MINISTERUL EDUCAȚIEI ȘI CERCETĂRII
UNIVERSITATEA TEHNICĂ A MOLDOVEI
FACULTATEA CALCULATOARE INFORMATICĂ ȘI MICROELECTRONICĂ
TEHNOLOGIA INFORMAȚIEI

RAPORT

DISCIPLINA: Analiza și proiectarea algoritmilor (APA)

LUCRAREA DE LABORATOR NR. 4

TEMA: "Metoda programării dinamice."

Student: _____ **Raevschi Grigore TI-231**

Coordonator: _____ **Asistent univ. Coșer Cătălin**

Chișinău 2024

Cuprins

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.

1 Introducere programarea dinamică:

Programarea dinamică este o tehnică fundamentală în domeniul algoritmicii, utilizată pentru a rezolva probleme complexe prin divizarea acestora în subprobleme mai simple. Această metodă se bazează pe principiul optimizării substructurale, care afirmă că soluția optimă a unei probleme poate fi construită din soluțiile optime ale subproblemelor sale. De obicei, programarea dinamică este aplicată în probleme de optimizare, unde se caută o soluție maximă sau minimă.

Un aspect esențial al programării dinamice este utilizarea memorizării, care implică stocarea rezultatelor subproblemelor deja rezolvate pentru a evita recalcularea acestora. Această abordare îmbunătățește semnificativ eficiența algoritmilor, reducând complexitatea temporală și spațială a acestora.

În cadrul lucrării de laborator, ne propunem să studiem metoda programării dinamice prin analiza și implementarea unor algoritmi specifici. De asemenea, vom compara această tehnică cu metoda greedy, o altă strategie de rezolvare a problemelor, care, deși mai simplă, nu garantează întotdeauna soluția optimă. Prin această comparație, vom evidenția avantajele și dezavantajele fiecărei abordări, contribuind astfel la o înțelegere mai profundă a aplicațiilor programării dinamice în diverse domenii.

1.1 Algoritm Dijkstra:

Algoritmul de mia jos, cunoscut și sub denumirea de "algoritmul lui Dijkstra", este un algoritm fundamental în teoria grafurilor, utilizat pentru a determina cele mai scurte căi de la un nod sursă la toate celelalte noduri dintr-un graf ponderat. Acest algoritm a fost dezvoltat de către Edsger W. Dijkstra în 1956 și a devenit un standard în domeniul informaticii, având aplicații diverse, de la rutare în rețele de calculatoare până la navigația GPS.

Algoritmul funcționează prin explorarea nodurilor grafului în mod iterativ, actualizând distanțele minime către fiecare nod în funcție de greutățile muchiilor. Inițial, toate distanțele sunt setate la un valorile mari, exceptând nodul sursă, care are distanța zero. Pe măsură ce algoritmul progresează, acesta selectează nodul cu cea mai mică distanță cunoscută, extinzând astfel căutarea către vecinii săi și ajustând distanțele în funcție de costurile întâlnite.

Un aspect esențial al algoritmului de mia jos este eficiența sa, care poate fi îmbunătățită prin utilizarea unor structuri de date adecvate, cum ar fi heap-urile, ceea ce permite reducerea complexității temporale. În cadrul lucrării de laborator, ne propunem să studiem algoritmul de mia jos, să analizăm implementarea sa și să explorăm compararea cu alte metode de calcul al distanțelor, evidențiind avantajele și limitările sale în diverse scenarii.

```
function Dijkstra(Graph, source):
    for each vertex v in Graph.Vertices:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0
    while Q is not empty:
        u ← vertex in Q with minimum dist[u]
        remove u from Q
        for each neighbor v of u still in Q:
            alt ← dist[u] + Graph.Edges(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u
    return dist[], prev[]
```

1.2 Algoritm Floyd:

Algoritmul Floyd-Warshall este o metodă eficientă pentru determinarea celor mai scurte căi între toate perechile de noduri dintr-un graf ponderat, având aplicații semnificative în teoria grafurilor și în domenii precum rețelistică, optimizarea rutelor și analiza rețelelor. Acest algoritm, dezvoltat de Robert Floyd și Stephen Warshall, se bazează pe o abordare dinamică și permite actualizarea distanțelor minime între noduri printr-o serie de iterații.

