UNIVERSITETI I PRISHTINËS "HASAN PRISHTINA"
FAKULTETI I INXHINIERISË ELEKTRIKE DHE KOMPJUTERIKE
DEPARTAMENTI: INXHINIERIA KOMPJUTERIKE – BACHELOR

Lënda: Dizajni dhe Analiza e Algoritmeve

Tema: Algoritmi i Dijkstras

Studentët: Lavdim Pireva
Shpend Jahiri

Mentorë: Prof.Ass.Dr. Avni Rexhepi
Msc. Dardan Shabani

Prishtinë, Maj 2020

Përmbajtja

Hyrje

Në procesin e prodhimit, organizimit dhe menaxhimit, ne duhet të zgjidhim shumë shpesh probleme me rrugë më të shkurtër. Për shembull, në procesin e prodhimit, për të përfunduar detyrat e prodhimit shpejt dhe me efikasitet, duhet të gjejmë rrugën më të shkurtër për të përfunduar çdo detyrë prodhimi, në procesin e menaxhimit, për të fituar përfitime të mëdha me kosto minimale duhet të hartojmë plane racionale, në rrjetin ekzistues të transportit për të transportuar sasi të mëdha të mallrave me kosto minimale duhet të organizojmë një rrugë të arsyeshme transporti. Të gjitha këto mund të përmblidhen si "problemi i rrugës më të shkurtër". Problemi i gjetjes së rrugës më të shkurtër midis kulmit të fillimit dhe kulmit terminal) është përdorur gjerësisht në fusha të ndryshme, siç janë: algoritmi i rrjetit kompjuterik, robot Pathfinder, navigimi i rrugës, modelimi i lojërave etj.

Algoritmi i Dijkstra zgjidh problemin e gjetjes së rrugës më të shkurtër nga një pikë në një graf (burimi) deri në një destinacion. Rezulton se mund të gjesh shtigjet më të shkurtëra nga një burim i caktuar në të gjitha pikat në një grafik në të njëjtën kohë, kështu që ky problem nganjëherë quhet problemi i shtigjeve më të shkurtra me burim të vetëm(SSSP). Ky punim do t'na ndihmojë të kuptojmë konceptet themelore të Algoritmit Dijkstra me ndihmën e teorisë, shembujve dhe ilustrimeve.

1.Grafet dhe algoritmet e tyre

Nje graf është një koleksion i çifteve – koleksion i njerëzve, koleksion i qytetëve, koleksion i yjeve, koleksion i vendeve pra ne pergjithësi koleksion i objekteve. Keto objekte ne aspektin e grafeve quhen ***nodes***(nyjet) ndërësa keto nyje jane te lidhura permes ***edges***(degët).

Ne matematike grafi paraqet bashkesinë e nyjëve dhe degëve qe i lidhin ato nyje. Ne figure eshte paraqitur grafi standard i cili paraqet grafin e pa orientuar.

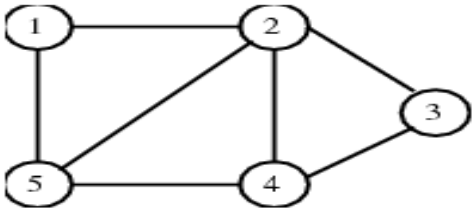


Figura 1: Grafi G = ({1,2,3,4,5},{{1,2},{1,5},{2,3},{2,4},{2,5},{3,4},{4,5}})

Pasi qe grafi është i pa orientuar atëherë renditja behet nga numri me i vogel deri te numri me i madh, pra ne shprehje shohim se {1,2,3,4,5} jane nyjet ndersa {1,2},{1,5} jane deget qe i lidhin nyjet perkatese. Pra ne shprehjen matematikore nuk verejm ndonje element p.sh {5,1}.

Element i rendësishëm i grafit është pesha apo kostoja e nyjeve. Kjo na tregon ndonje informacion për peshën p.sh nese kostoja është 5 atëherë nga nyja v1 deri tek nyja v5

atëherë ajo mund të paraqet distancën në kilometra apo distancën kohore ose ndonje informacion tjetër me rëndësi siq janë shpenzimet e karburantit. [1]

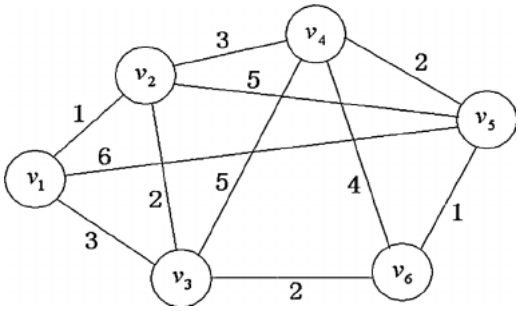


Figura 2: Grafi permes kostos

Perveq grafit te standardizuar e kemi edhe grafin e drejtuar apo te orientuar si më poshtë:

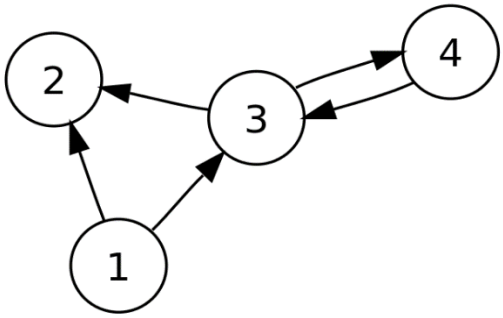


Figura 3: Grafi i drejtuar G = ({1,2,3,4},{{1,2},{1,3},{3,2},{3,4},{4,3}})

Algoritmet e grafeve janë: ***Depth - first*** (thellësia së pari), ***Breadth-first*** (gjerësia se pari); keto njihen si **algoritmet e përshkimit**.

