

---

# Busca em Extensão

---

*"Diz o tolo em seu coração: 'Deus não existe'.  
Eles se corromperam e cometem atos detestáveis;  
não há ninguém que faça o bem  
[do modo aceitável a Deus]".*

(Salmo 14:1)

# **Buscas em Grafos**

---

- Em inglês: **search**
  - Vou usar “busca” ao invés de “pesquisa”
- Porém, **busca em grafos** não é apenas para procurar algo
- A ideia é a de **percorrer o grafo**

# **Uma Busca Ingênua**

---

- Várias formas de “percorrer”
  - Diferentes propósitos/aplicações
- Uma forma simples, um tanto “ingênua”

Para cada vértice  $v$ , de 0 a  $n-1$   
analisar as entradas  $\text{Adj}[v]$

- O que dá para descobrir com algum algoritmo com esta forma?

# **Sobre as Buscas**

---

- Como poderíamos:
  - Descobrir se um grafo não-direcionado é “conectado”?
  - Descobrir se um digrafo tem ciclo?
  - Descobrir o menor caminho entre dois pontos?
  - Descobrir um caminho que passe por todas as arestas?
- É preciso percorrer o grafo de formas mais elaboradas
  - Buscas!

# **Buscas em Grafos**

---

- As buscas (ou pesquisas) em grafos são estratégias algorítmicas usadas para percorrer os vértices de um grafo, seguindo as suas arestas
  - É um arcabouço de algoritmo
- É a base para vários algoritmos de grafos
  - Mas nem todo algoritmo de grafo é chamado “busca”!

# **Buscas em Grafos**

---

- Veremos duas estratégias de busca básicas
  - Busca em extensão/largura
  - Busca em profundidade
- Depois, veremos alguns algoritmos que podem ser vistos como um outro tipo de busca
  - Busca em prioridade

---

# **Busca em Extensão**

---

# **Busca em Extensão**

---

- Também chamada de **busca em largura**, mas a tradução é difícil
- **Breadth-first search**
  - Tradução literal: “busca primeiro na largura”
- Um nome mais adequado (não usado):  
**“busca primeiro nos arredores”**

# **Busca em Extensão**

---

- Iniciada em um vértice  $s$  (start) qualquer
- Visita todos vizinhos de  $s$ 
  - Depois todos os vizinhos desses vizinhos
    - Depois todos os vizinhos dos vizinhos dos...
- Visita todos os vértices de distância  $k$ , depois todos os de distância  $k+1$

# Busca em Extensão

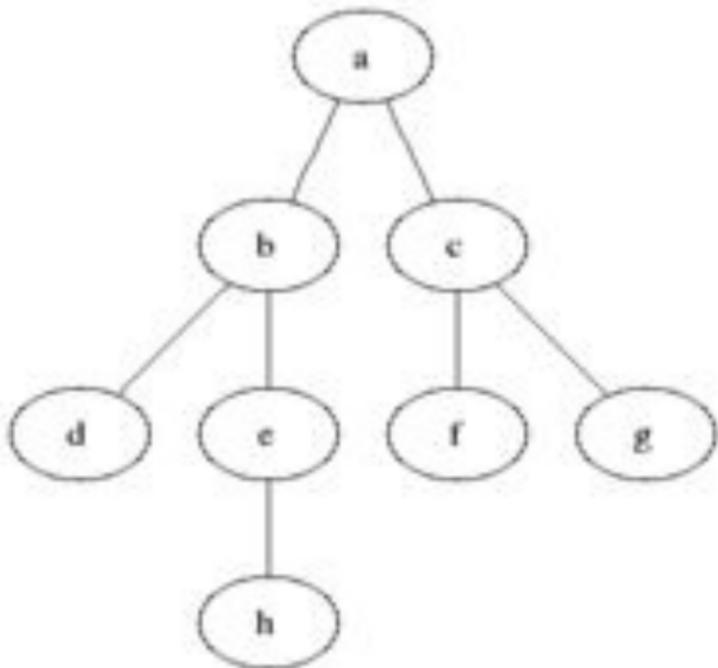
---

- Seguindo a definição do livro, cada vértice pode estar em três estados (cores)
  - **Branco**: não-visitado
  - **Cinza**: a ser visitado
  - **Preto**: já visitado
- Vértices cinzas formam uma “fronteira” entre brancos e pretos
  - São mantidos em uma fila **Q**