Algoritmul începe prin inițializarea unei matrice de distanțe, unde fiecare element reprezintă distanța minimă cunoscută între două noduri. În prima etapă, greutățile muchiilor sunt introduse în matrice, iar distanțele de la un nod la sine sunt setate la zero. Apoi, printr-o serie de iterații, algoritmul verifică dacă o cale între două noduri poate fi îmbunătățită prin intermediul unui nod intermediar. Această verificare se face pentru fiecare combinație de noduri, asigurându-se astfel că toate posibilele căi sunt explorate.

Unul dintre avantajele algoritmului Floyd-Warshall este că poate gestiona grafuri cu greutăți negative, atât timp cât nu există cicluri negative. Complexitatea sa temporală este O(V^3), ceea ce îl face mai puțin eficient pentru grafuri foarte mari, dar extrem de util în scenarii unde este necesară cunoașterea distanțelor minime între toate perechile de noduri. În cadrul lucrării de laborator, ne propunem să studiem algoritmul Floyd-Warshall, să analizăm implementarea sa și să comparăm eficiența acestuia cu alte metode de calcul al distanțelor în grafuri.

```
let dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)
for each edge (u, v) do
    dist[u][v] ← w(u, v)  // The weight of the edge (u, v)
for each vertex v do
    dist[v][v] ← 0
for k from 1 to |V|
    for i from 1 to |V|
        for j from 1 to |V|
            if dist[i][j] > dist[i][k] + dist[k][j]
                dist[i][j] ← dist[i][k] + dist[k][j]
            end if
```

2 Compararea algoritmilor Dijkstra și Floyd:

Tabelul 1.1 compară algoritmii Dijkstra și Floyd-Warshall, fiind foarte util pentru a înțelege cum fiecare dintre ei rezolvă problemele legate de găsirea celor mai scurte căi în grafuri. Ambele metode sunt folosite în domenii importante, cum ar fi rețelistică, optimizarea rutelor și analiza datelor. Tabelul de mai jos oferă o privire detaliată asupra caracteristicilor fiecărui algoritm, cum ar fi timpul și spațiul necesar pentru a funcționa, tipurile de grafuri pentru care sunt adecvate, precum și eficiența și modul în care pot fi aplicate. Această comparație va ajuta la alegerea algoritmului cel mai potrivit în funcție de natura problemei care trebuie rezolvată.

Tabel 1.1: Tabel comparativ între algoritmii Dijkstra și Floyd-Warshall

Caracteristică	Algoritmul Dijkstra	Algoritmul Floyd-Warshall
Tip de graf	Grafuri orientate/ponderate (fără cicluri negative)	Grafuri orientate/ponderate (poate gestiona greutăți negative, fără cicluri negative)
Complexitate temporală	$O(V^2)$ sau $O(E\ log\ V)$ (cu heap)	$O(V^3)$
Complexitate spațială	$O(V)$ (pentru distanțe)	$O(V^2)$ (matricea distanțelor)
Tipul soluției	Distanțe minime de la un singur nod la toate celelalte	Distanțele minime între toate perechile de noduri
Eficiență	Mai eficient pentru grafuri sparse	Mai eficient pentru grafuri dense
Implementare	Mai simplu de implementat	Poate fi mai complex din cauza matricei
Existența ciclurilor negative	Nu poate gestiona cicluri negative	Poate gestiona cicluri negative (fără a le include)
Utilizare	Rutare, navigație, rețele	Analiza rețelelor, optimizarea căilor

3 Compararea Tehnicii Greedy cu Metoda de Programare Dinamică:

Tehnica greedy și metoda de programare dinamică sunt două metode populare folosite pentru a rezolva probleme de optimizare, fiecare având propriile sale trăsături și beneficii.