Spanning Tree **(Algoritmi i pemës me shtrirje minimale)** është një algoritëm ku kjo peme formohet ne momentin kur grafit i'a eleminojm ato lidhjet e teperta ashtu qe me na e mundesu me mbet lidhjet minimale por që e sigurojn qarkullimin në të gjitha nyjet. Pra kjo është një nenbashkësi e degëve ne tërë grafin, prandaj procesi derisa e gjejm atë nenbashkësi kërkon mjaftë kohë sepse nese kemi **n** degë do te kemi 2^{n} nënbashkësi.[2]

[1]Internet: https://people.cs.pitt.edu/~milos/courses/cs441/lectures/Class25.pdf
[2] Internet: https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/03Graphs.pdf

Algoritmi Dijkstra-Prim është një algoritem per gjetjën e pemës minimale ndërmjet nyjëve ne graf. Ky algoritem është zhvilluar nga Edsger Dijkstra. Algoritmi i Dijkstra-Prim-it njihet edhe si ***"greedy"*** algoritem, pra algoritem lakmitar sepse synon çdo hap me e gjet vleren minimale të mundshme. **Greedy** algoritmet punojnë duke shikuar në nënbashkësinë e problemit më të madh dhe duke bërë vendimin më të mirë bazuar në atë informacion. Pra ky algoritem ne çdo hap i shton vetem ato dege minimale. Ideja e këtij algoritmi është që ky algoritem kalon në secilen nyje për dallim prej Dijkstras qe për me e gjet rrugen minimale nuk kalon në secilen nyje nga burimi në destinacion.

Algoritmi Kruskal është algoritem me tendencë të njejtë sikurse algoritmet më lartë pra në ndërtimin e pemës me shtrirje minimale por parimi i këtij algoritmi është qe ky fillon të ndërtoj pemen minimale duke zgjedhur degën me koston më të ulët ne graf për dallim nga algoritmet paraprake që fillojn nga nyja fillestare e grafit.[3]

2. Algoritmi Dijkstra

Algoritmi i Dijkstra-s mbështetet në metodën e përsëritjeve serike. Bashkësia e nevojshme (e kërkuar) e nyjave konstruktohet duke shtuar nga një nyje ne çdo përsëritje. Ky algoritem bazohet në një parim, ku fillon duke emëruar nyjën fillestare me 0 dhe nyjat tjera me ∞. Pra merret 0 per arsye qe nuk është berë asnjë përsëritje dhe merret infinit për arsye që nuk egziston asnjë rrugë nga nyja fillestare deri te ndonjë nyje tjetër e ndryshme nga nyja fillestare.[4]

[3] Prof.Dr.Qefsere Gjonbalaj Doko "Matematika III me matematikë diskrete"
[4] Internet: https://en.wikipedia.org/wiki/Dijkstra's_algorithm

2.1 Algoritmi në lidhje me problemin e rrugës më të shkurtë

Egzistojnë disa algoritme të ndryshme të cilat mundësojnë gjetjen e rrugës më të shkurtë në mes të dy nyjave të grafit me peshë. Ne do ta prezantojmë algoritmin të zbuluar nga matematikani gjerman Edsger Dijkstra ne 1959. Verzionin të cilin do ta përshkruajmë zgjidh problemet te grafet me peshë dhe jo të orientuara, ku të gjitha peshat janë pozitive.

Në qoftë se kemi nje graf me disa degë, me një nyje burimore dhe një nyje që e paraqet destinacionin e grafit athëhere tendenca e këtij algoritmi është me e gjetë sekuencën e degëve të cilave kur i'a mbledhim koston na jep koston minimale.

Në figurën më poshtë është paraqitur një graf per zbatimin e algoritmit të Dijkstra-s ashtu qe te fitojm rrugën më të shkurtë nga nyja A deri tek nyja G. Grafi duhet të jetë i lidhur, i padrejtuar dhe degët e tij të jenë me peshë:

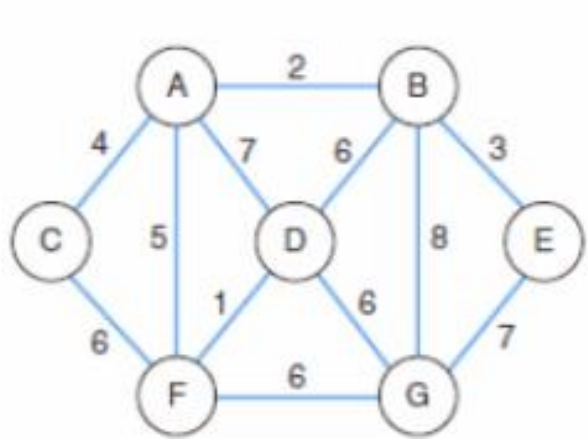


Figura 4: Grafi

Dijkstra e perdorë parimin duke filluar nga nyja fillestare apo ***"nyja burimore"*** dhe dega iniciale e cila është e lidhur drejtpërdrejt me nyjen burimore.[5]

