În continuare este prezentat algoritmul:

MST - Kruskal(G, w)

```
1:  $A \leftarrow \emptyset$ 
2: for fiecare vârf  $v \in V[G]$  do
3:   Make-Set ( $v$ )
4: sortează muchiile din  $M$  crescător în funcție de cost
5: for fiecare muchie  $(u, v) \in M$  do
6:   if Find-Set ( $u$ )  $\neq$  Find-Set ( $v$ )
7:     then  $A \leftarrow A \cup \{(u, v)\}$ 
8:     Union ( $u, v$ )
9: return  $A$ 
```

Modul de lucru al algoritmului Kruskal:

Liniile 1-3 inițializează mulțimea A cu mulțimea vidă și creează $|V|$ arbori, unul pentru fiecare vârf. Muchiile din M sunt ordonate crescător după cost, în linia 4. Bucla **for** din liniile 5-8 verifică, pentru fiecare muchie (u, v) , dacă punctele terminale u și v aparțin aceluiași arbore. Dacă fac parte din același arbore, atunci muchia (u, v) nu poate fi adăugată la pădure fără a se forma un ciclu și ea este respinsă. Altfel, cele două vârfuri aparțin unor arbori diferiți, și muchia (u, v) este adăugată la A în linia 7, vârfurile din cei doi arbori fiind reunite în linia 8.

2.2. Algoritmul lui Prim

Cel de-al doilea algoritm greedy pentru determinarea arborelui de acoperire minimal al unui graf se datorează lui Prim (1957). În acest algoritm, la fiecare pas, mulțimea A de muchii alese împreună cu mulțimea U a vârfurilor pe care le conectează formează un arbore parțial de cost minim pentru subgraful $\langle U, A \rangle$ al lui G . Inițial, mulțimea U a vârfurilor acestui arbore conține un singur vârf oarecare din V , care va fi rădăcina, iar mulțimea A a muchiilor este vidă. La fiecare pas, se alege o muchie de cost minim, care se adaugă la arborele precedent, dând naștere unui nou arbore parțial de cost minim. Arborele parțial de cost minim crește “natural”, cu

cate o ramură, până când va atinge toate vârfurile din V , adică până când $U = V$.

Cheia implementării eficiente a algoritmului lui Prim este să procedăm în așa fel încât să fie ușor să selectăm o nouă muchie pentru a fi adăugată la arborele format de muchiile din A . În pseudocodul de mai jos, graful conex G și rădăcina r a arborelui minim de acoperire, care urmează a fi dezvoltat, sunt privite ca date de intrare pentru algoritm. În timpul execuției algoritmului, toate vârfurile care *nu* sunt în arbore se află într-o coadă de prioritate Q bazată pe un câmp *key*. Pentru fiecare vârf v , $key[v]$ este costul minim al oricărei muchii care îl unește pe v cu un vârf din arbore. Prin convenție, $key[v] = \infty$ dacă nu există o astfel de muchie. Câmpul $\pi[v]$ reține „părintele” lui v din arbore. $Adj[u]$ este lista de adiacență cu vârfurile u .

MST - Prim(G, w, r)

```
1:  $Q \leftarrow V[G]$ 
2: for fiecare vârf  $u \in Q$ 
3:     do  $key[u] \leftarrow \infty$ 
4:  $key[r] \leftarrow 0$ 
5:  $\pi[r] \leftarrow \text{NIL}$ 
6: while  $Q \neq \emptyset$ 
7:     do  $u \leftarrow \text{Extract-Min}(Q)$ 
8:     for fiecare vârf  $v \in Adj[u]$ 
9:         do if  $v \in Q$  and  $w(u, v) < key[v]$ 
10:            then  $\pi[v] \leftarrow u$ 
11:                 $key[v] \leftarrow w(u, v)$ 
```

Modul de lucru al algoritmului Prim:

În liniile 1-4 se inițializează coada de prioritate Q , astfel încât aceasta să conțină toate vârfurile și se inițializează câmpul *key* al fiecărui vârf cu ∞ , excepție făcând rădăcina r , al cărei câmp *key* este inițializat cu 0. În linia 5 se inițializează $\pi[r]$ cu NIL, deoarece rădăcina r nu are nici un părinte. Pe parcursul execuției algoritmului, mulțimea $V - Q$ conține vârfurile arborelui curent. În

linia 7 este identificat un vârf $u \in Q$ incident unei muchii ușoare care traversează tăietura $(V - Q, Q)$ (cu excepția primei iterații, în care $u = v$ datorită liniei 4). Eliminarea lui u din mulțimea Q îl adaugă pe acesta mulțimii $V - Q$ a vârfurilor din arbore. În liniile 8-11 se actualizează câmpurile key și π ale fiecărui vârf v adiacent lui u , dar care nu se află în arbore. Actualizarea respectă condițiile $key[v] = w(v, \pi[v])$, și $(v, \pi[v])$ să fie o muchie ușoară care îl unește pe v cu vârf din arbore.

SARCINA DE BAZĂ:

1. De studiat tehnica greedy de proiectare a algoritmilor.
2. De implementat într-un limbaj de programare algoritmi Prim și Kruskal.
3. De făcut analiza empirică a algoritmilor Kruskal și Prim.
4. De alcătuit un raport.

Întrebări de control:

1. Descrieți metoda greedy.
2. Când se aplică algoritmul Kruskal și când algoritmul Prim ?
3. Cum pot fi îmbunătățite performanțele algoritmului Dijkstra?
4. Ce tip de date este comod de folosit la elaborarea programului al unui algoritm de tip greedy?
5. Care sunt avantajele și dezavantajele algoritmilor Prim și Kruskal?

LUCRAREA DE LABORATOR NR.4

Tema: Metoda programării dinamice

Scopul lucrării:

1. Studiarea metodei programării dinamice.
2. Analiza și implementarea algoritmilor de programare dinamică.
3. Compararea tehnicii greedy cu metoda de programare dinamică.

