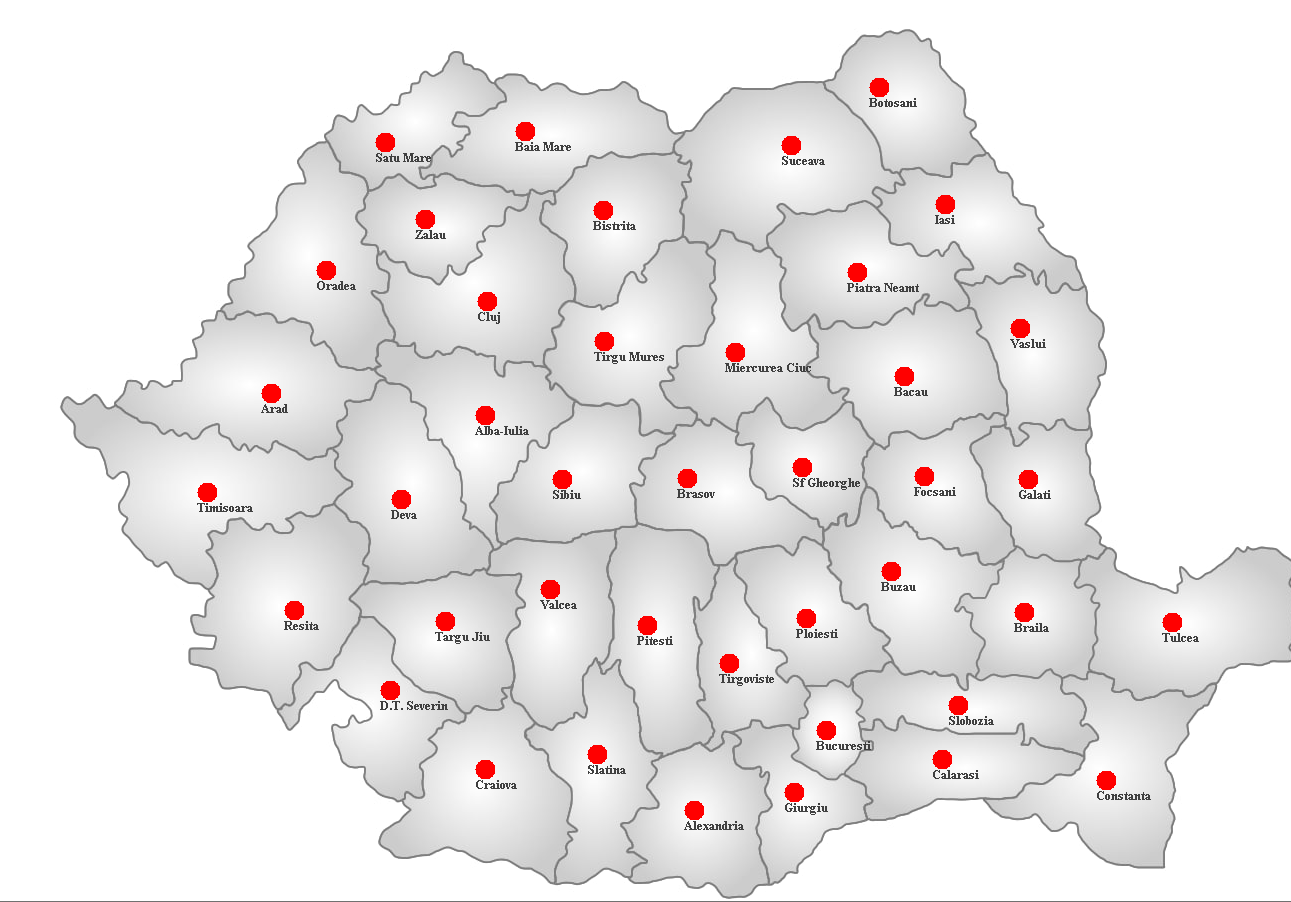
Universitatea Politehnica Timisoara

Proiect Sincretic

Tema: Graph neorientat cu interfata pentru resedintele de judet din Romania (Java Swing)