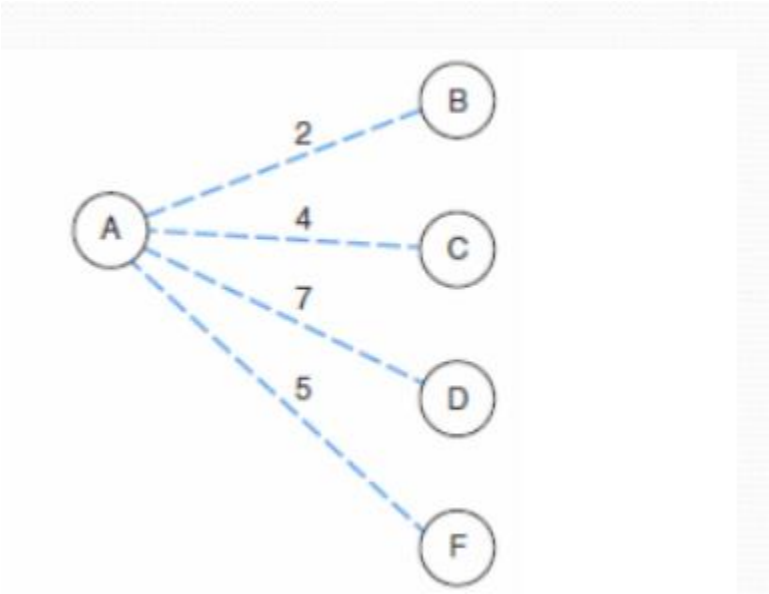


Figura 5: Hapi 1

Ne hapin e parë i paraqesum te gjitha degët të cilat janë të lidhura direkt me nyjën burimore. Këtu zgjedhet nyja me degën minimale, pra nyja B.

[5] Prof.Dr.Qefsere Gjonbalaj Doko "Matematika III me matematikë diskrete"

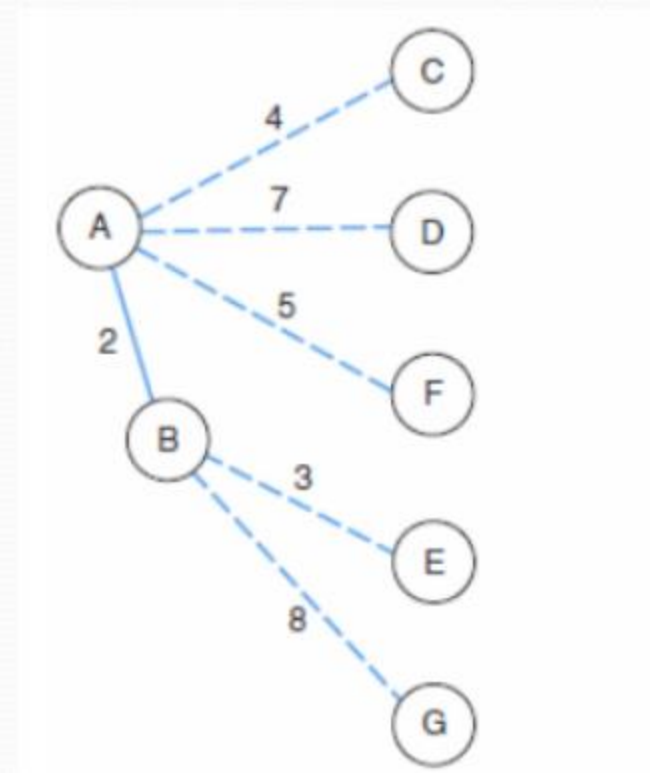


Figura 6: Hapi 2

Ne hapin e dytë e kemi lidhur nyjen B me nyjen fillestare, ndersa distanca me nyjet tjera llogaritet duke filluar nga nyja burimore A. Prandaj distanca në mes nyjes A dhe E është 5. Prandaj në hapin e ardhshëm zgjedhim nyjen me degen minimale, pra nyjen C. Pra ne cdo hap e ruan informacionin (koston) prej nyjes fillestare.

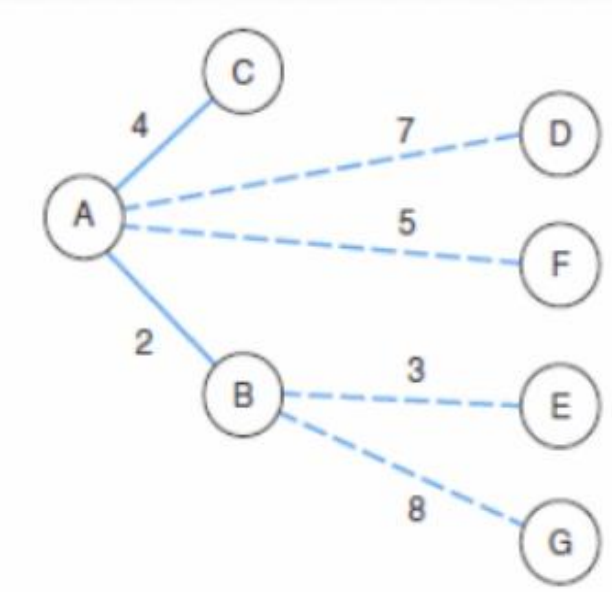


Figura 7: Hapi 3

Në hapin e tretë paraqiten kostot e radhes dhe vërejm që A dhe E kanë koston 5 ashtu sikurse A dhe F. Në këtë rast mundemi me e marr ose F ose E.

