LUCRARE DE LABORATOR NR. 1

Tema: Analiza algoritmilor

Scopul lucrării:

1. Analiza empirică a algoritmilor.
2. Analiza teoretică a algoritmilor.
3. Determinarea complexităţii temporale şi asimptotice a algoritmilor.

Note de curs:

1. Timpul de execuţie al algoritmilor

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

- algoritmul să fie simplu de înţeles, de codificat şi de depanat;
- algoritmul să folosească eficient resursele calculatorului, să aibă un timp de execuţie redus.

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

Daca programul urmează a fi rulat de un număr mare de ori, având şi un număr mare de date de prelucrat, trebuie ales algoritmul care duce la o execuţie mai rapidă. Chiar în această situaţie, ar trebui implementat mai înainte algoritmul mai simplu şi calculată reducerea de timp de execuţie 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 şi viteza de execuţie a instrucţiunilor programului;
- complexitatea algoritmului care stă la baza programului.

Deci timpul de rulare e o funcţie 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 execuţie al unui algoritm destinat rezolvării unei probleme de dimensiune n. Pentru a estima timpul de execuţie trebuie stabilit un *model de calcul* şi o unitate de măsură. Vom consideră un model de calcul (numit şi maşină de calcul cu acces aleator) caracterizat prin:

- Prelucrările se efectuează în mod secvenţial.
- Operaţiile *elementare* sunt efectuate în timp constant *indiferent* de valoarea operanzilor.
- Timpul de acces la informaţie nu depinde de poziţia acesteia (nu sunt diferenţe între prelucrarea primului element şi cea a ultimului element al unui tablou).

A stabili o unitate de măsură înseamnă a stabili care sunt operaţiile elementare şi a considera ca unitate de măsură timpul de execuţie a acestora. In acest fel timpul de execuţie va fi exprimat prin numărul de operaţii elementare executate. Operaţiile elementare sunt cele aritmetice (adunare, scădere, înmulţire, împărţire), comparaţiile şi cele logice (negaţie, conjuncţie şi disjuncţie). Cum scopul calculului timpului de execuţie este de a permite compararea algoritmilor, uneori este suficient să se contorizeze doar anumite tipuri de operaţii elementare, numite *operaţii de bază* (de exemplu în cazul unui algoritm de căutare sau de sortare se pot contoriza doar operaţiile de comparare) şi/sau să se considere că timpul de execuţie a acestora este unitar (deşi operaţiile de înmulţire şi împărţire sunt mai costisitoare decât cele de adunare şi scădere în analiza se poate considera că ele au acelaşi cost).

Timpul de execuţie al întregului algoritm se obţine însumând timpii de execuţie a prelucrărilor componente.

1.1. Importanţa celui mai defavorabil caz

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

În ceea ce priveşte analiza celui mai favorabil caz, acesta furnizează o margine inferioară a timpului de execuţie şi poate fi utilă pentru a identifica algoritmi ineficienţi (dacă un algoritm are un cost mare în cel mai favorabil caz, atunci el nu poate fi considerat o soluţie acceptabilă).

1.2. Timp mediu de execuţie

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

În aceste situaţii este utilă o altă măsură a complexităţii algoritmilor şi anume *timpul mediu de execuţie.* Acesta reprezintă o valoare medie a timpilor de execuţie calculată în raport cu distribuţia de probabilitate corespunzătoare spaţiului datelor de intrare.

2. Etapele analizei complexităţii

În analiza complexităţii unui algoritm se parcurg următoarele etape:

1. Se stabileşte dimensiunea problemei.
2. Se identifică operaţia de bază.
3. Se verifică dacă numărul de execuţii ale operaţiei 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 şi (dacă este posibil) cazul mediu.
4. Se află clasa de complexitate căruia îi aparţine algoritmul.

Câteva reguli generale pentru evaluarea timpului de execuţie în funcţie de mărimea datelor de intrare:

1. Timpul de execuţie a unei instrucţiuni de asignare, citire sau scriere, este constant şi se notează cu $O(1)$.
2. Timpul de rulare a unei secvenţe de instrucţiuni e determinat de regula de însumare, fiind proporţional cu cel mai lung timp din cei ai instrucţiunilor secvenţei.
3. Timpul de execuţie a unei instrucţiuni *if - then - else* este suma dintre timpul de evaluare a condiţiei ($O(1)$) şi cel mai mare dintre timpii de execuţie ai instrucţiunilor pentru condiţia adevărată sau falsă.
4. Timpul de execuţie a unei instrucţiuni de ciclare este suma, pentru toate iteraţiile, dintre timpul de execuţie a corpului instrucţiunii şi cel de evaluare a condiţiei de terminare ($O(1)$).
5. Pentru evaluarea timpului de execuţie a unei proceduri recursive, se asociază fiecărei proceduri recursive un timp necunoscut $T(n)$, unde n măsoară argumentele procedurii; se poate obţine o relaţie recurentă pentru $T(n)$, adică o ecuaţie pentru $T(n)$, în termeni $T(k)$, pentru diferite valori ale lui k.
6. Timpul de execuţie poate fi analizat chiar pentru programele scrise în pseudocod; pentru secvenţele care cuprind operaţii asupra unor structuri de date, se pot alege câteva implementări şi astfel se poate face comparaţie între performanţele implementărilor, în contextul aplicaţiei respective.

3. Analiza empirică a complexităţii algoritmilor

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

Aceasta poate fi utilă pentru:

- a obţine informaţii preliminare privind clasa de complexitate a unui algoritm;
- pentru a compara eficienţa a doi (sau mai mulţi) algoritmi destinaţi rezolvării aceleiaşi probleme;

- pentru a compara eficienţa mai multor implementări ale aceluiaşi algoritm;
- pentru a obţine informaţii privind eficienţa implementării unui algoritm pe un anumit calculator.

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

1. Se stabileşte scopul analizei.
2. Se alege metrica de eficienţă ce va fi utilizată (număr de execuţii ale unei/unor operaţii sau timp de execuţie a întregului algoritm sau a unei porţiuni din algoritm.
3. Se stabilesc proprietăţile datelor de intrare în raport cu care se face analiza (dimensiunea datelor sau proprietăţi 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 obţinute.

Alegerea măsurii de eficienţă depinde de scopul analizei. Dacă, de exemplu, se urmăreşte obţinerea unor informaţii privind clasa de complexitate sau chiar verificarea acurateţei unei estimări teoretice atunci este adecvată utilizarea numărului de operaţii efectuate. Dacă însă scopul este evaluarea comportării implementării unui algoritm atunci este potrivit timpul de execuţie.

