

**Universitatea Tehnică a Moldovei**

**DETERMINAREA RUTELOR OPTIME ÎNTR-UN SISTEM GPS**

**DETERMINATION OF OPTIMAL ROUTES IN A GPS SYSTEM**

|  |  |
| --- | --- |
| **Student:** | **gr. TI-221/TI-231,**  **Mardari Sandu**  **Bagrin Dan** |
| **Coordonator:** | **Andrievschi-Bagrin Veronica,**  [**Cătălin Coșeru**](https://www.facebook.com/catalin.coseru.56) |

**Chişinău, 2024**

**Cuprins**

ABREVIERI  [3](#_bookmark1)

INTRODUCERE  [4](#_bookmark1)

1 Analiza domeniului de studiu  [4](#_bookmark1)

1.1 [Definirea problemei și obiectivele proiectului 5](#_bookmark3)

[1.2 Contextul și motivația dezvoltării unui sistem GPS 6](#_bookmark7)

2 METODE ȘI ALGORITMI UTILIZAȚI  [13](#_bookmark8)

2.1 [Algoritmul Dijkstra 13](#_bookmark9)

2.2 [Algoritmul Floyd-Warshall 18](#_bookmark10)

2.3 [Compararea algoritmilor în contextul rutelor GPS 20](#_bookmark11)

3 IMPLEMENTAREA ȘI TESTAREA SISTEMULUI GPS  [22](#_bookmark11)

3.1 [Reprezentarea graica a grafului 23](#_bookmark11)

3.1 [Descrierea implementării software 24](#_bookmark11)

3.2 [Testarea performanței algoritmilor 25](#_bookmark12)

CONCLUZIE  [26](#_bookmark12)

ANEXE  [27](#_bookmark12)

**ABREVIERI**

**GPS -** (Global Positioning System) este un sistem global de navigație prin satelit care permite determinarea poziției geografice oriunde pe suprafața Pământului, cu ajutorul semnalelor transmise de o rețea de sateliți. Acesta furnizează informații despre locație, direcție, viteză și timp, fiind utilizat pe scară largă în navigație, geolocalizare, transport, aplicații mobile, agricultură de precizie și multe alte domenii.

**Software -** reprezintă ansamblul de programe, aplicații și date care rulează pe un sistem informatic pentru a îndeplini diverse funcții.

**ID (identificator)** - reprezintă un identificator unic utilizat pentru a distinge obiecte, înregistrări, elemente sau entități. Este un atribut, o valoare sau un simbol care identifică în mod unic un obiect într-un anumit sistem sau context.

**Distance (distanța)** - este o măsură a spațiului dintre două puncte, obiecte sau locații, exprimată de obicei în unități de lungime precum metri, kilometri, mile etc.

**INTRODUCERE**

Într-o lume în care mobilitatea și eficiența timpului sunt esențiale, tehnologia [GPS](#Lista) a devenit un pilon central în facilitarea deplasărilor zilnice, fie pentru călătorii personale, fie pentru livrări comerciale și transport public. Sisteme moderne de navigație GPS, echipate cu algoritmi avansați, permit utilizatorilor să identifice rapid și eficient cele mai bune rute între locațiile dorite, oferind o optimizare a timpului și a distanței parcurse.  
 Scopul central al proiectului constă în dezvoltarea unei aplicații GPS practice, capabile să sugereze rute optime pe baza locației de plecare și a destinației alese, utilizând algoritmi de calcul avansați, precum Dijkstra și Floyd-Warshall. Astfel, utilizatorii vor putea planifica traseele eficient, iar aplicația va oferi alternative adaptate nevoilor fiecărui utilizator, ținând cont de factori ca distanța și timpul de călătorie. Performanța tehnologică actuală, combinată cu puterea algoritmilor specializați, oferă posibilitatea unui sistem de navigație precis, flexibil și capabil să răspundă provocărilor mobilității moderne.  
 Lucrarea este structurată în trei capitole, fiecare abordând o componentă esențială a proiectului. În primul capitol, va fi analizat rolul și importanța sistemelor GPS moderne, cu accent pe impactul lor asupra eficienței și confortului în navigare. Vom explora funcționalitățile sistemelor actuale și relevanța optimizării în contextul unei mobilități urbane în continuă dezvoltare.  
 Cel de-al doilea capitol se concentrează asupra algoritmilor Dijkstra și Floyd-Warshall, explicând rațiunea alegerii acestora și detaliind modul lor de funcționare în cadrul aplicației. Fiecare algoritm va fi evaluat și adaptat pentru a răspunde cerințelor specifice ale navigației GPS, asigurând că utilizatorii vor beneficia de cele mai rapide și precise rute posibile.  
 În capitolul final, vom aborda aspectele practice ale implementării algoritmilor și interfaței aplicației. Această secțiune va include o descriere amănunțită a hărții virtuale a rețelei de drumuri, precum și modul în care utilizatorii pot interacționa cu aplicația pentru a-și planifica traseele. Astfel, ne propunem să oferim nu doar o aplicație de navigație performantă, ci și o experiență de utilizare intuitivă și accesibilă, pregătită să răspundă cerințelor diverse ale utilizatorilor de azi.

Astfel, proiectul nostru are ca obiectiv principal oferirea unei soluții de navigare performantă, care să integreze o experiență intuitivă și accesibilă, capabilă să răspundă cerințelor diverse ale utilizatorilor de azi. Într-o lume unde traficul și aglomerația reprezintă provocări constante, determinarea rutelor optime devine o necesitate, iar proiectul nostru reprezintă un efort inovator în direcția utilizării algoritmilor avansați pentru a facilita o navigare eficientă și rapidă. Această soluție contribuie la optimizarea traseelor și la reducerea timpului de călătorie, oferind o abordare inteligentă pentru îmbunătățirea mobilității în spațiile urbane și extraurbane.

1. **ANALIZA DOMENIULUI DE STUDIU**

Tehnologia de navigație a devenit o componentă esențială a vieții moderne, având un rol decisiv atât în activitățile de zi cu zi ale utilizatorilor obișnuiți, cât și în gestionarea logistică a operațiunilor din diverse industrii. Odată cu creșterea complexității rețelelor de drumuri și cu nevoia de mobilitate eficientă, cerințele pentru sistemele de navigație s-au diversificat. Utilizatorii nu mai caută doar rute simple, ci trasee optimizate care să le minimizeze timpul, să le economisească resursele și să le ofere o experiență de navigație sigură și intuitivă. Aceste cerințe au impulsionat dezvoltarea algoritmilor avansați care pot gestiona scenarii de navigație complexe și variate. În acest capitol vom analiza domeniul de studiu, identificând nevoile actuale de navigație și justificând implementarea unui sistem GPS avansat bazat pe algoritmi de calcul optimizatori precum Dijkstra și Floyd-Warshall.   
 Sistemele de navigație tradiționale au fost concepute pentru a răspunde nevoilor elementare, cum ar fi deplasările între două locații cunoscute, dar, pe măsură ce rețelele de drumuri s-au extins, au apărut și provocări mai complexe. În mediul actual, utilizatorii trebuie să ia în considerare mai mulți factori atunci când își aleg traseul, inclusiv condițiile de trafic, distanțele variabile și necesitatea de a accesa mai multe puncte de interes într-o singură deplasare. Acest lucru a dus la creșterea cererii pentru sisteme de navigație inteligente, capabile să ofere soluții rapide, precise și adaptabile la specificul fiecărui utilizator. Un sistem de navigație modern nu doar ghidează, ci și optimizează, adaptând traseele la o serie de condiții și preferințe specifice.   
 Pentru a răspunde acestor cerințe, este necesară integrarea unor algoritmi avansați care să fie capabili să gestioneze complexitatea tot mai mare a rețelelor de drumuri. Algoritmii Dijkstra și Floyd-Warshall, utilizați în cadrul acestui proiect, sunt cunoscuți pentru performanțele lor în determinarea celor mai scurte trasee și permit implementarea unui sistem GPS eficient. Algoritmul Dijkstra este specializat în găsirea celui mai scurt drum între două locații într-un graf, fiind deosebit de util în navigațiile punct-la-punct, unde scopul este optimizarea timpului și a distanței. Floyd-Warshall, pe de altă parte, este un algoritm care identifică toate rutele posibile într-un graf și permite găsirea celor mai bune trasee între orice perechi de puncte. Aceasta este o funcție esențială într-un sistem de navigație care trebuie să fie capabil să ofere trasee eficiente între locații multiple și să se adapteze la cerințele fiecărui utilizator.   
 Un alt aspect esențial în analiza domeniului de studiu este dezvoltarea unei interfețe intuitive, care să asigure o experiență de utilizare fluentă. O interfață bine concepută nu doar facilitează introducerea și vizualizarea locațiilor, dar face și navigația mai clară și mai ușor de gestionat. Utilizatorii au nevoie de o interfață care să le permită accesul rapid la setările de optimizare, cum ar fi selecția criteriilor (distanță minimă, timp minim) și posibilitatea de a salva și revizui traseele preferate. De asemenea, vizualizarea rutei pe o hartă virtuală interactivă îmbunătățește orientarea și permite urmărirea mai precisă a traseului sugerat.

Pe lângă beneficiile imediate pentru utilizatori, un sistem de navigație care utilizează algoritmi de optimizare are și un impact mai larg asupra sustenabilității și mobilității durabile. Reducerea timpului petrecut pe drumuri și minimizarea distanțelor ajută la reducerea amprentei de carbon și a consumului de resurse, contribuind astfel la inițiativele globale de protejare a mediului. Prin optimizarea traseelor, sistemul GPS propus în acest proiect contribuie indirect la reducerea poluării și la o utilizare mai eficientă a infrastructurii existente. Aceste beneficii reflectă nevoia de soluții sustenabile și demonstrează că, prin tehnologie, se pot aduce îmbunătățiri tangibile pentru mediu și pentru calitatea vieții utilizatorilor.  
 Astfel, acest proiect răspunde unei serii de provocări și cerințe actuale din domeniul navigației, îmbinând eficiența algoritmică cu securitatea datelor și cu o interfață prietenoasă și intuitivă. Analiza de față evidențiază importanța unui astfel de sistem, care nu doar simplifică navigația, ci și optimizează experiența utilizatorului, oferindu-i instrumentele necesare pentru a naviga rapid și sigur.