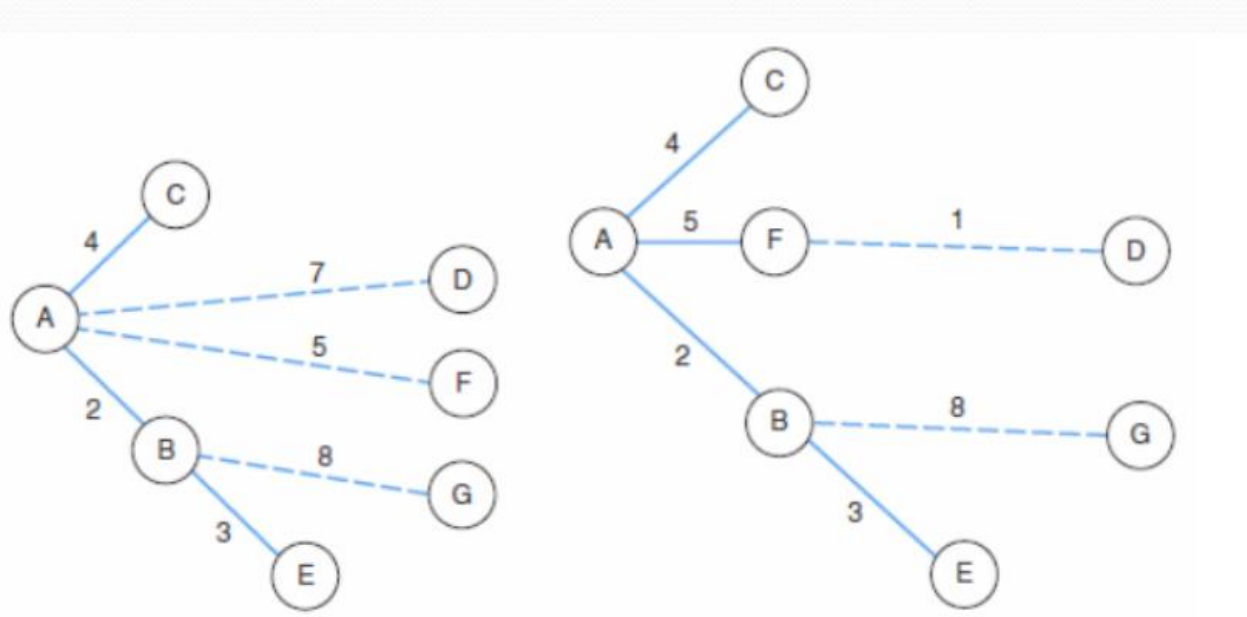


Figura 8: Hapi 4 dhe Hapi 5

Te hapi i katërt zgjedhim degen me kosto minimale pra nyjën F. Në hapin e 5-të, distanca nga nyja burimore deri te nyja D është 6, ndërsa nga nyja burimore deri tek nyja G është 10, prandaj zgjedhim nyjën D.

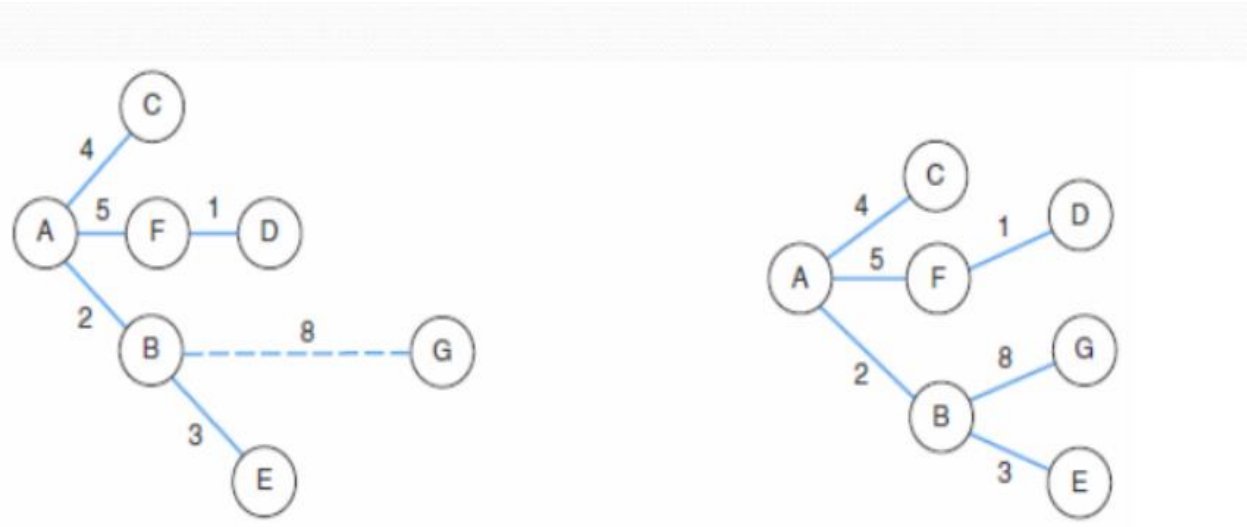


Figura 9: Hapi 6 dhe Hapi 7

Në hapin e gjashtë na ka mbetur vetem nyja G pa e vizituar, prandaj e marrim edhe nyjen G. Hapi i shtatë paraqet hapin final ku e kemi ndertuar rrugën me të shkurtër në mes të nyjes burimore dhe asaj në destinacion respektivisht nyjes A dhe nyjes G. Pra rruga e tillë është rruga me kosto 10.