Pentru a efectua o analiză empirică nu este suficient un singur set de date de intrare ci mai multe, care să pună în evidenţă 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 in practică. Pe de altă parte are importanţă şi analiza diferitelor valori sau configuraţii ale datelor de intrare. Dacă se analizează un algoritm care verifică dacă un număr este prim sau nu şi testarea se face doar pentru numere ce nu sunt prime sau doar pentru numere care sunt prime atunci nu se va obţine un rezultat relevant. Acelaşi 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 secvenţe al căror scop este monitorizarea execuţiei. Dacă metrica de eficienţă este numărul de execuţii ale unei operaţii atunci se utilizează un contor care se incrementează după fiecare execuţie a operaţiei respective. Dacă metrica este timpul de execuţie atunci trebuie înregistrat momentul intrării în secvenţa analizată şi momentul ieşirii. Majoritatea limbajelor de programare oferă funcţii 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 execuţia programului analizat. În special dacă este vorba de măsurarea timpului este indicat să se ruleze programul de test de mai multe ori şi să se calculeze valoarea medie a timpilor.

După execuţia 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 anume formă (dimensiune problemă, măsură de eficienţă).

4. Ordin de creştere

Pentru a aprecia eficienţa unui algoritm nu este necesară cunoaşterea expresiei detaliate a timpului de execuţie. Mai degrabă este interesant modul în care timpul de execuţie creşte o dată cu creşterea dimensiunii problemei. O măsură utilă în acest sens este *ordinul de creştere.* Acesta este determinat de *termenul dominant* din expresia timpului de execuţie. Când dimensiunea problemei este mare valoarea termenului dominant depăşeşte semnificativ valorile celorlalţi termeni astfel că aceştia din urmă pot fi neglijaţi.

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

dacă ordinul de creştere al timpului de execuţie al primului este mai mic decât al celuilalt.

Există următoarele cazuri când ordinul de creştere a timpului de execuţie, nu e cel mai bun criteriu de apreciere a performanţelor unui algoritm:

- dacă un program se rulează de puţine ori, se alege algoritmul cel mai uşor de implementat;
- dacă întreţinerea trebuie făcută de o altă persoană decât cea care l-a scris, un algoritm simplu, chiar mai puţin eficient, e de preferat unuia performant, dar foarte complex şi greu de înţeles;
- există algoritmi foarte eficienţi, dar care necesită un spaţiu de memorie foarte mare, astfel încât folosirea memoriei externe le diminuează foarte mult performanţele.

4.1. Notaţii asimptotice.

Fie $\mathbf{N}$ mulţimea numerelor naturale (pozitive său zero) şi $\mathbf{R}$ mulţimea numerelor reale. Notam prin $\mathbf{N}^+$ şi $\mathbf{R}^+$ mulţimea numerelor naturale, respectiv reale, strict pozitive, şi prin $\mathbf{R}^*$ mulţimea numerelor reale nenegative. Fie $f : \mathbf{N} \to \mathbf{R}^*$ o funcţie arbitrară. Definim mulţimea

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

Cu alte cuvinte, $O(f)$ (se citeşte "ordinul lui f") este mulţimea tuturor funcţiilor 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 şi 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 şi 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 notaţia $t(n) \in O(f(n))$, subânţelegând aici că $t(n)$ şi $f(n)$ sunt funcţii.

Notaţia asimptotică defineşte o relaţie de ordine parţială între funcţii şi deci, între eficienţa relativă a diferiţilor algoritmi care rezolvă o anumită problemă. Vom da în continuare o interpretare algebrică a notaţiei asimptotice. Pentru oricare două funcţii $f, g : \mathbf{N} \to \mathbf{R}^*$, definim următoarea relaţie binară: $f \le g$ dacă $O(f) \subseteq O(g)$. Relaţia "$\le$" este o *relaţie de ordine parţiala* în mulţimea funcţiilor definite pe **N** şi cu valori în $\mathbf{R}^*$. Definim şi o *relaţie de echivalenţă*: $f \equiv g$ dacă $O(f) = O(g)$.

În mulţimea $O(f)$ putem înlocui pe f cu orice altă funcţie echivalentă cu f. De exemplu, $\lg n \equiv \ln n \equiv \log n$ şi avem $O(\lg n) = O(\ln n) = O(\log n)$. Notând cu $O(1)$ ordinul funcţiilor mărginite superior de o constantă, obţinem 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 performanţei. Pentru o problemă dată, dorim mereu să obţinem un algoritm corespunzător unui ordin cât mai mic. Astfel, este o mare realizare dacă în locul unui algoritm exponenţial găsim un algoritm polinomial.

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

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

Există o anumită dualitate între notaţiile $O(f)$ şi $\Omega(f)$. Şi anume, pentru două funcţii oarecare $f, g : \mathbf{N} \to \mathbf{R}^*$, avem: $f \in O(g)$, dacă şi numai dacă
$g \in \Omega(f)$.

O situaţie fericită este atunci când timpul de execuţie al unui algoritm este limitat, atât inferior cât şi superior, de câte un multiplu real pozitiv al aceleiaşi funcţii. Introducem notaţia

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

numită *ordinul exact* al lui *f*. Pentru a compara ordinele a două funcţii, notaţia Θ nu este însă mai puternică decât notaţia O, în sensul că relaţia
$O(f) = O(g)$ este echivalentă cu $\Theta(f) = \Theta(g)$.

Se poate întâmpla că timpul de execuţie al unui algoritm să depindă simultan de mai mulţi parametri. Această situaţie este tipică pentru anumiţi algoritmi care operează cu grafuri şi în care timpul depinde atât de numărul de vârfuri, cât şi de numărul de muchii. Notaţia asimptotică se generalizează în mod natural şi pentru funcţii cu mai multe variabile. Astfel, pentru o funcţie arbitrară $f : \mathbf{N} \times \mathbf{N} \to \mathbf{R}^*$ definim

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

Similar, se obţin şi celelalte generalizări.

4.2. Operaţii asupra funcţiilor asimptotice

1. Dacă $T1(n)$ şi $T2(n)$ sunt timpii de execuţie a două secvenţe de program $P1$ şi $P2$, $T1(n)$ fiind $O(f(n))$, iar $T2(n)$ fiind $O(g(n))$, atunci timpul de execuţie $T1(n)+T2(n)$, al secvenţei $P1$ urmată de $P2$, va fi $O(\max(f(n),g(n)))$.
2. Dacă $T1(n)$ este $O(f(n))$ şi $T2(n)$ 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 execuţie 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: $a[j - 1] \leftarrow a[j]$;
6: $a[j] \leftarrow temp$
return $a[1..n]$ sortat

Timpii de rulare pentru instrucţiunile de asignare (4),(5) si (6) sânt $O(1)$, deci pentru secvenţa (4)-(6), timpul este $O(\max(1,1,1)) = O(1)$.