**1.1 Definirea problemei și obiectivele proiectului**

Proiectul de față abordează problema fundamentală a optimizării traseelor într-o rețea de drumuri complexă, având în vedere diferiți parametri precum distanța, timpul și costurile. În lipsa unor astfel de soluții, utilizatorii pot ajunge să consume mai multe resurse – timp, combustibil și bani – în încercarea de a naviga eficient între locații. Prin urmare, soluțiile moderne de navigație trebuie să fie construite pe algoritmi care pot identifica rapid și precis cele mai bune trasee, ținând cont de condițiile impuse de utilizator.  
 Pentru a îndeplini acest scop, proiectul definește mai multe obiective, fiecare contribuind la crearea unui sistem de navigație funcțional, adaptat și sigur pentru utilizatori. În continuare, sunt detaliate principalele obiective ale proiectului. Implementarea algoritmului Dijkstra pentru a găsi cel mai scurt drum între două puncte date într-un graf. Algoritmul Dijkstra este cunoscut pentru capacitatea sa de a găsi ruta cea mai scurtă între două locații specifice într-o rețea. Acesta identifică rapid traseul cel mai scurt între două puncte, reducând distanțele parcurse. De asemenea, prin selectarea continuă a drumului cu cost minim, algoritmul oferă o soluție practică și eficientă pentru navigare între puncte de interes unice. Implementarea algoritmului Floyd-Warshall pentru a calcula toate cele mai scurte drumuri între toate perechile de noduri dintr-un graf. Spre deosebire de Dijkstra, algoritmul Floyd-Warshall permite găsirea celor mai scurte rute între toate locațiile din rețea simultan. Acest obiectiv este important în cazul scenariilor complexe, în care utilizatorul trebuie să gestioneze trasee care includ multiple destinații. Astfel, Floyd-Warshall asigură flexibilitatea navigării între mai multe puncte într-un singur calcul, contribuind la o experiență de utilizare eficientă și adaptabilă.

Crearea unei interfețe grafice intuitive pentru introducerea nodurilor, locațiilor și arcelor. O interfață clară este esențială pentru o experiență de navigație fluidă. Obiectivul este de a dezvolta o interfață accesibilă, care să permită utilizatorilor să introducă locațiile, nodurile și conexiunile acestora într-un mod intuitiv. Astfel, utilizatorii pot crea rute personalizate și își pot vizualiza opțiunile de navigație într-un mod ușor de înțeles. Integrarea unei metode de afișare vizuală a rezultatelor și a rutei optime pe o hartă virtuală. Prezentarea traseului pe o hartă virtuală face navigarea mai intuitivă și mai accesibilă. Prin acest obiectiv, utilizatorii vor putea vedea traseul generat de aplicație într-un format vizual, ceea ce îi va ajuta să urmeze ruta cu ușurință și să înțeleagă mai bine opțiunile de navigație disponibile. Testarea și optimizarea performanței algoritmilor în funcție de dimensiunea și complexitatea rețelei. Algoritmii propuși trebuie să fie rapizi și eficienți, indiferent de complexitatea rețelei de drumuri. Prin testarea performanței acestora și optimizarea lor, aplicația va putea calcula rute precise într-un timp scurt, adaptându-se la dimensiunea și caracteristicile rețelelor de drumuri. Această optimizare va asigura o experiență fluentă pentru utilizatori, indiferent de complexitatea traseului. Oferirea de opțiuni pentru ajustarea criteriilor de optimizare. Utilizatorii vor putea personaliza navigația în funcție de criterii specifice, cum ar fi distanța minimă, timpul minim sau costul minim. Prin acest obiectiv, aplicația va permite o navigare flexibilă, adaptată preferințelor individuale, oferind soluții de traseu personalizate în funcție de nevoile fiecărui utilizator. Prin atingerea acestor obiective, proiectul va dezvolta un sistem de navigație adaptat cerințelor moderne, oferind o experiență eficientă, sigură și ușor de utilizat.

**1.2** **Contextul și motivația dezvoltării unui sistem GPS**

Dezvoltarea unui sistem GPS optimizat pentru calculul rutelor scurte și eficiente între locații multiple reprezintă o necesitate în contextul actual, marcat de cerințe crescânde de mobilitate și eficiență. În acest proiect, algoritmii Dijkstra și Floyd-Warshall sunt utilizați pentru a determina rutele optime într-o rețea de drumuri, având ca scop nu doar simplificarea navigației, ci și oferirea unei experiențe personalizate și sigure pentru utilizatori. Întrucât navigația nu mai este doar un instrument simplu de orientare, ci un sistem complex adaptat cerințelor moderne, proiectul își propune să depășească limitările clasice ale soluțiilor GPS, permițând utilizatorilor să acceseze trasee optimizate și personalizate conform preferințelor lor.   
 Un aspect central al acestui proiect este integrarea algoritmului Dijkstra, recunoscut pentru capacitatea sa de a calcula cel mai scurt drum între două puncte într-un graf. Acest algoritm este util în scenariile în care utilizatorii au nevoie de o rută rapidă între locații specifice, identificând traseul optim prin analiza distanțelor și a costurilor asociate fiecărei etape. Pe de altă parte, algoritmul Floyd-Warshall este esențial în cazurile în care este necesară o analiză complexă a tuturor rutelor posibile între fiecare pereche de puncte într-o rețea. Această capacitate de a găsi multiple trasee între diverse locații aduce un avantaj major proiectului, permițând utilizatorilor să navigheze între mai multe puncte de interes fără a fi constrânși de un traseu prestabilit.  
 Interfața utilizatorului reprezintă un alt element cheie în motivarea acestui proiect, având rolul de a face experiența de navigație intuitivă și accesibilă. Prin integrarea unei hărți virtuale interactive, utilizatorii pot vizualiza rutele recomandate într-un mod clar și detaliat, obținând o imagine de ansamblu asupra traseului și având posibilitatea de a urmări progresul în timp real. În plus, interfața trebuie să permită introducerea ușoară a locațiilor, ajustarea criteriilor de optimizare (de exemplu, distanță minimă sau timp minim) și vizualizarea detaliilor traseului calculat. Această abordare contribuie la simplificarea procesului de planificare a rutelor, oferind utilizatorilor control deplin asupra traseelor selectate și opțiunilor de optimizare.  
 Un alt motiv esențial pentru dezvoltarea acestui proiect este integrarea unor măsuri avansate de protecție a datelor, având în vedere că informațiile despre locație și traseele preferate sunt date personale sensibile. Sistemul GPS propus va implementa tehnologii de criptare și măsuri de limitare a accesului la date, protejând astfel confidențialitatea utilizatorilor și asigurându-le un nivel ridicat de securitate. Întrucât utilizatorii depind din ce în ce mai mult de aplicațiile de navigație pentru deplasările lor, este esențial ca informațiile despre locațiile vizitate și traseele parcurse să fie protejate de accesul neautorizat. Prin implementarea acestor măsuri, proiectul va contribui la crearea unui mediu digital sigur, în care utilizatorii pot naviga fără riscuri privind securitatea datelor.  
 Flexibilitatea și adaptabilitatea traseelor sunt alte aspecte motivante pentru acest proiect. Sistemul GPS propus va oferi utilizatorilor posibilitatea de a ajusta criteriile de optimizare în funcție de preferințele și nevoile lor specifice. Spre exemplu, un utilizator poate opta pentru minimizarea distanței, în timp ce altul poate prefera traseul cel mai rapid sau o combinație de criterii. Acest nivel de personalizare este rar întâlnit în soluțiile GPS convenționale și contribuie la crearea unei experiențe de navigație personalizate, adaptate fiecărui utilizator în parte. Capacitatea de a ajusta traseul în funcție de aceste criterii le permite utilizatorilor să utilizeze aplicația pentru o varietate de scopuri, fie că este vorba despre deplasări cotidiene sau despre trasee complexe care necesită opriri multiple.  
 Un alt element de motivare pentru dezvoltarea acestui sistem GPS este importanța unei utilizări eficiente a resurselor și optimizarea rutei în funcție de necesitățile de deplasare. Algoritmii Dijkstra și Floyd-Warshall, prin funcționalitățile lor avansate de calcul, contribuie la realizarea unui traseu optimizat, care să permită reducerea distanței și a timpului necesar parcurgerii rutei. Astfel, utilizatorii vor beneficia de un sistem GPS performant, capabil să le ofere soluții eficiente, indiferent de complexitatea rețelei de drumuri. Această abordare face ca sistemul să fie potrivit atât pentru utilizatorii individuali, cât și pentru organizațiile sau companiile care au nevoie de un instrument de navigație eficient și adaptabil.

**1.3 Integrarea funcționalităților principale în aplicație**

Pentru realizarea funcționalităților principale ale aplicației, s-a utilizat limbajul de programare Java împreună cu biblioteca JavaFX, oferind un cadru solid pentru dezvoltarea unei aplicații vizuale interactive. Acest subcapitol detaliază modul în care aplicația permite utilizatorilor să interacționeze cu sistemul, să definească un graf de drumuri, să adauge conexiuni între noduri și să aplice algoritmi pentru determinarea rutelor optime. Abordarea este una modulară, combinând elemente vizuale, logice și de reprezentare grafică.Structura aplicației este gândită astfel încât să faciliteze utilizarea intuitivă, fiecare componentă având un rol bine definit. Utilizatorul poate interacționa cu aplicația printr-o interfață grafică simplă și clară, care permite introducerea datelor despre noduri și muchii. După colectarea datelor, aplicația aplică algoritmii selectați (Dijkstra sau Floyd-Warshall) pentru a calcula distanțele minime dintre punctele specificate. Interfața aplicației este centralizată într-o fereastră principală creată cu ajutorul clasei Stage din JavaFX. Aceasta include componente precum câmpuri de text (TextField), butoane pentru diverse acțiuni (Button) și un panou grafic (Canvas) utilizat pentru vizualizarea grafului. Prin integrarea acestor elemente, utilizatorii pot introduce informații, configura graful și analiza rezultatele rutelor calculate. Aplicația integrează două aspecte principale: reprezentarea datelor despre rețeaua de drumuri și aplicarea algoritmilor de rutare. Graful este modelat ca o colecție de noduri, fiecare având conexiuni către alte noduri prin muchii cu ponderi specifice. Această reprezentare simplifică procesarea datelor și permite aplicarea eficientă a algoritmilor. Pentru a ilustra mai bine funcționalitatea aplicației, detaliem componentele sale principale și modul în care acestea interacționează.Graful este o componentă centrală a aplicației, reprezentând rețeaua de drumuri. Structura acestuia este gândită astfel încât să permită adăugarea dinamică a nodurilor și a conexiunilor dintre ele. Utilizatorul introduce numărul de noduri al rețelei printr-un câmp text, iar aplicația creează o structură inițială a grafului.

