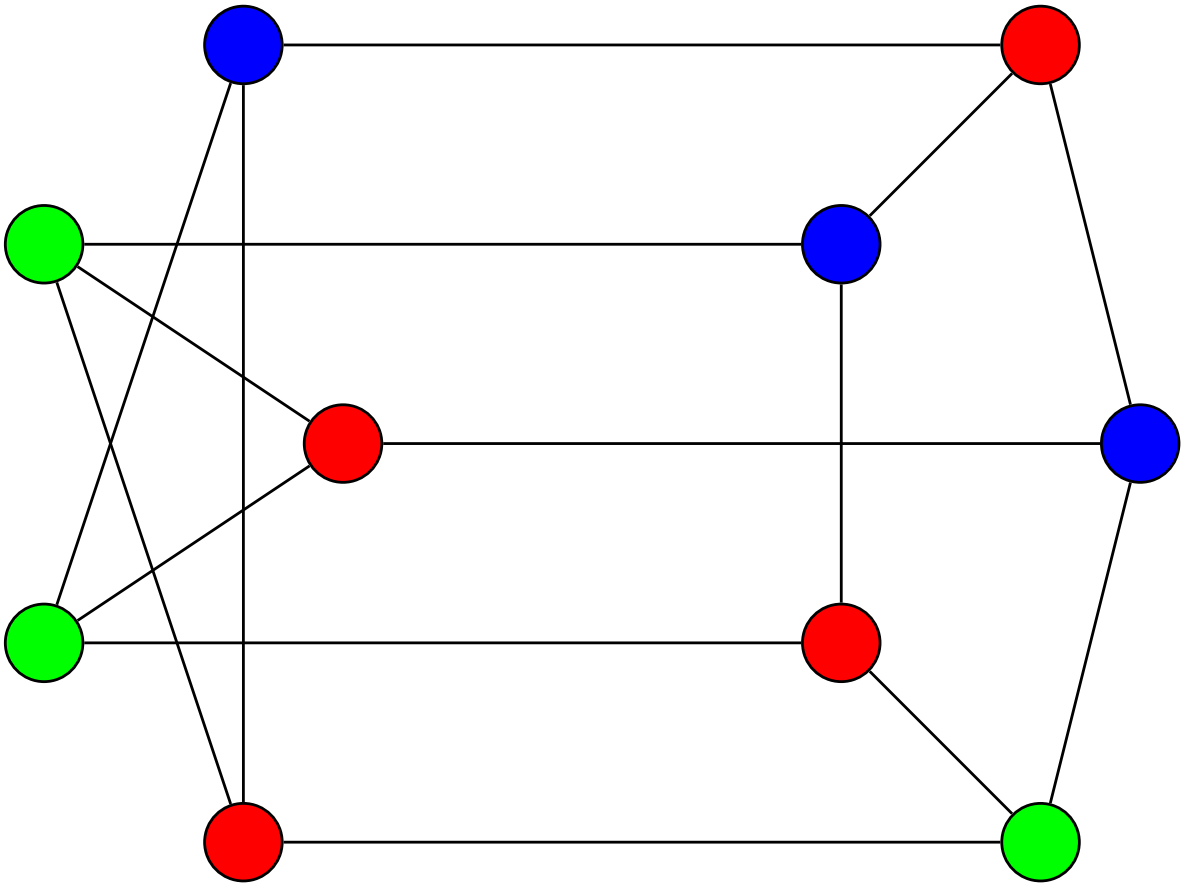
Centro de Matemática, Computação e Cognição
Universidade Federal do ABC
Profa. Carla Negri Lintzmayer

Notas de aula da disciplina Teoria dos Grafos



Esse material é um compilado dos seguintes:

- Sedgewick, R.. *Algorithms in C, part 5: graph algorithms*. 3rd ed. Addison-Wesley. 2002.
- Bondy, J. A.; Murty, U. S. R.. *Graph Theory*. Graduate Texts in Mathematics. Springer. New York. 2008.
- Lintzmayer, C. N.; Mota, G. O.. *Notas de aulas - Análise de algoritmos e estruturas de dados*. Em construção.

6 Aula 8: busca em grafos

- Algoritmos de busca em grafos são úteis para que possamos obter mais informações sobre a estrutura de um grafo (se há caminhos, se é conexo, se há ciclos, qual a distância...).
  - E são base para vários outros algoritmos importantes.
- As buscas começam em um vértice e descobrem tudo que é *alcançável* a partir dele.

