Universitatea Tehnică a Moldovei
Facultatea Calculatoare Informatică și Microelectronică
Departamentul Ingineria Software și Automatică

RAPORT

La lucrarea de laborator nr. 4

TEMA: „ Analiza şi implementarea algoritmilor de programare dinamică Dijkstra și Floyd-Warshall ”

Disciplina: Analiza și proiectarea algoritmilor

A efectuat studentul: gr.SI-201 Ivanova Evghenia
A verificat: asistentul universitar Buldumac Oleg

Chișinău 2021

Scopul lucrării : Analiza și implementarea algoritmilor Dijkstra și Floyd-Warshall pentru găsirea drumului de cost minim într-un graf ponderat orientat.

Sarcina : Să analizați și elaborați algoritmii Dijkstra și Floyd-Warshall pentru determinarea drumului de cost minim de la un nod sursă la oricare altul dintr-un graf orientat. În raport trebuie să aveți 2 tabele, unul pentru Dijkstra și al doilea pentru Floyd-Warshall. Trebuie să analizați algoritmii pentru 3 cazuri și anume cînd graful este rar, mediu și dens.

Considerații teoretice :

În general, există două tipuri de grafuri posibile:

1. Graf neorientat: folosind un graf neorientat, pentru fiecare pereche de noduri conectate, puteți trece de la un nod la altul în ambele direcții.
2. Graful orientat: puteți trece doar de la un nod la altul într-o direcție specifică pentru fiecare pereche de noduri conectate. Puteți folosi săgețile ce conectează două noduri.

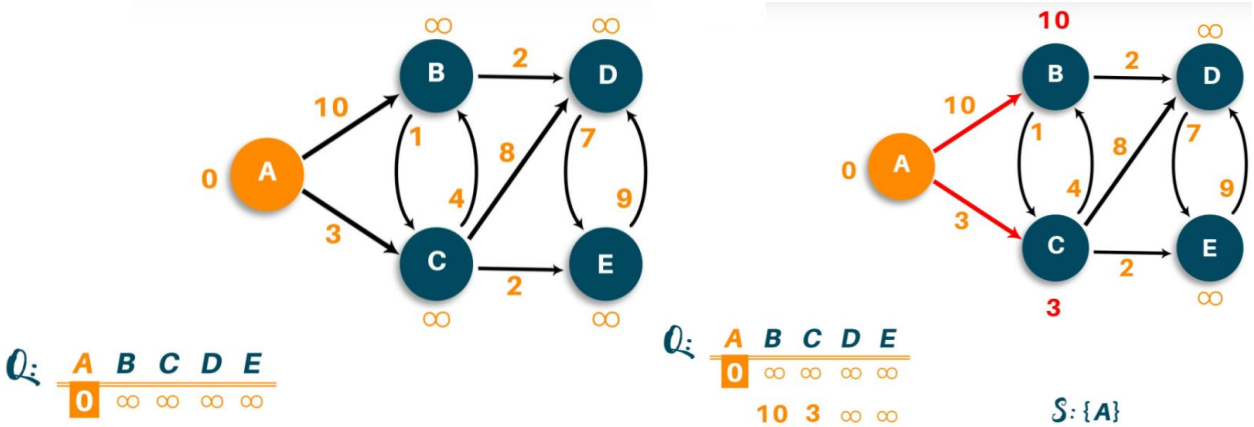
Ce este algoritmul lui Dijkstra?

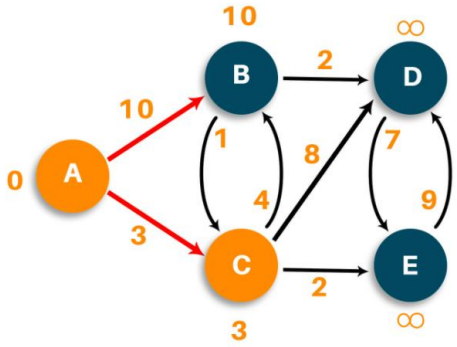
Algoritmul lui Dijkstra este cunoscut și ca algoritmul cu calea cea mai scurtă. Algoritmul creează arborele celor mai scurte căi de la vârful sursă de pornire spre toate celelalte puncte din graf. Diferă de arborele de acoperire minim, deoarece distanța cea mai scurtă dintre două vârfuri poate să nu fie inclusă în toate vârfurile grafului. Algoritmul funcționează prin construirea unui set de noduri care au o distanță minimă față de sursă. Aici, algoritmul lui Dijkstra folosește o abordare lacomă pentru a rezolva problema și a găsi cea mai bună soluție.