Cod pentru inițializarea grafului.

*private void initializeGraph(int nodes) {*

*graph.clear();*

*for (int i = 0; i < nodes; i++) {*

*graph.add(new HashMap<>());*

*}*

*}*Această metodă definește un graf gol, unde fiecare nod este reprezentat printr-o hartă de vecini și ponderi asociate. Nodurile sunt numerotate, iar utilizatorul poate adăuga muchii prin specificarea perechilor de noduri conectate și a costului asociat fiecărei conexiuni.  
 Componenta principală a interfeței grafice este formată dintr-o combinație de câmpuri de text, butoane și un panou grafic pentru vizualizarea grafului. Prin intermediul câmpurilor de text, utilizatorul introduce informații despre noduri, muchii și ponderi. Butoanele declanșează acțiuni precum inițializarea grafului, adăugarea conexiunilor și rularea algoritmilor de rutare.

Exemplu de buton pentru crearea grafului:

*TextField nodeCountField = new TextField();*

*nodeCountField.setPromptText("Introduceti numarul nodurilor: ");*

*Button createGraphButton = new Button("Creati graful");*

*createGraphButton.setOnAction(e -> {*

*try {*

*int nodeCount = Integer.parseInt(nodeCountField.getText());*

*initializeGraph(nodeCount);*

*drawGraph();*

*} catch (NumberFormatException ex) {*

*System.out.println("Error: Introduceti numere/cifre.");*

*}*

*});*

Acest fragment de cod asociază unui buton funcționalitatea de a inițializa structura grafică a rețelei, asigurând o interacțiune fluentă pentru utilizator. Dacă utilizatorul introduce date incorecte, aplicația emite mesaje de eroare pentru ghidare.   
 Vizualizarea grafului pe panoul grafic (Canvas) este realizată prin trasarea nodurilor și a muchiilor care le conectează. Nodurile sunt dispuse circular pentru claritate vizuală, iar fiecare muchie este desenată între punctele corespunzătoare.

Cod pentru desenarea grafului.

*private void drawGraph() {*

*GraphicsContext gc = canvas.getGraphicsContext2D();*

*gc.clearRect(0, 0, canvas.getWidth(), canvas.getHeight());*

*int nodeCount = graph.size();*

*double radius = 0.5;*

*double centerX = canvas.getWidth() / 2;*

*double centerY = canvas.getHeight() / 2;*

*int maxWeight = 0;*

*for (int i = 0; i < nodeCount; i++) {*

*for (Map.Entry<Integer, Integer> entry : graph.get(i).entrySet()) {*

*maxWeight = Math.max(maxWeight, entry.getValue());*

*}*

*}*

*for (int i = 0; i < nodeCount; i++) {*

*double angle = 2 \* Math.PI \* i / nodeCount;*

*double nodeRadius = 100;*

*double x = centerX + Math.cos(angle) \* nodeRadius;*

*double y = centerY + Math.sin(angle) \* nodeRadius;*

*gc.setFill(Color.BLACK);*

*gc.fillOval(x - radius, y - radius, radius \* 2, radius \* 2);*

*gc.fillText(String.valueOf(i + 1), x, y);*

*for (Map.Entry<Integer, Integer> entry : graph.get(i).entrySet()) {*

*int neighbor = entry.getKey();*

*int weight = entry.getValue();*

*double neighborAngle = 2 \* Math.PI \* neighbor / nodeCount;*

*double neighborX = centerX + Math.cos(neighborAngle) \* nodeRadius;*

*double neighborY = centerY + Math.sin(neighborAngle) \* nodeRadius;*

*gc.strokeLine(x, y, neighborX, neighborY);*

*double midX = (x + neighborX) / 2;*

*double midY = (y + neighborY) / 2;*

*double offsetX = 10;*

*double offsetY = -5;*

*gc.setFill(Color.RED);*

*gc.fillText(String.valueOf(weight), midX + offsetX, midY + offsetY);*

*}*

*}*

*}*

Fiecare nod este reprezentat printr-un cerc, iar etichetele text indică identificatorii lor. Muchiile sunt trasate între coordonatele nodurilor conectate, utilizând ponderile corespunzătoare.  
 Aplicația implementează doi algoritmi esențiali: Dijkstra și Floyd-Warshall. Algoritmul Dijkstra calculează distanțele minime de la un nod sursă la toate celelalte noduri, în timp ce Floyd-Warshall determină matricea distanțelor minime între toate perechile de noduri.

Exemplu de rulare a algoritmului Dijkstra.

*Button runDijkstraButton = new Button("Go (Dijkstra)");*

*runDijkstraButton.setOnAction(e -> {*

*int startNode = Integer.parseInt(startNodeField.getText()) - 1;*

*int[] distances = DijkstraClass.runDijkstra(startNode, graph);*

*displayResultsDijkstra(distances, startNode);*

*});*

Rezultatele algoritmului sunt afișate într-o zonă text dedicată, iar utilizatorul poate vedea distanțele minime calculate pentru fiecare nod.

Cod pentru afișarea rezultatelor Dijkstra.

*private void displayResultsDijkstra(int[] distances, int startNode) {*

*StringBuilder sb = new StringBuilder();*

*sb.append("Distanța minimă (Dijkstra) de la nodul ").append(startNode + 1).append(":\n");*

*for (int i = 0; i < distances.length; i++) {*

*sb.append("Până la nodul ").append(i + 1).append(": ")*

*.append(distances[i] == Integer.MAX\_VALUE ? "∞" : distances[i])*

*.append("\n");*

*}*

*resultArea.setText(sb.toString());*

*}*

Prin aceste funcționalități, aplicația asigură o experiență completă utilizatorului, de la configurarea rețelei până la analiza rezultatelor rutelor.

Exemplu de rulare a algoritmului Floyd-Warshall.

*Button runFloydWarshallButton = new Button("Go (Floyd-Warshall)");*

*runFloydWarshallButton.setOnAction(e -> {*

*int[][] distancesFloyd = floydWarshall.runFloydWarshall();*

*displayResultsFloydWarshall(distancesFloyd);*

*});*

Algoritmul calculează matricea distanțelor minime între toate perechile de noduri. Rezultatele sunt afișate într-o matrice text.

Cod pentru afișarea rezultatelor Floyd-Warshall.

*private void displayResultsFloydWarshall(int[][] distances) {*

*StringBuilder sb = new StringBuilder();*

*sb.append("Matricea distanțelor minime (Floyd-Warshall):\n");*

*for (int[] distance : distances) {*

*for (int i : distance) {*

*sb.append(i == 99999 ? "∞" : i).append("\t"); // Valoarea "99999" este considerată infinit*

*}*

*sb.append("\n"); // Linie nouă pentru fiecare rând din matrice*

*}*

*resultArea.setText(sb.toString()); // Afișează rezultatul în zona text*

*}*

1. **METODE ȘI ALGORITMI UTILIZAȚI**

În vederea determinării celor mai bune rute într-un sistem GPS, proiectul propune utilizarea unor algoritmi avansați de calcul optimizator. Algoritmii selectați au fost analizați și adaptați pentru a răspunde cerințelor specifice navigației, având în vedere complexitatea rețelelor de drumuri și nevoile variate ale utilizatorilor. Acești algoritmi permit nu doar calculul rutelor rapide și eficiente, ci și personalizarea traseelor în funcție de preferințele fiecăruia.  
 Proiectul se axează pe implementarea a două metode principale de optimizare a rutelor, fiecare cu avantaje distincte. Algoritmul Dijkstra este utilizat pentru identificarea celui mai scurt drum între două puncte într-un graf, fiind ideal pentru scenarii în care utilizatorii doresc optimizarea rapidă a unui traseu punct-la-punct. Pe de altă parte, algoritmul Floyd-Warshall permite calcularea celor mai scurte drumuri între toate perechile de puncte dintr-o rețea, oferind o flexibilitate sporită în cazul rutelor complexe care implică multiple locații.   
 Această abordare duală asigură atât viteza în procesare, cât și adaptabilitatea la diverse cerințe de navigație. Algoritmii sunt recunoscuți pentru eficiența lor în optimizarea rutelor și permit implementarea unui sistem care să răspundă nevoilor moderne de mobilitate. În continuare, fiecare metodă utilizată va fi detaliată, subliniind modul său de funcționare și avantajele practice în contextul proiectului.