Tehnica greedy, cunoscută și ca "lacomă", se bazează pe ideea de a face alegeri optime în fiecare etapă, sperând că aceste alegeri vor duce la o soluție bună în ansamblu. Această metodă este de obicei simplă și eficientă, fiind ușor de pus în practică. Exemplele tipice includ problemele legate de selecția activităților, problema rucsacului (în varianta fracționară) și algoritmii pentru găsirea arborelui minim. Totuși, nu toate problemele pot fi rezolvate eficient prin această abordare, deoarece nu asigură întotdeauna o soluție optimă în toate situațiile.

Pe de altă parte, programarea dinamică abordează problemele împărțindu-le în subprobleme mai mici, pe care le rezolvă într-un mod ordonat. Aceasta presupune salvarea rezultatelor subproblemelor pentru a evita recalcularea lor, ceea ce îmbunătățește eficiența. Programarea dinamică este foarte utilă pentru probleme care au caracteristici de substructură optimă și care implică subprobleme care se suprapun. Exemplele includ problema rucsacului (în varianta 0/1), calcularea numărului de căi pentru a ajunge la un anumit punct și problemele legate de secvențe.

În Tabelul 1.2 se face o comparație între acești doi algoritmi.

Tabel 1.2: Comparația între Tehnica Greedy și Programarea dinamică

Caracteristică	Tehnica Greedy	Programarea Dinamică
Abordare	Alegeri locale optime	Rezolvarea subproblemelor
Complexitate	De obicei mai mică	Poate fi mai mare, dar eficientă
Garantie de optimizare	Nu garantează soluții optime	Garantează soluții optime
Implementare	Mai simplă	Poate fi mai complexă
Exemple	Selecția activităților	Problema rucsacului (0/1)

3.1 Implementarea algoritmului Dijkstra:

În acest proiect am explorat implementarea algoritmului Dijkstra, o metodă eficientă utilizată pentru a găsi cel mai scurt drum într-un graf. Am ales acest algoritm datorită aplicațiilor sale practice în domenii precum rețelele de calculatoare, navigația GPS sau optimizarea rutelor. Lucrarea se concentrează pe rezolvarea unei probleme clasice: determinarea costului minim de a ajunge dintr-un nod inițial la celelalte noduri ale unui graf, reprezentat sub forma unei matrice de adiacență.

Pentru a testa performanța algoritmului, am folosit două tipuri de grafuri: unul dens, în care majoritatea nodurilor sunt conectate între ele, și unul rar, cu un număr redus de conexiuni. Această abordare ne permite să observăm modul în care structura grafului influențează timpul de execuție și numărul de operații necesare. În continuare, voi detalia implementarea algoritmului, punând accent pe pașii principali și pe modul în care aceștia contribuie la găsirea soluției.

Codul programului în **ANEXA A**.

3.2 Implementarea algoritmului Floyd

În acest proiect, am decis să implementez algoritmul Floyd-Warshall pentru a găsi cele mai scurte drumuri între toate perechile de noduri dintr-un graf. Alegerea acestui algoritm s-a bazat pe eficiența sa în a rezolva probleme de optimizare, fiind des utilizat în diverse aplicații practice, cum ar fi rețelele de transport sau comunicațiile.

Lucrarea analizează modul în care algoritmul parcurge graful și actualizează progresiv distanțele dintre noduri, verificând fiecare posibilitate de drum intermediar. Pentru a demonstra funcționarea acestuia, am folosit un exemplu de graf reprezentat printr-o matrice de adiacență, citită dintr-un fișier extern. În continuare, voi prezenta implementarea pas cu pas, explicând fiecare etapă și rezultatele obținute.

Codul programului în ANEXA B.

