GRAPHS

SHORTEST PATHS

Ruben Acuña

Spring 2022

EDGE-WEIGHTED DIGRAPH DATA TYPES

WEIGHTED GRAPHS

- Until this point, we've only talked about graphs in terms of vertices being connected.
- When we looked at using BFS to find the shortest path in the graph, we defined our shortest path in numbers of the number of edges.

- While this is useful, there is a more important generalization: weighted graphs.
- In a weighted graph, each node has an attributed, typically called weight, that associates a number with that edge. This value is intended to represent how costly an edge is to take in a particular path.
- So, the graphs we look at previously would have had a weight of 1 on all edges.

EDGES WITH ATTRIBUTES

Our main extension will be to create a class to hold the attributes of each edge.

Note that a decent number of graph libraries won't have "from" or "to", they will just store data.

public class	DirectedEdge	
	DirectedEdge(int v, int w, double weight)	
double	weight()	*weight of this edge*
int	from()	*vertex this edge points from*
int	to()	*vertex this edge points to*
String	toString()	*string representation*

ANNOTATING EDGES

- Nothing surprising here.
- There are two general approaches to annotating edges: 1) adding attributes directly to an edge class. 2) Making edge classes support generics or contain a hashtable.

```
public class DirectedEdge {
    private final int v;
    private final int w;
    private final double weight;

    public DirectedEdge(int v, int w, double weight) {
        this.v = v;
        this.w = w;
        this.weight = weight;
    }

    public double weight() {
        return weight;
    }

    public int from() {
        return v;
    }

    public int to() {
        return w;
    }

    public String toString() {
        return String.format("%d->%d %.2f", v, w, weight);
    }
}
```

REVISITING THE DIRECTED GRAPH ADT

Let's go back and update the ADT. For the most part, the functionality will stay the same but use DirectedEdge instead of just an Integer.

public class	EdgeWeightedDigraph	
	EdgeWeightedDigraph(int V)	*empty V-vertex digraph*
	EdgeWeightedDigraph(In in)	*construct from* in
int	V()	*number of vertices*
int	E()	*number of edges*
void	addEdge(DirectedEdge e)	*add* e *to this digraph*
Iterable<DirectedEdge>	adj(int v)	*edges pointing from* v
Iterable<DirectedEdge>	edges()	*all edges in this digraph*
String	toString()	*string representation*

WEIGHT GRAPH

- Similar to the previous digraph ADT – what's changed?

```
public class EdgeWeightedDigraph
{
    private final int V; // number of vertices
    private int E; // number of edges
    private LinkedList<DirectedEdge>[] adj; // adjacency lists

    public EdgeWeightedDigraph(int V) {
        this.V = V;
        this.E = 0;
        adj = (LinkedList<DirectedEdge>[]) new LinkedList[V];
        for (int v = 0; v < V; v++)
            adj[v] = new LinkedList<>();
    }

    public int V() { return V; }

    public int E() { return E; }

    public void addEdge(DirectedEdge e) {
        adj[e.from()].add(e);
        E++;
    }

    public Iterable<DirectedEdge> adj(int v) {
        return adj[v];
    }

    public Iterable<DirectedEdge> edges() {
        LinkedList<DirectedEdge> ll = new LinkedList<>();
        for (int v = 0; v < V; v++)
            for (DirectedEdge e : adj[v])
                ll.add(e);
        return ll;
    }
}
```

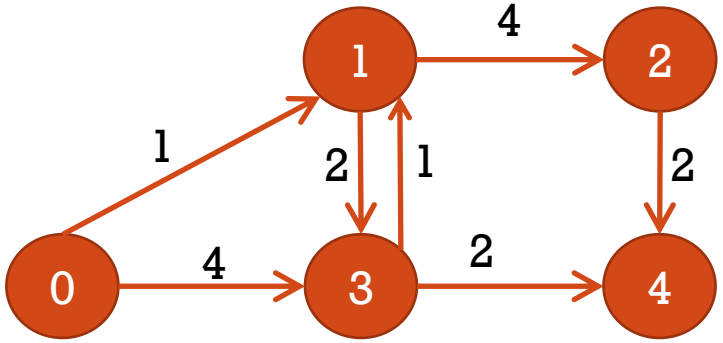
DIJKSTRA'S ALGORITHM

A SHORTEST PATH

- A path between two nodes A and B is a shortest path if there is no other path between them where the weights sum to lower value.

