**Ministerul Educației și Cercetării al Republicii Moldova Universitatea Tehnică a Moldovei Facultatea Calculatoare, Informatică și Microelectronică**

**Raport**

**Lucrarea de laborator nr.4:**

**Metoda programării dinamice.**

**Elaborat: st. gr. TI-232, Galiț Ștefan**

**Verificat: asist. univ. Coșeru Cătălin**

**Chișinău – 2024**

# Scopul lucrării:

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

# 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.

**Note teoretice:**

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 Qi (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 Qi 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 Qi obţine alte subprobleme dacă datele asupra cărora operează Qi 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 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,…, *QN* ordinea rezultată unei astfel de sortări. Prin *Qi* am notat o subproblemă (*Q*0 este subproblema cea mai mică). În cazul submulţimii de sumă dată, *Qi* este chiar descompunerea lui *i* în sumă de numere din vectorul dat. Fie *Si* soluţia problemei *Qi.* Soluţia subproblemei *QN* este soluţia problemei *P*.

Algoritmul general este:

**procedure** Rezolva(*P*)

{iniţializează (*S*0) } 1: **for** *i* ←1 to *N* do

2: **progresează** (*S*0, …, *Si*-1, *Si*); { aceasta procedura calculează soluţia problemei *Qi* pe baza soluţiilor problemelor *Q*0, …, *Qi*-1, cu o recurenţă de jos în sus.}

3**: return** (*SN*)

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*∙…).

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 → Z*+ şi

*Lungimea*: *D → Z*+.

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*] ≤ *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*.

1. 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.
2. 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.

1. **Probleme de drum minim în grafuri**
   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 câtre 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 câtre fiecare vârf al grafului. După ce se adaugă un nou vârf *v* la *S*, cel mai scurt drum special câtre *v* va fi, de asemenea, cel mai scurt dintre toate drumurile câtre

*v*. Când algoritmul se termină, toate vârfurile din graf sunt în *S*, deci toate drumurile de la sursă câtre 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*] = ∝, 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* ← {2, 3, ..., *n*} {*S* = *V* \*C* există doar implicit} 2: **for** *i* ← 2 **to** *n* **do** *D*[*i*] ← *L*[1, *i*]

3: **repeat** *n–*2 **times**

4: *v* ← vârful din *C* care minimizează *D*[*v*]

5: *C* ← *C* \ {*v*} {si, implicit, *S* ← *S* ∪ {*v*}} 6: **for** fiecare *w* ∈*C* **do**

7: *D*[*w*] ← min(*D*[*w*], *D*[*v*]+*L*[*v*, *w*])

**return** *D*

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

1. este în *S*, atunci *D*[*i*] dă lungimea celui mai scurt drum de la sursă câtre *i*;
2. nu este în *S*, atunci *D*[*i*] dă lungimea celui mai scurt drum special de la sursă câtre *i*.

La terminarea algoritmului, toate vârfurile grafului, cu excepţia unuia, sunt în *S*. Din proprietatea precedenta, rezulta că algoritmul lui Dijkstra funcţionează corect.

* 1. **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*] ≥ 0 pentru *i* ≠ *j*, *L*[*i*, *j*] = 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 *Dk* matricea *D* după iteraţia *k*. Verificarea necesară este atunci:

*Dk*[*i*, *j*] = min(*Dk*-1[*i*, *j*], *Dk*-1[*i*, *k*] *Dk*-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 *O*(*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 O* (*n*2), adică tot în *O* (*n*3). Algoritmul lui Floyd, datorită simplităţii lui, are însă constanta multiplicativă mai mică, fiind probabil mai rapid în practică.

**Rezultatul rulării programului:**

În figura (1) este prezentată analiza impirică ai algoritmilor pentru un graf rar, putem observa că algoritmul Dijkstra este mai rapid.

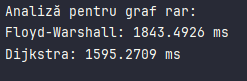


Figura.1 Analiza impirică pentru un graf rar.

În figura (2) este prezentată analiza impirică ai algoritmilor pentru un graf dens, că ambii algoritmi sunt aproximativ identici după viteză, însă Dijkstra e mai rapid.

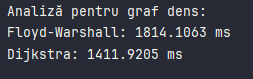


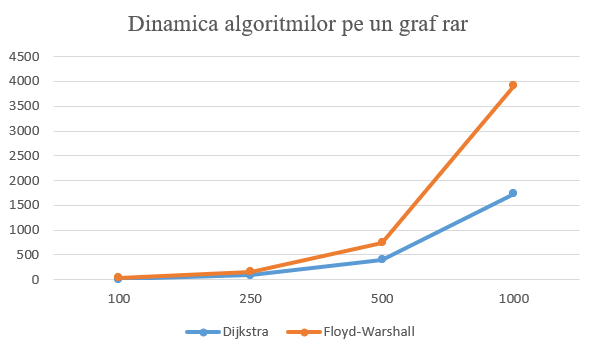
Figura.2 Analiza impirică pentru un graf dens.

**Dinamica algoritmilor bazată pe graf rar și dens și a numărului de noduri:**

În tabelul(1) și figura(3) este prezentată dinamica algoritmilor pe un graf rar.