Note de curs:

1. Programarea dinamică.

O problemă rezolvabilă prin metoda programării dinamice trebuie adusă mai întâi la o formă discretă în timp. Deciziile care se iau pentru a obține un rezultat trebuie să se poată lua pas cu pas. De asemenea, foarte importantă este ordinea în care acestea se iau. Programarea dinamică este (și nu luați aceste rânduri ca pe o definiție) în esență un proces decizional în mai multe etape: în starea inițială a problemei luăm prima decizie, care determină o nouă stare a problemei în care luăm o decizie. Termenul dinamic se referă chiar la acest lucru: problema este rezolvată în etape dependente de timp. Variabilele, sau funcțiile care descriu fiecare etapă trebuie să fie în așa fel definite încât să descrie complet un proces, deci pentru acest lucru va trebui să răspundem la două întrebări:

- care este etapa *inițială* (caz în care avem de a face cu un proces decizional descendent) sau care este etapa *finală* (caz în care avem de a face cu un proces decizional ascendent)?
- care este *regula* după care trecem dintr-o etapă în alta ? De obicei această regulă este exprimată printr-o *recurență*.

Deoarece, avem de a face cu o problemă care se rezolvă în mai multe etape, nu ne mai rămâne decât să vedem cum luăm deciziile dintr-o etapă în alta.

De exemplu, problema calculului *numerelor lui Fibonacci* se încadrează în categoria programării dinamice deoarece:

- este un proces în etape;
- fiecărei etape k îi corespunde calculul celui de al k -lea număr Fibonacci;

- există o singură decizie pentru a trece la o etapă superioară;

Determinarea unui drum ce leagă două orașe A și B și care trece printr-un număr minim de alte orașe este tot o problemă de programare dinamică deoarece:

- este un proces în etape,
- fiecărei etape k îi corespunde determinarea unui drum de lungime k ce pleacă din orașul A ,
- dar există mai multe decizii pentru trecerea la drumul de lungimea $k + 1$.

În cele ce urmează prin strategie înțelegem un șir de decizii. Conform principiului lui Bellman, numit *principiul optimalității* avem:

O strategie are proprietatea că oricare ar fi starea inițială și decizia inițială, deciziile rămase trebuie să constituie o strategie optimă privitoare la starea care rezultă din decizia anterioară.

Demonstrarea corectitudinii unui algoritm de programare dinamică se face, așa cum rezultă și din principiul optimalității, prin *inducție matematică*.

Definiția 1. Fie P problema care trebuie rezolvată. Simbolul P codifică problema inițială împreună cu dimensiunea datelor de intrare. O subproblemă Q_i (care este rezolvată la etapa i) are aceeași formă ca P , dar datele de intrare pe care le prelucrează sunt mai mici în comparație cu cele cu care lucrează P . Cuvântul dinamic vrea să sugereze tocmai acest lucru: uniformitatea subproblemelor

și rezolvarea lor în mod ascendent. Pe scurt Q_i se obține din P printr-o restricționare a datelor de intrare.

Din această definiție rezultă așa-zisa dependență (mai precis incluziune) dintre subprobleme. O subproblemă Q_i obține alte subprobleme dacă datele asupra cărora operează Q_i sunt mai mari decât datele asupra cărora operează subproblemele conținute, deci nu este vorba doar de o simplă dependență între subprobleme, acestea fiind incluse una în cealaltă.

Există două tipuri de subprobleme: *directe* care rezultă din relația de recurență și subprobleme *indirecte* care de fapt sunt sub-subprobleme ale problemei inițiale. De exemplu, în cazul numerelor lui Fibonacci, pentru determinarea termenului $F(5)$ subproblemele directe sunt $F(4)$ și $F(3)$, iar $F(2)$, $F(1)$ și $F(0)$ sunt subprobleme indirecte.

Definirea relațiilor dintre etape se face recursiv. Prin prisma unui matematician acest lucru este inconvenient, dar orice informatician știe că recursivitatea înseamnă resurse (timp și memorie) consumate. Pe lângă inconveniențele legate de apelurile recursive, o subproblemă este calculată de mai multe ori. Aceste lucruri pot fi evitate dacă subproblemele se calculează începând de jos în sus (adică de la cea mai “mică” la cea mai “mare”) și se rețin rezultatele obținute. În cazul numerelor Fibonacci, cea mai mică subproblemă este calcularea lui Fibonacci(0) și a lui Fibonacci(1), iar cea mai mare subproblemă este calcularea lui Fibonacci(N) unde N este un număr dat.

O problemă de programare dinamică se poate prezenta sub forma unui graf orientat. Fiecărui nod îi corespunde o etapă (sau o subproblemă), iar din relațiile de recurență se deduce modul de adăugare a arcelor. Mai precis, vom adăuga un arc de la starea (etapa) i la starea (etapa) j dacă starea j depinde direct de starea i . Dependența directă dintre etape este dată de relațiile de recurență.

Construirea unui astfel de graf este echivalentă cu rezolvarea problemei. Determinarea șirului deciziilor care au adus la soluție se reduce la o problemă de drum în grafuri.

Notăm cu P , problema pe care o dorim să o rezolvăm. Înțelesul pe care îl dăm lui P include și dimensiunea datelor de la intrare.

Am spus anterior că o etapă, sau o subproblemă are aceeași formă ca și problema de rezolvat la care sunt adăugate câteva restricții.

Pentru ca aceste probleme să poată fi calculate trebuie stabilită o ordine în care ele vor fi prelucrate. Considerând reprezentarea sub forma unui graf (nodurile corespund etapelor, muchiilor - deciziilor), va trebui să efectuăm o sortare topologică asupra nodurilor grafului. Fie Q_0, Q_1, \dots, Q_N ordinea rezultată unei astfel de sortări. Prin Q_i am notat o subproblemă (Q_0 este subproblema cea mai mică). În cazul submulțimii de sumă dată, Q_i este chiar descompunerea lui i în sumă de numere din vectorul dat. Fie S_i soluția problemei Q_i . Soluția subproblemei Q_N este soluția problemei P .

Algoritmul general este:

procedure Rezolva(P)

{inițializează (S_0) }

1: **for** $i \leftarrow 1$ to N do

2: **progresează** (S_0, \dots, S_{i-1}, S_i); {aceasta procedura calculează soluția problemei Q_i pe baza soluțiilor problemelor Q_0, \dots, Q_{i-1} , cu o recurență de jos în sus.}

3: **return** (S_N)

Demonstrarea corectitudinii unui astfel de algoritm se face prin inducție după i .