Proposição 6.1. *Se existe su-caminho e $uv \in E(G)$, então existe sv-caminho.*

- Seja T uma árvore que é subgrafo de um grafo G:
  - Se $V(T) = V(G)$, então T é geradora e podemos concluir que G é conexo.
  - Se $V(T) \neq V(G)$, existem duas possibilidades: ou não há arestas entre $V(T)$ e $V(G) \setminus V(T)$, caso em que G é desconexo, ou há.
  - No último caso, para qualquer aresta $xy \in E(G)$, onde $x \in V(T)$ e $y \in V(G) \setminus V(T)$, note que $T + xy$ é também uma árvore contida em G.

Proposição 6.2. *Se T é uma árvore, $u \in V(T)$ e $v \notin V(T)$, então $T + uv$ é uma árvore.*

- Resta escolher uma árvore inicial, pois já temos um algoritmo!
- Procedimentos assim costumam ser chamados de *busca* e a árvore resultante é chamada de *árvore de busca*.
  - Essa árvore é *enraizada* no vértice inicial s.
- Na prática, nem sempre se constrói uma árvore explicitamente, mas se faz uso de uma terminologia comum:
  - Cada vértice w no caminho de s a um vértice v é um *ancestral* de v, enquanto v é um *descendente* de w.
  - O ancestral imediato de um vértice v diferente da raiz é seu *predecessor*, $pred(v)$.
  - Podemos ver que $E(T) = \{(pred(v), v) \colon v \in V(T) \setminus \{s\}\}$, motivo pelo qual não é necessário construir a árvore explicitamente.
  - Ademais, $V(T) = \{v \in V(G) \colon pred(v) \neq Null\} \cup \{s\}$.

```
1: Função Busca(G, s)
2:     visitado[s] ← 1
3:     Enquanto houver aresta com um extremo visitado e outro não faça
4:         seja xy ∈ E(G) com visitado[x] = 1 e visitado[y] = 0
5:         visitado[y] ← 1
6:         pred[y] ← x
```

```
1: Função ChamaBusca(G)
2:     Para cada v ∈ V(G) faça
3:         visitado[v] ← 0
4:         pred[v] ← Null
5:     seja s ∈ V(G) "qualquer"
6:     Busca(G, s)
```

- O lema a seguir nos diz que Busca(G, s) marca todos os vértices que estão na mesma componente conexa de s. Em outras palavras, a árvore construída (mesmo que implicitamente) é geradora da componente conexa que contém s.

Lema 6.3. *No fim da execução de Busca(G, s), um vértice v está marcado se e somente se existe um sv-caminho em G.*

```
1: Função ImprimeCaminho(G, s, v, pred)
2:     Se v ≠ s então
3:         ImprimeCaminho(G, s, pred[v], pred)
4:     Imprima v
```

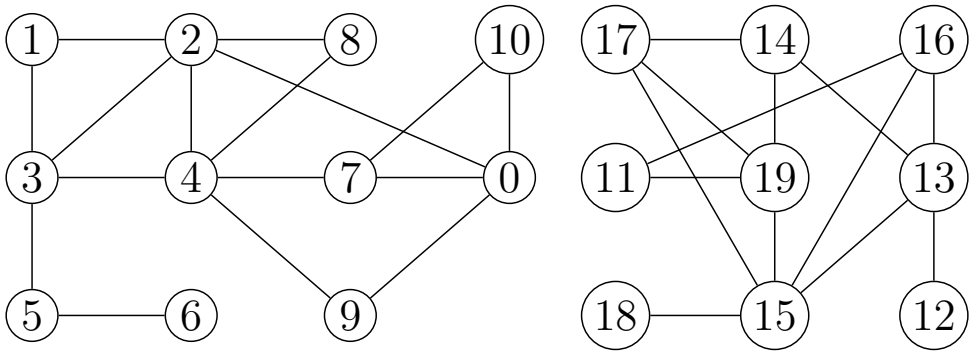
- Note que encontrar uma aresta xy na linha 4 envolve percorrer as vizinhanças dos vértices que já estão na árvore, uma a uma, para determinar qual vértice e aresta podem ser adicionados à árvore.

- Se o seu objetivo é apenas determinar se um grafo é conexo ou encontrar caminhos, qualquer algoritmo de busca serve (a ordem em que as vizinhanças são consideradas não importa).

- Mas algoritmos de busca nos quais critérios específicos são utilizados para determinar tal ordem podem prover informação adicional sobre a estrutura do grafo.