**Profesor indrumator: Dr. Ing. Mirella Mioc**

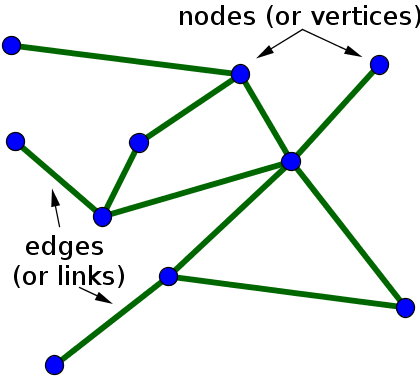
**Nume: Stoica Gheorghe-Denis**

**Facultatea de Automatica si Calculatoare**

**Informatica**

**Cuprins**

1. Introducere graf neorientat ponderat. Aplicabilitati
2. De ce graf neorientat ponderat in acest proiect?
3. Incarcarea oraselor din MySQL
4. Incarcarea drumurilor dintre orase din MySQL
5. BFS + algoritmul lui Dijkstra
6. Algoritmul lui Prim
7. Bibliografie
8. **Introducere graf neorientat ponderat. Aplicabilitati**

**Definitie:** Un grafic neorientat este un set de obiecte care sunt conectate între ele, adica toate marginile sunt bidirectionale.

La desenarea unui grafic neorientat, marginile sunt de obicei trasate ca linii între perechi de noduri, asa cum este ilustrat în figura urmatoare.

Graph neorientat, are structura matematica G = (V, E)

* V = multimea numarului de node-uri (vertices)
* E = multimea numarului de arce (edges)

Proprietate a grafului neorientat:

Daca avem arc intre nodul X->Y o sa avem arc si intre nodul Y->X. In general daca avem arc valoarea din lista a variabilei pentru costul drumului o sa fie notata cu 1 sau cu ponderea arcelor (costul drumului).

Aplicabilitati:

* Gasirea celui mai scurt drum de la un oras X la un oras Y(ex. Google Maps, Waze etc)
* Implementarea newsfeed-ului de la Facebook
* Medicina
* Conexiune LAN intre calculatoare
* Etc

1. **De ce graf neorientat ponderat in acest proiect?**

Am ales sa creez acest proiect in jurul grafului neorientat ponderat cu lista din cateva motive:

1. Eficienta mult mai buna la un numar mare de noduri
2. Neorientat din pricina faptului ca drumurile dintre localitatile resedinta de judet au sens atat de dus cat si de intors
3. Ponderat deoarece trebuie cunoscut costul respectivului drum pana in localitatea Y

Aplicabilitate in acest proiect

Acest proiect propune sa rezolve urmatoarele probleme cu ajutorul BFS + Dijkstra si algoritmului lui Prim:

* Cu ajutorul BFS + Dijkstra utilizatorul poate selecta localitatea de unde doreste sa plece si unde doreste sa ajunga astfel algoritmul va cauta cea mai scurta ruta posibila pentru a ajunge in timp optim.
* Pe de alta parte algoritmul lui Prim se aplica aici pentru gasirea celui mai scurt drum intre vecinii fiecarui nod(exemple la capitolul BFS + Dijkstra/ Algoritmul lui Prim).