Pentru instrucţiunea **if - then** (3), timpul este suma dintre cel pentru evaluarea condiţiei, $O(1)$ şi cel al secvenţei ce se execută la condiţie adevărată, tot $O(1)$, calculat mai sus, deci $O(1)$, acesta fiind şi timpul pentru o iteraţie a instrucţiunii **for** (2).

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

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

Pentru instrucţiunea **for** (1), având n-1 iteraţii, timpul de execuţie este:

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

deci este $O(n^2)$.

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

function *factorial*(*n*: integer)
1: **if** $n \leq 1$ **then**
2: *factorial*$\leftarrow 1$
3: **else** *factorial*$\leftarrow n*$*factorial*$(n - 1)$
return *factorial*(*n*)

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

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

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

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

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

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

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

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

SARCINA DE BAZĂ:

1. Efectuaţi analiza empirică a algoritmilor propuşi.
2. Determinaţi relaţia ce reprezintă complexitatea temporală pentru aceşti algoritmi.
3. Determinaţi complexitatea asimptotică a algoritmilor.
4. Faceţi o concluzie asupra lucrării efectuate.

Întrebări de control:

1. Enumeraţi factorii ce influenţează timpul de execuţie al algoritmului.
2. Când timpul de execuţie al algoritmului este dat de o relaţie de recurenţă ?
3. Care sunt regulile generale pentru evaluarea timpului de execuţie?
4. Care sunt etapele analizei empirice şi în care cazuri se face analiza empirică a algoritmilor?
5. Ce proprietăţi posedă funcţiile asimptotice?

LUCRARE DE LABORATOR NR. 2

Tema: Metoda divide et impera

Scopul lucrării:

1. Studierea metodei divide et impera.
2. Analiza şi implementarea algoritmilor bazaţi 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 aceleiaşi probleme.
2. Rezolvarea succesivă şi independentă a fiecăruia din aceste subcazuri.
3. Combinarea subsoluţiilor astfel obţinute pentru a găsi soluţia cazului iniţial.

Algoritmul formal al metodei divide et impera:

function *divimp*(*x*)
{returnează o soluţie pentru cazul *x*}
1: **if** *x* este suficient de mic
2: **then return** *adhoc*(*x*)
3: {descompune *x* în subcazurile $x_1, x_2, \ldots, x_k$}
4: **for** $i \leftarrow 1$ **to** *k* **do** $y_i \leftarrow divimp(x_i)$
5: {recompune $y_1, y_2, \ldots, y_k$ în scopul obţinerii soluţiei *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 definiţie recursivă. Uneori este posibil să eliminăm recursivitatea printr-un ciclu iterativ. Implementată pe o maşina convenţionala, versiunea iterativă *poate fi* ceva mai rapidă (in limitele unei constante multiplicative). Un alt avantaj al versiunii iterative ar fi faptul că economiseşte spaţiul de memorie. Versiunea recursiva foloseşte 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 sa-l sortam crescător. Prin tehnica divide et impera putem proceda astfel: separăm tabloul T în două părţi de mărimi cât mai apropiate, sortăm aceste părţi prin apeluri recursive, apoi interclasăm soluţiile pentru fiecare parte, fiind atenţi să păstrăm ordonarea crescătoare a elementelor. Obţinem 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: *mergesort* (*T*, *p*, *q*)
4: *mergesort*(*T*, *q*+1, *r*)
5: *merge* (*T*, *p*, *q*, *r*)

Acum putem sorta întreg tabloul T apelând procedura *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 şi 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 celuilalt tablou sunt transferate direct în tabloul final.

Algoritmul *mergesort* ilustrează perfect principiul divide et impera:

- divide problema în subprobleme;
- stăpâneşte subproblemele prin rezolvare;
- combină soluţiile subproblemelor.

In algoritmul *mergesort*, suma mărimilor subcazurilor este egală cu mărimea cazului iniţial. Această proprietate nu este în mod necesar valabilă pentru algoritmii divide et impera. Este esenţial 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 dedicata construirii subcazurilor şi nu combinării soluţiilor lor.

Ca prim pas, algoritmul alege un element *pivot* din tabloul care trebuie sortat. Tabloul este apoi partiţionat în două subtablouri, alcătuite de-o parte şi 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 partiţionare 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 obţin la fiecare partiţionare, 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 şi 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 ← Partition (T, p, r )
3:          quicksort (T, p, q)
4:          quicksort t(T, q+1, r)
```

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