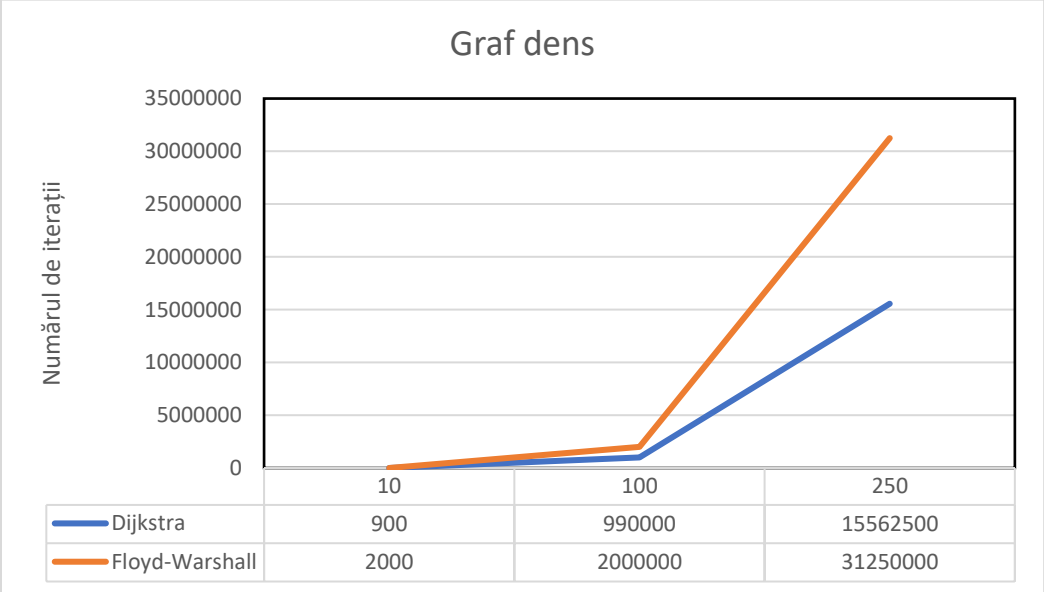
4 Analiza empirică

4.1 Graf dens

După cum se poate observa în Tabelul 1.3, analiza empirică a performanței algoritmului pentru grafuri dense oferă o imagine clară asupra timpului de execuție și a numărului de iterații necesare în diferite condiții. Aceste rezultate sunt esențiale pentru a înțelege comportamentul algoritmilor de procesare a grafurilor în funcție de densitatea conexiunilor între noduri. În continuare, vom explora impactul dimensiunii grafului și al densității acestuia asupra eficienței algoritmilor, precum și variațiile de performanță observate în funcție de parametrii specifici aleși în cadrul experimentelor.

Tabel 1.3: Analiza empirica pentru grafuri dense

Algoritmul	Nr.de noduri	Iteratii	Timpul
Dijkstra	250	15562500	4,690000
Dijkstra	100	990000	0,757000
Dijkstra	10	900	0,010000
Floyd–Warshall	250	31250000	2,060000
Floyd–Warshall	100	2000000	0,331000
Floyd–Warshall	10	2000	0,004000