**2.1 Algoritmul Dijkstra**

Algoritmul lui Dijkstra este una dintre cele mai utilizate metode pentru rezolvarea problemei celui mai scurt drum într-un graf ponderat, cu greutăți non-negative ale muchiilor. Creat de Edsger Dijkstra în 1956, acesta a fost inițial dezvoltat ca o demonstrație a eficienței algoritmice în optimizarea grafurilor și s-a dovedit a fi extrem de valoros în numeroase domenii practice. Esența sa constă în determinarea incrementală a celui mai scurt drum de la un nod sursă către toate celelalte noduri, folosind o combinație de inițializare a distanțelor și procesare iterativă a nodurilor prin intermediul unei cozi de priorități.  
 Algoritmul pornește prin atribuirea unei distanțe de zero nodului sursă și a unei distanțe inițiale de „infinit” pentru toate celelalte noduri. Această abordare simbolizează faptul că la început niciun drum către alte noduri nu este cunoscut, cu excepția sursei. Structura principală a algoritmului implică utilizarea unei cozi de priorități, unde nodurile sunt selectate pentru procesare în funcție de distanța curentă estimată. Fiecare nod este analizat, iar distanțele către vecinii săi sunt actualizate dacă se găsesc rute mai scurte prin nodul curent.  
 Complexitatea algoritmului depinde de implementarea structurii de date pentru coada de priorități. Dacă se utilizează un heap binar, algoritmul poate funcționa cu o complexitate de O((V+E)logV), unde V este numărul de noduri și E numărul de muchii din graf. Aceasta face ca metoda să fie extrem de eficientă în cazul grafurilor sparse, în care numărul de muchii este semnificativ mai mic decât pătratul numărului de noduri (V2). Acest avantaj îl face ideal pentru aplicații reale precum rețelele rutiere, unde multe locații sunt conectate doar cu vecinii lor imediați.  
 Un aspect important al algoritmului este determinismul său, ceea ce înseamnă că acesta oferă întotdeauna o soluție optimă dacă există una. Acest lucru îl diferențiază de algoritmii euristici, cum ar fi A\*, care pot oferi soluții mai rapide, dar care nu garantează întotdeauna un rezultat optim în funcție de implementarea euristicii. Totuși, algoritmul lui Dijkstra are o limitare semnificativă: nu poate gestiona greutăți negative ale muchiilor, deoarece o astfel de situație ar necesita revizuirea repetată a distanțelor, ceea ce contravine principiilor pe care se bazează. În astfel de cazuri, algoritmi precum Bellman-Ford sunt mai potriviți, deși aceștia au o complexitate mai mare.  
 Aplicabilitatea algoritmului lui Dijkstra este vastă și include numeroase domenii. În navigația GPS, algoritmul este folosit pentru a calcula rute optime între locații, ținând cont de distanță sau timp. În rețelele de calculatoare, este utilizat pentru a optimiza rutarea pachetelor de date, calculând cele mai scurte trasee între nodurile unei rețele. De asemenea, este aplicat în logistică pentru optimizarea rutelor de livrare, reducând costurile și timpul necesar transportului. Alte utilizări includ rețelele sociale, unde poate determina distanțele dintre utilizatori, sau sistemele de telecomunicații, pentru optimizarea conexiunilor între dispozitive.  
 Un alt avantaj major al algoritmului este simplitatea implementării sale. Algoritmul poate fi integrat cu ușurință în diverse aplicații [software](#Lista), datorită structurii sale clare și a logicii bazate pe bucle iterative. Această simplitate face ca algoritmul să fie o alegere populară chiar și pentru programatorii începători care lucrează cu probleme de optimizare a rutelor. Totuși, implementarea eficientă necesită utilizarea unor structuri de date adecvate, precum heap-urile binare sau Fibonacci, pentru a menține complexitatea scăzută și a asigura performanțe ridicate în cazul grafurilor mari. În ceea ce privește limitele algoritmului, acestea devin mai evidente în cazul grafurilor dense, unde numărul de muchii este aproape de V2. În astfel de situații, algoritmi precum Floyd-Warshall, care calculează toate perechile de distanțe într-un graf, pot fi mai potriviți. De asemenea, algoritmul este mai puțin eficient în cazul în care problema implică grafuri dinamice, unde greutățile muchiilor se pot schimba frecvent, deoarece fiecare actualizare a greutăților necesită rularea din nou a întregului algoritm. O altă limitare este că algoritmul presupune că toate nodurile sunt accesibile din nodul sursă, ceea ce poate necesita adaptări în cazul grafurilor orientate sau incomplete.

Clasa „DijkstraClass” definește structura generală a algoritmului și a funcționalităților acestuia, având o clasă statică internă denumită „Node”, care este folosită pentru a reprezenta nodurile din graf. Această structură modulară separă clar partea de modelare a nodurilor de logica principală a algoritmului, ceea ce îmbunătățește lizibilitatea și permite reutilizarea mai ușoară a clasei. Clasa internă „Node” este un exemplu simplu, dar puternic, de abstractizare orientată pe obiecte, ceea ce facilitează manipularea și procesarea nodurilor într-o manieră uniformă.

*package com.example.dijkstraproject;*

*import java.util.\*;*

*public class DijkstraClass {*

*static class Node {*

*int id;*

*int distance;*

*Node(int id, int distance) {*

*this.id = id;*

*this.distance = distance;*

*}*

*}*

Clasa Node este esențială deoarece oferă o reprezentare concisă a unui nod din graf, incluzând identificatorul nodului ([id](#Lista)) și valoarea distanței curente asociate acestuia ([distance](#Lista)). Această implementare permite instanțierea rapidă a obiectelor de tip nod, care pot fi manipulate eficient în coada de priorități. Constructorul este simplu și permite inițializarea atributelor id și distance în momentul creării nodului. Astfel, nodurile devin elementele de bază ale algoritmului, fiind folosite pentru a determina ordinea în care sunt procesate în cadrul cozii de priorități.

Metoda statică runDijkstra implementează logica propriu-zisă a algoritmului lui Dijkstra. Aceasta primește ca parametri nodul de start, care reprezintă punctul de plecare al algoritmului, și o reprezentare a grafului sub forma unei liste de hărți. Această reprezentare este practică, deoarece permite asocierea nodurilor cu greutățile arcelor, păstrând astfel toate informațiile necesare despre graf într-un format compact și eficient.

*public static int[] runDijkstra(int start, List<Map<Integer, Integer>> graph) {*

*int n = graph.size();*

*int[] distances = new int[n];*

*Arrays.fill(distances, Integer.MAX\_VALUE);*

*distances[start] = 0;*

Această parte a metodei realizează inițializarea vectorului distances, care stochează cele mai scurte distanțe calculate de la nodul de start la toate celelalte noduri. La început, toate nodurile sunt marcate ca inaccesibile prin atribuirea valorii maxime posibile (Integer.MAX\_VALUE). Această valoare simbolizează „infinitul” și este utilizată în algoritmi de grafuri pentru a indica faptul că nu există încă o cale cunoscută către acel nod. Nodul de start este tratat în mod special, iar distanța către acesta este setată la zero, indicând că nu există niciun cost asociat deplasării către el însuși.  
 O coadă de priorități este inițializată pentru a gestiona procesarea nodurilor în funcție de distanțele lor estimate. Această coadă folosește comparatorul Comparator.comparingInt(node -> node.distance) pentru a asigura ordonarea automată a nodurilor pe baza distanțelor. Nodul de start este adăugat în coadă cu distanța zero, ceea ce semnalizează algoritmului că acesta trebuie procesat primul. În paralel, un set este inițializat pentru a urmări nodurile deja procesate, prevenind astfel procesarea redundantă a acelorași noduri.

*PriorityQueue<Node> priorityQueue = new PriorityQueue<>(Comparator.comparingInt(node -> node.distance));*

*priorityQueue.add(new Node(start, 0));*

*Set<Integer> processed = new HashSet<>();*

Această abordare cu coada de priorități este esențială pentru eficiența algoritmului. În loc să parcurgă toate nodurile în mod secvențial, algoritmul selectează întotdeauna nodul cu cea mai mică distanță estimată, ceea ce reduce semnificativ numărul de comparații și îmbunătățește performanța generală.  
 În cadrul buclei principale a algoritmului, nodurile sunt extrase din coada de priorități în ordine crescătoare a distanței lor estimate. Nodurile deja procesate sunt ignorate, datorită verificării cu setul processed. Aceasta previne procesările redundante și permite algoritmului să se concentreze pe nodurile care necesită procesare.

*while (!priorityQueue.isEmpty()) {*

*Node node = priorityQueue.poll();*

*int u = node.id;*

*if (!processed.add(u)) {*

*continue;*

*}*

Extracția nodului cu metoda poll este un proces rapid, care profită de proprietățile cozii de priorități pentru a selecta nodul cu distanța minimă. Adăugarea nodurilor în setul processed asigură că fiecare nod este procesat o singură dată, ceea ce contribuie la corectitudinea și eficiența algoritmului.

Pentru fiecare vecin al nodului curent, algoritmul verifică dacă distanța calculată prin nodul curent este mai mică decât cea deja cunoscută. Dacă această condiție este îndeplinită, distanța este actualizată, iar vecinul este adăugat din nou în coada de priorități cu distanța sa actualizată. Acest proces garantează că toate rutele potențial mai scurte sunt explorate în mod eficient.

*for (Map.Entry<Integer, Integer> entry : graph.get(u).entrySet()) {*

*int v = entry.getKey();*

*int weight = entry.getValue();*

*if (distances[u] + weight < distances[v]) {*

*distances[v] = distances[u] + weight;*

*priorityQueue.add(new Node(v, distances[v]));*

*}*

*}*

*}*

Această parte a algoritmului implementează logica fundamentală a actualizării distanțelor. Utilizarea unui graf reprezentat ca o listă de hărți permite acces rapid la vecinii unui nod și la greutățile asociate arcelor. Actualizarea distanțelor și adăugarea nodurilor în coadă în funcție de noile distanțe optimizează procesul și asigură calcularea corectă a celor mai scurte distanțe.  
 La finalul algoritmului, vectorul de distanțe este returnat. Acesta conține cele mai scurte distanțe de la nodul de start la toate celelalte noduri din graf. Rezultatul poate fi utilizat în diverse aplicații, cum ar fi planificarea rutelor sau optimizarea navigației. Această ultimă secțiune finalizează algoritmul, oferind utilizatorului un rezultat precis și util sub forma unui vector. Modularitatea metodei runDijkstra și claritatea logicii sale interne facilitează integrarea acesteia în sisteme mai complexe.

**2.2 Algoritmul Floyd-Warshall**

Algoritmul Floyd-Warshall reprezintă un instrument esențial în teoria grafurilor, fiind utilizat pentru a determina cele mai scurte distanțe între toate perechile de noduri dintr-un graf ponderat. Este un algoritm general aplicabil atât grafurilor orientate, cât și celor neorientate, cu condiția ca ponderile muchiilor să fie nenegative sau să nu conducă la cicluri negative. Se bazează pe programarea dinamică și funcționează prin actualizarea iterativă a unei matrice a distanțelor, calculând trasee mai scurte prin noduri intermediare.  
 Inițial, algoritmul presupune construirea unei matrice de adiacență care să reflecte ponderile muchiilor directe între noduri. Dacă două noduri nu sunt conectate direct, algoritmul le atribuie o valoare infinită, simbolizând lipsa unei conexiuni. Nodurile sunt tratate individual, iar distanțele de la un nod la el însuși sunt setate la zero, fiind valori implicite în orice graf. Aceasta constituie baza pentru procesul iterativ care va urma. Procesul algoritmului implică trei bucle imbricate care examinează toate combinațiile posibile de noduri și verifică dacă trecerea printr-un nod intermediar reduce distanța deja calculată între două noduri. Dacă se găsește un drum mai scurt, matricea distanțelor este actualizată pentru a reflecta noua valoare minimă. Acest mecanism asigură că, după ce fiecare nod este luat în considerare ca intermediar, matricea distanțelor va conține cele mai scurte distanțe între toate perechile de noduri.  
 Complexitatea algoritmului este O(n^3) unde n este numărul de noduri din graf, făcându-l potrivit pentru grafurile de dimensiuni moderate. În cazul grafurilor mari sau sparse, poate deveni mai puțin eficient comparativ cu alte metode, cum ar fi algoritmul Dijkstra, care funcționează mai rapid pe grafuri cu mai puține muchii. Cu toate acestea, avantajul principal al algoritmului Floyd-Warshall constă în faptul că produce simultan soluții pentru toate perechile de noduri, eliminând necesitatea apelării repetate a altor algoritmi.  
 Un aspect semnificativ al algoritmului este abilitatea sa de a detecta cicluri de cost negativ. După completarea tuturor iterațiilor, dacă un element de pe diagonala principală a matricei distanțelor are o valoare negativă, acest lucru indică prezența unui astfel de ciclu. Detectarea ciclurilor negative este valoroasă în analiza rețelelor economice sau a altor sisteme complexe unde astfel de cicluri pot indica inconsistențe sau posibilități de optimizare. În aplicații practice, algoritmul este utilizat în diverse domenii, de la optimizarea traficului și rețelele de telecomunicații, până la analiza fluxurilor financiare. De asemenea, poate fi adaptat pentru a rezolva probleme legate de traseele cele mai scurte în rețelele de transport sau în rețelele rutiere, unde este esențială determinarea unui traseu eficient între mai multe locații.