```
procedure Partition (T, p, r )
1: x ← T[p]
2: i ← p –1;
3: j ← r + 1
4: while TRUE do
5:          repeat j ← j – 1 until T[j] ≤ x
6:          repeat i ← i + 1 until T[i] > x
7:          if i < j
8:                  then interschimbă T[i] ↔ T[j]
9:                  else return j
```

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

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

SARCINA DE BAZĂ:

1. Studiaţi noţiunile teoretice despre metoda divide et impera.
2. Implementaţi algoritmii Mergesort şi Quicksort.
3. Efectuaţi analiza empirică a algoritmilor Mergesort şi Quicksort.
4. Faceţi o concluzie asupra lucrării efectuate.

Întrebări de control:

1. De ce această metodă se numeşte divide et impera ?
2. Explicaţi noţiunea de algoritm recursiv.
3. Descrieţi etapele necesare pentru rezolvarea unei probleme prin metoda divide et impera.
4. Ce tip de funcţie descrie timpul de execuţie al unui algoritm de tipul divide et impera?
5. Care sunt avantajele şi dezavantajele algoritmilor divide et impera?

LUCRAREA DE LABORATOR NR.3

Tema: Algoritmi greedy

Scopul lucrării:

1. Studierea tehnicii greedy.
2. Analiza şi implementarea algoritmilor greedy.

Note de curs:

1. Tehnica greedy

Algoritmii *greedy* (greedy = lacom) sunt în general simpli şi sunt folosiţi 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 situaţii de acest fel avem:

- mulţime de *candidaţi* (lucrări de executat, vârfuri ale grafului etc);
- o funcţie care verifică dacă o anumită mulţime de candidaţi constituie o *soluţie posibilă*, nu neapărat optimă, a problemei;
- o funcţie care verifică dacă o mulţime de candidaţi este *fezabilă*, adică dacă este posibil să completăm această mulţime astfel încât să obţinem o soluţie posibilă, nu neapărat optimă, a problemei;
- o *funcţie de selecţie* care indică la orice moment care este cel mai promiţător dintre candidaţii încă nefolosiţi;
- o *funcţie obiectiv* care dă valoarea unei soluţii (timpul necesar executării tuturor lucrărilor într-o anumită ordine, lungimea drumului pe care l-am găsit etc); aceasta este funcţia pe care urmărim să o optimizăm (minimizăm/maximizăm).

Pentru a rezolva problema de optimizare, se caută o soluţie posibilă care să optimizeze valoarea funcţiei obiectiv. Un algoritm greedy construieşte soluţia pas cu pas. Iniţial, mulţimea candidaţilor selectaţi este vidă. La fiecare pas, se adaugă acestei mulţimi cel

mai promiţător candidat, conform funcţiei de selecţie. Dacă, după o astfel de adăugare, mulţimea de candidaţi selectaţi nu mai este fezabilă, se elimină ultimul candidat adăugat; acesta nu va mai fi niciodată considerat. Dacă, după adăugare, mulţimea de candidaţi selectaţi este fezabilă, ultimul candidat adăugat va rămâne de acum încolo în ea. De fiecare dată când se lărgeşte mulţimea candidaţilor selectaţi, se verifică dacă această mulţime nu constituie o soluţie posibilă a problemei. Dacă algoritmul greedy funcţionează corect, prima soluţie găsită va fi totodată o soluţie optimă a problemei. Soluţia optimă nu este în mod necesar unică: se poate că funcţia obiectiv să aibă aceeaşi valoare optimă pentru mai multe soluţii posibile. Funcţia de selecţie este de obicei derivată din funcţia obiectiv; uneori aceste două funcţii sunt chiar identice.

2. Arbori de acoperire cu cost minim

Fie $G = <V, M>$ un graf neorientat conex, unde V este mulţimea vârfurilor şi M este mulţimea muchiilor. Fiecare muchie are un *cost* nenegativ w (sau o *lungime* nenegativă). Problema este să găsim o submulţime $A \subseteq M$, 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 cat mai mică. Căutăm deci o submulţime A de cost total minim. Această problemă se mai numeşte şi *problema conectării oraşelor cu cost minim*, având numeroase aplicaţii.

Graful parţial $<V, A>$ este un arbore şi este *numit arborele de acoperire minim* al grafului G (*minimal spanning tree* (MST)). Un graf poate avea mai mulţi 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 mulţime de muchii este o *soluţie*, dacă constituie un arbore de acoperire al grafului G, şi este *fezabila*, dacă nu conţine cicluri. O mulţime fezabilă de muchii este *promiţătoare*, dacă poate fi completată pentru a forma soluţia optimă. O muchie *atinge* o

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

Mulţimea iniţiala a candidaţilor este *M*. Cei doi algoritmi greedy aleg muchiile una cate una intr-o anumita 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 şi care nu formează cu precedentele un ciclu. Alegem astfel *V*–1 muchii. Este uşor de dedus că obţinem în final un arbore.

În algoritmul lui Kruskal, la fiecare pas, graful parţial <*V, A*> 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 obţine arborele parţial de cost minim al grafului *G*.

Pentru a implementa algoritmul, trebuie să putem manipula submulţimile formate din vârfurile componentelor conexe. Folosim pentru aceasta o structura de date pentru mulţimi disjuncte pentru prezentarea mai multor mulţimi de elemente disjuncte [Cormen]. Fiecare mulţime conţine vârfurile unui arbore din pădurea curentă. Funcţia ***Find-Set*** (*u*) returnează un element reprezentativ din mulţimea care îl conţine pe *u*. Astfel, putem determina dacă două vârfuri *u* şi *v* aparţin aceluiaţi 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 lista de muchii cu costul asociat lor, astfel încât să putem ordona această listă în funcţie de cost. În continuare este prezentat algoritmul:

MST - Kruskal(G, w)
1: A$\leftarrow\varnothing$
2: **for** fiecare vârf $v \in V[G]$ **do**
3: ***Make-Set*** (v)
4: sortează muchiile din M crescător în funcţie 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 iniţializează mulţimea A cu mulţimea vidă şi 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 şi v aparţin aceluiaşi arbore. Dacă fac parte din acelaşi arbore, atunci muchia (u, v) nu poate fi adăugată la pădure fără a se forma un ciclu şi ea este respinsă. Altfel, cele două vârfuri aparţin unor arbori diferiţi, şi 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, mulţimea A de muchii alese împreună cu mulţimea U a vârfurilor pe care le conectează formează un arbore parţial de cost minim pentru subgraful $<U, A>$ al lui G. Iniţial, mulţimea U a vârfurilor acestui arbore conţine un singur vârf oarecare din V, care va fi rădăcina, iar mulţimea A a muchiilor este vidă. La fiecare pas, se alege o muchie de cost minim, care se adaugă la arborele precedent, dând naştere unui nou arbore parţial de cost minim. Arborele parţial de cost minim creste “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 aşa fel încât să fie uşor să selectăm o nouă muchie pentru a fi adăugată la arborele format de muchiile din *A*. În pseudocodul de mai jos, graful conex *G* şi rădăcina *r* a arborelui minim de acoperire, care urmează a fi dezvoltat, sunt privite ca date de intrare pentru algoritm. În timpul execuţiei 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 uneşte pe *v* cu un vărf din arbore. Prin convenţie, *key*[*v*] = ∞ dacă nu există o astfel de muchie. Câmpul $\pi[v]$ reţine „părintele" lui *v* din arbore. *Adj*[*u*] este lista de adiacenţă cu vârful *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: $\pi[r] \leftarrow$ NIL