Në fakt vërejmë qe ky algoritëm e gjenë njëkohësisht rrugen minimale me të gjitha nyjet tjera të cilat lidhen me nyjën burimore.[6]

2.2 Kompleksiteti Kohor

Në shkencat kompjuterike, kompleksiteti i kohës është kompleksiteti kompjuterik që përshkruan sasinë e kohës që duhet për tu ekzekutuar një algoritëm. Kompleksiteti kohor zakonisht vlerësohet duke llogaritur numrin e operacioneve elementare të kryera nga

[6] Prof.Dr.Qefsere Gjonbalaj Doko "Matematika III me matematikë diskrete"

algoritmi, duke supozuar se çdo operacion elementar kërkon një kohë fikse për tu kryer. Meqenëse koha e funksionimit të një algoritmi mund të ndryshojë varësisht nga madhësia e inputeve, zakonisht konsiderohet kompleksiteti i rastit më të keq, që është sasia maksimale e kohës që kërkohet për inputet e një madhësie të caktuar. Më pak e zakonshme, dhe që specifikohet në mënyrë të qartë, është kompleksiteti i rasteve mesatare, që është mesatarja e kohës së marrë në inputet e një madhësie të caktuar. [7] Në të dyja rastet, kompleksiteti kohor shprehet në përgjithësi si një funksion i madhësisë së inputeve. Ndërsa termi kompleksiteti i rastit më të mirë është përdorur në shkencat kompjuterike për të përshkruar sjelljen e një algoritmi në kushte optimale. Kufijtë e kohës së funksionimit të algoritmit të Dijkstra në një graf me skajet E dhe vertices V mund të shprehen si një funksion i numrit të skajeve, të shënuara |E|, dhe numri i vertikteve, i shënuar |V| duke përdorur shënimin big-O.[8]

Big-O jep një mënyrë tjetër për të folur në lidhje me mënyrën se si inputi ndikon në kohën për ekzekutim të algoritmit. Ai jep një kufi të sipërm të kohës së vrapimit.

Në algoritmin e Dijkstra, efikasiteti ndryshon në varësi të | V | = n DeleteMins dhe | E | azhurnime për radhët e përparësive që janë përdorur.

Nëse është përdorur Fibonnaci Heap, atëherë kompleksiteti është O (| E | + | V | log | V |), e cila është lidhja më e mirë. Operacioni DeleteMins merr O (log | V |).

Nëse përdoret standard binary heap, atëherë kompleksiteti është O (| E | log | E |),

| E | log | E | termi vjen nga | E | azhurnimet për grumbullimin standard.

Nëse grupi i përdorur është një radhë prioritare,atëherë kompleksiteti është O (| E | + $|V|^2$) O($|V|^2$) termi vjen nga | V | skanimet e setit të pakontrolluar të Neë Frontier për të gjetur kulm me vlerën më të vogël sDist.[9]

[7] Internet: https://en.wikipedia.org/wiki/Best,_worst_and_average_case
[8] Internet: https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/
[9] Internet: http://www-inst.eecs.berkeley.edu/~cs61bl/r//cur/graphs/dijkstra-algorithm-runtime.html?topic=lab24.topic&step=4&course=

2.3 Algoritmi i Dijkstras me Priority Queue (listë prioritare)

Në shkencat kompjuterike, një listë prioritare është një tip abstrakt i të dhënave i ngjashëm me strukturën e rregullt të të dhënave në radhë ose rafte, në të cilat secili element përveç kësaj ka një "përparësi" të lidhur me të. Në radhë prioritare, një element me përparësi të lartë shërbehet para një elementi me përparësi të ulët. Në disa zbatime, nëse dy elementë kanë të njëjtin përparësi, ato shërbehen sipas rendit në të cilin janë grumbulluar, ndërsa në zbatime të tjera, renditja e elementeve me të njëjtën përparësi është e papërcaktuar. Një listë prioritare mbështet operacionet e mëposhtme:

- Insert(x): shton elementin x në radhë.
- DecreaseKey(x, k): zvogëlon vlerën e çelësit të x për k, ku $k \leq x$.
- Extractmin() : fshin dhe shton elementin e Q me çelësin më të vogël

Teknikisht ky quhet edhe min-priority queue pasi që vazhdimisht bëhet ekstraktimi i çelësit minimal. Një listë prioritare mund të implementohet si një vektorë(array), listë e lidhur(linked list), pirg(heap) ose disa lloje tjera metodash si vektorë të parenditur (unordered array).

Lista prioritare mund të përdoret për të nxjerr minimumin në mënyrë efikase kur zbatohet algoritmi i Dijkstra, megjithëse shpesh herë është nevoja për aftësinë për të ndryshuar përparësinë e një kulmi të veçantë në listë prioritare. Poashtu si pasojë e implementimit të priority queue(listës prioritare) me algoritmin e dijkstras bën që të ndryshoj kompleksiteti kohor për Dijkstrën.[10]

[10] Internet: https://en.wikipedia.org/wiki/Priority_queue

2.3.1 Dijkstra e implementuar me Fibonnaci Heap

Fibonnaci Heap nganjëherë quhet struktura e të dhënave "dembele" sepse struktura e tyre menaxhohet më pak në mënyrë rigoroze derisa ato kryejnë operacione, sesa në shumë struktura të tjera të të dhënave. Pra ky “dembelizëm” u lejon atyre të kenë kompleksitet kohore jashtëzakonisht të shpejta të amortizuara për operacionet e zakonshme të pirgut. Algoritmi origjinal i rrugës më të shkurtër të Dijkstras nuk përdor një listë përparësie dhe shkon në kohën O (V^2). Kur përdoret Fibonacci heap si një listë përparësie, ajo funksionon në kohën O (E + V log V), e cila është asimptotikisht kompleksiteti më i shpejtë i njohur për këtë problem. Sidoqoftë, për shkak të kompleksitetit të tyre programues dhe për disa qëllime praktike, Fibonnaci Heap nuk është gjithmonë një përmirësim i nevojshëm ose i rëndësishëm. Fibonnaci Heap rezervohet më së miri për aplikacionet në të cilat numri i fshirjeve është i vogël në krahasim me përdorimin e operacioneve të tjera.