1. **Incarcarea oraselor din MySQL**

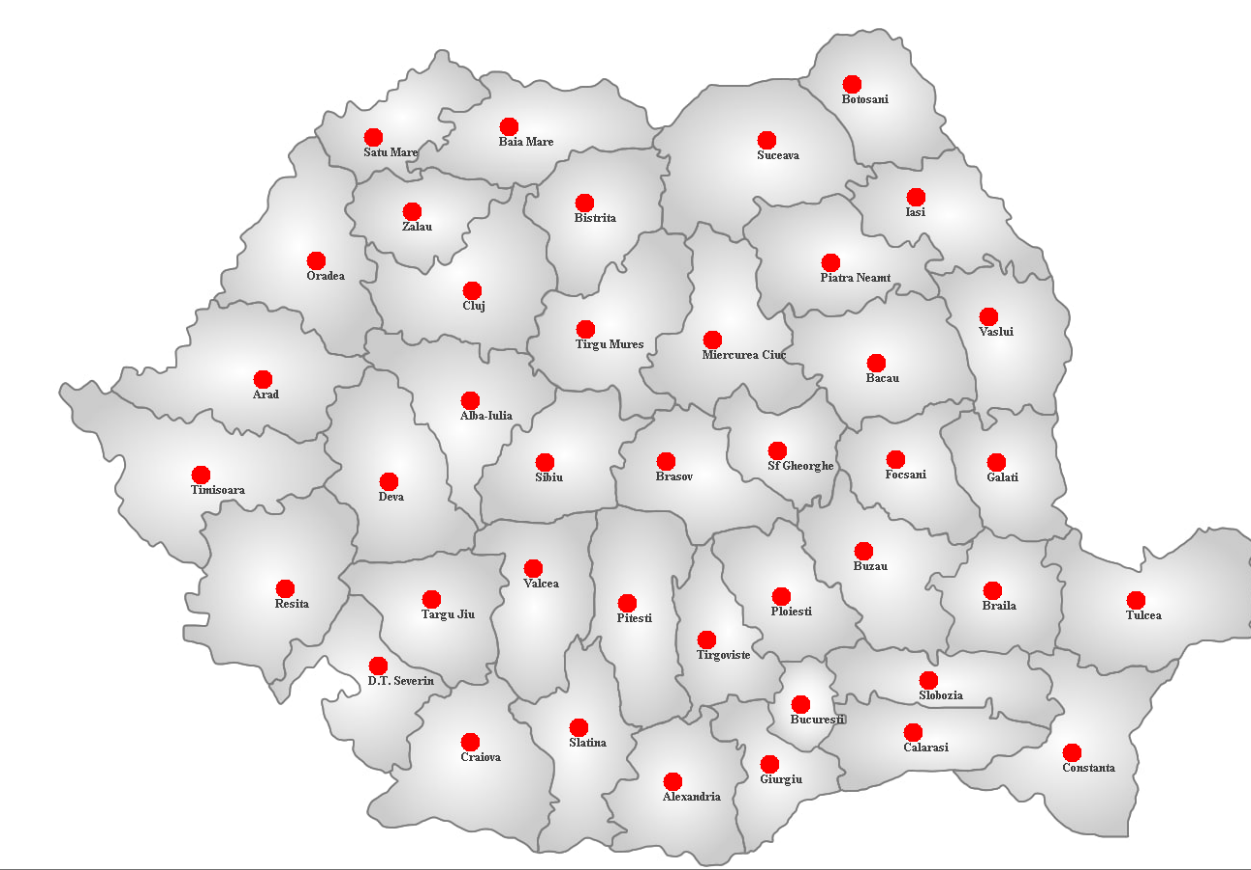
Pentru inserarea oraselor care sunt resedinte de judet ne vom folosi de o tabela numita nodes dintr-o baza de date MySQL pentru a prelua datele cum ar fi id-ul unic al orasului, numele acestuia, dar si coordonatele unde sa fie plasat respectivul oras pe interfata. Se presupune ca orasele introduse in baza de date sunt introduse corect si doar o data(sunt unice) astfel neexitand problema aparitiei unei duplicitati in program care ar duce la o posibila eroare.



Exemplu de orase din tabela 1

Complexitatea timpului pentru introducerea oraselor resedinta de judet in cadrul acestui proiect

1. Cel mai bun caz: O(1) – Atunci cand lista este goala
2. Cel mai comun(average) caz: O(41) – Deoarece exista 41 resedinte de judet, astfel inserandu-se de fiecare data la sfarsitul listei fiecare element aceasta trebuie parcursa pe rand
3. Cel mai rau caz: O(41) – Atunci cand sunt introduse toate orasele din baza de date



Inserare orase in interfata 1

Exemplu cod inserare in interfata programului + in lista

btn\_load.addActionListener(**new** ActionListener() {

**public** **void** actionPerformed(ActionEvent e) {

**int** count;

Connection conn;

PreparedStatement insert;

**try** {

Class.*forName*("com.mysql.cj.jdbc.Driver");

conn = DriverManager.*getConnection*("jdbc:mysql://betivii.xyz/r86436beti\_upt", "r86436beti\_denis", "P3}YU{ZbHEv3");

insert = conn.prepareStatement("SELECT \* FROM nodes");

ResultSet rs = insert.executeQuery();

