

GRAFOS

Gustavo Carvalho
(ghpc@cin.ufpe.br)

Universidade Federal de Pernambuco
Centro de Informática, 50740-560, Brazil



Agenda

1 Introdução

2 Representação

3 Travessias

4 Aplicações de DFS e BFS

5 Bibliografia



Introdução

Grafo: um conjunto de pontos (**vértices** ou **nós**) ligados por segmentos (**arestas** ou **arcos**)

$G = (V, E)$, onde $V \neq \emptyset$

Nomenclatura

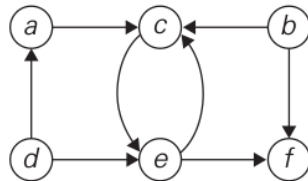
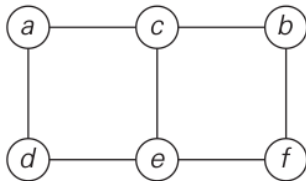
■ Grafos não-dirigidos

- O par $(u, v) = (v, u) \in E$
- u e v : vértices adjacentes
- u e v são extremidades de (u, v)
- (u, v) é uma aresta incidente aos vértices u e v
- Grafo não-dirigido: todas as arestas são não-dirigidas

■ Grafos dirigidos (*digraphs*)

- $(u, v) \neq (v, u)$
- u e v : cauda e cabeça da aresta (u, v)
- Grafo dirigido: todas as arestas são dirigidas

Introdução



1

$$V = \{a, b, c, d, e, f\},$$

$$E = \{\{a, c\}, \{a, d\}, \{b, c\}, \{b, f\}, \{c, e\}, \{d, e\}, \{e, f\}\}$$

$$V = \{a, b, c, d, e, f\},$$

$$E = \{(a, c), (b, c), (b, f), (c, e), (d, a), (d, e), (e, c), (e, f)\}$$

¹ Fonte: A. Levitin. Introduction to the Design and Analysis of Algorithms. 2011.

Introdução

Nomenclatura²:

- Grafo simples: não-dirigido, sem laços e sem arestas paralelas
- Multigrafo: não-dirigido, sem laços e com arestas paralelas
- Pseudografo: não-dirigido, com laços e com arestas paralelas
- Grafo dirigido simples: dirigido, sem laços e sem arestas paralelas
- Multigrafo dirigido: dirigido, com laços e com arestas paralelas
- Grafo misto

Exemplos:

- Cidades conectadas por vôos diretos
- Rotas (a pé) entre bairros
- Rotas (de carro) entre bairros
- Quem ganhou/perdeu/empatou a partida

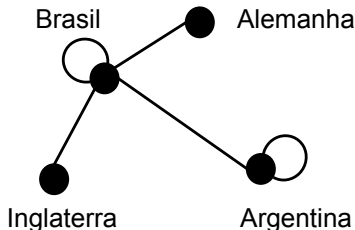
²Kenneth H. Rosen. Discrete Mathematics and Its Applications. 2011

Introdução

Seja G um grafo não dirigido:

- O **grau** de um vértice é seu número de arestas incidentes.
- Laços contribuem **duas vezes**.
- Notação: $\deg(a)$ (do inglês: *degree*)

Qual o grau dos vértices abaixo?



Introdução

Teorema do aperto de mãos

- Seja G um grafo não dirigido, então

$$2 | E | = \sum_{v \in V} \deg(v)$$

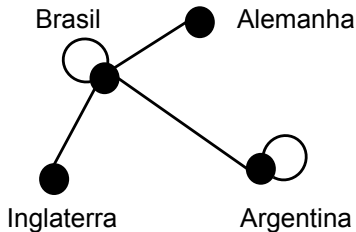
- Por quê?

- Cada aresta incide sobre 2 vértices.
- Cada aresta contribui com $1 + 1$ na soma dos graus.
- Então, a soma dos graus é o dobro do número de arestas.

Introdução

Calcule a soma dos graus do grafo abaixo

Compare este número com a quantidade de arestas

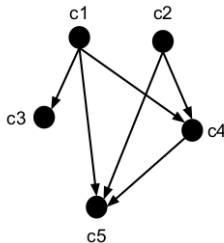


Introdução

Seja G um **grafo dirigido**, e v um vértice de G :

- **Grau de entrada** $\deg^-(v)$: número de arestas apontando para v .
- **Grau de saída** $\deg^+(v)$: número de arestas saindo de v .

