Grafos

Grafos

Orientados ou **não orientados**

Pesados (ou **etiquetados**) ou **não pesados** (**não etiquetados**)

Grafo $G = (V, E)$

V – conjunto dos **nós** (ou **vértices**)

$E \subseteq V^2$ – conjunto dos **arcos** (ou **arestas**)

$w : E \to \mathbb{R}$ – **peso** (ou **etiqueta**) de um arco

Vértices e arcos (1)

Se $G = (V, E)$ e $(u, v) \in E$

- O nó v diz-se adjacente ao nó u
- Os nós u e v são vizinhos

Se G é não orientado:

- Os nós u e v são as extremidades do arco (u, v)
- Os arcos (u, v) e (v, u) são o mesmo arco
- Logo, o nó u também é adjacente ao nó v
- O arco (u, v) liga os nós u e v
- O arco (u, v) é incidente no nó u e no nó v

Vértices e arcos (2)

Se G é orientado:

- O nó u é a origem do arco (u, v)
- O nó v é o destino do arco (u, v)
- O nó u é um predecessor (ou antecessor) do nó v
- O nó v é um sucessor do nó u
- O arco (u, v) sai, ou parte, do nó u
- O arco (u, v) chega ao nó v
- O arco (u, v) é incidente no nó v

O grau do nó u é o número de arcos $(u, v) \in E$

Subgrafos e grafos isomorfos

Subgrafo

Um subgrafo do grafo $G = (V, E)$ é um grafo $G' = (V', E')$ tal que $V' \subseteq V$ e $E' \subseteq E$

Grafos isomorfos

Os grafos $G_1 = (V_1, E_1)$ e $G_2 = (V_2, E_2)$ são isomorfos se existe uma função bijectiva $f : V_1 \to V_2$ tal que

$$(f(u), f(v)) \in E_2 \quad \text{sse} \quad (u, v) \in E_1$$

Caminhos

Um caminho num grafo $G = (V, E)$ qualquer é uma sequência não vazia de vértices $v_i \in V$

$$v_0\, v_1\, \ldots\, v_k \quad (k \geq 0)$$

tal que $(v_i, v_{i+1}) \in E$, para $i < k$

O comprimento do caminho $v_0 v_1 \ldots v_k$ é k, o número de arestas que contém

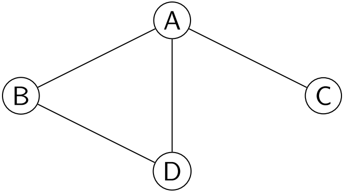
O caminho v_0 é o caminho de comprimento 0, de v_0 para v_0

Um caminho é simples se $v_i \neq v_j$ quando $i \neq j$

Exemplos de grafos

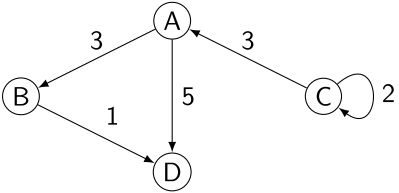
Grafo não orientado e não pesado

$$G = (\{A, B, C, D\}, \{(A, B), (B, D), (A, D), (C, A)\})$$



Grafo orientado pesado

$$G = (\{A, B, C, D\}, \{(A, B, 3), (B, D, 1), (A, D, 5), (C, A, 3), (C, C, 2)\})$$



Ciclos

Um ciclo, num grafo orientado, é um caminho em que

$$v_0 = v_k \quad \text{e} \quad k > 0$$

Num grafo não orientado, um caminho forma um ciclo se

$$v_0 = v_k \quad \text{e} \quad k \geq 3$$

Um ciclo é simples se $v_1, v_2, \ldots, v_k$ são distintos

Um grafo é acíclico se não contém qualquer ciclo simples

Representação / Implementação

Listas de adjacências

- Grafos esparsos $(\,|E| \ll |V|^2\,)$
- Permite descobrir rapidamente os vértices adjacentes a um vértice
- Complexidade espacial $\Theta(V + E)$