2.4 Aplikimi

Shumë më shumë probleme sesa që në fillim mund të mendoni se mund të zgjidhen me këtë algoritëm, duke e bërë algoritmin e Dijkstra një mjet të fuqishëm dhe të përgjithshëm.

Për shembull:

- Algoritmi i Dijkstra zbatohet për të gjetur automatikisht drejtime në mes vendndodhjeve fizike, të tilla si drejtimet e udhëtimit në faqet e internetit si Mapquest ose Google Maps.

- OSPF – Open Shortest Path First, të përdorur në Internet Routing. Ajo përdor një link-state në zonat individuale që përbëjnë hierarkinë. Llogaritja bazohet në Algoritmin e Dijkstras i cili përdoret për të llogaritur pemën më të shkurtër të rrugës brenda secilës zonë të rrjetit(network).
- Agjenda e fluturimeve, veçanarishtë në plotësimin e databazës së një udhëtimi me drejtimin më të shkurtër, kohën më të hershme të arritjes etj.
- Përdoret gjithashtu për të zgjidhur një sërë problemesh të rrugës më të shkurtër që shfaqen tek shtrirja e bimëve dhe strukturave, robotika, transporti, dhe modelimi i VLSI.[11]

2.5 Pseudokodi

```
function Dijkstra(Graph, source):
    for each vertex v in Graph:
        dist[v] := infinity
        previous[v] := undefined
    dist[source] := 0
    Q := the set of all nodes in Graph
    while Q is not empty:
        u := node in Q with smallest dist[ ]
        remove u from Q
        for each neighbor v of u:
            alt := dist[u] + dist_between(u, v)
```

[11] Internet: https://medium.com/@farruk/practical-dijkstras-algorithm-b329ade79a1e

```
            if alt < dist[v]
                dist[v] := alt
                previous[v] := u
    return previous[ ]
```

Në pseudokodin e mësipërm, fillimisht inicializohet distanca e nyjeve “infinit”, nyja paraprake e padefinuar, si dhe inicializimi i distances së burimit me 0, Q është seti i gjitha nyjeve në Graf ,u kulm kërkon për kulmin u në setin e nyjeve të Q që ka vlerën më të vogël të [u]. Gjatësia (u, v) kthen gjatësinë e bashkimit të skajit (d.m.th. distancën midis) dy nyjeve fqinje u dhe v. Variabla Alt është gjatësia e shtegut nga nyja fillestare (burimi) në nyjen fqinje v, nëse do të kalonte në nyjen u. Nëse kjo rrugë është më e shkurtër se rruga aktuale më e shkurtër e regjistruar për v, ajo rrugë aktuale zëvendësohet me këtë shteg alt.