Referitor la complexitatea unui algoritm de programare dinamică, putem spune că aceasta depinde de mai mulți factori: numărul de stări, numărul de decizii cu care se poate trece într-o stare, complexitatea subproblemei inițiale (Q_0)... . Ceea ce apare însă în toate problemele, este numărul de stări (etape). Deci complexitatea va avea forma $O(N \cdot \dots)$.

Noțiunea de complexitate pseudo – polinomială, de asemenea este legată uneori de *complexitatea unui algoritm de programare*

dinamică. Fie D mulțimea tuturor intrărilor corecte pentru o problemă dată. Definem două funcții:

$Max: D \rightarrow Z^+$ și

$Lungimea: D \rightarrow Z^+$.

Funcția Max indică valoarea maximă a datelor de intrare, iar funcția $Lungimea$ indică lungimea lor.

Un algoritm este numit *algoritm pseudo – polinomial*, dacă funcția sa de complexitate este mărginită superior de o funcție polinomială în două variabile: $Max[I]$ și $Lungime[I]$. Prin definiție orice algoritm polinomial este și pseudo – polinomial, deoarece se execută într-un timp mărginit de un polinom de $Lungime[I]$, dar nu toți algoritmi pseudo-polinomiali sunt polinomiali. Pentru problemele care au proprietatea că $Max[I]$ este mărginită de o funcție polinomială în variabila $Lungime [I]$ nu există nici o distincție între algoritmi polinomiali și cei pseudo-polinomiali. Să vedem când se poate și când nu se poate construi un algoritm pseudo-polinomial? Spunem despre o problemă că este *number-problem* dacă nu există nici o funcție polinomială p , astfel încât $Max[I] \leq p(Lungime[I])$ pentru orice intrare I corectă.

Singurele probleme NP-complete care sunt candidate pentru a fi rezolvate prin algoritmi pseudo-polinomială se află în această clasă de probleme.

Există mai multe clasificări ale problemelor de programare dinamică:

1. După natura deciziilor pot fi *deterministe* sau *nedeterministe*. La cele nedeterministe, în scrierea relațiilor de recurență intervine și o probabilitate (probabilitatea ca produsul să fie vândut, ca individul x să ajungă într-un punct dat, ca moneda să cadă pe partea cu stema, ca o mașină să se strice după ce a produs un număr de piese.)

2. După valoarea deciziilor pot fi *de optimizare* (determinarea drumurilor de cost minim între două noduri ale unui graf), *de neoptimizare* (determinarea numerelor Fibonacci).

3. După numărul de dimensiuni (sau numărul de parametri ai funcțiilor de recurență) pot fi *unidimensionale* (precum numerele lui Fibonacci), sau *multidimensionale*.

4. După numărul deciziilor care se pot lua la un pas pot fi *unidecisionale* (ca numerele lui Fibonacci), sau *multidecisionale* (ca drumul de lungime minimă între două noduri ale unui graf). Problemele unidecisionale sunt de obicei de neoptimizare.

5. După direcția deciziilor putem avea programare dinamică *înainte* în care starea i se calculează în funcție de stările $i + 1, i + 2, \dots$, sau programare dinamică *înapoi* în care starea i se calculează în funcție de stările $i - 1, i - 2, \dots$.

Dezvoltarea unui algoritm bazat pe programarea dinamică poate fi împărțită într-o secvență de patru pași:

1. Caracterizarea structurii unei soluții optime.
2. Definierea recursivă a valorii unei soluții optime.
3. Calculul valorii unei soluții optime într-o manieră de tip "bottom-up".
4. Construirea unei soluții optime din informația calculată.

Pașii 1-3 sunt baza unei abordări de tip programare dinamică. Pasul 4 poate fi omis dacă se dorește doar calculul unei singure soluții optime. În vederea realizării pasului 4, deseori se păstrează informație suplimentară de la execuția pasului 3, pentru a ușura construcția unei soluții optime.

2. Probleme de drum minim în grafuri

2.1. Cele mai scurte drumuri care pleacă din același punct

Fie $G = \langle V, A \rangle$ un graf orientat, unde V este mulțimea vârfurilor și A este mulțimea arcelor. Fiecare arc are o lungime nenegativă. Unul din vârfuri este ales ca vârf *sursă*. Problema este de a determina lungimea celui mai scurt drum de la sursă către fiecare vârf din graf.

Se va folosi un algoritm greedy, datorat lui Dijkstra (1959). Notăm cu C mulțimea vârfurilor disponibile (candidații) și cu S mulțimea vârfurilor deja selectate. În fiecare moment, S conține

acele vârfuri a căror distanță minimă de la sursă este deja cunoscută, în timp ce mulțimea C conține toate celelalte vârfuri. La început, S conține doar vârful sursă, iar în final S conține toate vârfurile grafului. La fiecare pas, adăugăm în S acel vârf din C a cărui distanță de la sursă este cea mai mică.

Se spune, că un drum de la sursă către un alt vârf este *special*, dacă toate vârfurile intermediare de-a lungul drumului aparțin lui S . Algoritmul lui Dijkstra lucrează în felul următor. La fiecare pas al algoritmului, un tablou D conține lungimea celui mai scurt drum special către fiecare vârf al grafului. După ce se adaugă un nou vârf v la S , cel mai scurt drum special către v va fi, de asemenea, cel mai scurt dintre toate drumurile către v . Când algoritmul se termină, toate vârfurile din graf sunt în S , deci toate drumurile de la sursă către celelalte vârfuri sunt speciale și valorile din D reprezintă soluția problemei.