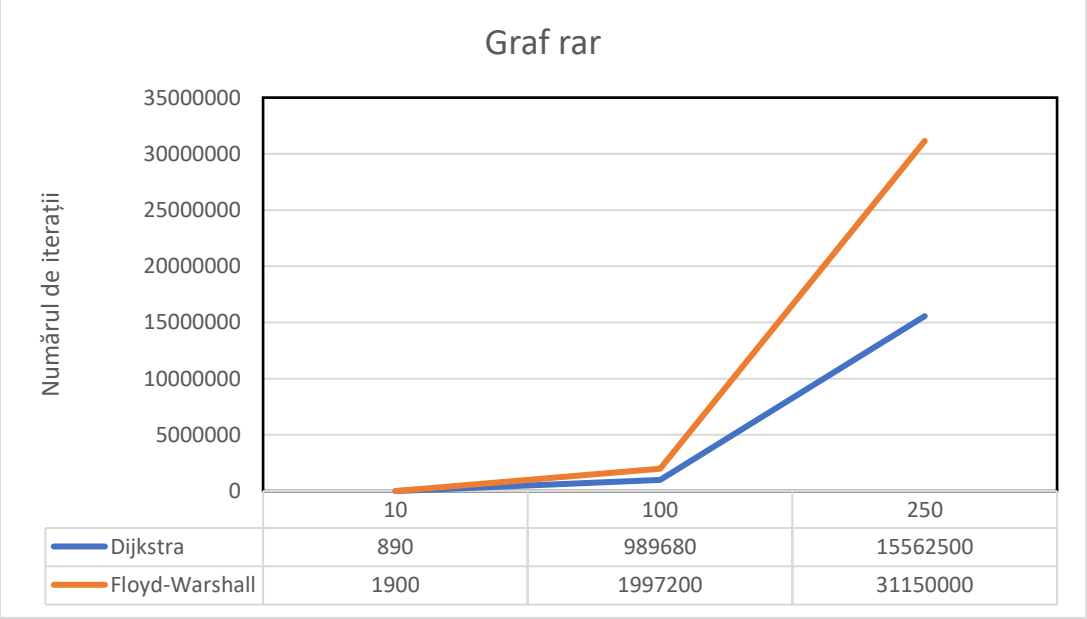


4.2 Graf rar

După cum se poate observa în Tabelul 1.4, analiza empirică pentru grafurile rare subliniază comportamentul algoritmilor în condiții de densitate scăzută a conexiunilor între noduri. În cazul unui graf rar, există mai puține muchii comparativ cu nodurile, ceea ce duce la o complexitate mai mică în anumite cazuri de calcul, dar și la posibile dificultăți în găsirea unor soluții rapide pentru anumite tipuri de probleme. Acest tabel oferă o privire detaliată asupra performanței algoritmilor implementați, evidențiind timpul de execuție și numărul de iterații în funcție de diferite seturi de date. În continuare, vom analiza cum aceste rezultate influențează alegerea metodelor de calcul pentru grafurile rare și impactul lor asupra complexității și eficienței algoritmice.

Tabel 1.4: Analiza empirica pentru grafuri rar

Algoritmul	Nr.de noduri	Iteratii	Timpul
Dijkstra	250	15562500	4,689600
Dijkstra	100	989680	0,757000
Dijkstra	10	890	0,010000
Floyd–Warshall	250	31150000	2,052000
Floyd–Warshall	100	1997200	0,330000
Floyd–Warshall	10	1900	0,003080



5 Concluzie

În cadrul acestei lucrări de laborator, am avut ocazia să studiem și să aplicăm metoda programării dinamice, o tehnică importantă în domeniul informaticii, care ajută la rezolvarea problemelor complexe prin descompunerea acestora în subprobleme mai mici, mai ușor de gestionat. Am analizat diverse exemple de algoritmi care utilizează această metodă și am implementat soluții pentru probleme care sunt frecvent întâlnite în practică, cum ar fi calculul drumurilor minime sau determinarea celei mai bune secvențe într-un set de date.

Un aspect important al lucrării a fost comparația între tehnica greedy și programarea dinamică. Am observat că, în timp ce ambele metode sunt folosite pentru optimizarea unui anumit rezultat, acestea se diferențiază în modul în care abordează problema. Algoritmul greedy face alegeri imediate, bazate pe situația curentă, fără a lua în considerare întregul context al problemei, ceea ce poate duce la soluții suboptime în anumite cazuri. Pe de altă parte, programarea dinamică examinează întregul spectru al soluțiilor posibile, construind progresiv soluția optimă pe baza rezultatelor parțiale, ceea ce garantează obținerea unui rezultat mai bun, dar cu un cost mai mare în termeni de resurse (timp și memorie).

