Shortest Paths

Outline

Shortest Paths

Shortest Paths

A shortest path from vertex s to vertex t in an edge-weighted digraph is a directed path from s to t with the property that no other such path has a lower weight

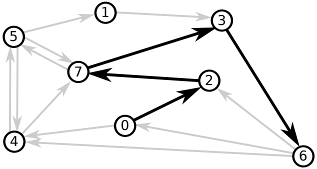
Shortest Paths

A shortest path from vertex *s* to vertex *t* in an edge-weighted digraph is a directed path from *s* to *t* with the property that no other such path has a lower weight

An edge-weighted graph and a shortest path

```
>_ ~/workspace/dsa/programs

$ more ../data/tinyEWD.txt
8
15
4 5 0.35
5 4 0.35
4 7 0.37
5 7 0.28
7 5 0.28
5 1 0.32
0 4 0.38
0 2 0.26
7 3 0.39
1 3 0.29
2 7 0.34
6 2 0.40
3 6 0.52
6 0 0.58
6 4 0.93
```



shortest path from 0 to 6

```
0 -> 2    0.26
2 -> 7    0.34
7 -> 3    0.39
3 -> 6    0.52
```

Shortest Paths

Shortest Paths

Variants: single source, single sink, source-sink, all pairs

Shortest Paths

Variants: single source, single sink, source-sink, all pairs

Typical shortest-paths applications

Application	Vertex	Edge
map	intersection	road
network	router	connection
schedule	job	precedence constraint
arbitrage	currency	exchange rate

Edge-Weighted Digraph API

Edge-Weighted Digraph API

EdgeWeightedDiGraph	
`EdgeWeightedDiGraph(int V)`	edge-weighted digraph with V vertices
`EdgeWeightedDiGraph(In in)`	edge-weighted digraph from input stream
`void addEdge(DirectedEdge e)`	add weighted directed edge e
`Iterable<DirectedEdge> adj(int v)`	edges pointing from v
`int V()`	number of vertices
`int E()`	number of edges

Edge-Weighted Digraph API

EdgeWeightedDiGraph

`EdgeWeightedDiGraph(int V)`	edge-weighted digraph with V vertices
`EdgeWeightedDiGraph(In in)`	edge-weighted digraph from input stream
`void addEdge(DirectedEdge e)`	add weighted directed edge e
`Iterable<DirectedEdge> adj(int v)`	edges pointing from v
`int V()`	number of vertices
`int E()`	number of edges

DiEdge

`DiEdge(int v, int w, double weight)`	create a directed weighted edge v-w
`int from()`	vertex this edge points from
`int to()`	vertex this edge points to
`double weight()`	weight of this edge

Edge-Weighted Digraph API

Edge-Weighted Digraph API

EdgeWeightedDigraph.java

```java
package dsa;

import stdlib.In;
import stdlib.StdOut;

public class EdgeWeightedDiGraph {
    private LinkedBag<DiEdge>[] adj;
    private int V;
    private int E;

    public EdgeWeightedDiGraph(int V) {
        adj = (LinkedBag<DiEdge>[]) new LinkedBag[V];
        for (int v = 0; v < V; v++) {
            adj[v] = new LinkedBag<DiEdge>();
        }
        this.V = V;
        this.E = 0;
    }

    public EdgeWeightedDiGraph(In in) {
        this(in.readInt());
        adj = (LinkedBag<DiEdge>[]) new LinkedBag[V];
        for (int v = 0; v < V; v++) {
            adj[v] = new LinkedBag<DiEdge>();
        }
        int E = in.readInt();
        for (int i = 0; i < E; i++) {
            int v = in.readInt();
            int w = in.readInt();
            double weight = in.readDouble();
            addEdge(new DiEdge(v, w, weight));
        }
    }

    public int V() {
```

Edge-Weighted Digraph API

EdgeWeightedDigraph.java

```java
        return V;
    }

    public int E() {
        return E;
    }

    public void addEdge(DiEdge e) {
        int v = e.from();
        int w = e.to();
        adj[v].add(e);
        E++;
    }

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

    public int outDegree(int v) {
        return adj[v].size();
    }

    public int inDegree(int v) {
        int inDegree = 0;
        for (LinkedBag<DiEdge> bag : adj) {
            for (DiEdge e : bag) {
                inDegree += e.to() == v ? 1 : 0;
            }
        }
        return inDegree;
    }

    public Iterable<DiEdge> edges() {
        LinkedBag<DiEdge> edges = new LinkedBag<DiEdge>();
        for (int v = 0; v < V; v++) {
```