Totuși, o limitare notabilă a algoritmului o reprezintă consumul ridicat de memorie, mai ales în cazul grafurilor mari. Deoarece stochează o matrice completă a distanțelor, cerințele de memorie cresc exponențial cu dimensiunea grafului. În astfel de situații, optimizările suplimentare sau algoritmii alternativi devin soluții preferabile. Cu toate acestea, simplitatea implementării și robustețea algoritmului îl mențin relevant pentru o gamă largă de aplicații.

Astfel, Floyd-Warshall rămâne unul dintre cele mai apreciate algoritmi datorită versatilității sale și a capacității de a rezolva eficient probleme complexe de grafuri, fiind o componentă esențială în instrumentarul matematicianului sau al inginerului software.

Prima etapă a algoritmului Floyd-Warshall presupune inițializarea unei matrice de distanțe care este o copie a matricei de adiacență a grafului. Dimensiunea matricei este determinată de numărul de noduri din graf (V), fiecare rând reprezentând un nod, iar fiecare coloană fiind o conexiune posibilă către alte noduri. În acest proces, fiecare valoare din matricea originală este transferată în matricea dist folosind metoda System.arraycopy, care copiază eficient datele dintr-o matrice în alta. Acest lucru asigură că distanțele inițiale dintre noduri, definite de graful inițial, sunt păstrate în matricea de lucru. Astfel, dist devine matricea care va fi modificată pe parcursul execuției algoritmului, în timp ce matricea originală a grafului rămâne intactă. Această etapă pregătește structura necesară pentru ca algoritmul să poată analiza și actualiza distanțele dintre noduri în următoarele etape.

*int V = graph.length;*

*int[][] dist = new int[V][V];*

*for (int i = 0; i < V; i++) {*

*System.arraycopy(graph[i], 0, dist[i], 0, V);*

*}*

Aceasta este partea esențială a algoritmului Floyd-Warshall, unde se examinează toate combinațiile posibile de drumuri dintre perechile de noduri (i, j), folosind fiecare nod k ca potențial intermediar. Primul ciclu for alege nodul intermediar k, iar buclele imbricate parcurg toate combinațiile de noduri sursă și destinație (i, j). Condițiile din această secțiune verifică dacă există drumuri valide de la i la k și de la k la j. Dacă aceste drumuri există, algoritmul compară suma distanțelor dist[i][k] + dist[k][j] cu distanța curentă dintre i și j (dist[i][j]). Dacă suma drumurilor intermediare este mai mică, valoarea din matricea dist este actualizată pentru a reflecta noua distanță minimă. Această abordare iterativă permite identificarea treptată a celor mai scurte drumuri dintre noduri, luând în considerare toate combinațiile posibile de rute.

*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] != INF && dist[k][j] != INF && dist[i][k] + dist[k][j] < dist[i][j]) {*

*dist[i][j] = dist[i][k] + dist[k][j];*

*}*

*}*

*}*

*}*

Această etapă este crucială pentru determinarea celei mai eficiente rute între toate perechile de noduri. Procesul permite explorarea tuturor drumurilor posibile, inclusiv a celor care implică mai mulți pași, și identificarea soluțiilor optime.  
 Ultimul pas al algoritmului Floyd-Warshall constă în returnarea matricei dist, care conține cele mai scurte distanțe calculate între toate nodurile grafului. După ce toate ciclurile și verificările au fost finalizate, matricea reflectă rezultatele complete și finale ale algoritmului. În această matrice, fiecare celulă (i, j) conține distanța minimă dintre nodul i și nodul j. Matricea este pregătită pentru a fi utilizată în aplicațiile care necesită analiza distanțelor, cum ar fi optimizarea traficului sau planificarea rutelor.

*return dist;*

Matricea finală poate fi integrată în sisteme mai mari, oferind o soluție completă pentru determinarea drumurilor optime în rețele complexe. Datorită acestei etape de returnare, utilizatorii au acces la rezultatele calculului pentru toate perechile de noduri din graf.

**2.3** **Compararea algoritmilor în contextul rutelor GPS**

Determinarea rutelor optime într-un sistem GPS este o problemă complexă, care necesită alegerea unor algoritmi eficienți și adaptabili la diverse scenarii. În cadrul acestui proiect, algoritmii Dijkstra și Floyd-Warshall au fost selectați datorită performanțelor lor recunoscute în teoria grafurilor și aplicațiile practice. Analiza detaliată a caracteristicilor fiecăruia relevă diferențele semnificative dintre metodele lor de funcționare, aplicabilitatea în diverse contexte și impactul asupra performanței sistemului GPS. Această comparație pune în evidență avantajele și limitările fiecărui algoritm, oferind o bază solidă pentru integrarea lor într-un sistem de navigație performant.  
 Un element central al analizei constă în evaluarea complexității algoritmice. Dijkstra este recunoscut pentru eficiența sa în determinarea celui mai scurt drum într-un graf ponderat, cu greutăți pozitive ale muchiilor. Metoda sa, bazată pe selectarea iterativă a nodului cu distanța minimă, permite optimizarea procesului de calcul pentru grafuri sparse. Utilizarea unei cozi de priorități bazate pe un heap binar reduce semnificativ timpul de procesare, rezultând o complexitate de O((V+E)⋅logV), unde V reprezintă numărul de noduri, iar E numărul de muchii din graf. Această caracteristică face ca Dijkstra să fie ideal pentru rețele rutiere obișnuite, cum ar fi cele din mediul urban, unde nodurile (intersecțiile) sunt conectate doar cu vecinii lor direcți.

În schimb, Floyd-Warshall se remarcă prin capacitatea sa de a determina cele mai scurte distanțe între toate perechile de noduri dintr-un graf. Spre deosebire de Dijkstra, acest algoritm utilizează o abordare bazată pe programare dinamică, actualizând iterativ o matrice a distanțelor. Complexitatea sa,   
 O(V3), îl face mai puțin eficient pentru grafuri sparse, însă oferă avantaje semnificative în cazul grafurilor dense, unde conexiunile între noduri sunt numeroase. De exemplu, într-o rețea rutieră la nivel național, unde fiecare oraș poate avea conexiuni multiple cu orașele din jur, Floyd-Warshall asigură o analiză completă a rețelei și furnizează soluții optime pentru toate perechile de drumuri posibile.   
 Aplicabilitatea algoritmilor diferă în funcție de complexitatea rețelei și de cerințele specifice utilizatorilor. Algoritmul Dijkstra este preferat în aplicațiile punctuale, cum ar fi calculul celei mai rapide rute între locația curentă și o destinație prestabilită. Într-un sistem GPS utilizat de șoferii obișnuiți, acest algoritm oferă răspunsuri rapide, ajustând traseele în timp real în funcție de condițiile de trafic. Această capacitate de a răspunde dinamic cerințelor utilizatorilor este crucială în scenarii precum gestionarea blocajelor rutiere sau evitarea accidentelor. De exemplu, un șofer prins într-un ambuteiaj poate beneficia de recalcularea instantanee a rutei, ceea ce îmbunătățește semnificativ experiența de utilizare.  
 În contrast, Floyd-Warshall este mai potrivit pentru analiza rețelelor complexe și pentru generarea de soluții globale. Acest algoritm este adesea utilizat în aplicații care necesită planificare strategică, cum ar fi logistica sau gestionarea transportului public. De exemplu, într-un sistem destinat companiilor de curierat, unde este necesară optimizarea rutelor pentru livrări multiple, algoritmul poate calcula simultan toate opțiunile disponibile, oferind managerilor de flotă o imagine completă a distanțelor și costurilor asociate. În mod similar, în contextul transportului public, Floyd-Warshall poate fi utilizat pentru a planifica rute eficiente care să includă opriri intermediare și să optimizeze timpul total de călătorie.  
 Fiecare dintre algoritmii analizați are limitări care influențează aplicabilitatea lor în anumite scenarii. Dijkstra, de exemplu, nu poate gestiona greutăți negative ale muchiilor. Deși această situație este rar întâlnită în rețelele rutiere, poate apărea în alte domenii, cum ar fi analiza economică sau rețelele de transport complexe. Această limitare poate fi depășită prin utilizarea algoritmului Bellman-Ford, care, deși mai lent, este capabil să gestioneze astfel de scenarii.  
 Pe lângă avantajele și limitările discutate, o altă dimensiune importantă în evaluarea algoritmilor Dijkstra și Floyd-Warshall este modul în care aceștia îndeplinesc cerințele de scalabilitate, fiabilitate și adaptabilitate în sisteme GPS.

Dijkstra este excelent pentru grafuri mari și sparse datorită complexității sale de O((V+E)⋅logV), mai ales când rețelele includ un număr redus de conexiuni per nod. Scalabilitatea sa este evidentă în aplicațiile care necesită calcule rapide pe rețele extinse, cum ar fi hărțile urbane detaliate. Cu toate acestea, în cazul grafurilor foarte dense, timpul său de execuție poate deveni mai mare comparativ cu Floyd-Warshall, deoarece trebuie să recalculeze distanțele pentru fiecare pereche de noduri în mod separat.

Floyd-Warshall oferă o abordare completă pentru grafuri dense, dar complexitatea cubică (O(V3)) îl face mai puțin scalabil pe măsură ce numărul de noduri crește. Această limitare poate fi problematică pentru rețele uriașe, unde timpul de execuție devine impracticabil.

Dijkstra este fiabil pentru aplicații în care rutele trebuie să fie calculate rapid și cu precizie în timp real. Fiabilitatea sa este susținută de utilizarea cozii de priorități și de faptul că garantează întotdeauna un rezultat optim pentru grafurile ponderate pozitiv.

