Grafos

Referências

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

Vídeos

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

Digrafos

Um **digrafo** (=*directed graph* ou *digraph*) consiste de um conjunto de **vértices** (*bolas*) e um conjunto de **arcos** (flechas).

Arcos

Um **arco** (*directed edges*) é um par ordenado de vértices.

Para cada arco `v-w` `v` é a **ponta inicial** e `w` é a ponta final.

Arcos anti-paralelos

Dois arcos são **anti-paralelos** se a ponta inicial de um é a ponta final do outros

TODO: Ilustração

Digrafos simétricos

Um digrafo é simétrico se cada um de seus arcos é anti-paralelo a outro.

Grau de entrada e saída

O **grau de entrada** de `v` é o número de arcos com ponta final em `v`.

O **grau de saída** de `v` é o número de arcos com ponta inicial em `v`.

TODO: Ilustração

Número de arcos

Um digrafo com `V` vértices tem no máximo $\texttt{V} \times (\texttt{V} - 1) = \mathrm{O}(\texttt{V}^2)$ arcos.

digrafo completo = todo par ordenado de vértices distintos é arco

digrafo denso = tem muitos arcos

digrafo esparso = tem poucos arcos

Especificação

Um digrafo pode ser especificado através de sua lista de arcos

```
d-f
b-d
a-c
b-e
e-f
a-b
```

Grafos

Um **grafo** é um digrafo simétrico.

TODO: Ilustração

Uma **aresta** (*edge*) é um par de arcos anti-paralelos.

Nota: para fazer programas, o que interessa mesmo são digrafos. A maneira natural de representar um grafo no computador é como um digrafo completo.

Digrafos no computador

Duas representações clássicas:

- matriz de adjacência (*adjacency matrix*)
- listas de adjacência (*adjacency list*)

Matriz de adjacência de digrafos

Matriz de adjacência de um digrafo tem linhas e colunas indexadas por vértices

```
adj[v][w] = true  se v-w é um arco
adj[v][w] = false em caso contrário
```

Espaço gasto é proporcional a $\texttt{V}^2$.

Representação tipicamente boa para digrafos densos.

Matriz de adjacência de grafos

Matriz de adjacência de um grafo tem linhas e colunas indexadas por vértices

```
adj[v][w] = true  se v-w é uma aresta
adj[v][w] = false em caso contrário
```

Espaço gasto é proporcional a V^2.

Representação tipicamente boa para grafos densos.

Matriz `adj` é simétrica;

Listas de adjacência

Na representação de um digrafo através de **listas de adjacência** tem-se, para cada vértice `v`, uma lista de vértices que são vizinhos de `v`.

```
0: 1, 2
1: 3
2: 1, 3
3:
```

Espaço gasto é proporcional a `V + E`

Representação tipicamente boa para digrafos espersos.

API

Agora, vejamos como o S&W tratam a representação de digrafos/grafos.

Digrafos (=*Digraphs*). Um grafo dirigido ou **digrafo** é um conjunto de vértices e uma coleção de **arestas dirigidas** ou **arcos** que são pares ordenados de vértices. Dizemos que um arco aponta do primeiro vértice no par para o segundo no par. Se `v-w` é um arco então dizemos que `v` é ponta inicial

Grafos (=*Graphs*). Um grafo é um conjunto de vértices e uma coleção de arestas que ligam pares de vértices. Usamos os inteiros `0` a `V-1` como nomes dos vértices de um grafo com `V` vértices.