Edge-Weighted Digraph API

EdgeWeightedDigraph.java

```java
            for (DiEdge e : adj(v)) {
                edges.add(e);
            }
        }
        return edges;
    }

    public String toString() {
        StringBuilder s = new StringBuilder();
        s.append(V + " " + E + "\n");
        for (int v = 0; v < V; v++) {
            s.append(v + ": ");
            for (DiEdge e : adj[v]) {
                s.append(e + "  ");
            }
            s.append("\n");
        }
        return s.toString().strip();
    }

    public static void main(String[] args) {
        In in = new In(args[0]);
        EdgeWeightedDiGraph G = new EdgeWeightedDiGraph(in);
        StdOut.println(G);
    }
}

class DiEdge {
    private int v;
    private int w;
    private double weight;

    public DiEdge(int v, int w, double weight) {
        this.v = v;
        this.w = w;
```

Edge-Weighted Digraph API

EdgeWeightedDigraph.java

```java
        this.weight = weight;
    }

    public int from() {
        return v;
    }

    public int to() {
        return w;
    }

    public double weight() {
        return weight;
    }

    public String toString() {
        return v + "->" + w + " " + String.format("%5.2f", weight);
    }

    public static void main(String[] args) {
        DiEdge e = new DiEdge(12, 34, 5.67);
        StdOut.println(e);
    }
}
```

Shortest Path API

Shortest Path API

Single-source shortest paths API

Dijkstra

`Dijkstra(EdgeWeightedDigraph G, int s)`	constructor
`double distTo(int v)`	distance from s to v, ∞ if no path
`boolean hasPathTo(int v)`	path from s to v?
`Iterable<DirectedEdge> pathTo(int v)`	path from s to v, `null` if none

Shortest Path API

Single-source shortest paths API

Dijkstra	
`Dijkstra(EdgeWeightedDigraph G, int s)`	constructor
`double distTo(int v)`	distance from s to v, ∞ if no path
`boolean hasPathTo(int v)`	path from s to v?
`Iterable<DirectedEdge> pathTo(int v)`	path from s to v, `null` if none

SP test client

```
public class Dijkstra {
    public static void main(String[] args) {
       In in = new In(args[0]);
        EdgeWeightedDiGraph G = new EdgeWeightedDiGraph(in);
        int s = Integer.parseInt(args[1]);
        Dijkstra sp = new Dijkstra(G, s);
        for (int t = 0; t < G.V(); t++) {
            if (sp.hasPathTo(t)) {
                StdOut.printf("%d to %d (%.2f)  ", s, t, sp.distTo(t));
                if (sp.hasPathTo(t)) {
                    for (DiEdge e : sp.pathTo(t)) {
                        StdOut.print(e + "   ");
                    }
                }
                StdOut.println();
            }
            else { StdOut.printf("%d to %d (no path)\n", s, t); }
        }
    }
}
```

Shortest Path API

Shortest Path API

```
>_ ~/workspace/dsa/programs

$ java dsa.Dijkstra ../data/tinyEWD.txt 0
0 to 0 (0.00):
0 to 1 (1.05): 0->4 0.38 4->5 0.35 5->1 0.32
0 to 2 (0.26): 0->2 0.26
0 to 3 (0.99): 0->2 0.26 2->7 0.34 7->3 0.39
0 to 4 (0.38): 0->4 0.38
0 to 5 (0.73): 0->4 0.38 4->5 0.35
0 to 6 (1.51): 0->2 0.26 2->7 0.34 7->3 0.39 3->6 0.52
0 to 7 (0.60): 0->2 0.26 2->7 0.34
```

Shortest Path API

Shortest Path API

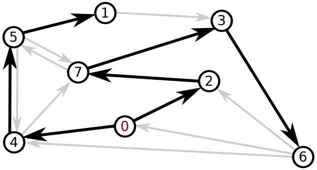
A shortest-paths tree solution (SPT) always exists

Shortest Path API

A shortest-paths tree solution (SPT) always exists

Data structures: can represent the SPT with two vertex-indexed arrays

- `distTo[v]` is length of shortest path from *s* to *v*
- `edgeTo[v]` is last edge on shortest path from *s* to *v*



