Lecture29
Friday, 3 June 2022 2:47 PM

graphs - maps, localnetwork (facebook)

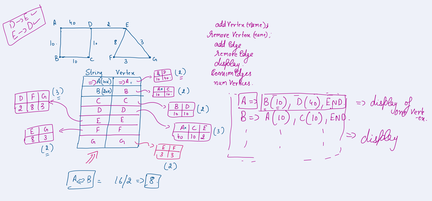
path.
shortest path & cost.
A->B A->B

{ (cost - number of lines)
→ weight of line → string
→ bidirectional → cost

A->G => ABCDEG
A->B => 8

Vertex => A, B, C, D, E, F, G
Edges => AB, BC, CD, DE, EF, FG, EG.

class Graph {
class Vertex {
String name; //A
HashMap<Vertex, Integer> nbrs; // — B | 8
key, value
Vertex (String) {
this.name = name;
this.nbrs = new hashmap<>();
}
display() {
System.out.println("a");
for(Vertex nbr : vtx.nbrs.keyset(); -> BAD
for(Vertex nbr : nbrs){
System.out.print(name + "[" + vtx.nbrs.get(nbr) + "]" ,
}
System.out.println("END");
}
}
Hashmap<String, Vertex> vtces;
Graph() {
this.vtces = new hashmap<>();
}



public int numVtces() {
return this.vtces.size();
}

public void addVertex(String name) {
if (vtces.containsKey(name)) {
return;
}
Vertex vtx = new Vertex(name);
this.vtces.put(name, vtx);
}

public void removeVertex(String name) {
if (!vtces.containsKey(name)) {
return;
}
Vertex vtx = this.vtces.get(name); //B
Set<Vertex> nbrs = vtx.nbrs.keyset();
for (Vertex nbr : nbrs) {
nbr.nbrs.remove(vtx);
}
this.vtces.remove(name);
}

public int numEdges() {
int ans = 0;
Collection<Vertex> vtces = this.vtces.values();
for (Vertex vtx : vtces) {
ans = ans + vtx.nbrs.size();
}
ans = ans/2;
return ans;
}

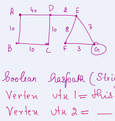
public void addEdge(String name1, String name2,
int cost) {
Vertex vtx1 = this.vtces.get(name1);
Vertex vtx2 = — — — (name2);] =>
if (vtx1 == null || vtx2 == null || vtx1.nbrs.containsKey(vtx2)) {
return;
}
vtx1.nbrs.put(vtx2, cost);
vtx2 — — put(vtx1, cost);
}

public void removeEdge( ) {

copy.

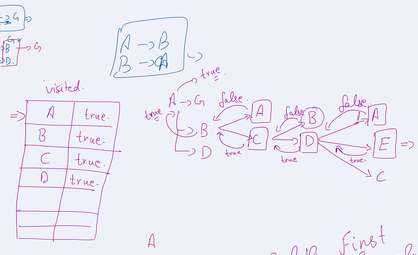
! vtx1.— containsKey(vtx2) {
return;
}
-> hashmap se remove. -> vtx1.—(vtx2),
vtx2 — —(vtx1)
}

public void display() {
Collection<Vertex> vtces = this.vtces.values();
for (Vertex vtx : vtces) {
vtx.display(); =>
}
}



public boolean hasPath(String name1, String name2) {
"A" "G"
DE -> true
DE -> x -> false.
Vertex vtx1 = this.vtces.get(name1);
Vertex vtx2 = — — get(name2);
if (vtx1 == null || vtx2 == null) {
return false;
}
HashMap<Vertex, Boolean> visited = new HashMap<>();
hasPath(vtx1, vtx2);
}
Recursion
private boolean hasPathRecursive(Vertex vtx1, Vertex vtx2, visited)
if (!visited.containsKey(vtx1)) {
visited.put(vtx1, true);
} else {
return false;
}
if (vtx1.nbrs.containsKey(vtx2)) {
return true;
}
Set<Vertex> nbrs = vtx1.nbrs.keyset();
for (Vertex nbr : nbrs) {
if (this.hasPath(nbr, vtx2)) {
return true;
}
}
return false;
}

A<=>G =>



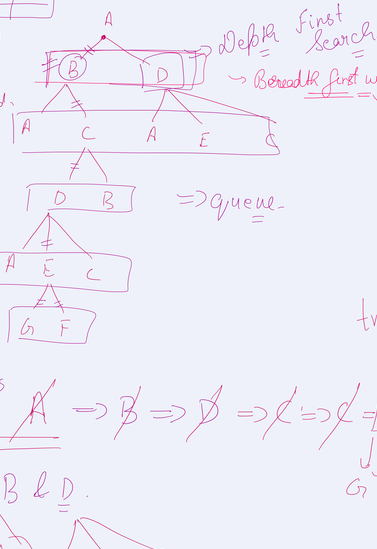
visited.

A	true.
B	true.
C	true.
D	true.

public boolean hasPathIterative(vtx1, vtx2, visited).
Queue<Vertex> q = new LL<>();
q.add(vtx1);
while(!q.isEmpty()) {
Vertex rv = q.remove();
if(!visited.containsKey(rv)) {
visited.put(rv, true);
if(rv.nbrs.contains(vtx2)) {
return true;
}
-> get nbrs of rv.
for (Vertex nbr : nbrs) {
if (!visited.containsKey(nbr)) {
q.add(nbr);
}
}
}
return false;
}

A	true.
B	true.
D	true
C	true
E	true.

Depth First Search.
-> Breadth first way



- dfs
- bfs
- Disconnected
- Connected Components
- Dijkstra's

=> tomorrow