6: **while** $Q \neq \varnothing$
7: **do** $u \leftarrow$ ***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 iniţializează coada de prioritate *Q*, astfel încât aceasta să conţină toate vârfurile şi se iniţializează câmpul *key* al fiecărui vârf cu ∞, excepţie fîcând rădăcina *r*, al cărei câmp *key* este iniţializat cu 0. În linia 5 se iniţializează $\pi[r]$ cu NIL, deoarece rădăcina *r* nu are nici un părinte. Pe parcursul execuţiei algoritmului, mulţimea $V - Q$ conţine vârfurile arborelui curent. În

linia 7 este identificat un vârf $u \in Q$ incident unei muchii uşoare care traversează tăietura ($V - Q$, Q) (cu excepţia primei iteraţii, în care $u = v$ datorită liniei 4). Eliminarea lui u din mulţimea Q îl adaugă pe acesta mulţimii $V - Q$ a vârfurilor din arbore. În liniile 8-11 se actualizează câmpurile *key* şi π ale fiecărui vârf v adiacent lui u, dar care nu se află în arbore. Actualizarea respectă condiţiile $key[v] = w(v, \pi[v])$, şi $(v, \pi[v])$ să fie o muchie uşoară care îl uneşte 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 algoritmii Prim şi Kruskal.
3. De făcut analiza empirică a algoritmilor Kruskal şi Prim.
4. De alcătuit un raport.

Întrebări de control:

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

LUCRAREA DE LABORATOR NR.4

Tema: Metoda programării dinamice

Scopul lucrării:

1. Studierea metodei programării dinamice.
2. Analiza şi 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 obţine un rezultat trebuie să se poată lua pas cu pas. De asemenea, foarte importantă este ordinea în care acestea se iau. Programarea dinamică este (şi nu luaţi aceste rânduri ca pe o definiţie) în esenţă un proces decizional în mai multe etape: în starea iniţială 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 funcţiile care descriu fiecare etapă trebuie să fie în aşa fel definite încât să descrie complet un proces, deci pentru acest lucru va trebui să răspundem la două întrebări:

- care este etapa *iniţială* (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 *recurenţă*.

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 Fibonaci* 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 oraşe *A* şi *B* şi care trece printr-un număr minim de alte oraşe 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 oraşul *A*,
- dar există mai multe decizii pentru trecerea la drumul de lungimea $k + 1$.

În cele ce urmează prin strategie înţelegem un şir de decizii. Conform principiului lui Bellman, numit ***principiul optimalităţii*** avem:

O strategie are proprietatea că oricare ar fi starea iniţială şi decizia iniţială, 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, aşa cum rezultă şi din principiul optimalităţii, prin *inducţie matematică*.

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

şi rezolvarea lor în mod ascendent. Pe scurt Q_i se obţine din P printr-o restricţionare a datelor de intrare.

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

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

Definirea relaţiilor dintre etape se face recursiv. Prin prisma unui matematician acest lucru este inconvenient, dar orice informatician ştie că recursivitatea înseamnă resurse (timp şi memorie) consumate. Pe lângă inconvenienţele 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") şi se reţin rezultatele obţinute. În cazul numerelor Fibonacci, cea mai mică subproblemă este calcularea lui Fibonacci(0) şi 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 relaţiile de recurenţă 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. Dependenţa directă dintre etape este dată de relaţiile de recurenţă.

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

Notăm cu *P*, problema pe care o dorim să o rezolvăm. Înţelesul pe care îl dăm lu *P* include şi dimensiunea datelor de la intrare.

Am spus anterior că o etapă, sau o subproblemă are aceeaşi formă ca şi problema de rezolvat la care sunt adăugate câteva restricţii.

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,…, Q_N ordinea rezultată unei astfel de sortări. Prin Q_i am notat o subproblemă (Q_0 este subproblema cea mai mică). În cazul submulţimii de sumă dată, Q_i este chiar descompunerea lui *i* în sumă de numere din vectorul dat. Fie S_i soluţia problemei Q_i. Soluţia subproblemei Q_N este soluţia problemei *P*.

Algoritmul general este:

procedure Rezolva(*P*)
{iniţializează (S_0) }
1: **for** $i \leftarrow 1$ to *N* do
2: **progresează** (S_0, …, S_{i-1}, S_i); { aceasta procedura calculează soluţia problemei Q_i pe baza soluţiilor problemelor Q_0, …, Q_{i-1}, cu o recurenţă de jos în sus.}
3**: return** (S_N)

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

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

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

dinamică. Fie *D* mulţimea tuturor intrărilor corecte pentru o problemă dată. Definim două funcţii:

Max: $D \rightarrow Z^+$ şi

Lungimea: $D \rightarrow Z^+$.

Funcţia *Max* indică valoarea maximă a datelor de intrare, iar funcţia *Lungimea* indică lungimea lor.

Un algoritm este numit *algoritm pseudo – polinomial*, dacă funcţia sa de complexitate este mărginită superior de o funcţie polinomială în două variabile: *Max*[*I*] şi *Lungime*[*I*]. Prin definiţie orice algoritm polinomial este şi pseudo – polinomial, deoarece se execută într-un timp mărginit de un polinom de *Lungime*[*I*], dar nu toţi algoritmii pseudo-polinomiali sunt polinomiali. Pentru problemele care au proprietatea că *Max*[*I*] este mărginită de o funcţie polinomială în variabila *Lungime* [*I*] nu există nici o distincţie între algoritmii polinomiali şi cei pseudo-polinomiali. Să vedem când se poate şi când nu se poate construi un algoritm pseudo-polinomial? Spunem despre o problemă că este *number-problem* dacă nu există nici o funcţie 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 relaţiilor de recurenţă intervine şi 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 maşină 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 funcţiilor de recurenţă) pot fi *unidimensionale* (precum numerele lui Fibonacci), sau *multidimensionale*.

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

5. După direcţia deciziilor putem avea programare dinamică *înainte* în care starea *i* se calculează în funcţie de stările $i + 1$, $i + 2$, …, sau programare dinamică *înapoi* în care starea *i* se calculează în funcţie de stările $i - 1$, $i - 2$, ….

Dezvoltarea unui algoritm bazat pe programarea dinamică poate fi împărţită într-o secvenţă de patru paşi:

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

Paşii 1-3 sunt baza unei abordări de tip programare dinamică. Pasul 4 poate fi omis dacă se doreşte doar calculul unei singure soluţii optime. In vederea realizării pasului 4, deseori se păstrează informaţie suplimentară de la execuţia pasului 3, pentru a uşura construcţia unei soluţii optimale.

2. Probleme de drum minim în grafuri

2.1. Cele mai scurte drumuri care pleacă din acelaşi punct

Fie $G = <V, A>$ un graf orientat, unde V este mulţimea vârfurilor şi A este mulţimea arcelo. Fiecare arc are o lungime nenegativa. Unul din vârfuri este ales că 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 mulţimea vârfurilor disponibile (candidaţii) şi cu S mulţimea vârfurilor deja selectate. În fiecare moment, S conţine