Floyd-Warshall este extrem de fiabil în grafuri dense, unde oferă soluții complete pentru toate perechile de noduri. Cu toate acestea, în cazul grafurilor sparse, utilizarea sa poate consuma resurse suplimentare fără a aduce beneficii semnificative.

Dijkstra excelează în adaptabilitate datorită posibilității sale de recalculare rapidă a rutelor în funcție de schimbările din rețea (ex. ambuteiaje, lucrări de întreținere). Aceasta îl face ideal pentru aplicații GPS în timp real. Floyd-Warshall, deși mai lent în actualizarea soluțiilor pentru schimbări locale în graf, este extrem de adaptabil în planificarea strategică, permițând includerea mai multor variabile (ex. costuri adiționale, restricții) în analiza rețelei.

Dijkstra este mai simplu de implementat și adaptat pentru aplicații practice cu o singură sursă, necesitând mai puține resurse și configurații inițiale. Floyd-Warshall, deși conceptul de programare dinamică este mai intuitiv, poate implica o implementare mai complexă și consumatoare de memorie, mai ales pentru grafuri mari.

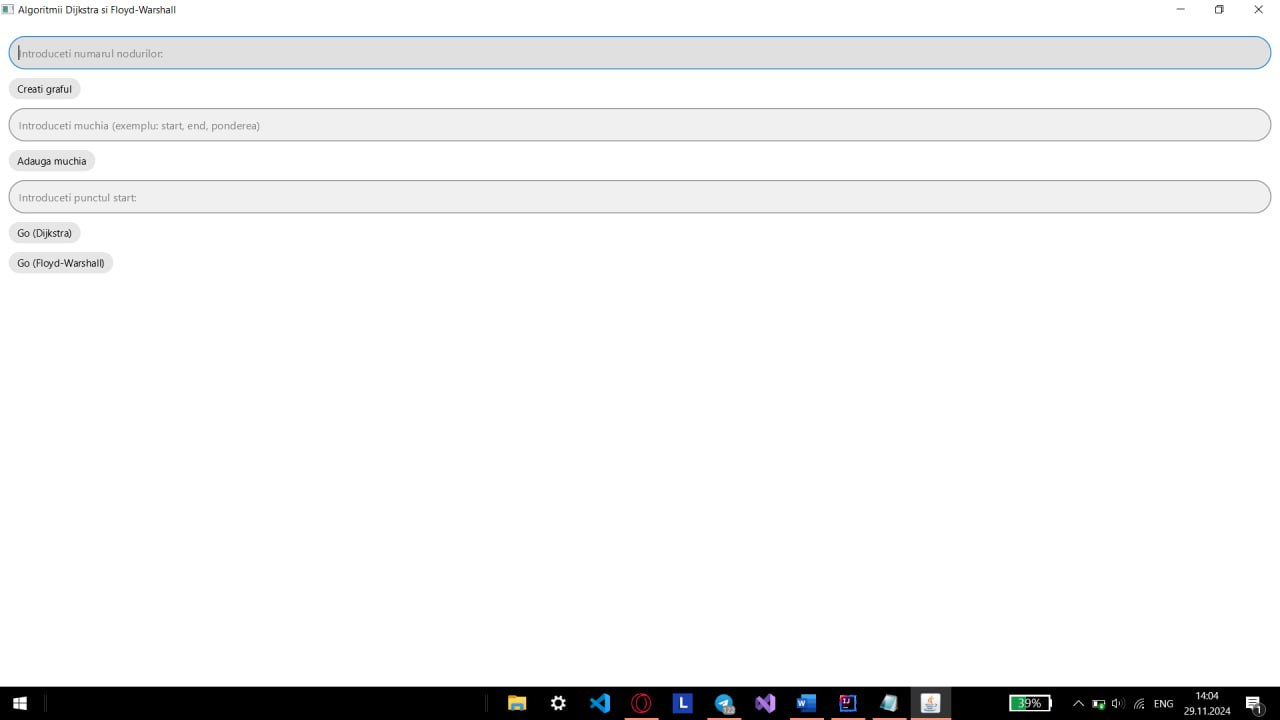
Dijkstra îndeplinește cel mai bine cerințele de eficiență operațională și timp real, fiind adecvat pentru aplicațiile cu o singură sursă de calcul al rutelor optime. Floyd-Warshall excelează în contextul planificării globale și analizei complexe, fiind ideal pentru aplicațiile care necesită vizualizarea completă a distanțelor și costurilor între toate perechile de noduri. Această comparație detaliată subliniază necesitatea de a alege algoritmul în funcție de cerințele specifice ale aplicației, oferind astfel o bază solidă pentru optimizarea performanței sistemului GPS.

**3. IMPLEMENTAREA ȘI TESTAREA SISTEMULUI GPS**

**3.1 Descrierea implementării software**

Aplicația este construită pentru a oferi utilizatorilor posibilitatea de a simula rețele de drumuri și de a calcula cele mai scurte rute între locații. Aceasta utilizează o interfață grafică intuitivă, implementată cu ajutorul bibliotecii JavaFX.

Componente principale sunt zona de introducere a datelor, zona de alegere a algoritmului, si zona de afișare a rezultatelor. Utilizatorii introduc numărul de noduri, muchiile (conexiunile între noduri) și ponderile acestora. Aplicația oferă posibilitatea de a selecta algoritmul de optimizare (Dijkstra sau Floyd-Warshall). Include secțiuni pentru afișarea distanțelor calculate și vizualizarea grafului.



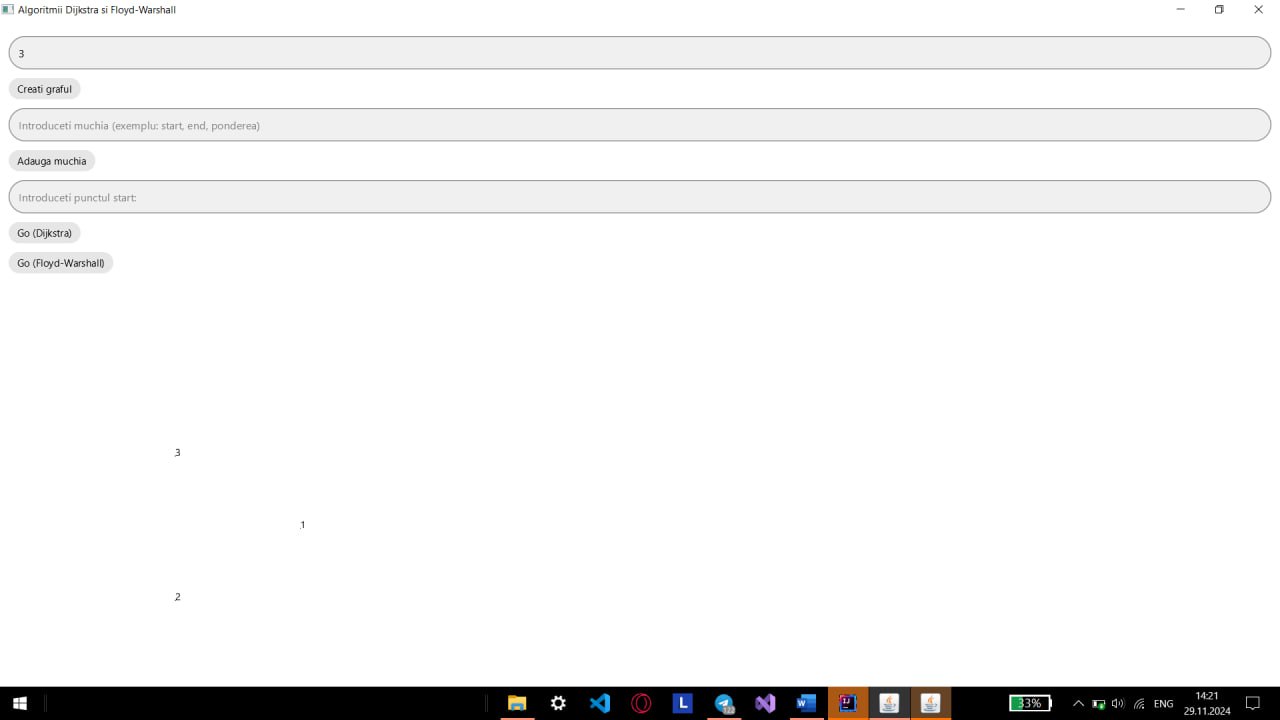
**Figura 3.1 - Interfața principală a aplicației, incluzând câmpurile pentru introducerea datelor, butoanele de control și zona de afișare a rezultatelor**

Avantaje ale implementării. Interfața grafică intuitivă reduce curba de învățare pentru utilizatori, făcând aplicația ușor de utilizat chiar și pentru cei fără cunoștințe tehnice avansate. Alegerea dintre doi algoritmi permite utilizatorilor să rezolve atât probleme punctuale, cât și probleme globale legate de rețele complexe. Structura modulară a aplicației optimizează timpul de procesare și gestionează eficient memoria.

**3.1 Reprezentarea grafică a grafului**

Reprezentarea grafică a grafului este o componentă centrală a aplicației și permite utilizatorilor să înțeleagă structura rețelei de drumuri într-un mod vizual. Graful este afișat pe un panou dedicat din interfața aplicației, unde fiecare nod este dispus geometric, iar conexiunile (muchiile) sunt adăugate dinamic în funcție de algoritmul selectat.

Fluxul de generare a grafului este Poziționarea nodurilor si Adăugarea muchiilor. În stadiul inițial, aplicația afișează doar nodurile grafului. Fiecare nod este reprezentat sub forma unui punct numerotat, dispus circular pentru claritate. Această etapă ajută utilizatorii să verifice structura de bază a grafului. Conexiunile dintre noduri (muchiile) devin vizibile doar după ce utilizatorul selectează și rulează un algoritm (Dijkstra sau Floyd-Warshall). Liniile trasate între noduri reflectă conexiunile calculate de algoritmul ales, iar ponderile (distanțele) sunt afișate lângă muchii.