Când eșuează algoritmul lui Dijkstra

Algoritmul lui Dijkstra funcționează numai cu graful care are ponderi pozitive. În timpul rulării unui algoritm, greutățile marginilor trebuie adăugate pentru a găsi calea cea mai scurtă între noduri. Dacă există o pondere negativă în graf, atunci algoritmul nu va funcționa. Amintiți-vă că, odată ce marcați nodul ca „vizitat”, calea curentă către nod este cea mai scurtă cale pentru a ajunge la acel nod. Prin urmare, dacă aveți ponderi negative, acesta poate modifica acest pas dacă greutatea totală este redusă.

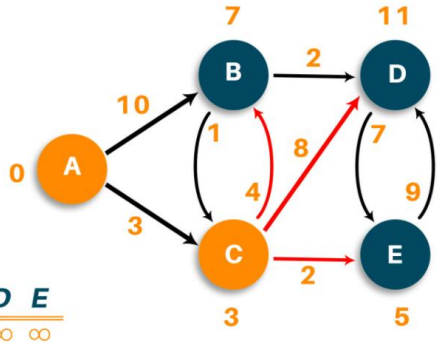
Exemplu





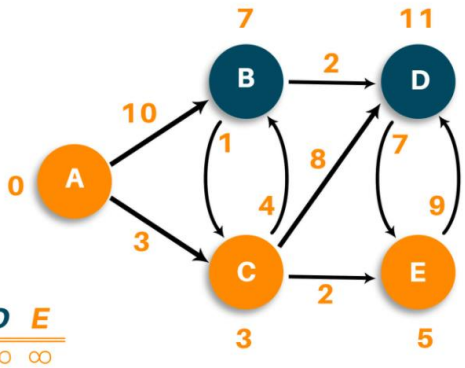
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
	10	3	∞	∞	

S: {A, C}



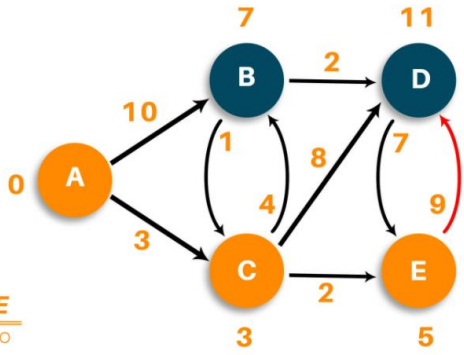
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
	10	3	∞	∞	
	7		11	5	

S: {A, C}



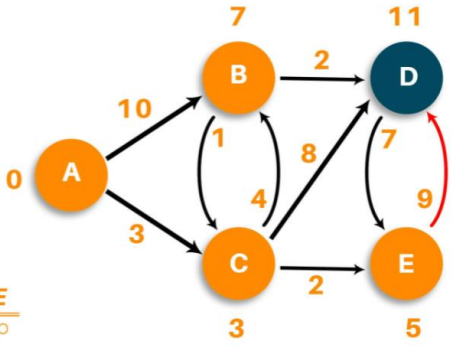
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
	10	3	∞	∞	
	7		11	5	

S: {A, C, E}



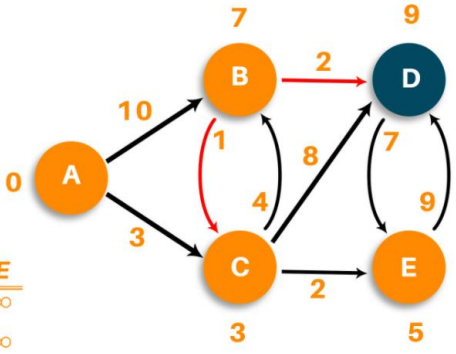
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
	10	3	∞	∞	
	7		11	5	
	7		11		