ResultSetMetaData Rss = rs.getMetaData();

count =Rss.getColumnCount();

**int** i = 0;

**while**(rs.next())

{

nod[i] = **new** Dot(Integer.*parseInt*(rs.getString(1)), Integer.*parseInt*(rs.getString(3)), Integer.*parseInt*(rs.getString(4)), rs.getString(2));

dotList.add(nod[i]);

i++;

progres.setValue(progres.getValue() + 1);

}

} **catch** (ClassNotFoundException e1) {

e1.printStackTrace();

}

**catch** (SQLException e1) {

e1.printStackTrace();

}

repaint();

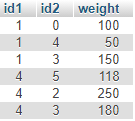
btn\_load.setEnabled(**false**);

}

});

1. **Incarcarea drumurilor dintre orase din MySQL**

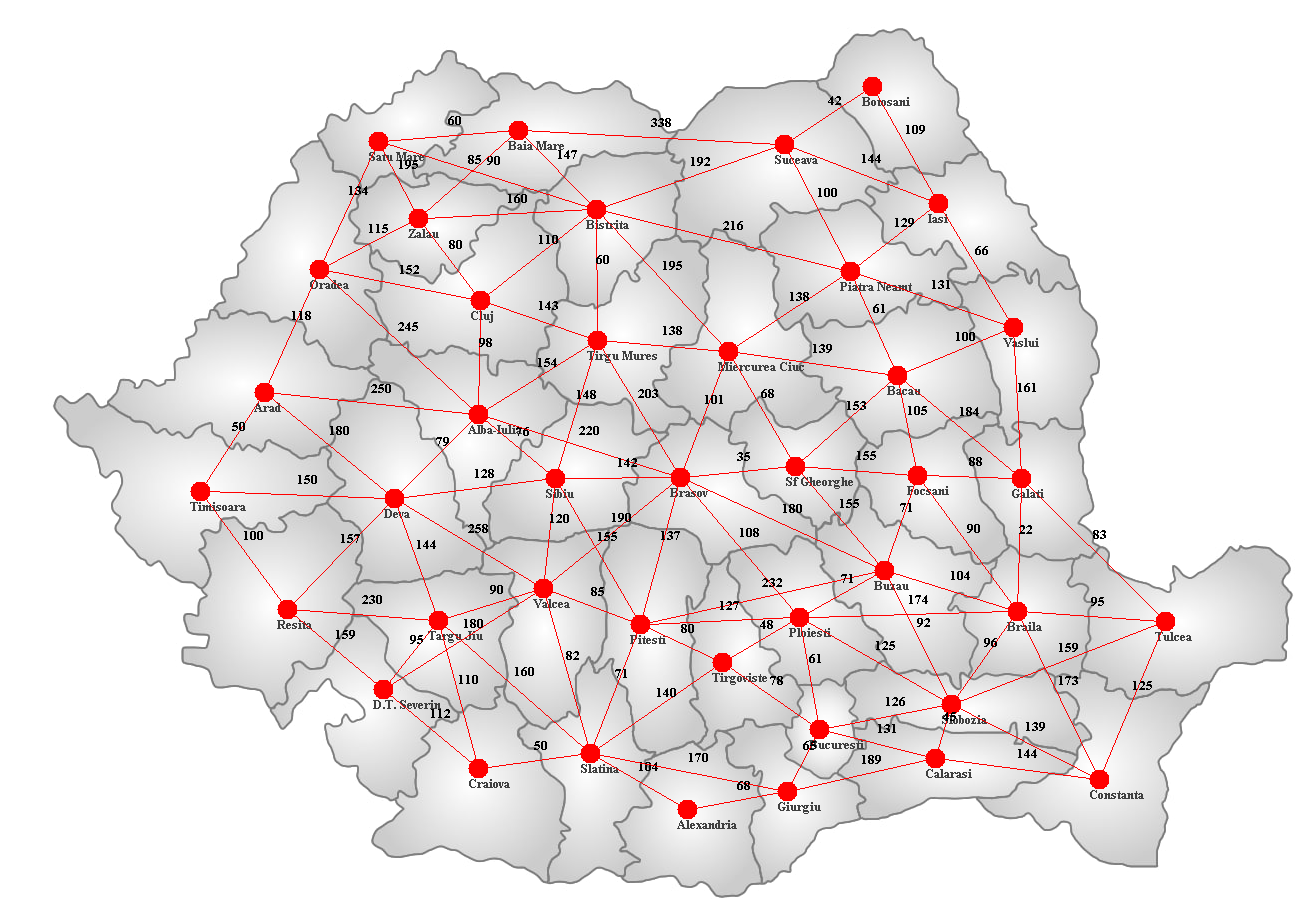
Pentru inserarea drumurilor dintre orasele care sunt resedinte de judet ne vom folosi de o tabela numita edges dintr-o baza de date MySQL pentru a prelua datele cum ar fi id1 unic al primului oras, id2 unic al celui de-al doilea oras intre care o sa se faca drumul, dar si weight, tradus fiind costul drumului intre cele 2 orase. Se presupune ca drumurile introduse intre respectivele orase in baza de date sunt introduse corect si doar o data(sunt unice) astfel neexitand problema aparitiei unei duplicitati in program care ar duce la o posibila eroare.