**Tabelul 1 Dinamica algoritmilor pe un graf rar în milisecunde.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 100 | 250 | 500 | 1000 |
| Dijkstra | 10.17 | 90.70 | 405.91 | 1738.55 |
| Floyd-Warshall | 29.38 | 65.55 | 343.84 | 2175.80 |

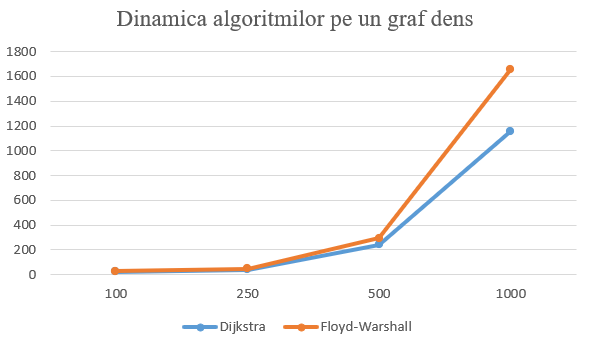
****

Figura(3). Dinamica algoritmilor pe un graf rar.

În tabelul(2) și figura(4) este prezentată dinamica algoritmilor pe un graf dens.

**Tabelul 2 Dinamica algoritmilor pe un graf dens în milisecunde.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 100 | 250 | 500 | 1000 |
| Dijkstra | 25.18 | 38.57 | 241.87 | 1155.93 |
| Floyd-Warshall | 4.26 | 10.7 | 54.04 | 500.97 |

****

Figura(4). Dinamica algoritmilor pe un graf dens

# Concluzie

Metodele de programare dinamică sunt esențiale pentru rezolvarea problemelor ce implică procese decizionale în mai multe etape, oferind soluții eficiente comparativ cu tehnicile greedy. Algoritmii Dijkstra și Floyd-Warshall au fost analizați și implementați, demonstrând performanțe variate în funcție de tipul grafului: Dijkstra este mai rapid pentru grafuri rare, în timp ce Floyd-Warshall excelează în grafuri dense. Aceste rezultate subliniază importanța alegerii algoritmului adecvat în funcție de structura datelor.

**Anexa A**

import java.util.Arrays;  
  
public class Dijkstra {  
 public static void dijkstra(int *graph*[][], int *src*, int *V*) {  
 int dist[] = new int[*V*];  
 boolean visited[] = new boolean[*V*];  
  
 Arrays.fill(dist, Integer.MAX\_VALUE);  
 dist[*src*] = 0;  
  
 for (int count = 0; count < *V* - 1; count++) {  
 int u = selectMinVertex(dist, visited, *V*);  
 visited[u] = true;  
  
 for (int v = 0; v < *V*; v++) {  
 if (!visited[v] && *graph*[u][v] != 0 && dist[u] != Integer.MAX\_VALUE &&  
 dist[u] + *graph*[u][v] < dist[v]) {  
 dist[v] = dist[u] + *graph*[u][v];  
 }  
 }  
 }  
  
 // printSolution(dist, V);  
 }  
  
 public static int selectMinVertex(int *dist*[], boolean *visited*[], int *V*) {  
 int min = Integer.MAX\_VALUE, minIndex = -1;  
  
 for (int v = 0; v < *V*; v++) {  
 if (!*visited*[v] && *dist*[v] < min) {  
 min = *dist*[v];  
 minIndex = v;  
 }  
 }  
  
 return minIndex;  
 }  
  
 public static void printSolution(int *dist*[], int *V*) {  
 System.out.println("Distanța de la sursă la fiecare nod:");  
 for (int i = 0; i < *V*; i++) {  
 System.out.println("Nod " + i + " -> " + *dist*[i]);  
 }  
 }  
  
  
}

*Anexa A* codul algoritmului Dijkstra.

**Anexa B**

import java.util.Arrays;  
  
public class FloydWarshall {  
 final static int INF = 99999;  
  
 public static void floydWarshall(int *graph*[][], int *V*) {  
 int dist[][] = new int[*V*][*V*];  
  
  
 for (int i = 0; i < *V*; i++)  
 for (int j = 0; j < *V*; j++)  
 dist[i][j] = *graph*[i][j];  
  
  
 for (int k = 0; k < *V*; k++) {  
 for (int i = 0; i < *V*; i++) {  
 for (int j = 0; j < *V*; j++) {  
 if (dist[i][k] + dist[k][j] < dist[i][j]) {  
 dist[i][j] = dist[i][k] + dist[k][j];  
 }  
 }  
 }  
 }  
  
 // printSolution(dist, V);  
 }  
  
 public static void printSolution(int *dist*[][], int *V*) {  
 System.out.println("Matricea cu cele mai scurte distanțe între fiecare pereche de noduri:");  
 for (int i = 0; i < *V*; i++) {  
 for (int j = 0; j < *V*; j++) {  
 if (*dist*[i][j] == INF)  
 System.out.print("INF ");  
 else  
 System.out.print(*dist*[i][j] + " ");  
 }  
 System.out.println();  
 }  
 }  
}

*Anexa B* codul algoritmului Prim.