Qual o $\deg^-(v)$ e o $\deg^+(v)$ para cada vértice v do grafo abaixo?



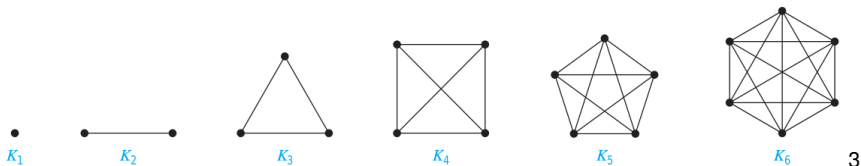
Teorema: $\sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v) = |E|$

Introdução

Seja G um grafo simples (não-dirigido, sem laços/arestas paralelas)

$$\blacksquare 0 \leq |E| \leq \frac{|V|(|V|-1)}{2}$$

Um grafo (simples) G é **completo** se existe uma aresta entre dois vértices quaisquer: $K_{|V|}$



Grafos **densos** e grafos **esparsos**

Grafos **ponderados**: com **peso** ou **custo** associado a arestas

³ Fonte: Kenneth H. Rosen. Discrete Mathematics and Its Applications. 2011.

Introdução

Um **caminho** de tamanho n de u para v , onde n é um inteiro positivo, é uma sequência de arestas e_1, \dots, e_n do grafo de forma que $e_1 = \{x_0, x_1\}$, $e_2 = \{x_1, x_2\}$, ..., $e_n = \{x_{n-1}, x_n\}$, onde $x_0 = u$ e $x_n = v$.

Um **circuito** (ou ciclo) é um caminho em que $u = v$.

Um caminho ou circuito é **simples** se não contém a mesma aresta mais de uma vez.

Circuito **euleriano**: circuito simples que contém cada aresta de G .

- Um multigrafo conectado (com $|V| \geq 2$) tem um circuito euleriano se, e somente se, cada vértice tiver grau par.

Circuito **hamiltoniano**: circuito simples que passa por cada vértice de G uma única vez.

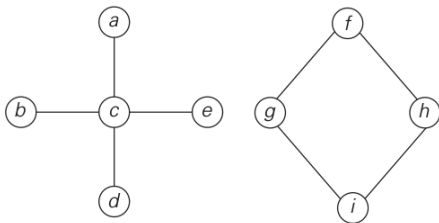


Introdução

Grafo **conectado**: se, para todo u e v , existe um caminho de u para v

Componente conectado: subgrafo máximo conectado

- O subgrafo de G é um grafo G' tal que: $V' \subseteq V$, $E' \subseteq E$



4

⁴Fonte: A. Levitin. Introduction to the Design and Analysis of Algorithms. 2011.

Agenda

1 Introdução

2 Representação

3 Travessias

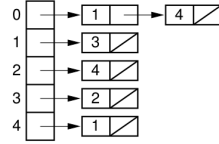
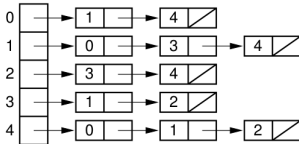
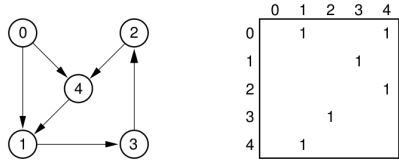
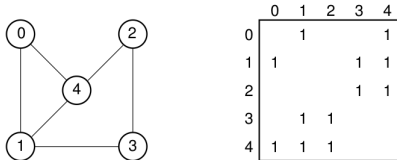
4 Aplicações de DFS e BFS

5 Bibliografia



Representação de grafos⁵

Matriz de adjacências ou lista de adjacências



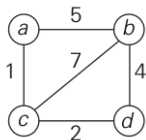
Eficiência **espacial**: grafo denso vs. grafo esparsos

Eficiência **temporal**: $\Theta(|V|^2)$ vs. $\Theta(|V| + |E|)$

⁵Fonte: C. Shaffer. Data Structures and Algorithm Analysis. 2013.