S: {A, C, E}



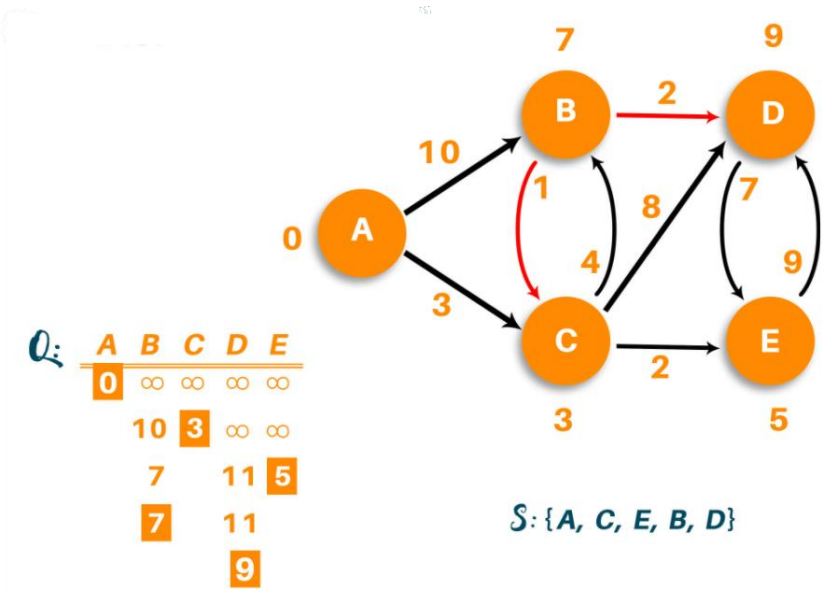
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
	10	3	∞	∞	
	7		11	5	
	7		11		

S: {A, C, E, B}



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
	10	3	∞	∞	
	7		11	5	
	7		11		
			9		

S: {A, C, E, B}



Complexitatea timpului

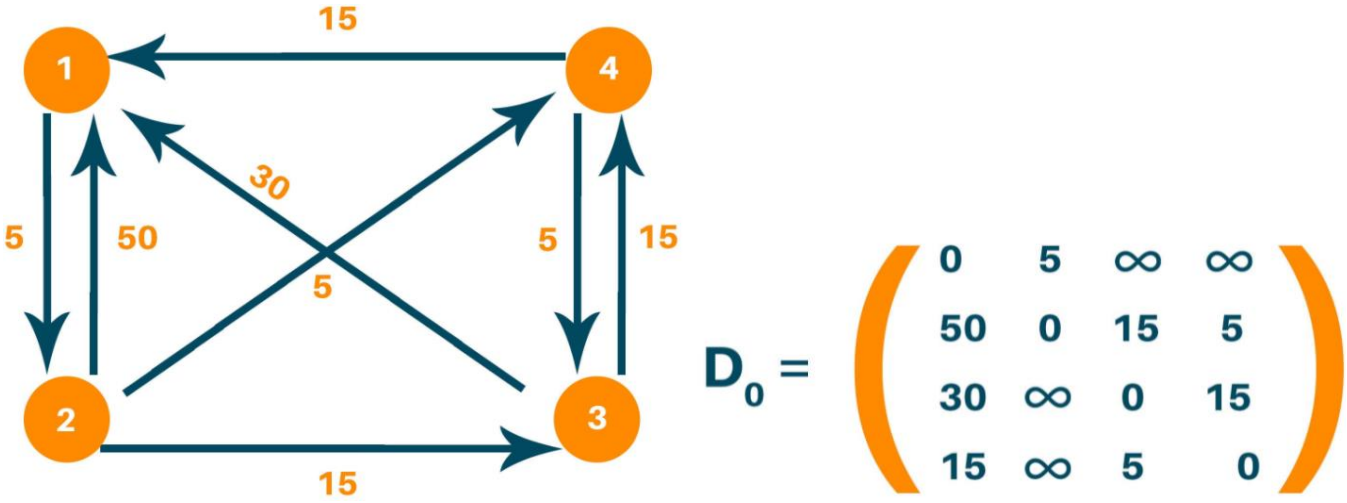
Complexitatea temporală a algoritmului lui Dijkstra este O(V^2) , unde V este numărul de vârfuri din graf. Totuși, dacă graful de intrare este reprezentat folosind o listă de adiacență, atunci complexitatea timpului poate fi redusă la O(E log V) folosind un heap binar.

