BFS

Referências

- Undirected graphs (S&W),
- slides (S&W)
- Directed graphs (S&W)
- slides (S&W)

Vídeos

- Undirected graphs (S&W)
- Directed graphs (S&W)

Busca em largura

```
public class BreadFirstDirectedPaths {
    private static final int INFINITY = Integer.MAX_VALUE; // Hmm. Prefiro G.V()
    private boolean[] marked;  // marked[v] = is there an s->v path?
    private int[] edgeTo;      // edgeTo[v] = last edge on shortest s->v path
    private int[] distTo;      // distTo[v] = length of shortest s->v path

    // Computes the shortest path from {@code s} and every other vertex in graph G.
    public BreadthFirstDirectedPaths(Digraph G, int s) {
        marked = new boolean[G.V()];
        edgeTo = new int[G.V()];
        distTo = new int[G.V()];
        for (int v = 0; v < G.V(); v++)
            distTo[v] = INFINITY;
        bfs(G, s);
    }

    // BFS from single source
    private void bfs(Digraph G, int s) {
        Queue<Integer> q = new Queue<Integer>();
        marked[s] = true;
        distTo[s] = 0;
        q.enqueue(s);
        while (!q.isEmpty()) {
            int v = q.dequeue();
            for (int w : G.adj(v)) {
                if (!marked[w]) {
                    edgeTo[w] = v;
                    distTo[w] = distTo[v] + 1;
                    marked[w] = true;
                    q.enqueue(w);
                }
```

```
            }
        }
    }

    // is there a directed path from the source  s to vertex v?
    public boolean hasPathTo(int v) {
        return marked[v];
    }

    // returns the number of edges in a shortest path from the source {@code s}
    public int distTo(int v) {
        return distTo[v];
    }

    // returns a shortest path from s to v, or null  if no such path.
     public Iterable<Integer> pathTo(int v) {
        if (!hasPathTo(v)) return null;
        Stack<Integer> path = new Stack<Integer>();
        int x;
        for (x = v; x != s; x = edgeTo[x])
            path.push(x);
        path.push(s);
        return path;
    }
}
```

Classe DigraphEdge

public class	DirectedEdge
.	DirectedEdge(int v, int w, double weight)
double	weight() # peso do arco
int	from() # origem do arco
int	to() # destino do arco
String	toString() # representação como string

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

    // initializes a directed edge from vertex v to vertex w with
    // the given  weight.
    public DirectedEdge(int v, int w, double weight) {
        this.v = v;
        this.w = w;
        this.weight = weight;
```

```
    }

    // returns the tail vertex of the directed edge.
     public int from() {
        return v;
    }

    // returns the head vertex of the directed edge.
     public int to() {
        return w;
    }

    // returns the weight of the directed edge.
    public double weight() {
        return weight;
    }

    // returns a string representation of the directed edge.
    public String toString() {
        return v + "->" + w + " " + String.format("%5.2f", weight);
    }

    // unit tests the DirectedEdge} data type.
    public static void main(String[] args) {
        DirectedEdge e = new DirectedEdge(12, 34, 5.67);
        StdOut.println(e);
    }
}
```

Classe EdgeWeightedDigraph

public class	EdgeWeightedDigraph
.	EdgeWeightedDigraph(int V)
.	EdgeWeightedDigraph(In in)
int	V()
int	E()
void	addEdge(DirectedEdge e)
Iterable<DirectedEdge>	adj(int v)
Iterable<DirectedEdge>	edges()
String	toString() # representação como string

```
public class EdgeWeightedDigraph {
    private static final String NEWLINE = System.getProperty("line.separator");

    private final int V;                // number of vertices in this digraph
    private int E;                      // number of edges in this digraph
```

```
private Bag<DirectedEdge>[] adj;    // adj[v] = adjacency list for vertex v
private int[] indegree;             // indegree[v] = indegree of vertex v

// initializes an empty edge-weighted digraph with V vertices and 0 edges.
public EdgeWeightedDigraph(int V) {
    this.V = V;
    this.E = 0;
    this.indegree = new int[V];
    adj = (Bag<DirectedEdge>[]) new Bag[V];
    for (int v = 0; v < V; v++)
        adj[v] = new Bag<DirectedEdge>();
}

// initializes a random edge-weighted digraph with V vertices and E edges.
public EdgeWeightedDigraph(int V, int E) {
    this(V);
    for (int i = 0; i < E; i++) {
        int v = StdRandom.uniform(V);
        int w = StdRandom.uniform(V);
        double weight = 0.01 * StdRandom.uniform(100);
        DirectedEdge e = new DirectedEdge(v, w, weight);
        addEdge(e);
    }
}

// initializes an edge-weighted digraph from the specified input stream.
// the format is the number of vertices V,
// followed by the number of edges E,
// followed by E pairs of vertices and edge weights,
// with each entry separated by whitespace.
public EdgeWeightedDigraph(In in) {
    this(in.readInt());
    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 DirectedEdge(v, w, weight));
    }
}

// initializes a new edge-weighted digraph that is a deep copy of G.
public EdgeWeightedDigraph(EdgeWeightedDigraph G) {
    this(G.V());
    this.E = G.E();
    for (int v = 0; v < G.V(); v++)
        this.indegree[v] = G.indegree(v);
    for (int v = 0; v < G.V(); v++) {
        // reverse so that adjacency list is in same order as original
        Stack<DirectedEdge> reverse = new Stack<DirectedEdge>();
```

```
            for (DirectedEdge e : G.adj[v]) {
                reverse.push(e);
            }
            for (DirectedEdge e : reverse) {
                adj[v].add(e);
            }
        }
    }

    // returns the number of vertices in this edge-weighted digraph.
    public int V() {
        return V;
    }

    // returns the number of edges in this edge-weighted digraph.
    public int E() {
        return E;
    }

    // adds the directed edge e to this edge-weighted digraph.
    public void addEdge(DirectedEdge e) {
        int v = e.from();
        int w = e.to();
        adj[v].add(e);
        indegree[w]++;
        E++;
    }


    // returns the directed edges incident from vertex {@code v}.
    public Iterable<DirectedEdge> adj(int v) {
        return adj[v];
    }

    // returns the number of directed edges incident from vertex v.
    public int outdegree(int v) {
        return adj[v].size();
    }

    // returns the number of directed edges incident to vertex {@code v}.
    public int indegree(int v) {
        return indegree[v];
    }

    // returns all directed edges in this edge-weighted digraph.
    public Iterable<DirectedEdge> edges() {
        Bag<DirectedEdge> list = new Bag<DirectedEdge>();
        for (int v = 0; v < V; v++) {
            for (DirectedEdge e : adj(v)) {
                list.add(e);
```

```
            }
        }
        return list;
    }

    // returns a string representation of this edge-weighted digraph.
    public String toString() {
        StringBuilder s = new StringBuilder();
        s.append(V + " " + E + NEWLINE);
        for (int v = 0; v < V; v++) {
            s.append(v + ": ");
            for (DirectedEdge e : adj[v]) {
                s.append(e + "  ");
            }
            s.append(NEWLINE);
        }
        return s.toString();
    }
}
```