`public class`	`Digraph` (`Graph`)	
-	`Graph(int V)`	cria um `V`-grafo sem arestas
-	`Graph(In in)`	lê um grafo a partir de `in`
int	`V()`	número de vértices
int	`E()`	número de arestas
void	`addEdge(int v, int w)`	insere a aresta `v-w`
Iterable	`adj(int v)`	vértices adjacentes a `v`
String	`toString()`	string representando o grafo
Digraph	`reverse()`	reverso do grafo

Classe `Digraph`

```
public class Digraph {
    private final int V;           // number of vertices in this digraph
    private int E;                 // number of edges in this digraph
    private Bag<Integer>[] adj;    // adj[v] = adjacency list for vertex v
    private int[] indegree;        // indegree[v] = indegree of vertex v

    public Digraph(int V) {
        this.V = V;
        this.E = 0;
        indegree = new int[V];
        adj = (Bag<Integer>[]) new Bag[V];
        for (int v = 0; v < V; v++) {
            adj[v] = new Bag<Integer>();
        }
    }

    public Digraph(In in) {
       \\
    }

    // Initializes a new digraph that is a deep copy of the specified digraph.
    public Digraph(Digraph G) {
        this(G.V());
        this.E = G.E();
        for (int v = 0; v < 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<Integer> reverse = new Stack<Integer>();
            for (int w : G.adj[v]) {
                reverse.push(w);
            }
            for (int w : reverse) {
                adj[v].add(w);
            }
        }
    }

    // return the number of vertices in this digraph
    public int V() {
        return V;
    }

    // return the number of edges in this digraph
    public int E() {
        return E;
    }

    // Adds the directed edge v→w to this digraph.
    public void addEdge(int v, int w) {
        adj[v].add(w);
        indegree[w]++;
        E++;
    }

    // returns the vertices adjacent from vertex {@code v} in this digraph.
    public Iterable<Integer> adj(int v) {
        validateVertex(v);
        return adj[v];
    }

    // returns the number of directed edges incident from vertex {@code 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];
    }


    // return the reverse of the digraph
    public Digraph reverse() {
        Digraph reverse = new Digraph(V);
```

```
    for (int v = 0; v < V; v++) {
        for (int w : adj(v)) {
            reverse.addEdge(w, v);
        }
    }
    return reverse;
}

// returns a string representation of the graph.
public String toString() {
    // StringBuilder s = new StringBuilder();
    String s = "";
    // s.append(V + " vertices, " + E + " edges " + NEWLINE);
    s += V + " vertices, " + E + " edges " + "\n";
    for (int v = 0; v < V; v++) {
        // s.append(String.format("%d: ", v));
        s += String.format("%d: ",v);
        for (int w : adj[v]) {
            // s.append(String.format("%d ", w));
            s += String.format("%d ",w);
        }
        // s.append(NEWLINE);
        s += "\n";
    }
    // return s.toString();
    return s;
}
```

Caminhos

Um **caminhos** num digrafo é qualquer sequência da forma $v_0 - v_1 - v_2 - \cdots - v_p$, onde $v_{k-1} - v_k$ é um arco para $k = 1, \ldots, p$.

O vértice v_0 é a **origem** ou **início** do caminho e o vértice e o vértice v_p é o seu **destino** ou **trérmino**.

Um caminhos é **simples** se não tem vértices repetidos.

Problema

Dados um digrafo `G` e dois vértices `s` e `t` decidir se existe uma caminho de `s` a `t`

`public class`	`Paths`	
-	`Paths(Graph G, int s)`	encontra caminhos a partir de s
boolean	`hasPath(int v)`	existe um caminho de `s` a `v`
Iterable	`pathTo(int v)`	caminho de `s` a `v` e `null` se não existir

Caminhos no computador

Uma maneira *compacta* para representar caminhos de um vértice a outros é uma *arborescência*.

Uma **arborescência** é um digrafo em que:

- existe exatamente um vértice com grau de entrada 0, a **raiz** da arborescência
- não existe vértice com grau de entrada maior que 1,
- cada um dos vértices é termino de uma caminho com origme na raiz.