acele vârfuri a căror distanţă minimă de la sursă este deja cunoscută, în timp ce mulţimea C conţine toate celelalte vârfuri. La început, S conţine doar vârful sursă, iar în final S conţine toate vârfurile grafului. La fiecare pas, adăugam în S acel vârf din C a cărui distanţă 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 aparţin lui S. Algoritmul lui Dijkstra lucrează în felul următor. La fiecare pas al algoritmului, un tablou D conţine 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 şi valorile din D reprezintă soluţia problemei.

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

function *Dijkstra*($L[1 .. n, 1 .. n]$)
1: $C \leftarrow \{2, 3, ..., n\}$ {$S = V \setminus C$ 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
a) este în S, atunci $D[i]$ dă lungimea celui mai scurt drum de la sursă către i;
b) 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 excepţia unuia, sunt în *S*. Din proprietatea precedenta, rezulta că algoritmul lui Dijkstra funcţionează corect.

2.2. Determinarea celor mai scurte drumuri intr-un graf

Fie $G = <V, A>$ un graf orientat, unde V este mulţimea vârfurilor şi A este mulţimea 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 şi 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 optimalităţii este valabil: dacă cel mai scurt drum de la i la j trece prin varful k, atunci porţiunea de drum de la i la k, cât şi cea de la k la j, trebuie să fie, de asemenea, optime.

Construim o matrice D care să conţină lungimea celui mai scurt drum între fiecare pereche de vârfuri. Algoritmul de programare dinamică iniţializează pe D cu L. Apoi, efectuează n iteraţii. După iteraţia k, D va conţine lungimile celor mai scurte drumuri care folosesc ca vârfuri intermediare doar vârfurile din $\{1, 2, ..., k\}$. După n iteraţii, obţinem rezultatul final. La iteraţia 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 $\{1, 2, ..., k–1\}$. Fie D_k matricea D după iteraţia 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 facut uz de principiul optimalităţii pentru a calcula lungimea celui mai scurt drum faţă 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 .. n, 1 .. n])
1:  array D[1 .. n, 1 .. n]
2:  D ← L
3:  for k ← 1 to n do
4:      for i ← 1 to n do
5:          for j ← 1 to n do
6:              D[i, j] ← 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 obtine un timp în $n\,\Theta(n^2)$, adică tot în $\Theta(n^3)$. Algoritmul lui Floyd, datorită simplităţii 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 algoritmii prezentaţi mai sus.
3. De făcut analiza empirică a acestor algoritmi pentru un graf rar şi pentru un graf dens.
4. De alcătuit un raport.

Întrebări de control:

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

LUCRAREA DE LABORATOR NR.5

Tema: Tehnica căutării cu revenire (backtracking)

Scopul lucrării:

1. Studierea metodei backtracking.
2. Analiza şi 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 (in traducere aproximativa, "cautare cu revenire") este un principiu fundamental de elaborare a algoritmilor pentru probleme de optimizare, sau de găsire a unor soluţii care indeplinesc anumite condiţii. Majoritatea problemelor ce pot fi rezolvate prin această tehnică pot fi reduse la determinarea unei submulţimi a unui produs cartezian de forma $A_1 \times A_2 \times ... \times A_n$ (cu submulţimile A_k finite). Fiecare element al submulţimii poate fi văzut ca o soluţie. O soluţie este de forma $s = (s_1, s_2, ..., s_n)$ cu $s_k \in A_k = \{a_1^k, a_2^k, ..., a_{m_k}^k\}, m_k = \text{card} A_k$. În majoritatea cazurilor nu orice element al produsului cartezian este soluţie ci doar cele care satisfac anumite restricţii.

Metoda backtracking constă în construirea soluţiei s completând pe rând toate componentele s_k pentru $k = 1, 2, ..., n$. Specifică este maniera de parcurgere a spaţiului soluţiilor:

- Soluţiile 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 mulţimile A_k există o

relaţie de ordine şi se realizează o parcurgere sistematică a spaţiului $A_1 \times A_2 \times ... \times A_n$);

- La completarea componentei k se verifică dacă soluţia parţială (s_1, s_2, ..., s_k), verifică condiţiile induse de restricţiile problemei. O soluţie parţială care satisface restricţiile problemei este numită soluţie fezabilă deoarece poate conduce la o soluţie a problemei.
- Dacă au fost încercate toate valorile corespunzătoare componentei k şi nu a fost găsită o soluţie sau dacă se doreşte determinarea unei noi soluţii atunci se revine la componenta (k-1) şi se încearcă următoarea valoare corespunzătoare acesteia ş.a.m.d. Această revenire la o componentă anterioară este specifică tehnicii backtracking.
- Procesul de căutare şi revenire este continuat până când este găsită o soluţie sau până când au fost testate toate configuraţiile posibile.

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

- Se alege o reprezentare a soluţiei sub forma unui vector cu n componente;
- Se identifică mulţimile A_1, A_2,..., A_n şi relaţiile de ordine care indică modul de parcurgere a fiecărei mulţimi;
- Pornind de la restricţiile problemei se stabilesc condiţiile de validitate ale soluţiilor parţiale (condiţiile de continuare).

În descrierea algoritmului se vor volosi notaţiile următoare: $A_k = \{a_1^k, a_2^k, ..., a_{m_k}^k\}$, 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,..., A_n)
1: $k \leftarrow 1$
2: $i_k \leftarrow 0$
3: **while** $k > 0$ **do**
4: $\quad i_k \leftarrow i_k + 1$

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

Observaţii. În problemele concrete pot apărea următoarele situaţii:

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

2. Condiţiile de continuare se deduc din restricţiile problemei.

3. Pentru unele probleme soluţia este obţinută în cazul în care $k = n$ iar pentru altele este posibil să fie satisfăcută o condiţie de găsire a soluţiei pentru $k < n$ (pentru unele probleme nu toate soluţiile conţin acelaşi număr de elemente).

4. La găsirea unei soluţii de cele mai multe ori aceasta se afişează sau se memorizează într-o zonă dedicată. Dacă se doreşte identificarea unei singure soluţii atunci căutarea se opreşte după găsirea acesteia, altfel procesul de căutare continuă prin analiza următorei valori a ultimei componente completate.

2. Problema colorării hărţilor

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

prezentată printr-o matrice de adiacenţă $n \times n$ 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 soluţiilor. O soluţie a problemei este o modalitate de colorare a ţărilor şi poate fi reprezentată printr-un vector $(s_1,..., s_n)$ cu $s_i \in \{1,..., m\}$ reprezentând culoarea asociată ţării i. Mulţimile de valori ale elementelor $A_1=...= A_n= \{1,..., m\}$.

Restricţii şi condiţii de continuare. Restricţia ca două ţări vecine să fie colorate diferit se specifică prin: $s_i \neq s_j$ pentru orice i şi j care au proprietatea $v_{ij} = 1$. Condiţia de continuare pe care trebuie s-o satisfacă soluţia parţială $(s_1,..., s_k)$ este: $s_k \neq s_i$ pentru orice $i < k$ cu proprietatea $v_{ik} = 1$.

Descrierea algoritmului.

Colorare (*n*)