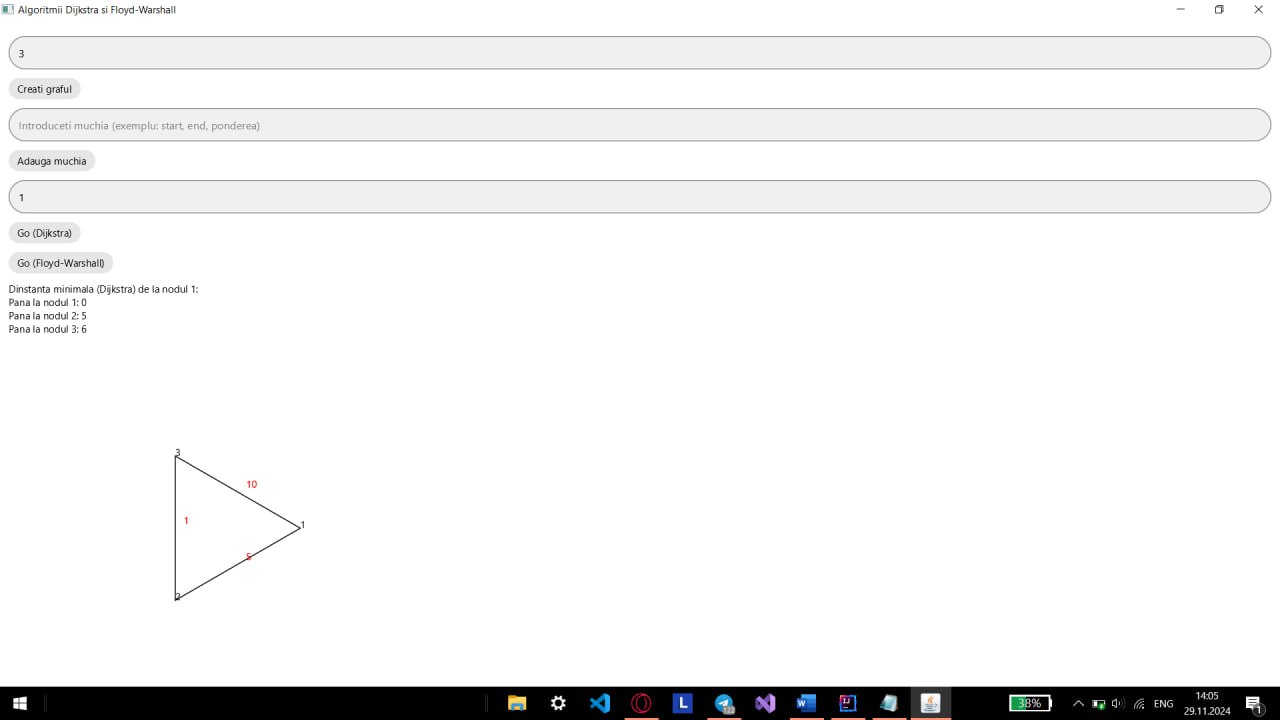


**Figura 3.2 - Poziționarea inițială a nodurilor grafului, fără conexiunile trasate, ilustrând etapa pregătitoare a reprezentării vizuale.**

Muchiile (conexiunile între noduri) sunt generate dinamic doar după ce un algoritm este selectat și rulat. Astfel, reprezentarea finală a grafului, inclusiv liniile și ponderile, este disponibilă abia în secțiunea de testare a algoritmilor.

**3.2 Testarea Algoritmul Dijkstra**

Acest algoritm calculează distanțele minime de la un nod sursă către toate celelalte noduri din graf. Rezultatele sunt afișate sub formă de listă, unde fiecare element indică nodul destinație și distanța minimă calculată.

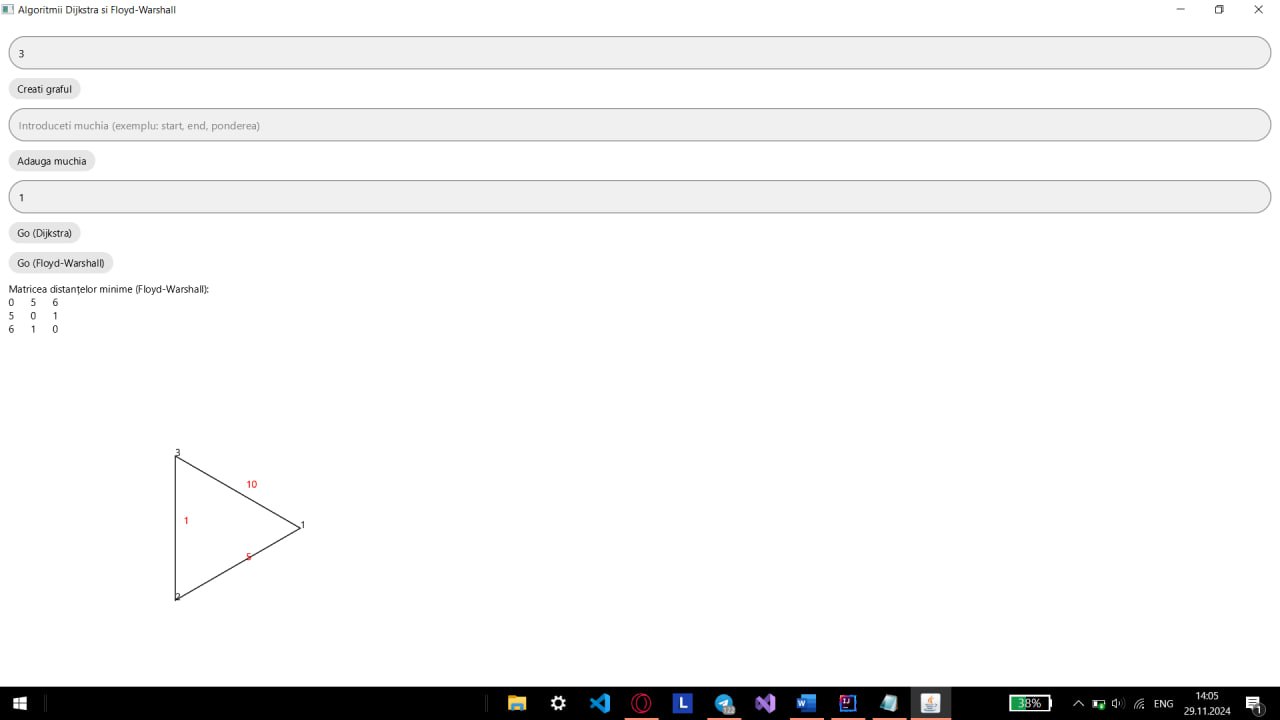


**Figura 3.3 - Rezultatele algoritmului Dijkstra afișate în interfață**

Se poate observa clar (în figura 3.3) distanța minimă de la nodul 1 către toate celelalte noduri.

**3.3 Algoritmul Floyd-Warshall**

Acest algoritm calculează matricea distanțelor minime între toate perechile de noduri. Rezultatele sunt afișate sub formă de matrice, unde fiecare element reprezintă distanța minimă între nodurile respective.



**Figura 3.4 - Matricea distanțelor minime și reprezentarea grafică a rezultatului algoritmului Floyd-Warshall.**

**CONCLUZIE**

Proiectul „Determinarea rutelor optime într-un sistem GPS” a reprezentat o abordare inovatoare și practică, concentrându-se pe utilizarea algoritmilor clasici pentru optimizarea rutelor într-o rețea de drumuri. Prin implementarea și testarea algoritmilor Dijkstra și Floyd-Warshall, proiectul a demonstrat posibilitatea de a obține soluții eficiente și adaptabile, aplicabile într-o gamă variată de domenii, cum ar fi transportul și logistica.  
 Un aspect important al proiectului a fost dezvoltarea și implementarea algoritmilor de optimizare. Utilizarea algoritmilor Dijkstra și Floyd-Warshall a evidențiat capacitatea de a calcula rapid rute optime în rețele complexe, cu o implementare solidă realizată în limbajul Java. De asemenea, aplicația dezvoltată a integrat o interfață grafică intuitivă, care a facilitat vizualizarea rețelelor de drumuri, a nodurilor și a conexiunilor dintre acestea, oferind utilizatorilor o perspectivă clară și ușor de înțeles asupra datelor procesate. Reprezentarea grafică a grafului a fost completată de o afișare clară a rezultatelor algoritmilor, ceea ce a contribuit la creșterea funcționalității și accesibilității aplicației.  
 Testarea performanței algoritmilor a fost un alt punct central al proiectului, iar rezultatele au confirmat eficiența și adaptabilitatea acestora. Algoritmul Dijkstra s-a remarcat prin rapiditatea și simplitatea sa, fiind ideal pentru navigații punctuale, în care utilizatorii au nevoie de calcule rapide între locații specifice. În schimb, algoritmul Floyd-Warshall s-a dovedit extrem de util pentru analiza completă a rețelelor, oferind o imagine de ansamblu detaliată a tuturor distanțelor posibile între noduri. Această diferență între cei doi algoritmi a subliniat complementaritatea lor, demonstrând că fiecare dintre aceștia are avantaje clare în funcție de scenariul de utilizare.  
 Rezultatele proiectului au arătat că integrarea algoritmilor de optimizare în sistemele GPS poate îmbunătăți semnificativ eficiența și ușurința utilizării acestora. Aplicația dezvoltată a fost concepută să fie modulară și scalabilă, ceea ce o face potrivită pentru extindere și adaptare la scenarii mai complexe. În ansamblu, proiectul a contribuit la avansarea tehnologiilor de optimizare a rutelor și a deschis noi perspective pentru utilizarea acestora în domenii precum logistica, transportul public și planificarea urbană. Acesta reprezintă un exemplu concret al modului în care algoritmii clasici pot fi integrați eficient în soluții moderne, oferind un echilibru între performanță, funcționalitate și accesibilitate.