Uma arborescência pode ser representada por um vetor de pais `edgeTo[]`, onde `edgeTo[v]` é o último vértice no caminho da origem até `v`.

```
private int[] edgeTo;
```

Assim, na nossa representação os caminhos estão "invertidos". O código a seguir imprime o caminho invertido da raiz `s` a `v`.

```
for (x = v; x != s; x = edgeTo[x])
    StdOut.print(x+"-");
StdOut.println(s);
```

DFS

A classe a seguir usa busca em profundidade (*Depth First Search* ou DFS) para encontrar caminhos de um vértice `s` a todos os mais vértice do grafo que são alcançaveis a partir de `s`.

```
public class DepthFirstDirectedPaths {
    private boolean[] marked;  // marked[v] = true if v is reachable from s
    private int[] edgeTo;      // edgeTo[v] = last vertex on path from s to v
    private final int s;       // source vertex
```

```
//  Computes a directed path from {@code s} to every other vertex in digraph {@code G}.
public DepthFirstDirectedPaths(Digraph G, int s) {
    marked = new boolean[G.V()];
    edgeTo = new int[G.V()];
    this.s = s;
    dfs(G, s);
}

private void dfs(Digraph G, int v) {
    marked[v] = true;
    for (int w : G.adj(v)) {
        if (!marked[w]) {
            edgeTo[w] = v;
            dfs(G, w);
        }
    }
}

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


// Returns a directed path from the source vertex {@code s} to vertex {@code v}, or
public Iterable<Integer> pathTo(int v) {
    if (!hasPathTo(v)) return null;
    Stack<Integer> path = new Stack<Integer>();
    for (int x = v; x != s; x = edgeTo[x])
        path.push(x);
    path.push(s);
    return path;
}

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

    int s = Integer.parseInt(args[1]);
    DepthFirstDirectedPaths dfs = new DepthFirstDirectedPaths(G, s);

    for (int v = 0; v < G.V(); v++) {
        if (dfs.hasPathTo(v)) {
            StdOut.printf("%d to %d:  ", s, v);
            for (int x : dfs.pathTo(v)) {
                if (x == s) StdOut.print(x);
                else        StdOut.print("-" + x);
            }
```

```
                StdOut.println();
            }

            else {
                StdOut.printf("%d to %d:  not connected\n", s, v);
            }

        }
    }

}
```

Apêndice

Classe `AdjMatrixDigraph`

Representação de digrafos através de matriz de adjacência.

```
public class AdjMatrixDigraph {
    private int V;
    private int E;
    private boolean[][] adj;

    // empty graph with V vertices
    public AdjMatrixDigraph(int V) {
        if (V < 0) throw new RuntimeException("Number of vertices must be nonnegative");
        this.V = V;
        this.E = 0;
        this.adj = new boolean[V][V];
    }

    // random graph with V vertices and E edges
    public AdjMatrixDigraph(int V, int E) {
        this(V);
        if (E < 0) throw new RuntimeException("Number of edges must be nonnegative");
        if (E > V*V) throw new RuntimeException("Too many edges");

        // can be inefficient
        while (this.E != E) {
            int v = StdRandom.uniform(V);
            int w = StdRandom.uniform(V);
            addEdge(v, w);
        }
    }

    // number of vertices and edges
    public int V() { return V; }
```

```
public int E() { return E; }


// add directed edge v->w
public void addEdge(int v, int w) {
    if (!adj[v][w]) E++;
    adj[v][w] = true;
}

// return list of neighbors of v
public Iterable<Integer> adj(int v) {
    return new AdjIterator(v);
}

// support iteration over graph vertices
private class AdjIterator implements Iterator<Integer>, Iterable<Integer> {
    private int v;
    private int w = 0;

    AdjIterator(int v) {
        this.v = v;
    }

    public Iterator<Integer> iterator() {
        return this;
    }

    public boolean hasNext() {
        while (w < V) {
            if (adj[v][w]) return true;
            w++;
        }
        return false;
    }

    public Integer next() {
        if (hasNext()) return w++;
        else           throw new NoSuchElementException();
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}


// string representation of Graph - takes quadratic time
public String toString() {
    String NEWLINE = System.getProperty("line.separator");
    StringBuilder s = new StringBuilder();
```

```
        s.append(V + " " + E + NEWLINE);
        for (int v = 0; v < V; v++) {
            s.append(v + ": ");
            for (int w : adj(v)) {
                s.append(w + " ");
            }
            s.append(NEWLINE);
        }
        return s.toString();
    }
```