```
1: k ← 1
2: s[k] ← 0
3: while k > 0 do
4:     s[k] ← s[k] +1
5:     valid ← FALSE
6:     while (valid = FALSE) and (s[k] ≤ m) do
7:         if validare (s[1..k]) then
8:             valid ← TRUE
9:         else s[k] ← s[k] +1
11:    if valid = TRUE then
12:        if k = n then afişare (s[1..n])
13:        else k← k + 1; s[k] ← 0
14:    else k← k - 1
```

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

```
1: for i ← 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 oraşe

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

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

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

Reprezentarea soluţiilor. În această problemă nu toate soluţiile au aceeaşi lungime. O soluţie a problemei este de forma (s_1, ..., s_m) cu $s_i \in \{1, 2, ..., n\}$ indicând oraşul care va fi parcurs în etapa *i* a traseului.

Restricţii şi condiţii de continuare. O soluţie (s_1,..., s_m) trebuie să satisfacă: $s_1 = p$ (oraşul de start), $s_m = q$ (oraşul destinaţie), $s_i \neq s_j$ pentru orice $i \neq j$(nu se trece de două ori prin acelaşi oraş) $v_{s_i s_{i+1}} = 1$ pentru i = 1, 2, ..., n – 1(între oraşele parcurse succesiv există drum direct). La completarea componentei *k*, condiţia de continuare este $s_k \neq s_i$ pentru i = 1, 2, ..., k – 1şi $v_{s_{k-1} s_k} = 1$. Condiţia de găsire a soluţiei nu este determinată de numărul de componente completate ci de faptul că $s_k = q$

Descrierea algoritmului.

***drumuri_recursiv*(*k*)**

1: **if** *s*[*k*-1] = *q* **then** *afişare* (*s*[1..*k*-1])
2: **else**
7: **if** *k* ≠ *n* + 1 **then**
8: **for** *i* ← 1, *n* **do**
9: *s*[*k*] ← *i*
11: **if** *validare* (*s*[1..*k*]) **then**
drumuri_recursiv(*k*+1)

***validare* (*s*[1..*k*])**
1: **if** *v*[*s*[*k*-1], *s*[*k*]] = 0 **then return** FALSE
2: **else**
3: **for** *i* ← 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 prezentaţi mai sus.
3. De făcut analiza acesui algoritm.
4. De alcătuit un raport.

Întrebări de control:

1. În ce constă metoda backtracking?
2. Cum se parcurge spaţiul soluţiilor?
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. Studierea algoritmilor genetici.
2. Analiza şi 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 spaţiul soluţiilor, în care se folosesc populaţii supuse unui proces de evoluţie similar într-o măsură oarecare cu cel întâlnit în natură.

Principalele noţiuni care permit analogia între rezolvarea problemelor de căutare şi evoluţia în natură sunt următoarele:

- *Populaţie*. O populaţie este formată din indivizi care trăiesc într-un mediu la care trebuie să se adapteze.
- *Fitness.* Fiecare individ din populaţie este adaptat mai mult sau mai puţin la mediu. Fitness-ul este o măsură a gradului de adaptare la mediu. Scopul evoluţiei este ca toţi indivizii să ajungă la un fitness cât mai bun.
- *Cromozom.* Este o mulţime ordonată de elmente, numite *gene* ale căror valori determină caracteristicile unui individ. În genetică poziţiile pe care se află genele în cadrul cromozomului se numesc *loci*, iar valorile pe care le pot lua se numesc *alele.*
- *Generaţie.* Este o etapă în evoluţia unei populaţii. Dacă privim evoluţia ca pe un proces iterativ în care o populaţie se transformă în alta atunci generaţia este o iteraţie în cadrul acestui proces.

- *Selecţie.* Procesul de selecţie naturală are ca efect supravieţuirea indivizilor cu grad ridicat de adaptare la mediu.
- *Reproducere.* Este procesul prin care se trece de la o generaţie la alta. Indivizii noii generaţii moştenesc caracteristici de la precursorii lor dar pot dobândi şi caracteristici noi ca urmare a unor procese de mutaşie care au un caracter aleator. În cazul în care în procesul de reproducere intervin cel puţin doi pîrinţi caracteristicile moştenite ale urmaşului se obţin prin combinarea caracteristicilor părinţilor.

2. Structura generală a unui algoritm genetic

Algoritmii genetici sunt procese iterative prin care o populaţie iniţializată aleator este transformată succesiv prin selecţie, încrucişare şi mutaţie până la atingerea unui număr anumit de iteraţii sau până la îndeplinirea unui alt criteriu de oprire.

Există un număr foarte mare de variante de algoritmi genetici datorită flexibilităţii 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 $P(t)= (x_1(t), x_2(t),..., x_m(t))$ reprezintă populaţia corespunzătoare generaţiei t iar $f(x_i)$ reprezintă gradul de adaptare la mediu al elementului x_i:

***genetic algoritm* (*P*(*t*))**

1: *Iniţializare*: $P(0)= (x_1(0), ..., x_m(0))$, $t = 0$

2: **repeat**

3: *evaluarea populaţiei curente:* calcul $f(x_i(t))$ pentru $i = 1, 2, ..., m$

4: *selecţia părinţilor*: $P(t) \rightarrow P^1$

5: *generarea urmaşilor prin încrucişare*: $P^1 \rightarrow P^2$

6: *modificarea urmaşilor prin mutaţie*: $P^2 \rightarrow P^3$

7: *evaluarea populaţiei de urmaşi*: calcularea valorilor lui f pentru elementele lui P^3

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

9: *incrementarea contorului de generaţii*: $t = t + 1$
10: **until** $t < t_{max}$

La proiectarea unui algoritm genetic trebuie să se stabilească:

- *Modul de codificare.* Se specifică modul în care fiecărei configuraţii din spaţiul de căutare i se asociază un cromozom.
- *Funcţia de adaptare.* Se construieşte funcţia care exprimă gradul de adaptare la mediu pornind de la restricţiile şi funcţia scop a problemei.
- *Modul de iniţializare şi dimensiunea populaţiei.* Se pot utiliza populaţii de dimensiune fixă sau de dimensiune variabilă. De cele mai multe ori populaţia se iniţializează cu elemente aleatoare din spaţiul de căutare.
- *Mecanismul de selecţie* a părinţilor şi a supravieţuitorilor.
- *Mecanismul de combinare* a părinţilor pentru a genera urmaşi.
- *Mecanismul de mutaţie* prin care se asigură modificarea cromozomilor generaţi.
- *Criteriul de oprire*. Atunci când nu se cunoaşte un criteriu specific problemei se optează pentru un număr maxim de iteraţii.