Presupunem că vârfurile sunt numerotate, $V = \{1, 2, \dots, n\}$, vârful 1 fiind sursa, și că matricea L dă lungimea fiecărui arc, cu $L[i, j] = +\infty$, dacă arcul (i, j) nu există. Soluția se va construi în tabloul $D[2 .. n]$. Algoritmul este:

```

function Dijkstra( $L[1 .. n, 1 .. n]$ )
1:  $C \leftarrow \{2, 3, \dots, n\}$      $\{S = V \setminus C$  există doar implicit $\}$ 
2: for  $i \leftarrow 2$  to  $n$  do  $D[i] \leftarrow L[1, i]$ 
3: repeat  $n-2$  times
4:      $v \leftarrow$  vârful din  $C$  care minimizează  $D[v]$ 
5:      $C \leftarrow C \setminus \{v\}$      $\{$ si, implicit,  $S \leftarrow S \cup \{v\}\}$ 
6:     for fiecare  $w \in C$  do
7:          $D[w] \leftarrow \min(D[w], D[v]+L[v, w])$ 
return  $D$ 

```

Proprietatea 1. În algoritmul lui Dijkstra, dacă un vârf i

- este în S , atunci $D[i]$ dă lungimea celui mai scurt drum de la sursă către i ;
- nu este în S , atunci $D[i]$ dă lungimea celui mai scurt drum special de la sursă către i .

La terminarea algoritmului, toate vârfurile grafului, cu excepția unuia, sunt în S . Din proprietatea precedentă, rezulta că algoritmul lui Dijkstra funcționează corect.

2.2. Determinarea celor mai scurte drumuri într-un graf

Fie $G = \langle V, A \rangle$ un graf orientat, unde V este mulțimea vârfurilor și A este mulțimea arcelor. Fiecărui arc i se asociază o lungime nenegativă. Să se calculeze lungimea celui mai scurt drum între fiecare pereche de varfuri.

Vom presupune că vârfurile sunt numerotate de la 1 la n și că matricea L dă lungimea fiecărui arc: $L[i, i] = 0$, $L[i, j] \geq 0$ pentru $i \neq j$, $L[i, j] = +\infty$ dacă arcul (i, j) nu există.

Principiul optimalității este valabil: dacă cel mai scurt drum de la i la j trece prin varful k , atunci porțiunea de drum de la i la k , cât și cea de la k la j , trebuie să fie, de asemenea, optime.

Construim o matrice D care să conțină lungimea celui mai scurt drum între fiecare pereche de vârfuri. Algoritmul de programare dinamică inițializează pe D cu L . Apoi, efectuează n iterații. După iterația k , D va conține lungimile celor mai scurte drumuri care folosesc ca vârfuri intermediare doar vârfurile din $\{1, 2, \dots, k\}$. După n iterații, obținem rezultatul final. La iterația k , algoritmul trebuie să verifice, pentru fiecare pereche de vârfuri (i, j) , dacă există sau nu un drum, trecând prin varful k , care este mai bun decât actualul drum optim ce trece doar prin vârfurile din $\{1, 2, \dots, k-1\}$. Fie D_k matricea D după iterația k . Verificarea necesară este atunci:

$$D_k[i, j] = \min(D_{k-1}[i, j], D_{k-1}[i, k] + D_{k-1}[k, j])$$

unde s-a făcut uz de principiul optimalității pentru a calcula lungimea celui mai scurt drum față de k . Implicit, s-a considerat că un drum optim care trece prin k nu poate trece de două ori prin k .

Acest algoritm simplu este datorat lui Floyd (1962):

```

function Floyd( $L[1 \dots n, 1 \dots n]$ )
1: array  $D[1 \dots n, 1 \dots n]$ 
2:  $D \leftarrow L$ 
3: for  $k \leftarrow 1$  to  $n$  do
4:   for  $i \leftarrow 1$  to  $n$  do
5:     for  $j \leftarrow 1$  to  $n$  do
6:        $D[i, j] \leftarrow \min(D[i, j], D[i, k] + D[k, j])$ 
return  $D$ 

```

Se poate deduce că algoritmul lui Floyd necesită un timp în $\Theta(n^3)$. Un alt mod de a rezolva această problemă este să se aplice algoritmul *Dijkstra* prezentat mai sus de n ori, alegând mereu un alt vârf sursă. Se obține un timp în $n \Theta(n^2)$, adică tot în $\Theta(n^3)$. Algoritmul lui Floyd, datorită simplității lui, are însă constanta multiplicativă mai mică, fiind probabil mai rapid în practică.

SARCINA DE BAZĂ:

1. De studiat metoda programării dinamice de proiectare a algoritmilor.
2. De implementat într-un limbaj de programare algoritmi prezențați mai sus.
3. De făcut analiza empirică a acestor algoritmi pentru un graf rar și pentru un graf dens.
4. De alcătuit un raport.

Întrebări de control:

1. Descrieți metoda programare dinamică.
2. De ce această metodă se numește programare dinamică?
3. Care este diferența între metoda divide et impera și metoda programării dinamice?
4. Care este clasificarea problemelor de programare dinamică?
5. Ce se întâmplă în cazul când costurile arcelor grafurilor prelucrate cu algoritmi Dijkstra și Floyd sunt negative?

LUCRAREA DE LABORATOR NR.5

Tema: Tehnica căutării cu revenire (backtracking)

Scopul lucrării:

1. Studiarea metodei backtracking.
2. Analiza și implementarea algoritmilor bazați pe metoda backtracking.
3. Compararea metodei backtracking cu metodele studiate pînă acum.

Note de curs:

1. Principiul de bază al metodei backtracking

Backtracking (în traducere aproximativă, “cautare cu revenire”) este un principiu fundamental de elaborare a algoritmilor pentru probleme de optimizare, sau de găsire a unor soluții care îndeplinesc anumite condiții. Majoritatea problemelor ce pot fi rezolvate prin această tehnică pot fi reduse la determinarea unei submulțimi a unui produs cartezian de forma $A_1 \times A_2 \times \dots \times A_n$ (cu submulțimile A_k finite). Fiecare element al submulțimii poate fi văzut ca o soluție. O soluție este de forma $s = (s_1, s_2, \dots, s_n)$ cu $s_k \in A_k = \{a_1^k, a_2^k, \dots, a_{m_k}^k\}$, $m_k = \text{card}A_k$. În majoritatea cazurilor nu orice element al produsului cartezian este soluție ci doar cele care satisfac anumite restricții.