- How many paths are there in the graph? Say from 0 to 4.

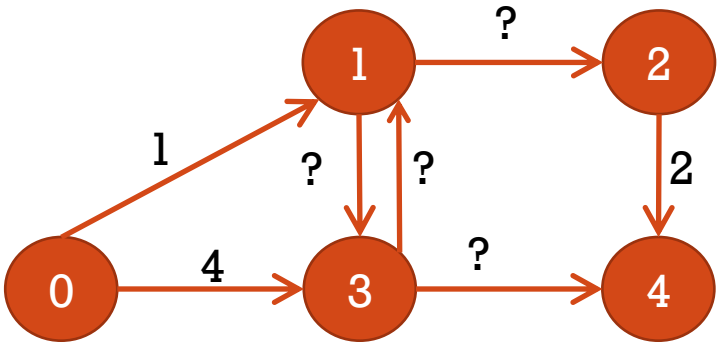
- What's the shortest path from 0 to 4?



Our end result will be a "distTo" value for each node in the graph, that contains the length of the shortest path to get to that node. We will also maintain the edgeTo data that tells us how we could get to a particular node.

A SHORTEST PATH

- A quick thought: are there any graphs we might have trouble finding a shortest path for? Suppose of course that some path does exist.



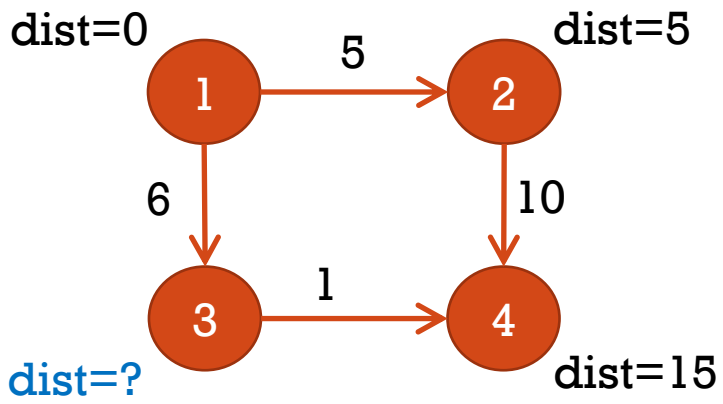
RELAXATION

Basic idea:

- Pick a node.
- If it provides a better (shorter) way to reach any of its neighbors, then update the neighbors

```
private void relax(EdgeWeightedDigraph G, int v) {
        for(DirectedEdge e : G.adj(v)) {
            int w = e.to();
            if(distTo[w] > distTo[v] + e.weight()) {
                distTo[w] = distTo[v] + e.weight();
                edgeTo[w] = e;
            }
        }
    }
```

RELAXATION



DIJKSTRA'S ALGORITHM

- We need to relax each node in a graph. The issue is that we need to relax them in an order such that we discover the shortest paths.

- Idea:
- Maintain a list of all nodes in the graph.
- Pick out the node with the minimum distTo.*
- Relax that node.
- Remove that node from the list and go to pick out new node.

- *Could also use a priority queue.

CODE

```
private DirectedEdge[] edgeTo;
private double[] distTo;
private LinkedList<Integer> nodes;

public Dijkstra(EdgeWeightedDigraph G, int s) {
    edgeTo = new DirectedEdge[G.V()];
    distTo = new double[G.V()];

    for (int v = 0; v < G.V(); v++) {
        distTo[v] = Double.POSITIVE_INFINITY;
        nodes.add(v);
    }
    distTo[s] = 0.0;

    while(!nodes.isEmpty()) {

        int v = nodes.getFirst();   //find node with min dist
        for(int n : nodes)
            if(distTo[n] < distTo[v])
                v = n;

        nodes.remove(new Integer(v));

        relax(G, v);
    }
}

private void relax(EdgeWeightedDigraph G, int v) {
    for(DirectedEdge e : G.adj(v)) {
        int w = e.to();
        if(distTo[w] > distTo[v] + e.weight()) {
            distTo[w] = distTo[v] + e.weight();
            edgeTo[w] = e;
        }
    }
}
```

ALGORITHM TRACE

- Run Dijkstra's algorithm on this graph to find all the shortest paths.