Matriz de adjacências

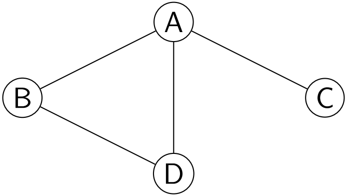
- Grafos densos $(\,|E| = O(V^2)\,)$
- Permite verificar rapidamente se $(u, v) \in E$
- Complexidade espacial $\Theta(V^2)$

Na notação O, V e E significam, respectivamente, $|V|$ e $|E|$

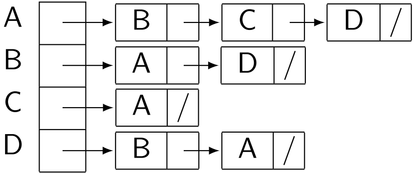
Representação / Implementação

Grafo não orientado e não pesado

Grafo $G = (\{A, B, C, D\}, \{(A, B), (B, D), (A, D), (C, A)\})$



Listas de adjacências



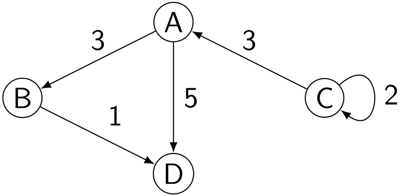
Matriz de adjacências

	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	0
D	1	1	0	0

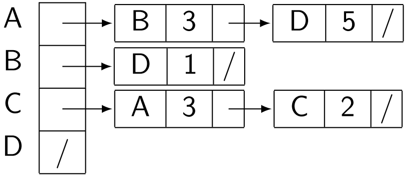
Representação / Implementação

Grafo orientado pesado

Grafo $G = (\{A, B, C, D\}, \{(A, B, 3), (B, D, 1), (A, D, 5), (C, A, 3), (C, C, 2)\})$



Listas de adjacências



Matriz de adjacências

	A	B	C	D
A	0	3	0	5
B	0	0	0	1
C	3	0	2	0
D	0	0	0	0

Implementação em Java

Grafo orientado não pesado

```
class Graph {
  int nodes; // number of nodes
  List<Integer>[] adjacents; // adjacency lists

  @SuppressWarnings("unchecked")
  public Graph(int nodes)
  {
    this.nodes = nodes;

    adjacents = new List[nodes];

    for (int i = 0; i < nodes; ++i)
      adjacents[i] = new LinkedList<>();
  }

  /* Adds the (directed) edge (U,V) to the graph. */
  public void addEdge(int u, int v)
  {
    adjacents[u].add(v);
  }

  ...
}
```

Percursos básicos em grafos

Percurso em largura

Nós são visitados por ordem crescente de distância ao nó em que o percurso se inicia

Percurso em profundidade

Nós são visitados pela ordem por que são encontrados

Percurso em largura (a partir do vértice *s*)

BFS(G, s)

```
for each vertex u in G.V - {s} do
    u.color <- WHITE
    u.d <- INFINITY
    u.p <- NIL
s.color <- GREY
s.d <- 0
s.p <- NIL
Q <- EMPTY                          // fila (FIFO)
ENQUEUE(Q, s)
while Q != EMPTY do
    u <- DEQUEUE(Q)                 // próximo nó a explorar
    for each vertex v in G.adj[u] do
        if v.color = WHITE then
            v.color <- GREY
            v.d <- u.d + 1
            v.p <- u
            ENQUEUE(Q, v)
    u.color <- BLACK                // u foi explorado
```

Percurso em largura

Breadth-first search

Descobre um caminho mais curto de um vértice *s* a qualquer outro vértice

Calcula o seu comprimento (linhas 3, 6 e 15)

Constrói a árvore da pesquisa em largura (linhas 4, 7 e 16), que permite reconstruir o caminho identificado

Atributos dos vértices

`color`	`WHITE`	não descoberto
	`GREY`	descoberto, mas não processado
	`BLACK`	processado
`d`	distância a *s*	
`p`	antecessor do nó no caminho a partir de *s*	

