AED2 - Algoritmos e Estr. de Dados II

Aula 15: Busca em Largura (BFS)

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br

CEFET-MG - Campus Timoteo
Dep. Engenharia de Computação

Fevereiro de 2026

Sumário

1. Motivação
2. BFS: Ideia Central
3. O Algoritmo
4. Exemplo Passo a Passo
5. Implementação em Java
6. BFS em Grids
7. Multi-Source BFS
8. Aplicações e Armadilhas
9. BFS vs DFS
10. Exercícios

Motivação: O Problema do Menor Caminho

Problema real: Você está em uma rede social. Quantos "saltos" separam você de outra pessoa?

- Nós = pessoas; Arestas = amizades
- Queremos a **menor quantidade de arestas** entre dois nós
- Pesos? Não! Todas as arestas valem **1**

Pergunta

Por que não usar DFS para isso?

Exemplos do dia a dia

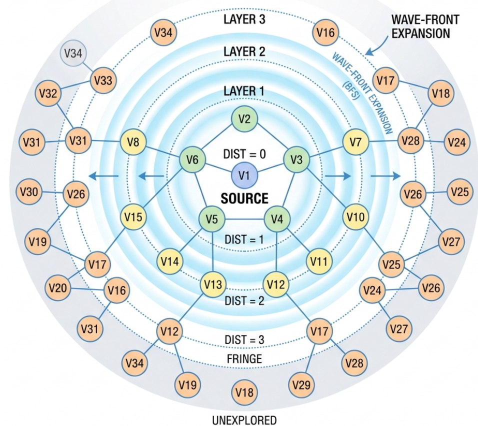
- GPS: menor número de ruas
- Jogos: menor caminho em labirintos
- Redes: menor número de roteadores
- Web: grau de separação entre páginas

BFS: Explorando em Ondas

Ideia: Explore todos os vizinhos *antes* de ir mais fundo.

Imagine jogar uma pedra num lago:

- **Onda 0:** nó inicial s
- **Onda 1:** vizinhos de s (distância 1)
- **Onda 2:** vizinhos dos vizinhos (distância 2)
- **Onda k:** nós a exatamente k passos de s



Garantia

O primeiro momento em que BFS alcança um vértice v, a distância registrada é **mínima**.

A Fila: Estrutura Central da BFS

Por que uma Fila (FIFO)?

Queremos processar os nós *na ordem em que foram descobertos*:

- Todos os nós de distância 1 antes dos de distância 2
- Todos os de distância 2 antes dos de distância 3
- . . . e assim por diante

Estados de um vértice:

- **Não descoberto:** nunca visto
- **Na fila:** descoberto, aguardando
- **Processado:** saiu da fila, vizinhos checados

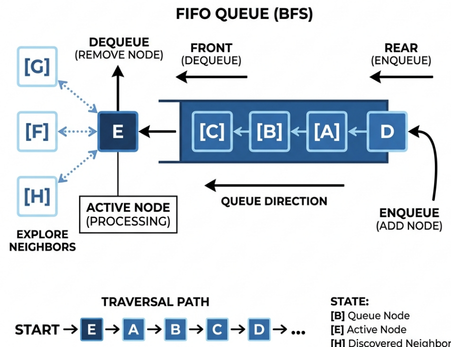


Figure: Estado da fila durante a BFS.

Pseudocódigo da BFS

BFS(G, s)

1. Para todo $v \in V$: $\text{dist}[v] \leftarrow \infty$, $\text{pai}[v] \leftarrow \text{nil}$
2. $\text{dist}[s] \leftarrow 0$
3. $Q \leftarrow \{s\}$ // *enfileira s*
4. **enquanto** $Q \neq \emptyset$:
   - 4.1 $u \leftarrow \textsc{Dequeue}(Q)$
   - 4.2 **para** cada $v \in \text{Adj}[u]$:
     - 4.2.1 **se** $\text{dist}[v] = \infty$:
     - 4.2.2 $\text{dist}[v] \leftarrow \text{dist}[u] + 1$
     - 4.2.3 $\text{pai}[v] \leftarrow u$
     - 4.2.4 $\textsc{Enqueue}(Q, v)$

Pontos-chave

- Marcamos **ao enfileirar**, não ao processar — evita duplicatas
- `pai[]` permite **reconstruir** o caminho
- $\text{dist}[v] = \infty$ significa "não visitado"

Complexidade

- **Tempo:** $O(V + E)$
- **Espaço:** $O(V)$

Exemplo: Grafo com 7 Vértices

Grafo não-direcionado com vértices $\{0, 1, 2, 3, 4, 5, 6\}$:

Arestas: 0—1, 0—2, 1—3, 1—4, 2—5, 2—6

Lista de Adjacência:

- 0: [1, 2]
- 1: [0, 3, 4]
- 2: [0, 5, 6]
- 3: [1] 4: [1] 5: [2] 6: [2]

Origem: vértice 0

Estrutura em árvore

0

1 **2**

3 **4** **5** **6**

Nível 0: {0}
Nível 1: {1, 2}
Nível 2: {3, 4, 5, 6}

Rastreamento da BFS — Passo a Passo

Passo	Processa	Fila após	dist[] atualizado
Início	—	[0]	dist[0]=0, demais=∞
1	**0**	[1, 2]	dist[1]=1, dist[2]=1
2	**1**	[2, 3, 4]	dist[3]=2, dist[4]=2
3	**2**	[3, 4, 5, 6]	dist[5]=2, dist[6]=2
4	**3**	[4, 5, 6]	(0 já visitado, nada novo)
5	**4**	[5, 6]	(0 já visitado)
6	**5**	[6]	(0 já visitado)
7	**6**	[]	(0 já visitado)

Distâncias finais a partir de 0:

- dist[0]=0, dist[1]=1, dist[2]=1
- dist[3]=2, dist[4]=2, dist[5]=2, dist[6]=2

Observe

Vértices do mesmo nível são processados *consecutivamente*. A fila nunca mistura níveis k e $k+2$ sem antes terminar $k+1$.