**BIBLIOGRAFIE**

1. Algoritmul lui Dijkstra  
    Disponibil: <https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/>
2. Algoritmul Floyd-Warshall  
   Disponibil: [https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/](https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/" \t "_new)
3. Oracle JavaFX Documentation - Documentația oficială oferă ghiduri și exemple pentru dezvoltarea interfețelor grafice în JavaFX.  
   Disponibil: <https://docs.oracle.com/javafx/>
4. JavaFX GUI Full Course  
   Disponibil: [https://www.youtube.com/watch?v=9XJicRt\_FaI](https://www.youtube.com/watch?v=9XJicRt_FaI" \t "_new)
5. Algoritmul lui Dijkstra Explicat  
   Disponibil: <https://www.youtube.com/watch?v=XB4MIexjvY0>
6. Algoritmul lui Floyd-Warshall  
   Disponibil: <https://www.youtube.com/watch?v=YbY8cVwWAvw>

**ANEXE**

**Anexa 1: Main (FX)**

**package com.example.dijkstraproject;**

**import javafx.application.Application;**

**import javafx.geometry.Insets;**

**import javafx.scene.Scene;**

**import javafx.scene.canvas.Canvas;**

**import javafx.scene.canvas.GraphicsContext;**

**import javafx.scene.control.Button;**

**import javafx.scene.control.TextArea;**

**import javafx.scene.control.TextField;**

**import javafx.scene.layout.VBox;**

**import javafx.scene.paint.Color;**

**import javafx.stage.Stage;**

**import java.util.\*;**

**public class HelloApplication extends Application {**

**private final List<Map<Integer, Integer>> graph = new ArrayList<>();**

**private final RoyWarshell.FloydWarshallClass floydWarshall = new RoyWarshell.FloydWarshallClass();**

**private final TextArea resultArea = new TextArea();**

**private final Canvas canvas = new Canvas(500, 250);**

**public static void main(String[] args) {**

**launch(args);**

**}**

**@Override**

**public void start(Stage primaryStage) {**

**VBox root = new VBox(10);**

**Scene scene = new Scene(root, 1000, 600);**

**Application.setUserAgentStylesheet(getClass().getResource("/style.css").toExternalForm());**

**root.setPadding(new Insets(10, 10, 10, 10));**

**root.getStylesheets();**

**TextField nodeCountField = new TextField();**

**nodeCountField.setPromptText("Introduceti numarul nodurilor: ");**

**Button createGraphButton = new Button("Creati graful");**

**createGraphButton.setOnAction(e -> {**

**try {**

**int nodeCount = Integer.parseInt(nodeCountField.getText());**

**initializeGraph(nodeCount);**

**drawGraph();**

**floydWarshall.initializeGraph(nodeCount);**

**} catch (NumberFormatException ex) {**

**System.out.println("Error: Introduceti numere/cifre.");**

**}**

**});**

**TextField edgeInputField = new TextField();**

**edgeInputField.setPromptText("Introduceti muchia (exemplu: start, end, ponderea)");**

**Button addEdgeButton = new Button("Adauga muchia");**

**addEdgeButton.setOnAction(e -> {**

**String[] edgeData = edgeInputField.getText().split(" ");**

**int start = Integer.parseInt(edgeData[0]) - 1;**

**int end = Integer.parseInt(edgeData[1]) - 1;**

**int weight = Integer.parseInt(edgeData[2]);**

**addEdge(start, end, weight);**

**drawGraph();**

**floydWarshall.addEdge(start, end, weight);**

**edgeInputField.clear();**

**});**

**TextField startNodeField = new TextField();**

**startNodeField.setPromptText("Introduceti punctul start: ");**

**Button runDijkstraButton = new Button("Go (Dijkstra)");**

**runDijkstraButton.setOnAction(e -> {**

**int startNode = Integer.parseInt(startNodeField.getText()) - 1;**

**int[] distancesDijkstra = DijkstraClass.runDijkstra(startNode, graph);**

**displayResultsDijkstra(distancesDijkstra, startNode);**

**});**

**Button runFloydWarshallButton = new Button("Go (Floyd-Warshall)");**

**runFloydWarshallButton.setOnAction(e -> {**

**int[][] distancesFloyd = floydWarshall.runFloydWarshall();**

**displayResultsFloydWarshall(distancesFloyd);**

**});**

**resultArea.setEditable(false);**

**root.getChildren().addAll(nodeCountField, createGraphButton, edgeInputField, addEdgeButton, startNodeField, runDijkstraButton, runFloydWarshallButton, resultArea, canvas);**

**primaryStage.setScene(scene);**

**primaryStage.setTitle("Algoritmii Dijkstra si Floyd-Warshall");**

**primaryStage.show();**

**}**

**// Initializam graful**

**private void initializeGraph(int nodes) {**

**graph.clear();**

**for (int i = 0; i < nodes; i++) {**

**graph.add(new HashMap<>());**

**}**

**}**

**// Adaugam muchia in graf**

**private void addEdge(int start, int end, int weight) {**

**graph.get(start).put(end, weight);**

**graph.get(end).put(start, weight);**

**}**

**// Desenarea grafului**

**private void drawGraph() {**

**GraphicsContext gc = canvas.getGraphicsContext2D();**

**gc.clearRect(0, 0, canvas.getWidth(), canvas.getHeight());**

**int nodeCount = graph.size();**

**double radius = 0.5;**

**double centerX = canvas.getWidth() / 2;**

**double centerY = canvas.getHeight() / 2;**

**int maxWeight = 0;**

**for (int i = 0; i < nodeCount; i++) {**

**for (Map.Entry<Integer, Integer> entry : graph.get(i).entrySet()) {**

**maxWeight = Math.max(maxWeight, entry.getValue());**

**}**

**}**

**for (int i = 0; i < nodeCount; i++) {**

**double angle = 2 \* Math.PI \* i / nodeCount;**

**double nodeRadius = 100;**

**double x = centerX + Math.cos(angle) \* nodeRadius;**

**double y = centerY + Math.sin(angle) \* nodeRadius;**

**gc.setFill(Color.BLACK);**

**gc.fillOval(x - radius, y - radius, radius \* 2, radius \* 2);**

**gc.fillText(String.valueOf(i + 1), x, y);**

**for (Map.Entry<Integer, Integer> entry : graph.get(i).entrySet()) {**

**int neighbor = entry.getKey();**

**int weight = entry.getValue();**

**double neighborAngle = 2 \* Math.PI \* neighbor / nodeCount;**

**double neighborX = centerX + Math.cos(neighborAngle) \* nodeRadius;**

**double neighborY = centerY + Math.sin(neighborAngle) \* nodeRadius;**

**gc.strokeLine(x, y, neighborX, neighborY);**

**double midX = (x + neighborX) / 2;**

**double midY = (y + neighborY) / 2;**

**double offsetX = 10;**

**double offsetY = -5;**

**gc.setFill(Color.RED);**

**gc.fillText(String.valueOf(weight), midX + offsetX, midY + offsetY);**

**}**

**}**

**}**

**// Printarea Dijkstra**

**private void displayResultsDijkstra(int[] distances, int startNode) {**

**StringBuilder sb = new StringBuilder();**

**sb.append("Dinstanta minimala (Dijkstra) de la nodul ").append(startNode + 1).append(":\n");**

**for (int i = 0; i < distances.length; i++) {**

**sb.append("Pana la nodul ").append(i + 1).append(": ")**

**.append(distances[i] == Integer.MAX\_VALUE ? "∞" : distances[i])**

**.append("\n");**

**}**

**resultArea.setText(sb.toString());**

**}**

**// Printarea Floyd-Warshall**

**private void displayResultsFloydWarshall(int[][] distances) {**

**StringBuilder sb = new StringBuilder();**

**sb.append("Matricea distanțelor minime (Floyd-Warshall):\n");**

**for (int[] distance : distances) {**

**for (int i : distance) sb.append(i == 99999 ? "∞" : i).append("\t");**

**sb.append("\n");**

**}**

**resultArea.setText(sb.toString());**

**}**

**}**

**Anexa 2: Algoritmul lui Dijkstra**

**package com.example.dijkstraproject;**

**import java.util.\*;**

**public class DijkstraClass {**

**static class Node {**

**int id;**

**int distance;**

**Node(int id, int distance) {**

**this.id = id;**

**this.distance = distance;**

**}**

**}**

**// Algoritmul lui Dijkstra**

**public static int[] runDijkstra(int start, List<Map<Integer, Integer>> graph) {**

**int n = graph.size();**

**int[] distances = new int[n];**

**Arrays.fill(distances, Integer.MAX\_VALUE);**

**distances[start] = 0;**

**PriorityQueue<Node> priorityQueue = new PriorityQueue<>(Comparator.comparingInt(node -> node.distance));**

**priorityQueue.add(new Node(start, 0));**

**Set<Integer> processed = new HashSet<>();**

**while (!priorityQueue.isEmpty()) {**

**Node node = priorityQueue.poll();**

**int u = node.id;**

**if (!processed.add(u)) {**

**continue;**

**}**

**for (Map.Entry<Integer, Integer> entry : graph.get(u).entrySet()) {**

**int v = entry.getKey();**

**int weight = entry.getValue();**

**if (distances[u] + weight < distances[v]) {**

**distances[v] = distances[u] + weight;**

**priorityQueue.add(new Node(v, distances[v]));**

**}**

**}**

**}**

**return distances;**

**}**

**}**

**Anexa 3: Algoritmul lui Floyd-Warshell**

**package com.example.dijkstraproject;**

**public class RoyWarshell {**

**public static class FloydWarshallClass {**

**private final int INF = 99999;**

**private int[][] graph;**

**// Intializarea**

**public void initializeGraph(int nodes) {**

**graph = new int[nodes][nodes];**

**for (int i = 0; i < nodes; i++) {**

**for (int j = 0; j < nodes; j++) {**

**if (i == j) {**

**graph[i][j] = 0;**

**} else {**

**graph[i][j] = INF;**

**}**

**}**

**}**

**}**

**// Adaugam Muchia**

**public void addEdge(int start, int end, int weight) {**

**graph[start][end] = weight;**

**graph[end][start] = weight;**

**}**

**// Algoritmul Floyd-Warshall**

**public int[][] runFloydWarshall() {**

**int V = graph.length;**

**int[][] dist = new int[V][V];**

**for (int i = 0; i < V; i++) {**

**System.arraycopy(graph[i], 0, dist[i], 0, V);**

**}**

**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] != INF && dist[k][j] != INF && dist[i][k] + dist[k][j] < dist[i][j]) {**

**dist[i][j] = dist[i][k] + dist[k][j];**

**}**

**}**

**}**

**}**

**return dist;**

**}**

**}**

**}**

**Anexa 4: CSS (FX)**

**/\*Butoane\*/**

**.button {**

**-fx-background-color: #e7e6e6;**

**-fx-text-fill: black;**

**-fx-font-size: 12px;**

**-fx-padding: 5px 10px;**

**-fx-border-radius: 20px;**

**-fx-background-radius: 15px;**

**}**

**.button:hover {**

**-fx-background-color: #bbbbbb;**

**-fx-transition: background-color 10.5 ease;**

**}**

**.button:pressed {**

**-fx-background-color: #807d7d;**

**-fx-transition: background-color 0.5s ease;**

**}**

**/\* Input text \*/**

**.text-field {**

**-fx-background-color: #f0f0f0;**

**-fx-background-radius: 20px;**

**-fx-border-color: gray;**

**-fx-border-radius: 20px;**

**-fx-padding: 10px;**

**-fx-font-size: 13px;**

**}**

**.text-field:focused {**

**-fx-background-color: #e0e0e0;**

**-fx-border-color: #0078d4;**

**}**

**.root{**

**-fx-background-color: white;**

**}**