6.1 Busca em largura (BFS - *Breadth-First Search*)

- A ideia é expandir a árvore pela vizinhança do vértice que foi visitado há mais tempo:
  - ordem "primeiro a entrar, primeiro a sair";
  - os vértices são explorados por camadas (s na 0, seus vizinhos na 1, etc. ...).
- Esse algoritmo pode ser usado para:
  - encontrar componentes conexas;
  - calcular distância entre vértices;
  - encontrar caminhos entre vértices;
  - detectar ciclos;
  - verificar se um grafo é bipartido (e dar uma bipartição em caso positivo);
  - encontrar uma árvore geradora ou uma floresta geradora.
- Exemplo de execução:




```
1: Função ChamaBFS(G)
2:     sejam visitado, pred e D vetores indexados por vértices
3:     Para cada v ∈ V(G) faça
4:         D[v] ← ∞
5:         visitado[v] ← 0
6:         pred[v] ← Null
7:     seja s ∈ V(G)
8:     BFS(G, s)
```

```
1:  Função BFS(G, s)
2:      visitado[s] ← 1
3:      D[s] ← 0
4:      crie uma fila vazia F
5:      ENFILEIRA(F, s)
6:      Enquanto F ≠ ∅ faça
7:          u ← DESENFILEIRA(F)
8:          Para cada v ∈ N(u) faça
9:              Se visitado[v] = 0 então
10:                 visitado[v] ← 1
11:                 pred[v] ← u
12:                 D[v] ← D[u] + 1
13:                 ENFILEIRA(F, v)
```

Lema 6.4. *Seja G um grafo e $s \in V(G)$. No fim da execução de BFS(G, s), um vértice v está visitado se e somente se existe sv-caminho em G.*

- Tempo de execução de BFS(G, s):
  - A inicialização (linhas 2 a 5) leva tempo total $\Theta(1)$.
  - Note que antes de enfileirar um vértice v, atualizamos $visitado[v]$ de 0 para 1 (linha 10) e não modificamos mais esse atributo.
  - Assim, todo vértice para o qual existe um sv-caminho entra somente uma vez na fila e nunca mais passará no teste da linha 9.
  - Como a linha 7 sempre remove alguém da fila, o teste do laço **enquanto** (linha 6) é executado $O(V)$ vezes, assim como a chamada a DESENFILEIRA (linha 7).
  - Resta então analisar a quantidade de vezes que o conteúdo do laço **para** da linha 8 é executado.
  - Se utilizarmos matriz de adjacências, então o laço **para** leva tempo $\Theta(V)$ para executar uma única vez. Como ele executa uma vez para cada iteração do laço **enquanto**, o tempo de execução total é $\leq \sum_{u \in V(G)} \Theta(V) = O(V^2)$. Assim, o tempo total do algoritmo é $O(V) + O(V^2) = O(V^2)$.
  - Se utilizarmos listas de adjacências, então o laço **para** leva tempo $\Theta(d(u))$ para executar uma única vez. Como ele executa uma vez para cada iteração do laço **enquanto**, o tempo total de execução é $\leq \sum_{u \in V(G)} \Theta(d(u)) = O(E)$. Assim, o tempo total do algoritmo é $O(V) + O(E) = O(V + E)$.

- Portanto, a BFS é linear no tamanho da entrada.
- O Lema 6.4 mostra que BFS(G, s) visita todos os vértices que estão na mesma componente conexa de s.
- Vamos agora mostrar $D[v]$, calculado por BFS(G, s), é igual a dist(s, v), para todo $v \in V(G)$. Precisaremos dos dois seguintes resultados auxiliares.

Lema 6.5. *Sejam G um grafo e $s \in V(G)$. Na execução de BFS(G, s), se u e v são dois vértices que estão na fila e u entrou na fila antes de v, então*

$$D[u] \leq D[v] \leq D[u] + 1 \ .$$

Demonstração. Vamos mostrar o resultado por indução na quantidade de iterações do laço **enquanto** na execução de BFS(G, s).

Como caso base, considere que houve zero iterações do laço. Nesse caso, a fila possui apenas s e não há o que provar.

Suponha agora que logo após a $(\ell - 1)$-ésima iteração do laço **enquanto** a fila é $F = (x_1, \ldots, x_k)$ e que vale $D[x_i] \leq D[x_j] \leq D[x_i] + 1$ para todos os pares x_i e x_j com $i < j$ (isto é, x_i entrou na fila antes de x_j).

Considere agora a ℓ-ésima iteração do laço **enquanto**. Seja $F = (x_1, \ldots, x_k)$ no início dessa iteração. Durante a iteração, o algoritmo remove x_1 de F e adiciona seus vizinhos não visitados, digamos $w_1, \ldots, w_h$ a F, de modo que agora temos $F = (x_2, \ldots, x_k, w_1, \ldots, w_h)$. Ademais, o algoritmo faz $D[w_j] = D[x_1] + 1$ para todo w_j. Utilizando a hipótese de indução, sabemos que para todo $1 \leq i \leq k$ temos

$$D[x_1] \leq D[x_i] \leq D[x_1] + 1 \ .$$

Assim, para qualquer w_j e para qualquer x_i temos, pela desigualdade acima,

$$D[x_i] \leq D[x_1] + 1 = D[w_j] = D[x_1] + 1 \leq D[x_i] + 1 \ .$$

Portanto, pares de vértices do tipo x_i, w_j satisfazem a conclusão do lema, para quaisquer $2 \leq i \leq k$ e $1 \leq j \leq h$. Já sabíamos, por hipótese de indução, que pares do tipo x_i, x_j também satisfazem a conclusão do lema. Ademais, pares de vizinhos de x_1, do tipo w_i, w_j, também satisfazem pois têm a mesma estimativa de distância ($D[x_1]+1$). Portanto, todos os pares de vértices em $\{x_2, \ldots, x_k, w_1, \ldots, w_h\}$ satisfazem a conclusão do lema. □