Análise da complexidade temporal de BFS (1)

Grafo implementado através de listas de adjacências

BFS(G, s)

```
1  for each vertex u in G.V - {s} do
2      u.color <- WHITE
3      u.d <- INFINITY
4      u.p <- NIL
```

- Ciclo das linhas 1–4 é executado $|V| - 1$ vezes

```
5  s.color <- GREY
6  s.d <- 0
7  s.p <- NIL
8  Q <- EMPTY                       // fila (FIFO)
9  ENQUEUE(Q, s)
```

- Linhas 5–9 com custo constante

Análise da complexidade temporal de BFS (2)

- Ciclo das linhas 10–18 é executado $|V|$ vezes, no pior caso

```
10  while Q != EMPTY do
11      u <- DEQUEUE(Q)              // próximo nó a explorar
12      for each vertex v in G.adj[u] do
13          if v.color = WHITE then
14              v.color <- GREY
15              v.d <- u.d + 1
16              v.p <- u
17              ENQUEUE(Q, v)
18      u.color <- BLACK             // u foi explorado
```

- Mas o ciclo das linhas 12–17 é executado, no pior caso

$$\sum_{u \in V} |\,\texttt{G.adj}[u]\,| = |E| \text{ (orientado) ou } 2 \times |E| \text{ (não orientado) vezes}$$

porque cada vértice só pode entrar na fila uma vez

Análise da complexidade temporal de BFS (3)

Considerando que todas as operações, incluindo a criação de uma fila vazia, ENQUEUE e DEQUEUE, têm custo $\Theta(1)$

- O ciclo das linhas 1–4 tem custo $\Theta(V)$
- Conjuntamente, os ciclos das linhas 10–18 e 12–17 têm custo $O(E)$ (pior caso)

Logo, a complexidade temporal de BFS é $O(V + E)$

Análise da complexidade temporal de BFS (4)

Grafo implementado através da matriz de adjacências

Na linha 12, é necessário percorrer uma linha da matriz, com $|V|$ elementos

```
12'     for each vertex v in G.V do
13'         if G.adjm[u,v] and v.color = WHITE then
```

Como o ciclo das linhas 10–18 é executado $|V|$ vezes, no pior caso, o custo combinado dos dois ciclos é $O(V^2)$

- Corresponde a aceder a todas as posições de uma matriz $|V| \times |V|$

Neste caso, a complexidade temporal de BFS será $O(V^2)$

Complexidade espacial de BFS

Memória usada pelo algoritmo

- 3 valores escalares (`color`, `d` e `p`) por cada vértice

$$\Theta(V)$$

- Uma fila, que poderá ter, no pior caso, $|V| - 1$ vértices

$$O(V)$$

A complexidade espacial de BFS é $\Theta(V)$

BFS em Java (1)

```
public static final int INFINITY = Integer.MAX_VALUE;
public static final int NONE = -1;

private static enum Colour  WHITE, GREY, BLACK ;

public int[] bfs(int s)
{
  Colour[] colour = new Colour[nodes];
  int[] d = new int[nodes];          // distância para S
  int[] p = new int[nodes];          // predecessor no caminho de S

  for (int u = 0; u < nodes; ++u)
    {
      colour[u] = Colour.WHITE;
      d[u] = INFINITY;
      p[u] = NONE;
    }

  colour[s] = Colour.GREY;
  d[s] = 0;

  Queue<Integer> Q = new LinkedList<>();

  Q.add(s);

  ...
```

BFS em Java (2)

```
  ...
  while (!Q.isEmpty())
    {
      int u = Q.remove();                      // visita nó U

      for (Integer v : adjacents[u])
        if (colour[v] == Colour.WHITE)
          {
            colour[v] = Colour.GREY;           // V é um novo nó
            d[v] = d[u] + 1;
            p[v] = u;

            Q.add(v);
          }

      colour[u] = Colour.BLACK;                // nó U está tratado
    }

    return d ou p ou ...
}
```