Complexitatea spațială a algoritmului lui Dijkstra este O(V) , unde V este numărul total de vârfuri ale grafului. Acest lucru se datorează faptului că trebuie să stocăm toate aceste vârfuri în listă ca ieșire.

Ce este algoritmul Floyd Warshall?

La fel ca algoritmul lui Dijkstra, algoritmul Floyd Warshall este folosit pentru a găsi calea cea mai scurtă între toate vârfurile din graful ponderat. Acest algoritm funcționează atât cu grafurile orientate, cât și cu neorientate, dar nu funcționează cu graful cu cicluri negative. Prin urmare, dacă distanța de la vârful v față de sine este negativă, atunci putem presupune că graful are prezența unui ciclu negativ. Acest algoritm urmează abordarea de programare dinamică ca model de lucru. Aici algoritmul nu construiește calea în sine, dar poate reconstrui calea cu o simplă modificare. Algoritmul Floyd Warshall este cunoscut și ca algoritm Roy Warshall sau algoritm Roy-Floyd.

Cum funcționează algoritmul Floyd Warshall



$$D_1 = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix} \quad D_2 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$$D_3 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 45 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix} \quad D_4 = \begin{pmatrix} 0 & 5 & 15 & 10 \\ 20 & 0 & 10 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

Complexitatea timpului

Există trei bucle pentru a calcula calea cea mai scurtă din graf și fiecare dintre aceste bucle are complexități constante. Prin urmare, datorită acestui fapt, complexitatea temporală a algoritmului Floyd Warshall este $O(n^3)$. De asemenea, complexitatea spațială a algoritmului Floyd Warshall este $O(n^2)$.