Representação de grafos

Grafos **ponderados**: as duas representações são possíveis



	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>
<i>a</i>	∞	5	1	∞
<i>b</i>	5	∞	7	4
<i>c</i>	1	7	∞	2
<i>d</i>	∞	4	2	∞

<i>a</i>	$\rightarrow b, 5 \rightarrow c, 1$
<i>b</i>	$\rightarrow a, 5 \rightarrow c, 7 \rightarrow d, 4$
<i>c</i>	$\rightarrow a, 1 \rightarrow b, 7 \rightarrow d, 2$
<i>d</i>	$\rightarrow b, 4 \rightarrow c, 2$

6

6

Fonte: A. Levitin. Introduction to the Design and Analysis of Algorithms. 2011.

Representação de grafos

Operações do ADT grafo

- `int n(G g);`
- `int e(G g);`
- `int first(G g, int v);`
- `int next(G g, int v, int w);`
- `void setEdge(G g, int i, int j, int wt);`
- `void delEdge(G g, int i, int j);`
- `boolean isEdge(G g, int i, int j);`
- `int weight(G g, int i, int j);`
- `void setMark(G g, int v, int val);`
- `int getMark(G g, int v);`

Representação de grafos: matriz de adjacências

Estrutura de dados G :

```
1  int[][] matrix ;           // matriz de adjacências
2  int numEdge ;             // quantidade de arestas
3  int[] Mark ;              // array auxiliar de marcação
```

Algoritmo: G create_graph(int n)

```
1   $g.Mark \leftarrow \text{new int}[n]$ ;
2   $g.matrix \leftarrow \text{new int}[n][n]$ ;
3   $g.numEdge \leftarrow 0$ ;
4  return  $g$ ;
```

Representação de grafos: matriz de adjacências

Algoritmo: `int n(G g)`

1 **return** `length(g.Mark);`

Algoritmo: `int e(G g)`

1 **return** `g.numEdge;`

Algoritmo: `int first(G g, int v)`

1 **for** $i \leftarrow 0$ **to** `length(g.Mark) - 1` **do**
2 **if** `g.matrix[v][i] \neq 0` **then return** i ;
3 **return** `length(g.Mark);`

Representação de grafos: matriz de adjacências

Algoritmo: int next(G g, int v, int w)

```

1  for  $i \leftarrow w + 1$  to  $\text{length}(g.\text{Mark}) - 1$  do
2  |   if  $g.\text{matrix}[v][i] \neq 0$  then return  $i$ ;
3  return  $\text{length}(g.\text{Mark})$ ;

```

Algoritmo: void setEdge(G g, int i, int j, int wt)

```

1  if  $wt = 0$  then error ;
2  if  $g.\text{matrix}[i][j] = 0$  then  $g.\text{numEdge}++$ ;
3   $g.\text{matrix}[i][j] \leftarrow wt$ ;

```

Algoritmo: void delEdge(G g, int i, int j)

```

1  if  $g.\text{matrix}[i][j] \neq 0$  then  $g.\text{numEdge}--$ ;
2   $g.\text{matrix}[i][j] \leftarrow 0$ ;

```

Representação de grafos: matriz de adjacências

Algoritmo: boolean isEdge(G g, int i, int j)

1 **return** $g.matrix[i][j] = 0$;

Algoritmo: int weight(G g, int i, int j)

1 **return** $g.matrix[i][j]$;

Algoritmo: void setMark(G g, int v, int val)

1 $g.Mark[v] \leftarrow val$;

Algoritmo: int getMark(G g, int v)

1 **return** $g.Mark[v]$;

Representação de grafos: lista de adjacências

- `int n(G g);`
retorna *length(g.Mark)*
- `int e(G g);`
retorna *g.numEdge*
- `int first(G g, int v);`
retorna o primeiro elemento na lista de *v*
- `int next(G g, int v, int w);`
se *w* está na lista de *v*, retorna o elemento após *w* na lista de *v*
- `void setEdge(G g, int i, int j, int wt);`
se *j* já existe na lista de *i*, atualiza o peso
se *j* não existe na lista de *i*, insere *j* (ordenado por *j*)
na lista de *i* com peso *wt*
e incrementa *g.numEdge*