Inserare drumuri intre orase 1

Complexitatea timpului pentru graful neorientat ponderat cu liste in cadrul acestui proiect

1. Cel mai bun caz: O(1^2) – Atunci cand lista de arce este goala
2. Cel mai comun(average) caz: O(107) – Deoarece exista 107 drumuri intre orase, astfel inserandu-se de fiecare data la sfarsitul listei fiecare element aceasta trebuie parcursa pe rand
3. Cel mai rau caz: O(107) – Atunci cand sunt introduse toate drumurile intre orase din baza de date



Inserare drumuri in interfata 1

Exemplu cod inserare in interfata programului + in lista

btnLoadEdges.addActionListener(**new** ActionListener() {

**public** **void** actionPerformed(ActionEvent e) {

**int** count;

Connection conn;

PreparedStatement insert;

**try** {

Class.*forName*("com.mysql.cj.jdbc.Driver");

conn = DriverManager.*getConnection*("jdbc:mysql://betivii.xyz/r86436beti\_upt", "r86436beti\_denis", "P3}YU{ZbHEv3");

insert = conn.prepareStatement("SELECT \* FROM edges");

ResultSet rs = insert.executeQuery();

ResultSetMetaData Rss = rs.getMetaData();

count =Rss.getColumnCount();

**int** i = 0;

**while**(rs.next())

{

edge[i] = **new** Edge();

edge[i].setEdge(nod[Integer.*parseInt*(rs.getString(1))], nod[Integer.*parseInt*(rs.getString(2))]);

edge[i].setWeight(Integer.*parseInt*(rs.getString(3)));

edgeList.add(edge[i]);

i++;

progres\_edge.setValue(progres.getValue() + 1);

}

} **catch** (ClassNotFoundException e1) {

e1.printStackTrace();

}

**catch** (SQLException e1) {

e1.printStackTrace();

}

btnLoadEdges.setEnabled(**false**);

}

});

1. **BFS + algoritmul lui Dijkstra**

Edsger Dijkstra 1

Algoritmul lui Edsger W. Dijkstra este folosit pentru a gasi cele mai scurte cai intr-un grafic, care poate fi folosit pentru a reprezenta retelele de drumuri.

Reprezentat in concordanta cu proiectul acesta, algoritmul lui Dijkstra ne ajuta sa gasim drumul cel mai scurt dintre 2 orase selectate de catre utilizator. Este printre cel mai folosit algoritm pentru cautarea drumului cel mai scurt intre 2 noduri.