Metoda backtracking constă în construirea soluției s completând pe rând toate componentele s_k pentru $k = 1, 2, \dots, n$. Specifică este maniera de parcurgere a spațiului soluțiilor:

- Soluțiile sunt construite succesiv, la fiecare etapă fiind completată câte o componentă (la fel ca tehnica greedy însă ulterior se poate reveni asupra alegerii unei componente);
- Alegerea unei valori pentru o componentă se face într-o anumită ordine (aceasta presupune că pe mulțimile A_k există o

relație de ordine și se realizează o parcurgere sistematică a spațiului $A_1 \times A_2 \times \dots \times A_n$;

- La completarea componentei k se verifică dacă soluția parțială (s_1, s_2, \dots, s_k) , verifică condițiile induse de restricțiile problemei. O soluție parțială care satisface restricțiile problemei este numită soluție fezabilă deoarece poate conduce la o soluție a problemei.

- Dacă au fost încercate toate valorile corespunzătoare componentei k și nu a fost găsită o soluție sau dacă se dorește determinarea unei noi soluții atunci se revine la componenta $(k-1)$ și se încearcă următoarea valoare corespunzătoare acesteia ș.a.m.d. Această revenire la o componentă anterioară este specifică tehnicii backtracking.

- Procesul de căutare și revenire este continuat până când este găsită o soluție sau până când au fost testate toate configurațiile posibile.

În aplicarea metodei pentru rezolvarea unei probleme concrete se parcurg următoarele etape:

- Se alege o reprezentare a soluției sub forma unui vector cu n componente;

- Se identifică mulțimile A_1, A_2, \dots, A_n și relațiile de ordine care indică modul de parcurgere a fiecărei mulțimi;

- Pornind de la restricțiile problemei se stabilesc condițiile de validitate ale soluțiilor parțiale (condițiile de continuare).

În descrierea algoritmului se vor folosi notațiile următoare: $A_k = \{a_1^k, a_2^k, \dots, a_{m_k}^k\}$, m_k – numărul de elemente din A_k ; k indică componenta curentă din s ; i_k reprezintă indicele elementului curent din A_k .

Structura generală a algoritmului este:

backtracking(n, A_1, \dots, A_n)

1: $k \leftarrow 1$

2: $i_k \leftarrow 0$

3: **while** $k > 0$ **do**

4: $i_k \leftarrow i_k + 1$

```

5:  valid ← FALSE
    //căutarea unei componente valide pentru poziția k
6:  while (valid = FALSE) and ( $i_k \leq m_k$ ) do
7:       $s_k \leftarrow a_{i_k}^k$ 
8:      if ( $s_1, \dots, s_k$ ) „satisface condițiile de continuare” then
9:          valid ← TRUE
10:     else  $i_k \leftarrow i_k + 1$ 
11:  if valid = TRUE then
12:     if ( $s_1, \dots, s_k$ ) este soluție then „s-a găsit soluția”
13:     else  $k \leftarrow k + 1$ ;  $i_k \leftarrow 0$ 
14:  else  $k \leftarrow k - 1$ 

```

Observații. În problemele concrete pot apărea următoarele situații:

1. Mulțimile A_1, \dots, A_n nu sunt distincte sau elementele lor pot fi generate pe parcursul algoritmului fără a fi necesară transmiterea lor ca argument algoritmului.

2. Condițiile de continuare se deduc din restricțiile problemei.

3. Pentru unele probleme soluția este obținută în cazul în care $k = n$ iar pentru altele este posibil să fie satisfăcută o condiție de găsire a soluției pentru $k < n$ (pentru unele probleme nu toate soluțiile conțin același număr de elemente).

4. La găsirea unei soluții de cele mai multe ori aceasta se afișează sau se memorizează într-o zonă dedicată. Dacă se dorește identificarea unei singure soluții atunci căutarea se oprește după găsirea acesteia, altfel procesul de căutare continuă prin analiza următoarei valori a ultimei componente completate.

2. Problema colorării hărților

Formularea problemei. Se consideră o hartă cu n țări care urmează a fi colorată folosind $m < n$ culori, astfel încât oricare două țări vecine să fie colorate diferit. Relația de vecinătate dintre țări este

prezentată printr-o matrice de adiacență $n \times n$ ale cărei elemente sunt:

$$v_{ij} = \begin{cases} 1, & \text{dacă } i \text{ este vecină cu } j \\ 0, & \text{dacă } i \text{ nu este vecină cu } j \end{cases}$$

Reprezentarea soluțiilor. O soluție a problemei este o modalitate de colorare a țărilor și poate fi reprezentată printr-un vector (s_1, \dots, s_n) cu $s_i \in \{1, \dots, m\}$ reprezentând culoarea asociată țării i . Mulțimile de valori ale elementelor $A_1 = \dots = A_n = \{1, \dots, m\}$.

Restricții și condiții de continuare. Restricția ca două țări vecine să fie colorate diferit se specifică prin: $s_i \neq s_j$ pentru orice i și j care au proprietatea $v_{ij} = 1$. Condiția de continuare pe care trebuie s-o satisfacă soluția parțială (s_1, \dots, s_k) este: $s_k \neq s_i$ pentru orice $i < k$ cu proprietatea $v_{ik} = 1$.

Descrierea algoritmului.

Colorare (n)

```
1:  $k \leftarrow 1$ 
2:  $s[k] \leftarrow 0$ 
3: while  $k > 0$  do
4:    $s[k] \leftarrow s[k] + 1$ 
5:    $valid \leftarrow \text{FALSE}$ 
6:   while  $(valid = \text{FALSE})$  and  $(s[k] \leq m)$  do
7:     if  $validare(s[1..k])$  then
8:        $valid \leftarrow \text{TRUE}$ 
9:     else  $s[k] \leftarrow s[k] + 1$ 
11:  if  $valid = \text{TRUE}$  then
12:    if  $k = n$  then  $afișare(s[1..n])$ 
13:    else  $k \leftarrow k + 1; s[k] \leftarrow 0$ 
14:  else  $k \leftarrow k - 1$ 
```

Validare ($s[1..k]$)

```
1: for  $i \leftarrow 1, k - 1$  do
2:   if  $(s[k] = s[i])$  and  $(v[i, k] = 1)$  then return FALSE
3: return TRUE
```

3. Determinarea tuturor drumurilor dintre două orașe

Formularea problemei. Se consideră o mulțime de n orașe $\{o_1, o_2, \dots, o_n\}$ și o matrice binară cu n linii și n coloane prin care se specifică între care orașe există drumuri directe. Elementele matricei sunt de forma:

$$v_{ij} = \begin{cases} 1, & \text{dacă există drum direct între } i \text{ și } j \\ 0, & \text{altfel} \end{cases}$$

Se cere să se determine toate drumurile care leagă orașul o_p de orașul o_q .