Prin implementarea și analiza acestora, am reușit să observăm avantajele și dezavantajele fiecărei metode în funcție de complexitatea și natura problemei abordate. În timp ce tehnica greedy este rapidă și eficientă în situațiile unde alegerea imediată este suficientă pentru a obține un rezultat bun, programarea dinamică devine esențială atunci când este necesar un rezultat optim, chiar dacă acest lucru presupune un cost suplimentar de calcul. În final, am realizat că alegerea metodei de rezolvare a unei probleme depinde în mare măsură de specificul acesteia, iar cunoașterea și înțelegerea fiecărei tehnici reprezintă un pas important în devenirea unui programator eficient.

Această lucrare de laborator m-a ajutat să aprofundez concepte importante legate de programarea dinamică și să îmi dezvolt abilități esențiale pentru rezolvarea problemelor complexe. Totodată, mi-a oferit o înțelegere mai clară a modului în care diferite tehnici pot fi utilizate pentru a optimiza soluțiile și pentru a rezolva problemele într-un mod mai eficient. În concluzie, prin studiul acestor tehnici și prin aplicarea lor în practică, am dobândit o pregătire mai bună pentru a aborda probleme de programare dinamică în viitor, fie în scopuri academice, fie în proiecte profesionale.

6 Bibliografie

[1] „Floyd–Warshall algorithm.”
https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm

[2] „Floyd Warshall Algorithm.”
https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/

[3] „Dijkstra's algorithm. ”
https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

[4] „How to find Shortest Paths from Source to all Vertices using Dijkstra’s Algorithm.”
https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

[5] „Evaluarea performanțelor algoritmilor Modified Dijkstra și Floyd-Warshall.”
https://www.slideshare.net/slideshow/performance-evaluation-of-the-floyd-warshall-and-the-modified/24240882

[6] „Dijkstra vs Floyd-Warshall Algorithms.”
https://www.baeldung.com/cs/dijkstra-vs-floyd-warshall

[7] „Complexitatea Algoritmilor.”
https://www.cs.ubbcluj.ro/wpcontent/uploads/MuresanHoreaBogdan_ComplexitateaAlgoritmilor.pdf

[8] „Articol 08 - Grafuri”
https://ocw.cs.pub.ro/courses/sd-ca/2016/articole/articol-08

[9] „Greedy algorithms vs. dynamic programming: How to choose”
https://www.educative.io/blog/greedy-algorithm-vs-dynamic-programming

[10] „Floyd-Warshall Algorithm in C”
https://www.geeksforgeeks.org/floyd-warshall-algorithm-in-c/

ANEXA A: Implementarea Dikstra

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <time.h>
#include <locale.h>

#define V 6
int it = 0;
// Algoritmul Dijkstra
void Dijkstra(int GR[V][V], int st) {
    int distance[V], visited[V];
    int i, count, u, min, index;

    for (i = 0; i < V; i++) {
        distance[i] = INT_MAX; // Inițializare distanțe la infinit
        visited[i] = 0;        // Niciun vârf nu este vizitat
    }

    distance[st] = 0; // Distanța către sine este 0

    for (count = 0; count < V - 1; count++) {
        min = INT_MAX;

        // Găsește nodul cu cea mai mică distanță
        for (i = 0; i < V; i++) {
            if (!visited[i] && distance[i] <= min) {
                min = distance[i];
                index = i;
            }
        }

        u = index; // Vârful curent
        visited[u] = 1;
```

```
        // Actualizează distanțele pentru vecinii nevizitați
        for (i = 0; i < V; i++) {
            it++;
            if (!visited[i] && GR[u][i] && distance[u] != INT_MAX &&
                distance[u] + GR[u][i] < distance[i]) {
                distance[i] = distance[u] + GR[u][i];
            }
        }
    }

    printf("Costul din vârful inițial până în restul vârfurilor:\n");
    printf("-----------\n");
    for (i = 0; i < V; i++) {
        if (distance[i] != INT_MAX)
            printf("%d -> %d = %d\n", st + 1, i + 1, distance[i]);
        else
            printf("%d -> %d = inaccesibil\n", st + 1, i + 1);
    }
    printf("-----------\n");
}

int main() {
    setlocale(LC_ALL, ""); // Setează localizarea pentru afișare
    clock_t start, end;
    int GR[V][V];
    FILE *f;
    int i, j, x;

    start = clock();