Complexitatea timpului pentru BFS + Dijkstra

1. Cel mai bun caz: O(V+E logV) – V reprezinta orasele, iar E reprezinta drumurile intre ele, acest caz apare cand exista o coada de prioritate
2. Cel mai rau caz: O(V^2) – V reprezinta orasele, acest caz presupune implementarea clasica

Variabile utilizate in procesul algoritmului lui Dijkstra + BFS

private ArrayList<Dot> nodes; -> Lista cu orasele

private EdgeList<Edge> edges; -> Lista cu drumurile intre orasele

private Set<Dot> visitedNodes; -> Lista pentru orasele vizitate

private Set<Dot> unvisitedNodes; -> Lista pentru orasele nevizitate

private Map<Dot, Integer> totalWeight; -> Map pentru calcularea costului drumului a mai multor posibile rute

private Map<Dot,Dot> prevNodes; -> Map pentru orasele vizitate in trecut

Functii utilizate in procesul algoritmului lui Dijkstra + BFS

public ShortestPathDjsBfs() -> instantiaza constructorul care initiaza lista cu orasele si drumurile intre ele

public void execute(Dot start) -> aceasta functie incepe procesul de executie al algoritmului lui Dijkstra + BFS, aceasta instantiaza variabilele visitedNodes, unvisitedNodes de top HashSet<>, iar totalWeight si prevNodes de tip HashMap<> urmand ca dupa intr-un while sa caute nodul minim din nevizitate si care are si un drum

private Dot getMinimum(Set<Dot> dots) -> aceasta functie returneaza valoarea celui mai scurt drum pana la urmatorul nod

public void findMinimalWeights(Dot node) -> functie care cauta cel mai scurt drum si il introduce in totalWeight si in prevNodes

private int getDistance(Dot node, Dot target) -> returneaza distanta dintre 2 orase

private int getShortestDistance(Dot node) -> returneaza cea mai mica distanta

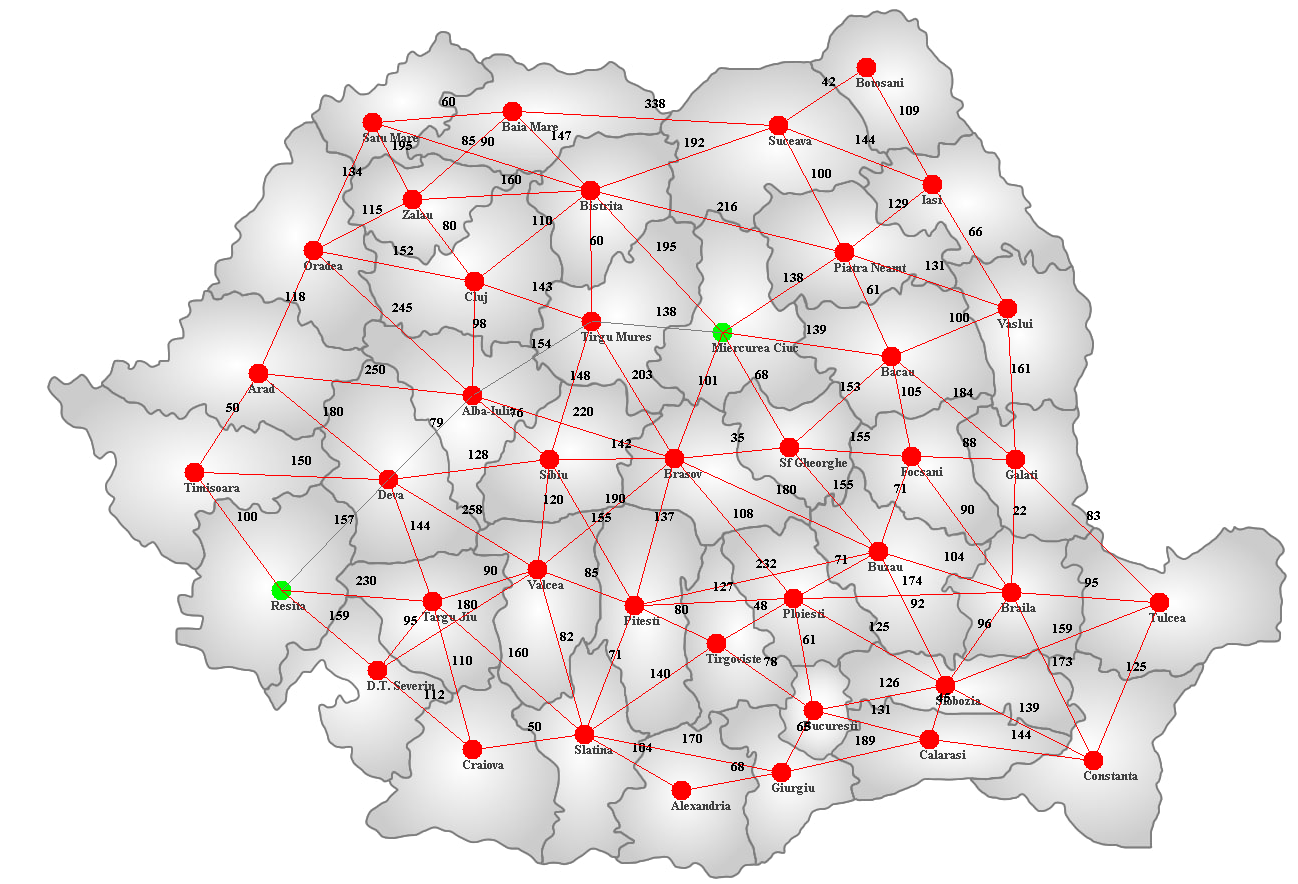
private DotList<Dot> getNeighbours(Dot node) -> returneaza vecinii orasului dat ca parametru