Codul programului :

```
#include <bits/stdc++.h>

#define MAX 999

using namespace std;

int iter, V;


void initializare(int **mat){
        for(int i = 0; i < V; i++)
                for(int j = 0; j < V; j++)
                        mat[i][j] = 0;
}


void initializareFloyd(int **mat){ //initilizare cu infinit in loc de 0
        for(int i = 0; i < V; i++)
                for(int j = 0; j < V; j++)
                   if(mat[i][j] == 0 && i != j) mat[i][j] = MAX;
}
```

```
void generare(double p, int **mat){
        int d = 1, i = 0, j = d + i, x = p*V*(V-1); //calculeaza cate muchii trebuie sa genereze
        srand(time(NULL));
        for(int k = 0; k < x; k+=2){
//parcurge matricea pe diagonale,paralele cu cea principala si genereaza atatea muchii x
                if(j >= V){
                        d ++;
                        i = 0;
                        j = i + d;
                }
                mat[j][i] = rand()% 100+1;
                mat[i++][j++] = rand()% 100+1;
        }
        /*for(i = 0; i < V; i++)
      for(j = 0; j < V; j++)
        if(mat[i][j] != 0) cout << " " << i << " -> " << j << "\t\t" << mat[i][j] << endl;*/
}

int minDist(int dist[], bool set[]){
   int min = MAX, index;
   for(int v = 0; v < V; v++)
      if(set[v] == false && dist[v] <= min) min = dist[v], index = v;
   return index;
}

void path(int parent[], int j, int **mat){
   if(parent[j] == -1) return;
   path(parent, parent[j], mat);
   cout << "--(" << mat[parent[j]][j] << ")-->" << j;
}
```

```
void Dijkstra(int **mat, int src){
    cout << "\n\tDjikstra\n\n";
    iter = 0;
          int dist[V], parent[V];
    bool set[V];
    for(int i = 0; i < V; i++)
       dist[i] = MAX, set[i] = false, parent[0] = -1; //set dist infinit,nod vizitate cu false,nod anter -1
    dist[src] = 0;
    for(int n = 0; n < V-1; n++){
       int u = minDist(dist, set);
       set[u] = true; //marcheaza nodul curent vizitat
       for(int v = 0; v < V; v++){
          if(!set[v] && mat[u][v] && dist[u] != MAX && dist[u] + mat[u][v] < dist[v]){
             dist[v] = dist[u] + mat[u][v];
             parent[v] = u;
             iter ++;
                          }
                    }
    }
    /*for(int i = 1; i < V; i++){
       cout  << "\n " << src << " -> " << setw(3) << i << setw(10) << dist[i] << "     " << src;
       path(parent, i, mat);
          }*/
    cout << "\n " << iter << " iteratii   ";
}

void pathFloyd(int **path, int v, int u, int **mat, int *pref){
    if(path[v][u] == v) return;
    pathFloyd(path, v, path[v][u], mat, pref);
          *pref = path[v][u];
```

```
    cout << "--(" << mat[path[v][path[v][u]]][path[v][u]] << ")-->" << path[v][u];
}

void Floyd(int **mat){
    cout << "\n\tFloyd Warshall\n\n";
    iter = 0;
    int **cost, **path; //matricea consturilor si drumurilor
    cost = new int*[V];
    path = new int*[V];
        for(int i = 0; i < V; i++)
                cost[i] = new int[V];
        for(int i = 0; i < V; i++)
                path[i] = new int[V];
    for(int v = 0; v < V; v++)
       for(int u = 0; u < V; u++){
          cost[v][u] = mat[v][u]; //matricea costurilor devine graful initial
          if(v == u) path[v][u] = 0; //pe diagonala in matricea drumurilor se pune 0
          else if(cost[v][u] != MAX) path[v][u] = v; //daca exista muchie se pune in matricea drumurilor
          else path[v][u] = -1; //daca nu este muchie se pune -1
       }
    for(int k = 0; k < V; k++) //k nod intermediar, incepe cu 0 si termina cu V
       for(int v = 0; v < V; v++) //v rand
          for(int u = 0; u < V; u++){ //u coloana
             if(cost[v][k] != MAX && cost[k][u] != MAX && cost[v][k] + cost[k][u] < cost[v][u]){
                cost[v][u] = cost[v][k] + cost[k][u];
                path[v][u] = path[k][u];
                iter ++;
             }
          }
          /*for(int v = 0; v < 1; v++)
```

```
        for(int u = 0; u < V; u++)
            if(u != v && path[v][u] != -1){
                int pref = v;
                cout << v << " -> " << setw(3) << u << setw(10) << cost[v][u] << "     " << v;
                pathFloyd(path, v, u, mat, &pref);
                cout << "--(" << mat[pref][u] << ")-->";
                cout << u << endl;
            }*/
    cout << " " << iter << " iteratii   ";
}

int main(){
            int **mat;
            double p;
            cout << " Introdu nr de varfuri ";
            cin >> V;
            mat = new int*[V];
            for(int i = 0; i < V; i++)
                        mat[i] = new int[V];

            cout << "\n Graf Rar\n\n";
            p = 0.4;
            initializare(mat);
            generare(p, mat);
            auto startR = chrono::high_resolution_clock::now();
            Dijkstra(mat, 0);
            auto endR = chrono::high_resolution_clock::now();
cout <<" timp de "<< chrono::duration_cast<chrono::nanoseconds>(endR - startR).count()*1e-9 << " s\n\n ";
            initializareFloyd(mat);
            auto startRF = chrono::high_resolution_clock::now();
```

```
        Floyd(mat);
        auto endRF = chrono::high_resolution_clock::now();
cout <<" timp de "<<chrono::duration_cast<chrono::nanoseconds>(endRF-startRF).count()*1e-9<<" s\n\n ";

        cout << "\n Graf Mediu\n\n";
        p = 0.6;
        initializare(mat);
        generare(p, mat);
        auto startM = chrono::high_resolution_clock::now();
        Dijkstra(mat, 0);
        auto endM = chrono::high_resolution_clock::now();
cout<<" timp de "<< chrono::duration_cast<chrono::nanoseconds>(endM - startM).count()*1e-9 << " s\n\n ";
        initializareFloyd(mat);
         auto startMF = chrono::high_resolution_clock::now();
        Floyd(mat);
        auto endMF = chrono::high_resolution_clock::now();
cout<<" timp de "<<chrono::duration_cast<chrono::nanoseconds>(endMF-startMF).count()*1e-9<<" s\n\n ";

        cout << "\n Grad Dens\n\n";
        p = 0.9;
        initializare(mat);
        generare(p, mat);
        auto startD = chrono::high_resolution_clock::now();
        Dijkstra(mat, 0);
        auto endD = chrono::high_resolution_clock::now();
cout<<" timp de "<< chrono::duration_cast<chrono::nanoseconds>(endD - startD).count()* 1e-9 << " s\n\n ";
        initializareFloyd(mat);
        auto startDF = chrono::high_resolution_clock::now();
        Floyd(mat);
       auto endDF = chrono::high_resolution_clock::now();
```

```
cout<<" timp de "<<chrono::duration_cast<chrono::nanoseconds>(endDF-startDF).count()*1e-9<<" s\n\n ";
}
```