Reprezentarea soluțiilor. În această problemă nu toate soluțiile au aceeași lungime. O soluție a problemei este de forma (s_1, \dots, s_m) cu $s_i \in \{1, 2, \dots, n\}$ indicând orașul care va fi parcurs în etapa i a traseului.

Restricții și condiții de continuare. O soluție (s_1, \dots, s_m) trebuie să satisfacă: $s_1 = p$ (orașul de start), $s_m = q$ (orașul destinație), $s_i \neq s_j$ pentru orice $i \neq j$ (nu se trece de două ori prin același oraș) $v_{s_i, s_{i+1}} = 1$ pentru $i = 1, 2, \dots, m-1$ (între orașele parcurse succesiv există drum direct). La completarea componentei k , condiția de continuare este $s_k \neq s_i$ pentru $i = 1, 2, \dots, k-1$ și $v_{s_{k-1}, s_k} = 1$. Condiția de găsim a soluției nu este determinată de numărul de componente completate ci de faptul că $s_k = q$

Descrierea algoritmului.

drumuri_rekursiv(k)

1: **if** $s[k-1] = q$ **then** *afișare* ($s[1..k-1]$)

2: **else**

7: **if** $k \neq n + 1$ **then**

8: **for** $i \leftarrow 1, n$ **do**

9: $s[k] \leftarrow i$

11: **if** *validare* ($s[1..k]$) **then**

drumuri_rekursiv($k+1$)

validare ($s[1..k]$)

```
1: if  $v[s[k-1], s[k]] = 0$  then return FALSE
2: else
3:   for  $i \leftarrow 1, k - 1$  do
2:     if ( $s[k] = s[i]$ ) then return FALSE
3: return TRUE
```

SARCINA DE BAZĂ:

1. De studiat tehnica backtracking de proiectare a algoritmilor.
2. De implementat într-un limbaj de programare unul dintre algoritmii prezentați mai sus.
3. De făcut analiza aceluși algoritm.
4. De alcătuit un raport.

Întrebări de control:

1. În ce constă metoda backtracking?
2. Cum se parcurge spațiul soluțiilor?
3. La rezolvarea unei probleme concrete ce etape se parcurg?
4. Ce este asemanator și diferit de tehnica greedy?
5. Care sunt condițiile de oprire a procesului de căutare în metoda backtracking?

LUCRAREA DE LABORATOR NR.6

Tema: Algoritmi genetici

Scopul lucrării:

1. Studiarea algoritmilor genetici.
2. Analiza și implementarea algoritmilor genetici.
3. Compararea algoritmilor genetici cu metoda backtracking .

Note de curs:

1. Introducere

Algoritmii genetici fac parte din clasa mai generală a algoritmilor evolutivi. Algoritmii evolutivi sunt metode de rezolvare a problemelor bazate pe o căutare în spațiul soluțiilor, în care se folosesc populații supuse unui proces de evoluție similar într-o măsură oarecare cu cel întâlnit în natură.

Principalele noțiuni care permit analogia între rezolvarea problemelor de căutare și evoluția în natură sunt următoarele:

- *Populație*. O populație este formată din indivizi care trăiesc într-un mediu la care trebuie să se adapteze.

- *Fitness*. Fiecare individ din populație este adaptat mai mult sau mai puțin la mediu. Fitness-ul este o măsură a gradului de adaptare la mediu. Scopul evoluției este ca toți indivizii să ajungă la un fitness cât mai bun.

- *Cromozom*. Este o mulțime ordonată de elemente, numite *gene* ale căror valori determină caracteristicile unui individ. În genetică pozițiile pe care se află genele în cadrul cromozomului se numesc *loci*, iar valorile pe care le pot lua se numesc *alele*.

- *Generație*. Este o etapă în evoluția unei populații. Dacă privim evoluția ca pe un proces iterativ în care o populație se transformă în alta atunci generația este o iterație în cadrul acestui proces.

- *Selecție.* Procesul de selecție naturală are ca efect supraviețuirea indivizilor cu grad ridicat de adaptare la mediu.

- *Reproducere.* Este procesul prin care se trece de la o generație la alta. Indivizii noii generații moștenesc caracteristici de la precursorii lor dar pot dobândi și caracteristici noi ca urmare a unor procese de mutație care au un caracter aleator. În cazul în care în procesul de reproducere intervin cel puțin doi părinți caracteristicile moștenite ale urmașului se obțin prin combinarea caracteristicilor părinților.

2. Structura generală a unui algoritm genetic

Algoritmii genetici sunt procese iterative prin care o populație inițializată aleator este transformată succesiv prin selecție, încrucișare și mutație până la atingerea unui număr anumit de iterații sau până la îndeplinirea unui alt criteriu de oprire.

Există un număr foarte mare de variante de algoritmi genetici datorită flexibilității lor. În general pentru fiecare problemă concretă apar elemente particulare în cadrul algoritmului. Cu toate acestea, majoritatea algoritmilor genetici se pot încadra în următoarea structură generală, unde $P(t) = (x_1(t), x_2(t), \dots, x_m(t))$ reprezintă populația corespunzătoare generației t iar $f(x_i)$ reprezintă gradul de adaptare la mediu al elementului x_i :

genetic algorithm ($P(t)$)

1: *Inițializare:* $P(0) = (x_1(0), \dots, x_m(0)), t = 0$

2: **repeat**

3: *evaluarea populației curente:* calcul $f(x_i(t))$ pentru $i = 1, 2, \dots, m$

4: *selecția părinților:* $P(t) \rightarrow P^1$

5: *generarea urmașilor prin încrucișare:* $P^1 \rightarrow P^2$

6: *modificarea urmașilor prin mutație:* $P^2 \rightarrow P^3$

7: *evaluarea populației de urmași:* calcularea valorilor lui f pentru elementele lui P^3

8: *selecția supraviețuitorilor:* $\{P(t), P^3\} \rightarrow P(t+1)$

9: *incrementarea contorului de generații: $t = t + 1$*

10: **until** $t < t_{\max}$

La proiectarea unui algoritm genetic trebuie să se stabilească:

- *Modul de codificare.* Se specifică modul în care fiecărei configurații din spațiul de căutare i se asociază un cromozom.

