DFS

Referências

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

Vídeos

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

Aula passada

- digrafos
- grafos = digrafos simétricos
- representação de grafos através de vetor de listas de adjacência

API

`public class`	`Digraph` (`Graph`)	consumo de tempo
-	`Graph(int V)`	~`V`
int	`V()`	~1
int	`E()`	~1
void	`addEdge(int v, int w)`	~1
Iterable	`adj(int v)`	~grau de saída de `v`
String	`toString()`	~`V+E`
Digraph	`reverse()`	~`V+E`

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`.

O consumo de tempo desse método é proporcional `V+E` se o digrafo for implementado com lista de adjacências.

O consumo de tempo desse métod é proporcional a V^2 se o digrafo for implementado como matriz de adjacências

```
public class DepthFirstDirectedPaths {
    private boolean[] marked;  // marked[v] = true if v is reachable from s

    private final int s;       // source vertex
    private int[] edgeTo;      // edgeTo[v] = last vertex on path from s to v
```

```java
//  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()];   // O(V)
    edgeTo = new int[G.V()];       // O(V)
    this.s = s;                    // O(1)
    dfs(G, s);                     // O(V+E)
}

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;
}
```

Certificados

Como é possível verificar a resposta?

Como é possível verificar que existe caminho?

Como é possível verificar que não existe caminho?

Cortes

Um **corte** é uma bipartição de um conjunto de vértices.

Um arco pertence ou atravessa um corte `(S,T)` se tiver uma ponta em `S` e outra em `T`.

st-cortes

Um corte (S,T) é um st-corte se s está em S e t está em T.

Para provarmos que não existe um caminho de s a tbasta exibirmos um st-corte em que todo arco tem ponta inicial em T e ponta final em S.

```
S = { v :  marked[v] }
T = { v : !marked[v] }
```

Conclusão

Para quaisquer vértices s e t de um digrafo, vale uma e apenas uma das seguintes afirmações:

- existe uma caminho de s a t
- existe um st-corte (S,T) em que todo arco tem ponta inicial em T e ponta final em S.

Ciclos

Um ciclo em um digrafo é qualquer sequência da forma $v_0 - v_1 - v_2 - \cdots - v_{p-1} - v_p$ tal que $v_{k-1} - v_k$ é um arco para $k = 1, 2, \ldots, p$ e $v_0 == v_p$.

Certificados

Como é possível verificar a resposta?

Como é possível verificar que existe ciclo?

Como é possível verificar que não existe ciclo?

DFS

O consumo de tempo desse método é proporcional `V+E` se o digrafo for implementado com lista de adjacências.

O consumo de tempo desse métod é proporcional a V^2 se o digrafo for implementado como matriz de adjacências

```
public class DFS {
    private boolean[] marked;
    private final int s;  // DepthFirstPaths

    public DFS(Digraph G) {
        marked = new boolean[G.V()];
        for (int v = 0; v < G.V(); v++)
            if (!marked(v)) {
                 dfs(G,v);
            }
    }

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

DFSPaths

Dado um digrafo `G` e um vértice `s`, encontra um caminho de `s` a `t` para todo vértice `t` alcançavel a partir de `s`.

```
public class DFSPaths {
    private boolean[] marked;
    private final int s;  // DepthFirstPaths
    private int[] edgeTo; // DepthFirstPaths


    public DFSPaths(Digraph G, int s) { // DepthFirstPaths
        marked = new boolean[G.V()];
        edgeTo = new int[G.V()]; // DepthFirstPaths
        dfs(G, s);
    }


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

    public boolena hasPath(int v) { // DepthFirstPaths
        return marked[v];
    }

    public Iterable<Integer> pathTo(int v) { // DepthFirstPaths
        if (!hasPath(v)) return null;
        Stack<Integer> path = new Stack<Integer>();
        for (int w = v; w != s; w = edgeTo[w])
            path.push(w);
        path.push(s);
        return path;
    }
}
```

DFSCC

Determinas os componentes de um dado **grafo** `G`.

```
public class DFSCC {
    private boolean[] marked;
    private int[] edgeTo; // DepthFirstPaths

    private int[] id; // CC
    private int count; // CC

    public DFCC(Graph G) { // CC, TwoColor
        marked = new boolean[G.V()];
        edgeTo = new int[G.V()]; // DepthFirstPaths
        id     = new int[G.V()]; // CC
        for (int v = 0; v < G.V(); v++)
            if (!marked(v)) {
                 dfs(G,v);
                 count++; // CC
            }
    }

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

    public int id(int v) { // CC
        return id[v];
    }

    public int count(int v) { // CC
        return count;
    }
```

DFSBipartite

Determina se um dado **grafo** é bipartido. Caso o grafo não seja bipartido, produz um ciclo de comprimento ímpar como certificado.

```
public class DFSBipartite {
    private boolean[] marked;        // marked[v] = has vertex v been marked?
    private int[] edgeTo;            // edgeTo[v] = previous vertex on path to v
    private boolean[] color; // TwoColor
    private boolean isTwoColorable = true; // TwoColor
    private Stack<Integer> cycle;
    private int onCycle = -1;

    // Determines whether the graph G is bipartite
    public DFSBipartite(Graph G) { //  TwoColor
        marked = new boolean[G.V()];
        edgeTo = new int[G.V()]; // DepthFirstPaths
        color  = new boolean[G.V()]; // TwoColor
        for (int v = 0; v < G.V(); v++)
            if (!marked(v)) {
                 dfs(G,v);
            }
    }

    private void dfs(Digraph G, int v) {
         marke[v] = true;
         for (int w : G.adj[v]) {
             if (!marked(w)) {
                  color[w] = !color[v];; // TwoColor
                  edgeTo[w] = v;
                  dfs(G, w);
                  if (hasCycle()) return;
             }
             else if (color[v] == color[w])
                  isTwoColorable = false;
                  onCycle = v;
                  edgeTo[v] = w; // fecha o ciclo
             }
    }

    public boolean isBipartite() { // TwoColor
        return isTwoColorable;
    }

    // retorna um ciclo de comprimento ímpar se o grafo não é bipartido
    public Iterable<Integer> cycle() {
        if (isTwoColorable) return null;
        if (cycle != null) return cycle;
        cycle = new Stack<Integer>();
        for (int x = edgeTo[onCycle]; x != onCycle; x = edgeTo[x])
```

```
        cycle.push(x);
    cycle.push(onCycle);
    return cycle;
}
```