public Map<Dot, Dot> getPrev() -> returneaza prevNodes

public boolean checkNeighbour(Edge e) -> returneaza daca e vecin sau nu orasul

public int getIndexOf(Edge e) -> returneaza indexul din lista al drumului

public void getPath(Dot a, Dot b) -> returneaza pe interfata drumul cel mai scurt de la orasul1 selectat de catre utilizator, la orasul2, colorand in verde respectivele orase selectate, si in gri drumurile gasite optime de catre algoritm



Reprezentare prin BFS + Dijkstra 1

1. **Algoritmul lui Prim**

Algoritmul gaseste un MST pentru un graf neorientat. Isi obtine arborele gasind un subset de margini care include fiecare varf. Algoritmul functioneaza construind un varf la un moment dat și adaugand la fiecare pas o conexiune mai ieftina la alta.

Raportat la acest proiect ne vom folosii de algoritmul lui Prim pentru a gasii printr-o simpla apasare de buton cel mai scurt drum dintre fiecare oras si vecinii sai.

Complexitatea timpului pentru algoritmul lui Prim

1. Cel mai bun caz: O(E logV) – V reprezinta orasele, iar E reprezinta drumurile intre ele, acest caz apare cand exista o coada de prioritate
2. Cel mai rau caz: O(V^2) – V reprezinta orasele, acest caz presupune implementarea clasica

Variabile utilizate in procesul algoritmului lui Prim

private ArrayList<Dot> nodes; -> Lista cu orasele

private EdgeList<Edge> edges; -> Lista cu drumurile intre orasele

private int []nodes\_visited = new int[41]; -> Array pentru orasele vizitate

private int totalWeight = 0; -> Suma totala a costurilor tuturor drumurilor dintre orase