Lema 6.6. *Sejam G um grafo e $s \in V(G)$. Ao fim de BFS(G, s) vale que $D[v] \geq \text{dist}(s, v)$ para todo $v \in V(G)$.*

Demonstração. Vamos mostrar o resultado por indução na quantidade k de vértices adicionados à fila.

Se $k = 1$, então o único vértice adicionado à fila é s, antes do laço **enquanto** começar. Nesse ponto, temos $D[s] = 0 \geq \text{dist}(s, s) = 0$ e $D[v] = \infty \geq \text{dist}(s, v)$ para todo $v \neq s$, e portanto o resultado é válido.

Suponha agora que se x é um dos primeiros $k - 1$ vértices inseridos na fila, então $D[x] \geq \text{dist}(s, x)$. Considere o momento em que o algoritmo realiza a k-ésima inserção na fila, e seja v o vértice que foi adicionado. Note que v foi considerado no laço **para** da linha 8 por ser vizinho de algum vértice u que foi removido da fila. Então u foi um dos $k - 1$ primeiros a serem inseridos na fila e, por hipótese de indução, temos que $D[u] \geq \text{dist}(s, u)$. Note que para qualquer aresta uv temos $\text{dist}(s, v) \leq \text{dist}(s, u) + 1$. Assim, combinando esse fato com o que é feito na linha 12, obtemos

$$D[v] = D[u] + 1 \geq \text{dist}(s, u) + 1 \geq \text{dist}(s, v) \ .$$

Como um vértice entra na fila somente uma vez, o valor em $D[v]$ não muda mais durante a execução do algoritmo. Logo, $D[v] \geq \text{dist}(s, v)$ para todo $v \in V(G)$ ao fim da execução. □

Teorema 6.7. *Sejam G um grafo e $s \in V(G)$. Ao fim de BFS(G, s) vale que $D[v] = \text{dist}(s, v)$ para todo $v \in V(G)$.*

Demonstração. Já sabemos, pelo Lema 6.6, que $D[v] \geq \text{dist}(s, v)$ para todo $v \in V(G)$. Agora mostraremos que $D[v] \leq \text{dist}(s, v)$ para todo $v \in V(G)$.

Suponha, para fins de contradição, que ao fim da execução de BFS(G, s) existe ao menos um $x \in V(G)$ com $D[x] > \text{dist}(s, x)$. Seja v o vértice com menor valor $\text{dist}(s, v)$ para o qual isso acontece, isto é, tal que $D[v] > \text{dist}(s, v)$.

Considere um sv-caminho mínimo $(s, \ldots, u, v)$. Note que $\text{dist}(s, v) = \text{dist}(s, u)+1$. Pela escolha de v e como $\text{dist}(s, u) < \text{dist}(s, v)$, temos que $D[u] = \text{dist}(s, u)$. Assim,

$$D[v] > \text{dist}(s, v) = \text{dist}(s, u) + 1 = D[u] + 1 \ . \tag{1}$$

Considere o momento em que BFS(G, s) remove u de F. Se nesse momento o

vértice v já estava visitado, então algum outro vizinho $w \neq u$ de v já entrou e saiu da fila, visitando v. Nesse caso, fizemos $D[v] = D[w] + 1$ e, pelo Lema 6.5, $D[w] \leq D[u]$, de forma que $D[v] \leq D[u] + 1$, uma contradição com (1). Assim, assuma que v não havia sido visitado neste momento. Nesse caso, quando v entrar na fila (certamente entra, pois é vizinho de u), teremos $D[v] = D[u] + 1$, que é também uma contradição com (1). □

- A árvore T resultante da BFS é chamada *árvore de busca em largura* ou *árvore de BFS*.
  - Ela é geradora na componente conexa que contém s.
  - O sv-caminho existente nela tem $\text{dist}(s, v)$ arestas.
  - O que podemos dizer sobre $\text{dist}(x, y)$ quando $x \neq s$ e $y \neq s$?
  - O que podemos dizer de uma aresta $e \in E(G) \setminus E(T)$?
  - Como encontrar ciclos?
  - Como encontrar todas as componentes conexas?