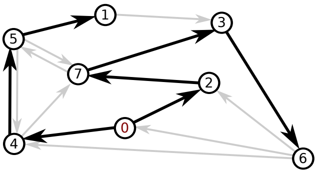
	edgeTo[]		distTo[]
0	null		0
1	5 -> 1	0.32	1.05
2	0 -> 2	0.26	0.26
3	7 -> 3	0.37	0.97
4	0 -> 4	0.38	0.38
5	4 -> 5	0.35	0.73
6	3 -> 6	0.52	1.49
7	2 -> 7	0.34	0.60

Shortest Path API

A shortest-paths tree solution (SPT) always exists

Data structures: can represent the SPT with two vertex-indexed arrays

- `distTo[v]` is length of shortest path from s to v
- `edgeTo[v]` is last edge on shortest path from s to v



	edgeTo[]	distTo[]
0	null	0
1	5 -> 1 0.32	1.05
2	0 -> 2 0.26	0.26
3	7 -> 3 0.37	0.97
4	0 -> 4 0.38	0.38
5	4 -> 5 0.35	0.73
6	3 -> 6 0.52	1.49
7	2 -> 7 0.34	0.60

Edge relaxation: relax edge $e = v \to w$

- `distTo[v]` is length of shortest known path from s to v
- `distTo[w]` is length of shortest known path from s to w
- `edgeTo[w]` is last edge on shortest known path from s to w
- if $e = v \to w$ gives shorter path to w through v, update both `distTo[w]` and `edgeTo[w]`

Shortest Path API

Shortest Path API

Edge relaxation (implementation)

```
private void relax(DiEdge e) {
    int v = e.from(), w = e.to();
    if (distTo[w] > distTo[v] + e.weight()) {
        distTo[w] = distTo[v] + e.weight();
        edgeTo[w] = e;
    }
}
```

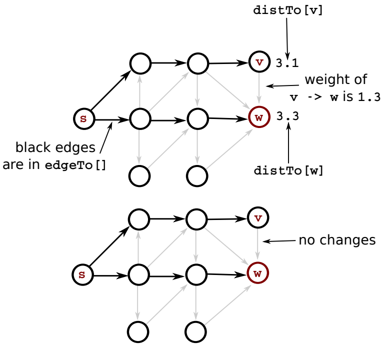
Shortest Path API

Edge relaxation (implementation)

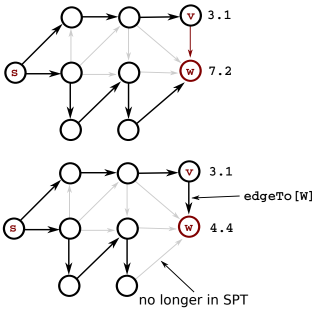
```
private void relax(DiEdge e) {
    int v = e.from(), w = e.to();
    if (distTo[w] > distTo[v] + e.weight()) {
        distTo[w] = distTo[v] + e.weight();
        edgeTo[w] = e;
    }
}
```

Edge relaxation (two cases)

`v -> w` is ineligible



`v -> w` is eligible



Dijkstra's Algorithm

Dijkstra's Algorithm

Dijkstra's algorithm computes a SPT in any edge-weighted digraph with nonnegative weights, as follows

- Considers vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value)
- Adds vertex to tree and relaxes all edges pointing from that vertex

Dijkstra's Algorithm

Dijkstra's algorithm computes a SPT in any edge-weighted digraph with nonnegative weights, as follows

- Considers vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value)
- Adds vertex to tree and relaxes all edges pointing from that vertex

Dijkstra's algorithm using a binary heap based priority queue computes a SPT in an edge-weighted digraph in time proportional to $E \log V$ in the worst case

Dijkstra's Algorithm

Dijkstra's Algorithm

Dijkstra.java

```java
package dsa;

import stdlib.In;
import stdlib.StdOut;

public class Dijkstra {
    private int s;
    private DiEdge[] edgeTo;
    private double[] distTo;
    private IndexMinPQ<Double> pq;

    public Dijkstra(EdgeWeightedDiGraph G, int s) {
        this.s = s;
        edgeTo = new DiEdge[G.V()];
        distTo = new double[G.V()];
        for (int v = 0; v < G.V(); v++) {
            distTo[v] = Double.POSITIVE_INFINITY;
        }
        distTo[s] = 0.0;
        pq = new IndexMinPQ<Double>(G.V());
        pq.insert(s, distTo[s]);
        while (!pq.isEmpty()) {
            int v = pq.delMin();
            for (DiEdge e : G.adj(v)) {
                relax(e);
            }
        }
    }

    public boolean hasPathTo(int v) {
        return distTo[v] < Double.POSITIVE_INFINITY;
    }

    public Iterable<DiEdge> pathTo(int v) {
        if (!hasPathTo(v)) {
```