- *Funcția de adaptare.* Se construiește funcția care exprimă gradul de adaptare la mediu pornind de la restricțiile și funcția scop a problemei.

- *Modul de inițializare și dimensiunea populației.* Se pot utiliza populații de dimensiune fixă sau de dimensiune variabilă. De cele mai multe ori populația se inițializează cu elemente aleatoare din spațiul de căutare.

- *Mecanismul de selecție* a părinților și a supraviețuitorilor.

- *Mecanismul de combinare* a părinților pentru a genera urmași.

- *Mecanismul de mutație* prin care se asigură modificarea cromozomilor generați.

- *Criteriul de oprire.* Atunci când nu se cunoaște un criteriu specific problemei se optează pentru un număr maxim de iterații.

2.1. Codificarea cromozomilor

Codificarea spațiului de căutare este una dintre cele mai importante etape în proiectarea unui algoritm genetic deoarece determină modul în care se va desfășura procesul evolutiv. Această etapă se referă la următoarele aspecte:

- Structuri de date folosite;
- Regula de codificare;
- Regula de decodificare.

Structuri de date. În algoritmi genetici cromozomii sunt reprezentați prin structuri liniare cu număr fix de elemente (tablouri) cel mai des folosite sau cu număr variabil de elemente (liste înlănțuite).

Reguli de codificare. Modul în care sunt codificați cromozomii depinde de problema concretă. Există următoarele variante de codificare:

- *Codificare binară.* Varianta clasică de codificare a cromozomilor. Într-o astfel de codificare cromozomii sunt vectori cu elementele din mulțimea $\{0, 1\}$ iar spațiul de căutare este $S\{0, 1\}^n$, cu n gene (numărul de gene este corelat cu dimensiunea problemei). Codificarea binară este utilizată în problemele de optimizare combinatorială în care configurațiile pot fi specificate ca vectori binari.

- *Codificarea reală.* Este adecvată pentru problemele de optimizare pe domenii continue. În acest caz cromozomii sunt niște vectori cu elemente reale, fiind chiar elementele domeniului de definiție al funcției scop. Avantajul codificării reale este faptul că este naturală și nu necesită proceduri speciale de codificare și decodificare.

- *Codificarea specifică.* Fiecare cromozom este reprezentat sub forma unei permutări:

Cromozomul A	1 5 3 4 6 3 4 8 9 2
Cromozomul B	9 2 4 3 3 6 1 5 4 8

Această metodă de codificare se folosește în cazul problemei voiajorului comercial. O permutare reprezintă ordinea în care voiajorul comercial vizitează orașele.

Reguli de decodificare. Decodificarea asigură pregătirea evaluării configurației. Ea este necesară doar pentru codificarea binară și pentru codificarea specifică.

2.2. Populația inițială

Populația inițială este o mulțime de cromozomi care trebuie să fie suficient de mare pentru a se putea realiza un număr suficient de combinații între cromozomii componenți, dar nu prea mare, deoarece aceasta poate încetini algoritmul. De obicei populația inițială este una aleatoare, dar pentru o mai bună performanță, o

parte din cromozomi pot fi generați prin metode euristice. În acest mod s-ar putea ajunge la soluție mai repede.

2.3. Funcția de adecvare (fitness-ul)

Fitness-ul unui cromozom este speranța lui de viață sau gradul de adaptare la mediu. Scopul este de a obține cromozomi cu fitness cât mai mare sau cât mai mic, adică un fitness la extremă. Fitness-ul este exprimat de o funcție ce urmează a fi optimizată minimizată sau maximizată.

2.4. Selecția cromozomilor

Cromozomii sunt selectați din populație pentru a fi părinți într-o încrucișare.

Există mai multe metode de selecție, printre care:

1. Selecție aleatoare. Doi părinți sunt aleși aleator din populație. Aceasta se face generând două numere aleatoare între 1 și dimensiunea populației.

2. Selecție cu ajutorul ruletei. Părinții sunt selectați în conformitate cu fitness-ul lor. Cu cât sunt mai buni, cu atât șansa lor de a fi aleși este mai mare. Ne imaginăm o ruletă în care sunt dispuși toți cromozomii din populație. Fiecărui cromozom îi corespunde un sector al ruletei, direct proporțional, ca mărime, cu fitness-ul cromozomului respectiv. În acest fel cromozomii cu fitness mare au atașate sectoare mai mari, iar cei cu fitness mic au atașate sectoare mai mici. La aruncarea bilei pe ruletă există mai multe șanse de alegere pentru cromozomii cu fitness mai mare.

Simularea roții de ruletă se face în felul următor:

- Se calculează suma tuturor fitness-urilor cromozomilor (S).
- Se generează un număr aleator r între 1 și S .
- Se parcurge populația și se calculează suma fitness-urilor cromozomilor până ajungem la valoarea r . Acel cromozom îl alegem.

3. Selecție aranjată (Selecție pe baza rangurilor). Se ordonează crescător valorile funcției de adecvare pentru toți cromozomii. Se renumerează cu numere întregi din intervalul $[1, \dots, \text{dimensiunea populației}]$. Cromozomul cu fitness-ul cel mai mic are numărul 1 etc., iar cel mai mare are numărul egal cu dimensiunea populației. Aceste numere sunt considerate fitness-uri și pe ele se aplică selecția proporțională. Se observă că cromozomii cu fitness mai mare au la fel cele mai mari șanse de a fi aleși, dar de data aceasta nu mai sunt așa de mari în comparație cu șansele celorlalți.