# Animações

---

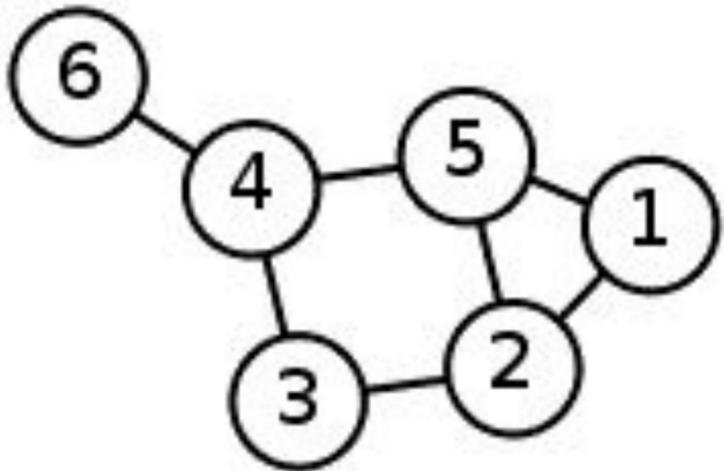
- Busca em extensão em uma árvore



# Animações

---

- Busca em extensão em um grafo qualquer



# Busca em Extensão

---

- Pseudo-código (v1)

```
BFS (grafo G, vértice s)
    atribui BRANCO a todos os vértices
    põe s em uma fila Q
    atribui CINZA a s
    enquanto Q não for vazia
        u = primeiro da fila (remove)
        para todo v em Adj[u] que for BRANCO
            adiciona v no fim de Q
            atribui CINZA a v
        atribui preto a u
```

# Sobre as Cores

---

- Usaremos as seguintes variáveis
  - **cor[]** : guarda a cor de cada vértice
- Exemplo: **cor[5]**
  - Valor que representa a cor do vértice 5
- Qual tipo de dado usar?
  - Pode usar um “enum”
  - Pode usar tipo “int”, separando 3 valores

# Sobre as Cores

---

- Alternativa: tratar como **atributo** do vértice

- Criar uma classe/struct Vertice, ou VerticeInfo, etc.
  - Definir atributo “cor”:

v.cor

- Problema: classe **Vertice** (do grafo) com atributos específicos de algoritmo

# Busca em Extensão

---

- Pseudo-código (v2)

```
BFS (grafo G, vértice s)
    para cada vértice u
        cor[u] = BRANCO;
    cor[s] = CINZA;
    Enfileira(Q, s);
    While ( ! EstahVazia(Q) )
        u = Desenfileira(Q);
        para cada v na lista Adj[u]
            if (cor[v] == BRANCO)
                cor[v] = CINZA;
                Enfileira (v, Q);
    cor[u] = PRETO;
```

# Sobre as Cores

---

- Na verdade, para a maioria das aplicações basta marcar **branco** (não-atingido) e **cinza** (atingido)
- Pode ser meramente um array de booleanos

---

# **Calculando Menores Caminhos**

---

# Aplicação

---

- A busca em extensão pode ser usada para encontrar os **caminhos de menor comprimento** (ou de comprimento mínimo)
  - Vou chamar de **menores caminhos**
- Em especial, cada chamada do algoritmo vai calcular os menores caminhos que saem de  $s$  e chegam para cada um dos demais vértices

# **Aplicação**

---

- Vamos mostrar um pseudo-código melhorado capaz de calcular os caminhos mínimos
- Algumas questões a serem tratadas no pseudo-código:
  - Como representar o caminho percorrido de  $s$  até cada vértice?
  - Como representar o comprimento de cada um desses caminhos?