Dijkstra's Algorithm

Dijkstra.java

```java
            return null;
        }
        LinkedStack<DiEdge> path = new LinkedStack<DiEdge>();
        for (DiEdge e = edgeTo[v]; e != null; e = edgeTo[e.from()]) {
            path.push(e);
        }
        return path;
    }

    public double distTo(int v) {
        return distTo[v];
    }

    private void relax(DiEdge e) {
        int v = e.from(), w = e.to();
        if (distTo[w] > distTo[v] + e.weight()) {
            edgeTo[w] = e;
            distTo[w] = distTo[v] + e.weight();
            if (pq.contains(w)) {
                pq.change(w, distTo[w]);
            } else {
                pq.insert(w, distTo[w]);
            }
        }
    }

    public static void main(String[] args) {
        In in = new In(args[0]);
        int s = Integer.parseInt(args[1]);
        EdgeWeightedDiGraph G = new EdgeWeightedDiGraph(in);
        Dijkstra sp = new Dijkstra(G, s);
        for (int t = 0; t < G.V(); t++) {
            if (sp.hasPathTo(t)) {
                StdOut.printf("%d to %d (%.2f): ", s, t, sp.distTo(t));
                for (DiEdge e : sp.pathTo(t)) {
```

Dijkstra's Algorithm

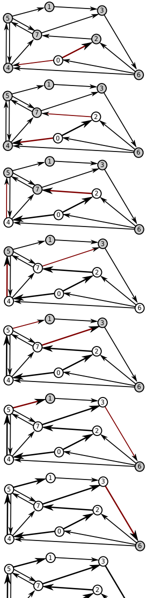
Dijkstra.java

```
                    StdOut.print(e + "    ");
                }
                StdOut.println();
            } else {
                StdOut.printf("%d to %d: not connected\n", s, t);
            }
        }
    }
}
```

Dijkstra's Algorithm

Dijkstra's Algorithm

Trace



red: on pq

	edgeTo[]	distTo[]
0		0.00
1		
2	0 -> 2 0.26	0.26
3		
4	0 -> 4 0.38	0.38
5		
6		
7		

index

priority

black: on SPT

	edgeTo[]	distTo[]
0		0.00
1		
2	0 -> 2 0.26	0.26
3		
4	0 -> 4 0.38	0.38
5		
6		
7	2 -> 7 0.34	0.60

	edgeTo[]	distTo[]
0		0.00
1		
2	0 -> 2 0.26	0.26
3		
4	0 -> 4 0.38	0.38
5	4 -> 5 0.35	0.73
6		
7	2 -> 7 0.34	0.60

	edgeTo[]	distTo[]
0		0.00
1		
2	0 -> 2 0.26	0.26
3	7 -> 3 0.37	0.97
4	0 -> 4 0.38	0.38
5	4 -> 5 0.35	0.73
6		
7	2 -> 7 0.34	0.60

	edgeTo[]	distTo[]
0		0.00
1	5 -> 1 0.32	1.05
2	0 -> 2 0.26	0.26
3	7 -> 3 0.37	0.97
4	0 -> 4 0.38	0.38
5	4 -> 5 0.35	0.73
6		
7	2 -> 7 0.34	0.60

	edgeTo[]	distTo[]
0		0.00
1	5 -> 1 0.32	1.05
2	0 -> 2 0.26	0.26
3	7 -> 3 0.37	0.97
4	0 -> 4 0.38	0.38
5	4 -> 5 0.35	0.73
6	3 -> 6 0.52	1.49
7	2 -> 7 0.34	0.60

	edgeTo[]	distTo[]
0		0.00
1	5 -> 1 0.32	1.05
2	0 -> 2 0.26	0.26
3	7 -> 3 0.37	0.97
4	0 -> 4 0.38	0.38
5	4 -> 5 0.35	0.73
6	3 -> 6 0.52	1.49
7	2 -> 7 0.34	0.60

	edgeTo[]	distTo[]
0		0.00
1	5 -> 1 0.32	1.05
2	0 -> 2 0.26	0.26
3	7 -> 3 0.37	0.97
4	0 -> 4 0.38	0.38