Representação de grafos: lista de adjacências

- `void delEdge(G g, int i, int j);`
se j está na lista de i , remove da lista e decrementa $g.numEdge$
- `boolean isEdge(G g, int i, int j);`
procura na lista de i pelo elemento j
- `int weight(G g, int i, int j);`
procura na lista de i pelo elemento j e retorna seu peso
- `void setMark(G g, int v, int val);`
igual à representação com matriz
- `int getMark(G g, int v);`
igual à representação com matriz

Agenda

- 1 Introdução
- 2 Representação
- 3 Travessias**
- 4 Aplicações de DFS e BFS
- 5 Bibliografia



Travessia em grafos

De forma geral:

Algoritmo: void graphTraverse(G g)

```

1  for  $v \leftarrow 0$  to  $n(g) - 1$  do
2     $\lfloor$  setMark( $g, v, UNVISITED$ );
3  for  $v \leftarrow 0$  to  $n(g) - 1$  do
4     $\lfloor$  if getMark( $g, v$ ) = UNVISITED then
5       $\lfloor$  traverse( $g, v$ );

```

A operação *traverse* pode ser feita em profundidade ou largura

- **DFS:** depth-first search (**busca em profundidade**)
- **BFS:** breadth-first search (**busca em largura**)

Eficiência temporal: $\Theta(|V| + |E|)$

Travessia em grafos: DFS

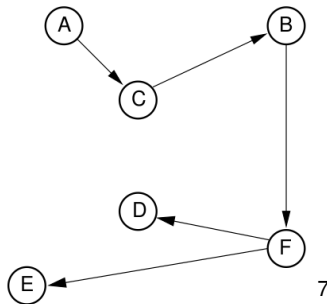
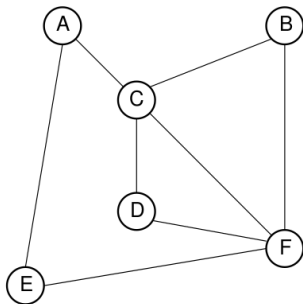
Faz uso de uma pilha (implícita na recursão)

Algoritmo: void DFS(G g , int v)

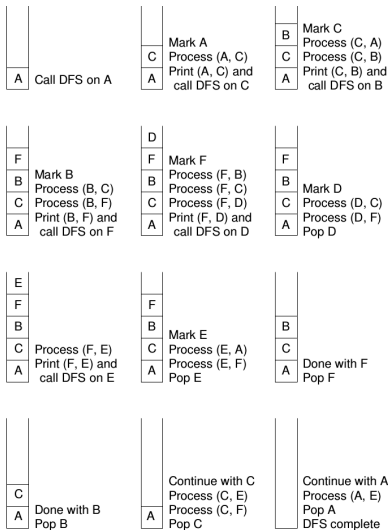
```
1  preVisit( $g$ ,  $v$ );  
2  setMark( $g$ ,  $v$ , VISITED);  
3   $w \leftarrow \textit{first}(g, v)$ ;  
4  while  $w < n(g)$  do  
5    if getMark( $g$ ,  $w$ ) = UNVISITED then  
6    |   DFS( $g$ ,  $w$ );  
7    |    $w \leftarrow \textit{next}(g, v, w)$ ;  
8  posVisit( $g$ ,  $v$ );
```

Travessia em grafos: DFS

Começando a busca de A



Travessia em grafos: DFS



8

Travessia em grafos: BFS

Faz uso de uma fila (explicitamente)

Algoritmo: void BFS(G g , int $start$)

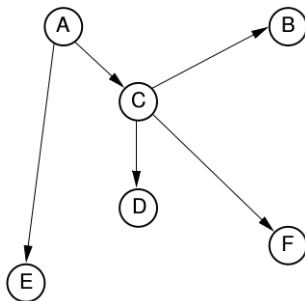
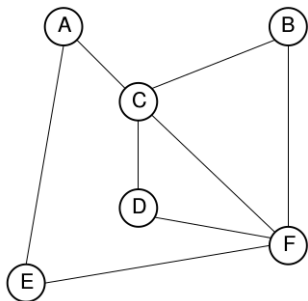
```

1   $Q \leftarrow create\_queue()$ ;
2   $enqueue(Q, start)$ ;
3   $setMark(g, start, VISITED)$ ;
4  while  $length(Q) > 0$  do
5       $v \leftarrow dequeue(Q)$ ;
6       $preVisit(g, v)$ ;
7       $w \leftarrow first(g, v)$ ;
8      while  $w < n(g)$  do
9          if  $getMark(g, w) = UNVISITED$  then
10              $setMark(g, w, VISITED)$ ;
11              $enqueue(Q, w)$ ;
12          $w \leftarrow next(g, v, w)$ ;
13  $posVisit(g, v)$ ;

```

Travessia em grafos: BFS

Começando a busca de A



9

⁹Fonte: C. Shaffer. Data Structures and Algorithm Analysis. 2013.