4. Selecție de tip turneu. Se selectează uniform aleator câte k cromozomi din populație iar dintre acestea se alege cel care are cea mai mare valoare pentru funcția de adecvare. Cel mai frecvent se utilizează $k=2$.

5. Selecție prin trunchiere. Se folosește atunci când dintr-o populație sau o reuniune de populații se cere selectarea unei subpopulații. Sunt selectate atâtea elemente câte sunt necesare pentru a completa subpopulația în ordinea descrescătoare a valorii funcției de adecvare.

6. Elitism. Prin elitism se înțelege supraviețuirea celui mai bun dintre elementele generate până la un moment dat. Nu toate variantele de selecție satisfac această proprietate. De exemplu, selecția prin trunchiere este elitistă, însă cea de tip turneu nu este elitistă. O metodă de selecție poate fi transformată în una elitistă prin amplasarea explicită a celui mai bun element al populației curente în populația corespunzătoare generației următoare.

2.5. Încrucișarea (crossover)

Încrucișarea va depinde de tipul de codificare a cromozomilor și de particularitățile problemei pe care o rezolvăm. Vom analiza câteva metode de încrucișări:

- **Încrucișare într-un singur punct.** Se alege un punct de încrucișare. Din primul părinte se copie prima secvență, din al doilea părinte a doua secvență în raport cu punctul de încrucișare.

101001011 + 110111111 = 101001111

- **Încrucișare în două sau mai multe punct.** Sunt alese mai multe puncte de încrucișare. Secvențele dintre cele puncte alese sunt luate alternativ, din primul părinte apoi din al doilea s.a.m.d.

$$101001011 + 110111111 = 101111011$$

- **Încrucișare uniformă.** Biții sunt copiați aleator din primul și al doilea părinte.

$$101001011 + 110111111 = 101111011$$

- **Încrucișare aritmetică.** Pentru crearea de noi urmași putem folosi operațiile precum și funcții aritmetice (**and, or, not** etc)

$$101001011 + 110111111 = 100001011 \text{ (and)}$$

2.6. Mutația

Asigura alterarea valorii unor gene pentru a evita situațiile în care o anumită gena nu apare în populație fiindcă nu a fost generată de la început. În acest fel se asigură diversitatea populației. Prin mutație se evită căderea soluțiilor problemei într-un optim local. De regulă această operație se efectuează de nu prea multe ori, cu o probabilitate de circa 0,5%. În dependență de codificarea cromozomilor avem diferite metode de mutație.

Codificarea binară

Inversarea biților Biții selectați sunt inversați $1 \leftrightarrow 0$.

$$110010011 \rightarrow 111010011$$

Interschimbarea biților Biții selectați își schimbă valorile între ei.

$$110010011 \rightarrow 111010011$$

Codificarea sub formă de permutare

Transpoziție Asemănător interschimbării biților de la codificarea binară. Se efectuează o transpoziție asupra permutării respective.

$$(123467895) \rightarrow (129467835)$$

Codificarea sub formă de valori

Se adaugă sau se scade un număr mic la valorile selectate.

$$(4.63 \ 5.32 \ 2.89 \ 3.41 \ 9.67) \rightarrow (4.85 \ 5.32 \ 2.89 \ 3.63 \ 9.67)$$

2.7. Criteriul de oprire

Algoritmii genetici se termină de obicei, după ce s-au creat un număr predefinit de populații noi. În alte cazuri se poate determina dacă soluția s-a îmbunătățit de la o populație la alta. În caz afirmativ se continua cu crearea de noi populații, iar în caz negative se afișează cel mai bun cromozom din populația curentă.

SARCINA DE BAZĂ:

1. De studiat algoritmul genetic general și metodele specifice problemei de creare a algoritmului.
2. De implementat într-un limbaj de programare unul dintre algoritmii prezentați mai sus.
3. De făcut analiza acesui algoritm.
4. De alcătuit un raport.

Întrebări de control:

1. De unde este inspirată ideea algoritmilor genetici? Cu ce noțiuni se lucrează?
2. Ce tip de probleme putem rezolva cu acești algoritmi?
3. Descrieți structura generală a unui algoritm genetic?
4. Ce este asemanator și diferit de tehnica greedy?
5. Care sunt condițiile de oprire a unui algoritm genetic?

BIBLIOGRAFIE

1. Oltean M. Proiectarea și implementarea algoritmilor. Cluj 1999 Tipoholding, Editura Computer Libris Agora.
2. T. Cormen, Ch. Leiserson, R. Rivest. Introducere în algoritmi. Computer Libris Agora, Cluj-Napoca, 2000.
3. Donald E. Knuth. Fundamental Algorithms, volume 1 of The Art of Computer Programming. Addison-Wesley, 1973.
4. Donald E. Knuth. Seminumerical Algorithms, volume 2 of The Art of Computer Programming. Addison-Wesley, 1981.
5. Donald E. Knuth. Sorting and Searching, volume 3 of The Art of Computer Programming. Addison-Wesley, 1973.
6. А. Ахо, Д. Ульман, Д. Хопкрофт. Структуры данных и алгоритмы. Издательский дом «Вильямс», 2000.
7. А. Ахо, Д. Ульман, Д. Хопкрофт. Построение и анализ вычислительных алгоритмов. М.: Мир, 1979.
8. Erickson J. Combinatorial Algorithms <http://www.uiuc.edu/~jeffe/>
9. Tomescu, I.; Probleme de combinatorică, și teoria grafurilor. Editura Didactică și Pedagogică, București, 1981.

CUPRINS

LUCRARE DE LABORATOR NR. 1	
Analiza algoritmilor	
LUCRARE DE LABORATOR NR. 2	
Metoda divide et impera	
LUCRARE DE LABORATOR NR. 3	
Algoritmi greedy	
LUCRARE DE LABORATOR NR. 4	
Metoda programării dinamice	
LUCRARE DE LABORATOR NR. 5	
Tehnica căutării cu revenire (backtracking)	
LUCRARE DE LABORATOR NR. 6	
Algoritmi genetici	
BIBLIOGRAFIE	