2.1. Codificarea cromozomilor

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

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

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

Reguli de codificare. Modul în care sunt codificaţi 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 mulţimea {0, 1} iar spaţiul 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 configuraţiile pot fi specificate ca vectori binari.
- *Codificarea reală.* Este adecvată pentru problemele de optimizare pe domenii continue. În acest caz cromozomii sunt nişte vectori cu elemente reale, fiind chiar elementele domeniului de definiţie al funcţiei scop. Avantajul codificării reale este faptul că este naturală şi nu necesită proceduri speciale de codificare şi 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 foloseşte în cazul problemei voiajorului comercial. O permutare reprezintă ordinea în care voiajorul comercial vizitează oraşele.

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

2.2. Populaţia iniţială

Populaţia iniţială este o mulţime de cromozomi care trebuie să fie suficient de mare pentru a se putea realiza un număr suficient de combinaţii între cromozomii componenţi, dar nu prea mare, deoarece aceasta poate încetini algoritmul. De obicei populaţia iniţială este una aleatoare, dar pentru o mai bună performanţă, o

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

2.3. Funcţia de adecvare (fitness-ul)

Fitness-ul unui cromozom este speranţa lui de viaţă sau gradul de adaptare la mediu. Scopul este de a obţime cromozomi cu fitnnes cât mai mare sau cât mai mic, adică un fitness la extremă. Fitness-ul este exprimat de o funcţie ce urmează a fi optimizată minimizată sau maximizată.

2.4. Selecţia cromozomilor

Cromozomii sunt selectaţi din populaţie pentru a fi părinţi într-o încrucişare.

Există mai multe metode de selecţie, printre care:

1. Selecţie aleatoare. Doi părinţi sunt aleşi aleator din populaţie. Aceasta se face generând două numere aleatoare între 1 şi dimensiunea populaţiei.

2. Selecţie cu ajutorul ruletei. Părinţii sunt selectaţi în conformitate cu fitness-ul lor. Cu cât sunt mai buni, cu atât şansa lor de a fi aleşi este mai mare. Ne imaginăm o ruletă în care sunt dispuşi toţi cromozomii din populaţie. Fiecărui cromozom îi corespunde un sector al ruletei, direct proporţional, ca mărime, cu fitness-ul cromozomului respectiv. În acest fel cromozomii cu fitness mare au ataşate sectoare mai mari, iar cei cu fitness mic au ataşate sectoare mai mici. La aruncarea bilei pe ruletă există mai multe şanse de alegere pentru cromozomii cu fitness mai mare.

Simularea roţii de ruletă se face în felul următor:

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

3. Selecţie aranjată (Selecţie pe baza rangurilor). Se ordonează crescător valorile funcţiei de adecvare pentru toţi cromozomii. Se renumerotează cu numere întregi din intervalul [1, ..., *dimensiunea populaţiei*]. Cromozomul cu fitness-ul cel mai mic are numărul 1etc., iar cel mai mare are numărul egal cu dimensiunea populaţiei. Aceste numere sunt considerate fitness-uri şi pe ele se aplică selecţia proporţională. Se observă că cromozomii cu fitness mai mare au la fel cele mai mari şanse de a fi aleşi, dar de data aceasta nu mai sunt aşa de mari în comparaţie cu şansele celorlalţi.

***4. Selecţie de tip turneu*.** Se selectează uniform aleator câte *k* cromozomi din populaţie iar dintre aceştea se alege cel care are cea mai mare valoare pentru funcţia de adecvare. Cel mai frecvent se utilizează *k*=2.

***5. Seleţie prin trunchiere*.** Se foloseşte atunci când dintr-o populaţie sau o reuniune de populaţii se cere selectarea unei subpopulaţii. Sunt selectate atâtea elemente câte sunt necesare pentru a completa subpopulaţia în ordinea descrescătoare a valorii funcţiei de adecvare.

***6. Elitism*.** Prin elitism se înţelege supravieţuirea celui mai bun dintre elementele generate până la un moment dat. Nu toate variantele de selecţie satisfac această proprietate. De exemplu, selecţia prin trunchiere este elitistă, însă cea de tip turneu nu este elitistă. O metodă de selecţie poate fi transformată în una elitistă prin amplasarea explicită a celui mai bun element al populaţiei curente în populaţia corespunzătoare generaţiei următoare.

2.5. Încrucişarea (crossover)

Încrucișarea va depinde de tipul de codificare a cromozomilor și de particularitățile problemei pe care o rezolvăm. Vom analiza cîteva 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 + 11011**1111** = **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.
10100**10**11 + 110**111**1**11** = **101111011**
- ***Încrucișare uniformă***. Biții sunt copiați aleator din primul și al doilea părinte.
1**0**100**1011** + **1**10**1**1111 = **101111011**
- ***Încrucișare aritmetică.*** Pentru crearea de noi urmați putem folosi operațiile precum și funcții aritmetice (**and, or, not** etc)
101001011 + 110111111 = **100001011 (and)**

2.6. Mutația

Asigura alterarea valorii unor gene pentru a evita situatiile in care o anumita gena nu apare in populatie fiindca nu a fost generata de la inceput. In acest fel se asigura diversitatea populatiei. Prin mutație se evită căderea soluțiilor problemei într-un optim local. De regulă această operație se efectuiază 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↔0.
11**0**010011→11**1**010011

Interschimbarea biților Biții selectați își schimbă valorile între ei.
11**0**010011→11**1**010011

Codificarea sub formă de permutare

Transpoziție Asemănător interschimbării biților de la codificarea binară. Se efectuiază o transpoziție asupra permutării respective.
(12**3**4678**9**5)→(12**9**4678**3**5)

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)→(**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 prezentaţi 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, şi teoria grafurilor. Editura Didactică şi Pedagogică, Bucureşti, 1981.

CUPRINS