# Busca em Extensão

---

- Usaremos as seguintes variáveis
  - **cor[]** : guarda a cor de cada vértice
  - **d[]** : guarda o comprimento (distância) do caminho desde s até um vértice qualquer
  - **π[]** ou **ante[]**: guarda o antecessor de cada vértice, no caminho de s até ele
    - Ou seja, ante[u] guarda o antecessor de u no caminho

# Busca em Extensão

---

- Pseudo-código (v3)

```
BFS (grafo G, vértice s)
```

```
    para cada vértice u
        cor[u] = BRANCO; d[u] = ∞; ante[u] = NIL;
    cor[s] = CINZA; d[s] = 0;
    Enfileira(Q, s);

    While ( ! EstahVazia(Q) )
        u = Desenfileira(Q);

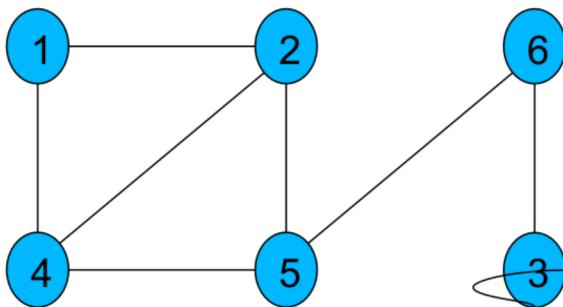
        para cada v na lista Adj[u]
            if (cor[v] == BRANCO)
                cor[v] = CINZA; d[v] = d[u]+1; ante[v] = u;
                Enfileira (v, Q);

        cor[u] = PRETO;
```

# Exemplo

---

- Busca no grafo



- Mostrar listas de adjacências
- Executar **BFS(G,1)** mostrando evolução da fila **Q** a cada iteração

# **Saídas do Algoritmo**

---

- Podemos considerar que as saídas do algoritmo são os arrays
  - $d[]$
  - $\text{pred}[]$
- Veremos, a seguir, as propriedades desses dois arrays após a execução da busca em extensão

## **Array $d[]$**

---

- O array  $d[]$  guarda o comprimento mínimo do caminho de  $s$  até cada vértice
- Ou seja,  $d[u] = \text{comprimento do menor caminho de } s \text{ a } u$
- Se um vértice  $u$  não for conectado a  $s$ , então teremos  $d[u]=\infty$

# **Array $d[]$**

---

- Exemplo:
  - Executar **BFS(G,1)** no grafo anterior (ou seja,  $s=1$ )
  - Após a execução,  $d[6]$  terá o comprimento do caminho que vai do vértice 1 até o vértice 6, ou seja, teremos
$$d[6] = 3$$

## **Array ante[]**

---

- O array ante[] permite recuperar seqüência de vértices que formam o caminho de mínimo partindo de s até cada vértice do grafo
- Na verdade, ante[u] = antecessor do vértice u, no caminho de comprimento mínimo
- Para recuperar o caminho completo (invertido) até um vértice u, é preciso acessar o antecessor de cada vértice sucessivamente até chegar a s

$u \leftarrow \text{ante}[u] \leftarrow \text{ante}[\text{ante}[u]] \leftarrow \dots \leftarrow s$

## **Array $\text{ante}[]$ (cont.)**

---

- No exemplo do grafo anterior, teríamos:
  - $\text{ante}[6] = 5$
  - $\text{ante}[5] = 2$
  - $\text{ante}[2] = 1$  (lembrando que, naquele exemplo,  $s=1$ )
- Isso representa o seguinte caminho de comprimento mínimo (representado invertido):

$6 \leftarrow 5 \leftarrow 2 \leftarrow 1$

# Imprimir Caminho

---

- Imprimir menor caminho entre s e v
  - Considerando s e ante[] como globais
  - Precisa ter rodada BFS com início no mesmo vértice s antes

```
PRINT-PATH (vértice v)
```

```
if (v == s)
    print(s);
else
    if (ante[v] == NIL)
        print("não há caminho");
    else
        PRINT-PATH(s, ante[v]);
        print(v);
```