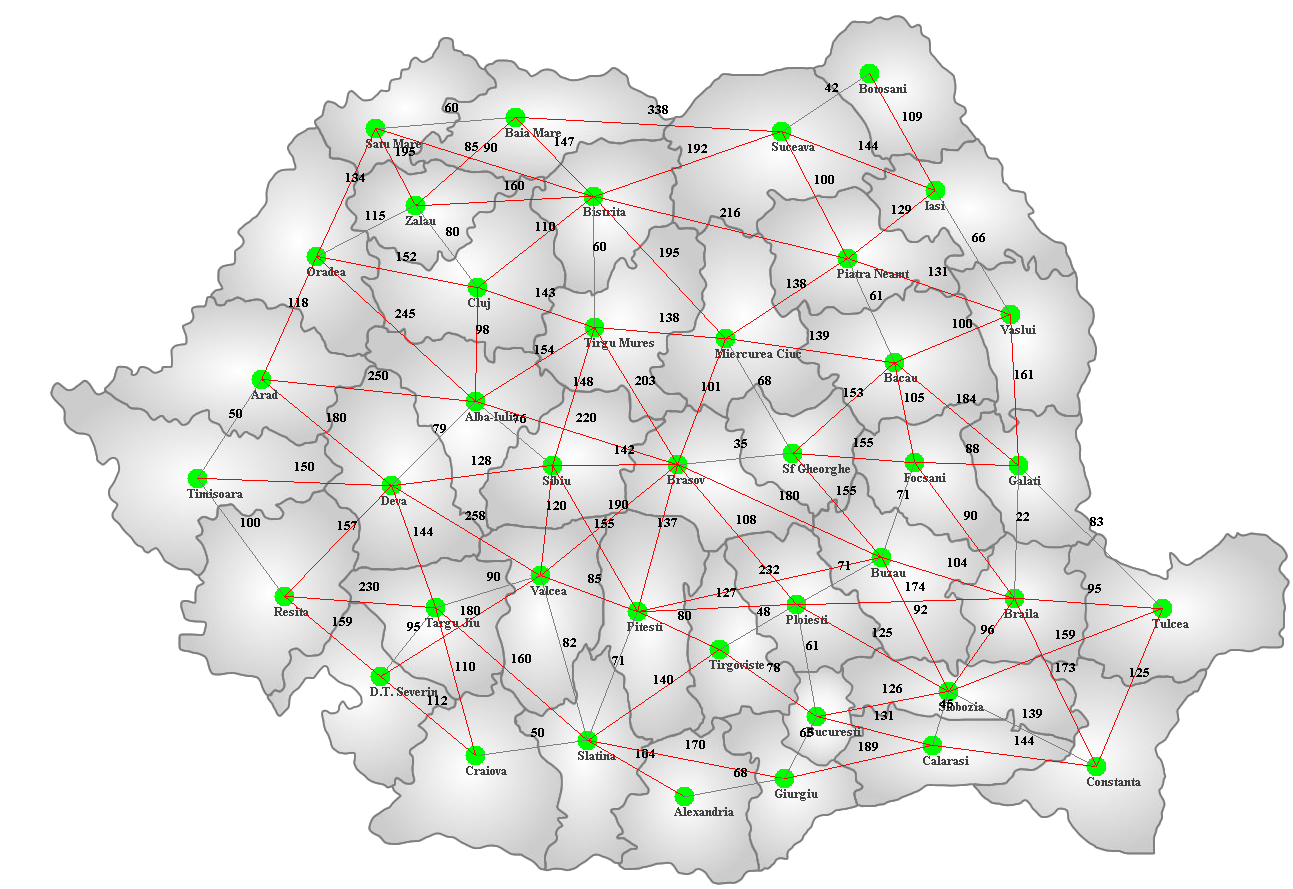
private int count = 0; -> contor pentru parcurgerea fiecarui oras

Functii utilizate in procesul algoritmului lui Prim

public ShortestPathPrim() -> instantiaza constructorul care initiaza lista cu orasele si drumurile intre ele

public void execute(Dot start) -> aceasta functie incepe procesul de executie al algoritmului lui Prim, aceasta instantiaza variabilele nodes\_visited pe 0, min cu valoarea 32676, dar si 2 variabile de tip Dot pentru a stoca nodul 1 si nodul 2, urmand ca dupa intr-un while sa puna nodul 1 = nodes.get(i) adica sa ia fiecare nod in parte si sa ii caute in nodul 2 vecinul cu cel mai scurt drum; Se printeaza in consola un mesaj informativ, iar pe interfata se vor colora orasele parcurse dar si drumul cel in scurt intre fiecare

public int checkEdge(Dot start, Dot dest) -> verifica daca exista drum intre doua orase si ii returneaza pozitia, in caz negativ returneaza -1



1. **Bibliografie**

<https://mathinsight.org/image/small_undirected_network_labeled>

<https://www.programiz.com/dsa/prim-algorithm>

<https://github.com/dharmendranamdev/Data-Structure-and-Algorithms/blob/master/Graph/src/minimumSpanningTreeAlgo/PrimsAlgo_AdjMat.java>

<https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)>

<https://www.quora.com/What-are-real-life-applications-of-graphs>

<https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm#:~:text=Dijkstra's%20algorithm%20(%2F%CB%88da%C9%AA,%2C%20for%20example%2C%20road%20networks.&text=For%20a%20given%20source%20node,that%20node%20and%20every%20other>.

<https://www.geeksforgeeks.org/dijkstras-algorithm-for-adjacency-list-representation-greedy-algo-8/>

<https://en.wikipedia.org/wiki/Prim%27s_algorithm>