Link – ul codului plasat pe GitHub

https://github.com/AgentSI/AgentSI/blob/main/APA_4.cpp

Execuția programului :

```
Introdu nr de varfuri 100

Graf Rar

       Djikstra

312 iteratii    timp de 0.000992 s

       Floyd Warshall

37512 iteratii    timp de 0.008067 s

Graf Mediu

       Djikstra

330 iteratii    timp de 0.000996 s

       Floyd Warshall

41523 iteratii    timp de 0.008541 s

Grad Dens

       Djikstra

393 iteratii    timp de 0.000996 s

       Floyd Warshall

49838 iteratii    timp de 0.007119 s
```

```
Introdu nr de varfuri 200

Graf Rar

       Djikstra

730 iteratii    timp de 0.004376 s

       Floyd Warshall

177535 iteratii    timp de 0.106664 s

Graf Mediu

       Djikstra

772 iteratii    timp de 0.004013 s

       Floyd Warshall

204963 iteratii    timp de 0.100463 s

Grad Dens

       Djikstra

826 iteratii    timp de 0.001173 s

       Floyd Warshall

240781 iteratii    timp de 0.053051 s
```

```
Introdu nr de varfuri 300

Graf Rar

       Djikstra

1123 iteratii    timp de 0.006678 s

       Floyd Warshall

439774 iteratii    timp de 0.200946 s

Graf Mediu

       Djikstra

1205 iteratii    timp de 0.004262 s

       Floyd Warshall

493831 iteratii    timp de 0.138204 s

Grad Dens

       Djikstra

1304 iteratii    timp de 0.006281 s

       Floyd Warshall

579264 iteratii    timp de 0.139587 s
```

Tabelul cu datele de ieșire :

Dijkstra		**Iterații**	**Timp**
Rar	100	312	0.000992
	200	730	0.004376
	300	1123	0.006678
Mediu	100	330	0.000996
	200	772	0.004013
	300	1205	0.004262
Dens	100	393	0.000996
	200	826	0.001173
	300	1304	0.006281

Floyd-Warshall		Iterații	Timp
Rar	100	37512	0.008067
	200	177535	0.106664
	300	439774	0.200946
Mediu	100	41523	0.008541
	200	204963	0.100463
	300	493831	0.138204
Dens	100	49838	0.007119
	200	240781	0.053051
	300	579264	0.139587

Concluzia :

Grafurile sunt folosite ca o conexiune între obiecte, oameni sau entități, iar algoritmul lui Dijkstra găsește cea mai scurtă distanță dintre două puncte dintr-un graf. Algoritmul lui Dijkstra o importanță în lumea reală, fiind cel mai recomandat.

Am studiat algoritmul Floyd Warshall, fiind diferit de algoritmul lui Dijkstra pentru că găsește calea cea mai scurtă între toate nodurile dintr-un graf ponderat. Am studiat algoritmul Floyd Warshall împreună cu un exemplu care explică algoritmul în detaliu. Am observat complexitatea timpului pentru a rula algoritmul pe orice graf ponderat. În cele din urmă, am înțeles aplicarea algoritmului Floyd Warshall care ne poate ajuta să-l aplicăm în viața reală.

Am analizat acești doi algoritmi în baza timpului de execuție și numărului de iterații. După care am observat că algoritmul Floyd-Warshall are o complexitate spațială pătratică și temporală cubică, ambele demonstrând a fi mai mari decât a algoritmul Dijkstra ce are o complexitate liniară – spațială și pătratică – temporală.

Așa cum Dijkstra este un algoritm greedy și determină drumurile din nodul sursă în celelalte noduri, Floyd-Warshall este un algoritm de programare dinamică și determină drumul în toate perechile de noduri ale grafului.