Travessia em grafos: BFS

A	
---	--

Initial call to BFS on A.
Mark A and put on the queue.

E	B	D	F
---	---	---	---

Dequeue C.
Process (C, A). Ignore.
Process (C, B).
Mark and enqueue B. Print (C, B).
Process (C, D).
Mark and enqueue D. Print (C, D).
Process (C, F).
Mark and enqueue F. Print (C, F).

D	F
---	---

Dequeue B.
Process (B, C). Ignore.
Process (B, F). Ignore.

--	--

Dequeue F.
Process (F, B). Ignore.
Process (F, C). Ignore.
Process (F, D). Ignore.
BFS is complete.

C	E
---	---

Dequeue A.
Process (A, C).
Mark and enqueue C. Print (A, C)
Process (A, E).
Mark and enqueue E. Print(A, E).

B	D	F
---	---	---

Dequeue E.
Process (E, A). Ignore.
Process (E, F). Ignore.

F

Dequeue D.
Process (D, C). Ignore.
Process (D, F). Ignore.

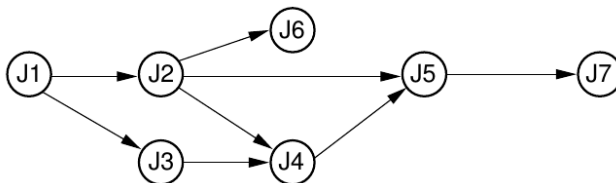
Agenda

- 1 Introdução
- 2 Representação
- 3 Travessias
- 4 Aplicações de DFS e BFS**
- 5 Bibliografia

DFS: ordenação topológica

Seja G um grafo dirigido, listar todos os vértices em uma ordem onde, para toda aresta (u, v) , u é listado antes de v .

- Só possui solução se, e somente se, G for um **dag**
- dag: *directed acyclic graph*



11

¹¹ Fonte: C. Shaffer. Data Structures and Algorithm Analysis. 2013.

DFS: ordenação topológica

Assumindo que G é um **dag**

Algoritmo: void toposort(G g , int v , STACK s)

```

1  setMark( $g$ ,  $v$ , VISITED);
2   $w \leftarrow \text{first}(g, v)$ ;
3  while  $w < n(g)$  do
4      if getMark( $g$ ,  $w$ ) = UNVISITED then
5          |   toposort( $g$ ,  $w$ ,  $s$ );
6          |    $w \leftarrow \text{next}(g, v, w)$ ;
7  push( $s$ ,  $v$ );           // teremos em  $s$  a ordenação topológica

```

BFS: menor caminho (sem peso)

Similar à busca em largura com algumas modificações

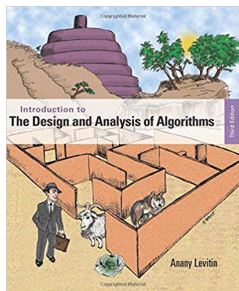
- Iniciar a busca a partir da **origem**
- Ao chegar no **destino**, encerrar a busca
- Array auxiliar de **predecessores** (para reconstruir o caminho)

Agenda

- 1 Introdução
- 2 Representação
- 3 Travessias
- 4 Aplicações de DFS e BFS
- 5 Bibliografia**



Bibliografia + leitura recomendada



Capítulo 1 (pp. 27–31)
Capítulo 3 (pp. 122–128)
Capítulo 4 (pp. 138–141)
Anany Levitin.

Introduction to the Design and Analysis of Algorithms.
 3a edição. Pearson. 2011.



Capítulo 11 (pp. 371–388)
Clifford Shaffer.
Data Structures and Algorithm Analysis.
 Dover, 2013.



GRAFOS

Gustavo Carvalho
(ghpc@cin.ufpe.br)

Universidade Federal de Pernambuco
Centro de Informática, 50740-560, Brazil