2.6 Bllokdiagrami i Dijkstras

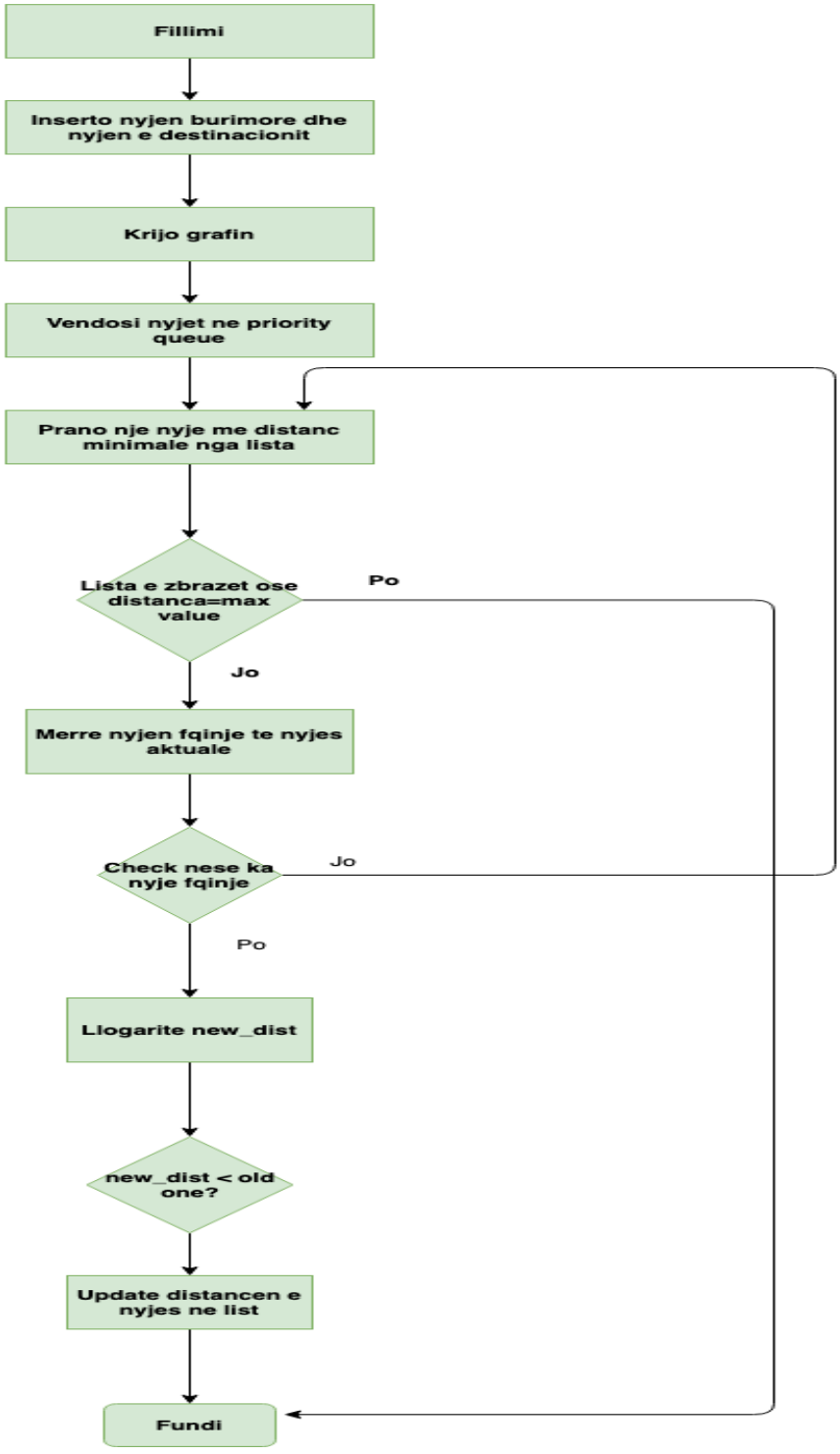


Figura 10: Bllok diagrami

2.7 Implementimi ne Java

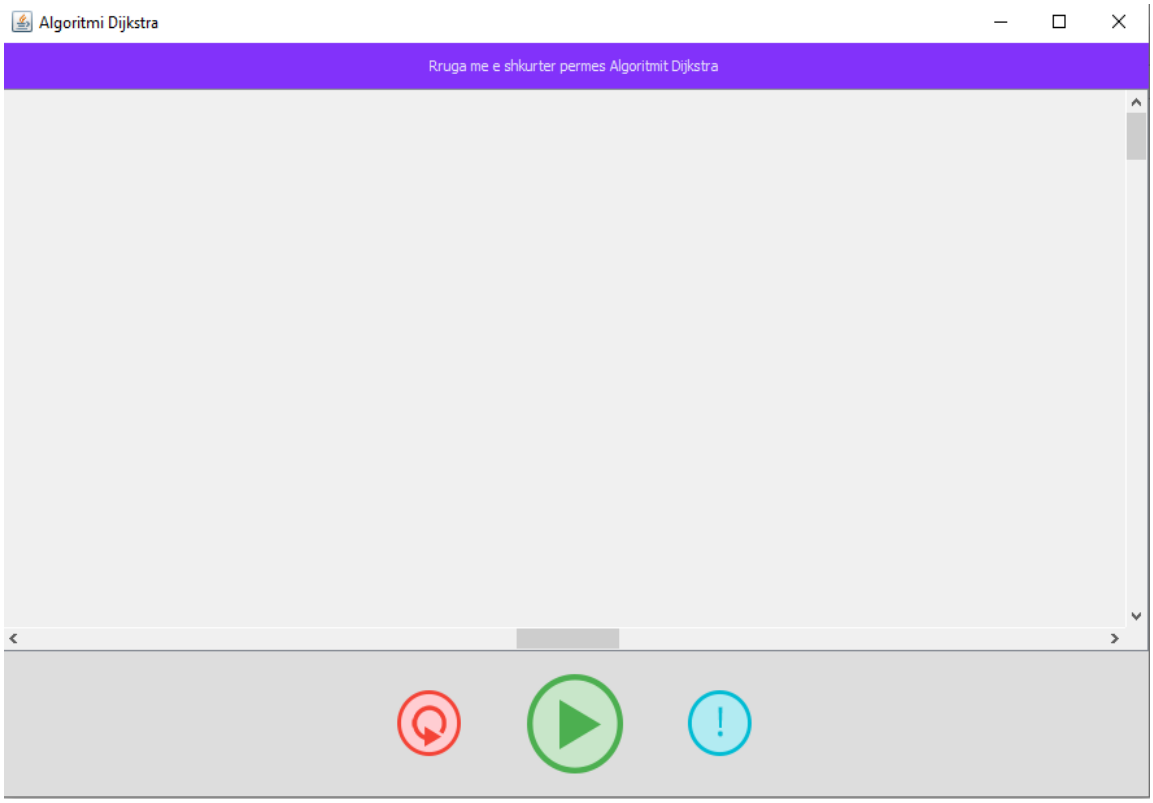
```
public class DijkstraAlgorithm {
    private boolean safe = false;
    private String message = null;

    private Graph graph;

    private Map<Node, Node> predecessors;
    private Map<Node, Integer> distances;

    private PriorityQueue<Node> unvisited;
    private HashSet<Node> visited;

    public class NodeComparator implements Comparator<Node>  {

       @Override
         public int compare(Node node1, Node node2) {
              int distanca = distances.get(node1) - distances.get(node2);
             return distanca;
         }
    };

    public DijkstraAlgorithm(Graph graph){
        this.graph = graph;
        predecessors = new HashMap<>();
        distances = new HashMap<>();
        for(Node node : graph.getNodes()){
            distances.put(node, Integer.MAX_VALUE);
        }

        visited = new HashSet<>();
safe = evaluate();
    }
    // Validimet
    private boolean evaluate(){
        if(graph.getSource()==null){
            message = "Nyja burimore duhet te jete prezente ne graf";
            return false;
        }

        if(graph.getDestination()==null){
            message = "Nyja e destinacionit duhet te jete prezente ne graf";
            return false;
        }

        for(Node node : graph.getNodes()){
            if(!graph.isNodeReachable(node)){
                message = "Secila nyje duhet te jete pjese e grafit";
                return false;
            }
```

```
        }

        return true;
    }

    public void run() throws IllegalStateException {
        if(!safe) {
            throw new IllegalStateException(message);
        }
        unvisited = new PriorityQueue<>(graph.getNodes().size(), neë
NodeComparator());

        Node source = graph.getSource();
        distances.put(source, 0);
        visited.add(source);


        for (Edge neighbor : getNeighbors(source)){
            Node adjacent = getAdjacent(neighbor, source);
            distances.put(adjacent, neighbor.getWeight());
            predecessors.put(adjacent, source);
            unvisited.add(adjacent);
        }

        while (!unvisited.isEmpty()){
            Node current = unvisited.poll();
            updateDistance(current);
            unvisited.remove(current);
            visited.add(current);

        }

        for(Node node : graph.getNodes()) {
            node.setPath(getPath(node));
        }

        graph.setSolved(true);

    }

    private void updateDistance(Node node){
        int distance = distances.get(node);
        for (Edge neighbor : getNeighbors(node)){
            Node adjacent = getAdjacent(neighbor, node);
            if(visited.contains(adjacent))
                continue;


            int current_dist = distances.get(adjacent);

            int new_dist = distance + neighbor.getWeight();

            if(new_dist < current_dist) {
                distances.put(adjacent, new_dist);
                predecessors.put(adjacent, node);
```

```
                unvisited.add(adjacent);
            }
        }
    }

    // Metoda per me e marr nyjen fqinje nyjes me degen perkatese qe vjen si input
    private Node getAdjacent(Edge edge, Node node) {
        if(edge.getNodeOne()!=node && edge.getNodeTwo()!=node)
            return null;

        return node==edge.getNodeTwo()?edge.getNodeOne():edge.getNodeTwo();
    }
    // Metoda per me i marr deget fqinje te nyjes perkatese qe vjen si input
    private List<Edge> getNeighbors(Node node) {
        List<Edge> neighbors = new ArrayList<>();

        for(Edge edge : graph.getEdges()){
            if(edge.getNodeOne()==node ||edge.getNodeTwo()==node)
                neighbors.add(edge);
        }

        return neighbors;
    }

    // Metoda per me marr distacen e nyjes
    public Integer getDistance(Node node){
        return distances.get(node);
    }

    public List<Node> getDestinationPath() {
        return getPath(graph.getDestination());
    }


    public List<Node> getPath(Node node){
        List<Node> path = new ArrayList<>();

        Node current = node;
        path.add(current);
        while (current!=graph.getSource()){
            current = predecessors.get(current);
            path.add(current);
        }

        return path;
    }

}
```