    // Deschiderea fișierului
    f = fopen("g6des.txt", "r"); // pentru graf dens
    // f = fopen("g6des_rar.txt", "r"); // Pentru graf rar
    if (f == NULL) {
```

```
        perror("Eroare la deschiderea fișierului");
        return 1;
    }

    // Citirea matricei de adiacență
    for (i = 0; i < V; i++) {
        for (j = 0; j < V; j++) {
            if (fscanf(f, "%d", &x) != 1) {
                perror("Eroare la citirea fișierului");
                fclose(f);
                return 1;
            }
            GR[i][j] = x;
        }
    }
    fclose(f);

    // Rulează Dijkstra pentru fiecare vârf
    for (i = 0; i < V; i++) {
        Dijkstra(GR, i);
    }

    end = clock();

    // Timpul de execuție
    printf("\n\nTimpul de execuție este: %.8f secunde\n", ((double)(end - start)) /
CLOCKS_PER_SEC);
    printf("Număr de iterații: %d\n", it);

    return 0;
}
```

ANEXA B: Implementarea Floyd

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <locale.h>

#define maxV 1000

int i, j, n;
int it = 0;
int GR[maxV][maxV];

// Algoritmul Floyd-Warshall
void FU(int D[][maxV], int V)
{
    int k;

    // Inițializare: diagonală principală cu 0
    for (i = 0; i < V; i++)
    {
        D[i][i] = 0;
    }

    // Algoritmul propriu-zis
    for (k = 0; k < V; k++)
    {
        for (i = 0; i < V; i++)
        {
            for (j = 0; j < V; j++)
            {
                it++;
                if (D[i][k] && D[k][j] && i != j)
                {
                    if (D[i][k] + D[k][j] < D[i][j] || D[i][j] == 0)
```

```
                    {
                        D[i][j] = D[i][k] + D[k][j];
                    }
                }
                it++;
            }
        }
    }

    // Afișarea matricei drumurilor minime
    printf("\nMatricea Drumurilor Minime:\n");
    for (i = 0; i < V; i++)
    {
        for (j = 0; j < V; j++)
        {
            if (D[i][j] == 0 && i != j)
                printf("INF\t");
            else
                printf("%d\t", D[i][j]);
        }
        printf("\n");
    }
}

int main()
{
    setlocale(LC_ALL, "");
    clock_t start, end;

    n = 6; // Numărul de noduri
    int x;

    // Citirea din fișier
    FILE *f = fopen("g6des.txt", "r");
    if (f == NULL)
```

```
    {
        perror("Eroare la deschiderea fișierului");
        return 1;
    }

    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
            if (fscanf(f, "%d", &x) != 1)
            {
                perror("Eroare la citirea fișierului");
                fclose(f);
                return 1;
            }
            GR[i][j] = x;
        }
    }
    fclose(f);

    // Calcularea și afișarea drumurilor minime
    printf("\tFloyd-Warshall:\n");
    printf("-------------------------------------------\n");
    start = clock();
    FU(GR, n);
    end = clock();
    printf("-------------------------------------------\n");

    // Timpul de execuție și numărul de iterații
    printf("\n\nTimpul de execuție este: %.8f secunde\n", ((double)(end - start)) /
CLOCKS_PER_SEC);
    printf("Număr de iterații: %d\n", it);

    return 0;
}
```