**Ministerul Educaţiei și Cercetării al Republicii Moldova**

**Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică și Microelectronică**

**RAPORT**

Lucrarea de laborator nr.4

*la Analiza și Proiectarea Algoritmilor*

A efectuat:

st. gr. TI-216 Roșca Dorin

A verificat : Lisnic Inga

Chişinău - 2022

**Lucrare de laborator nr 4**

**Tema:** Metoda programarii 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

**Rezumat succint la tema lucrării de laborator:**

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 printro 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.

**Codul programului in C**

**Algoritmele Dijkstra si Floyd:**

**#include** <iostream>

**#include** <limits.h>

**#include** <iomanip>

**#define** V 100

int iter1 = 0, iter2 = 0;

int randGraph[V][V];

void **GenRandomGraphs**(int Margine, int Vertex)

{

**for** (size\_t i = 0; i **<** Vertex; i++)

**for** (size\_t j = 0; j **<** Vertex; j++)

            randGraph[i][j] = 0;

    int count = 0;

    int i, j;

**while** (count **<** Margine)

    {

        i = **rand**() % Vertex;

        j = **rand**() % Vertex;

**if** (i **==** j **||** randGraph[i][j] **||** randGraph[j][i])

**continue**;

        randGraph[i][j] = randGraph[j][i] = **rand**() % 100;

        count++;

    }

}

//*Djikstra*

int **MinDistance**(int distance[], bool visited[])

{

    int min = INT\_MAX, minIndex;

**for** (size\_t v = 0; v **<** V; v++)

    {

**if** (**!**visited[v] **&&** distance[v] **<** min)

        {

            min = distance[v];

            minIndex = v;

        }

    }

**return** minIndex;

}

void **PrintDijkstraSolution**(int src, int distance[])

{

    std::cout << "Distance from Source (Dijkstra)" << std::endl;

**for** (size\_t v = 0; v **<** V; v++)

    {

**if** (distance[v] **!=** INT\_MAX)

            std::cout << src << "->" << v << " : " << distance[v] << std::endl;

    }

}

void **Dijkstra**(int graph[V][V], int source)

{

    int distance[V];

    bool visited[V];

    for (size\_t i = 0; i < V; i++)

    {

        distance[i] = INT\_MAX, visited[i] = false;

    }

    distance[source] = 0;

    for (size\_t edge = 0; edge < V - 1; edge++)

    {

        int u = MinDistance(distance, visited);

        visited[u] = true;

        for (size\_t v = 0; v < V; v++)

        {

            if (!visited[v] && graph[u][v] && distance[u] != INT\_MAX && distance[u] + graph[u][v] < distance[v])

            {

                distance[v] = distance[u] + graph[u][v];

            }

        }

    }

    PrintDijkstraSolution(source, distance);

}

//Floyd

void PrintFloydSolution(int distance[V][V], int src)

{

    std::cout << "Distance from Source (Floyd)" << std::endl;

    for (size\_t v = 0; v < V; v++)

        if (distance[src][v] != INT\_MAX)

            std::cout << src << "->" << v << " : " << distance[src][v] << std::endl;

}

void Floyd(int graph[V][V], int source)

{

    int distance[V][V];

    for (size\_t i = 0; i < V; i++)

        for (size\_t j = 0; j < V; j++)

            if (i == j)

                distance[i][j] = 0;

            else if (graph[i][j] == 0)

                distance[i][j] = INT\_MAX;

            else

                distance[i][j] = graph[i][j];

    for (size\_t k = 0; k < V; k++)

        for (size\_t u = 0; u < V; u++)

            for (size\_t v = 0; v < V; v++)

            {

                if (distance[u][v] > distance[u][k] + distance[k][v] && distance[u][k] != INT\_MAX && distance[k][v] != INT\_MAX)

                    distance[u][v] = distance[u][k] + distance[k][v];

            }

**PrintFloydSolution**(distance, source);

}

int **main**()

{

    double timeTaken;

**GenRandomGraphs**(4950, V);

    std::cout << "Algoritmul Dijkstra" << std::endl;

**Dijkstra**(randGraph, 0);

    std::cout << std::endl;

    std::cout << "Algoritmul Floyd" << std::endl;

**Floyd**(randGraph, 0);

    std::cout << std::endl;

**return** 0;

}

**Concluzie**

Efectuînd lucrarea dată am studiat algoritmele Dijkstra si FloydAceste 2 algoritme au acelasi scop-gasirea costul minim intre 2 varfuri a grafului insa efectuiaza operatii diferite :

1.Algoritmul Dijkstra 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

2.Algoritmul Floyd 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.