2.8 Aplikacioni

Aplikacioni është implementuar në gjuhën programuese Java. Në vazhdim shohim se si aplikacioni duket.



Në aplikacion përdoruesi duhët të caktoj nyjën fillestare dhe atë të destinacionit. Pastaj duhët të formoj grafin, pra të krijoj nyjët dhe degët tjera. Bydefault të gjitha degët kanë peshë fillestare 1, të cilën shfrytëzuesi mund ta ndryshoj. Kemi tre butona: reset, execute dhe info. Nese klikojm në butonin reset athëherë grafi i krijuar fshihet për tu krijuar prap, butoni execute e gjenë rrugën më të shkurtër mes nyjës se burimit dhe destinacionit ndërsa butoni info shfaq një modul që tregon se si të krijohet grafi, si të bëhet fshirja e një nyje dhe disa informacione tjera me rëndësi.

Referencat

https://en.wikipedia.org/wiki/Dijkstra's_algorithm

https://en.wikipedia.org/wiki/Priority_queue

https://people.maths.bris.ac.uk/~csxam/teaching/dsa/dijkstra.pdf

https://www.cs.dartmouth.edu/~thc/cs10/lectures/0509/0509.html

https://blog.aos.sh/2018/02/24/understanding-dijkstras-algorithm/

https://kbaile03.github.io/projects/fibo_dijk/fibo_dijk.html

https://www.codingame.com/playgrounds/1608/shortest-paths-with-dijkstras-algorithm/dijkstras-algorithm

https://www.coursera.org/lecture/algorithms-part2/dijkstras-algorithm-2e9Ic

https://www.baeldung.com/java-graphs

http://www1.cs.columbia.edu/~bert/courses/3137/hw3_files/GraphDraw.java

https://docs.oracle.com/javase/tutorial/2d/advanced/quality.html

https://docs.oracle.com/javase/7/docs/api/java/awt/event/MouseMotionListener.html

https://stackoverflow.com/questions/1094539/how-to-draw-a-decent-looking-circle-in-java