## **Array ante[] (cont.)**

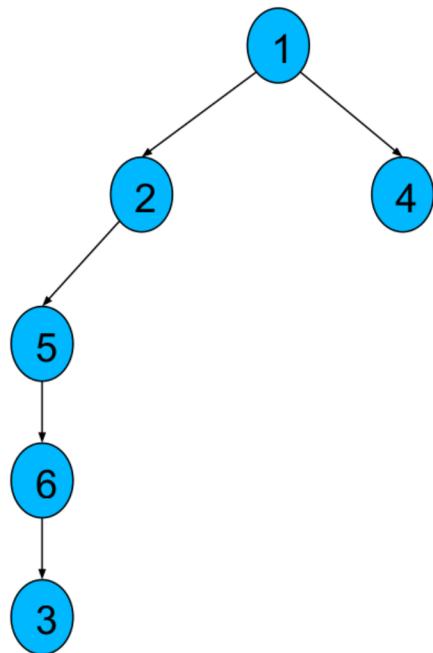
---

- O array ante[] define uma árvore com raiz no vértice de início s
- Ela pode ser chamada de “árvore de menores caminhos com origem em s”
- Podemos entender ante[u] como sendo o pai de u nessa árvore

# Propriedades

---

- Árvore de caminhos de comprimento mínimo com origem no vértice 1



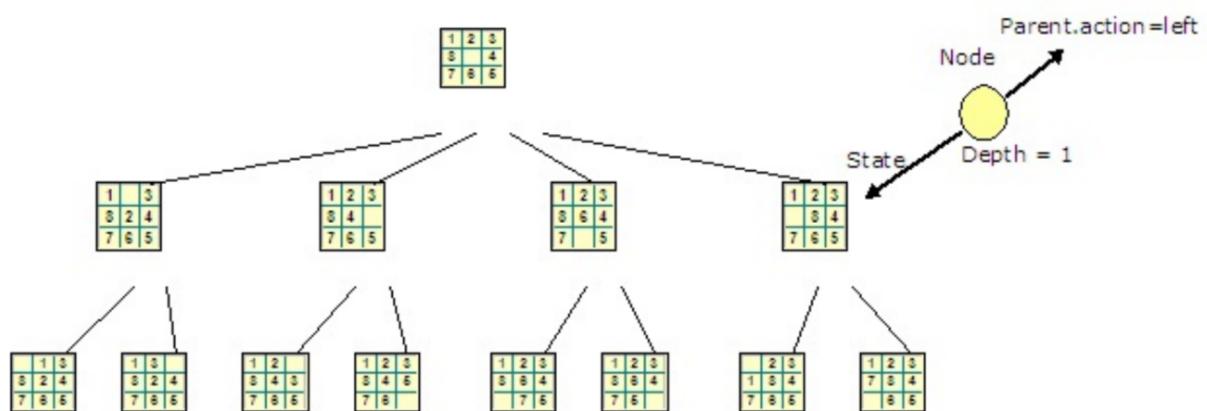
# **Aplicação**

---

- Situações reais que podem ser tratadas como o problema de achar os menores caminhos em um grafo sem peso
  - Menor caminho em um tabuleiro, movendo uma casa por vez (ex: sokoban)
  - Em um mapa, achar o caminho que passa pelo menor número de ruas

# Aplicação

- A busca em extensão também é estudada como técnica de busca da IA:
  - Achar a solução mais rápida para quebra-cabeças com peças deslizantes (como o que vem com o Vista)



---

# **Comentários Finais**

---

# **Complexidade de Tempo**

---

- Etapas mais “repetidas”:
  - Inicializa atributos para cada vértice
    - Proporcional a **V** instruções
  - Analisa todas as arestas (analisa as listas de adjacências de cada vértice)
    - Proporcional a **E** instruções
- Assim, a complexidade seria precisamente expressa assim:

$$\Theta(V+E)$$

# Complexidade de Tempo

---

- Em geral, vamos considerar aplicações em que, em média, cada vértice é extremo de, pelo menos, 1 aresta
  - Grau médio  $\geq 1$
  - Logo:  $V \leq \text{soma dos graus} = 2.E$   
 $V = O(E)$
- Neste caso, a complexidade de tempo da busca em largura pode ser expressa assim:

**O(E)**

# **Observações Finais**

---

- O nome **busca em extensão**, na verdade, não expressa um algoritmo específico
- Ele designa uma estratégia usada para escolher a ordem de visita dos vértices
- Aqui, essa estratégia foi usada especificamente em um **algoritmo para achar os menores caminhos**

# Observações Finais

---

- Porém, essa estratégia pode ser usada como base para **outros algoritmos**, que calculam outras propriedades
- Exemplo:
  - Testar se um grafo não-direcionado é conectado
- Estes outros algoritmos também podem ser chamados de “buscas em extensão”