Reconstruindo o Caminho com `pai[]`

Vetor pai[] após BFS a partir de 0:

vértice	0	1	2	3	4	5	6
pai	-1	0	0	1	1	2	2

Caminho de 0 até 6:

1. Parta de 6: pai[6] $= 2$
2. Vá a 2: pai[2] $= 0$
3. Vá a 0: pai[0] $=$ -1 $\rightarrow$ raiz!
4. Inverta: $0 \rightarrow 2 \rightarrow 6$

Caminho 0 → 6

$$\mathbf{0} \xrightarrow{+1} \mathbf{2} \xrightarrow{+1} \mathbf{6}$$

Distância $=$ **2 arestas**

Algoritmo getPath

```
v = target
while v != -1:
  path.add(v)
  v = pai[v]
reverse(path)
```


Prof. Aléssio Miranda Júnior - alessio@cefetmg.br
CEFET-MG - Campus Timóteo

Implementação em Java — BFS Completa

```java
import java.util.*;

public class BFS {
    public static int[] bfs(int start, int V, List<List<Integer>> adj) {
        int[] dist = new int[V];
        int[] pai  = new int[V];
        Arrays.fill(dist, -1);
        Arrays.fill(pai,  -1);

        Queue<Integer> fila = new LinkedList<>();
        dist[start] = 0;
        fila.add(start);

        while (!fila.isEmpty()) {
            int u = fila.poll();
            for (int v : adj.get(u)) {
                if (dist[v] == -1) {      // nao visitado
                    dist[v] = dist[u] + 1;
                    pai[v]  = u;
                    fila.add(v);
                }
            }
```

BFS em Matrizes (Grids)

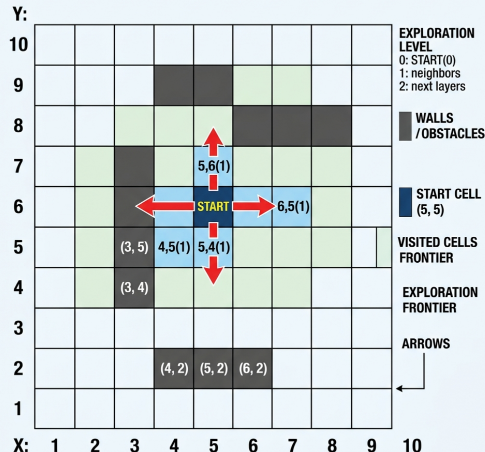
Muitos problemas usam **matrizes 2D** como grafos:

- Cada célula `(r,c)` é um vértice
- Vizinhos: cima, baixo, esq., dir.
- Célula `'#'` = parede (bloqueada)

Exemplo — Labirinto 5×5**:**

```
S . # . .
. . # . #
# . . . #
# # # . .
. . . . E
```

S=início, **E**=fim, **#**=parede



Implementação BFS em Grid

```java
int[] dr = {-1, 1, 0,  0};  // cima, baixo
int[] dc = { 0, 0, -1, 1};  // esq, dir

int bfsGrid(char[][] g, int sr, int sc, int er, int ec) {
    int R = g.length, C = g[0].length;
    int[][] dist = new int[R][C];
    for (int[] row : dist) Arrays.fill(row, -1);

    Queue<int[]> fila = new LinkedList<>();
    dist[sr][sc] = 0;
    fila.add(new int[]{sr, sc});

    while (!fila.isEmpty()) {
        int[] p = fila.poll();
        int r = p[0], c = p[1];
        if (r == er && c == ec) return dist[r][c];

        for (int d = 0; d < 4; d++) {
            int nr = r + dr[d], nc = c + dc[d];
            if (nr>=0 && nr<R && nc>=0 && nc<C
                    && g[nr][nc] != '#' && dist[nr][nc]==-1) {
                dist[nr][nc] = dist[r][c] + 1;
```

Multi-Source BFS

Problema: Dado um grid com *múltiplas* fontes, qual a distância de cada célula à fonte mais próxima?

Exemplo: Três quartéis de bombeiros em uma cidade. Qual o tempo de chegada a cada bairro?

Solução elegante

1. Enfileire **todas** as fontes com dist=0
2. Execute BFS normalmente

As "ondas" se expandem simultaneamente. Cada célula recebe a distância da fonte mais próxima.

Inicialização Multi-Source

```
for (int[] src :  fontes) {
  dist[src[0]][src[1]] = 0;
  fila.add(src);
}
// BFS normal a seguir
```

Aplicações

- Distância a saídas de emergência
- Propagação de fogo/vírus
- Voronoi em grids

Aplicações Clássicas da BFS

1. Caminho mínimo (sem peso)

Redes, labirintos, puzzles. Qualquer grafo onde todas as arestas custam 1.

2. Componentes Conexos

Execute BFS a partir de cada vértice não visitado. Cada chamada descobre um componente.

3. Verificar Bipartição

Tente colorir o grafo com 2 cores durante a BFS. Se encontrar conflito, não é bipartido.

4. Nível de um Nó (Árvores)

Profundidade de todos os nós com uma única BFS a partir da raiz.

5. Edmonds-Karp (Fluxo)

Usa BFS para encontrar o caminho aumentante mais curto no algoritmo de fluxo máximo.

6. Web Crawlers

Exploram links nível a nível para indexar páginas próximas primeiro.

Armadilhas Comuns

Erro 1: Marcar ao processar, não ao enfileirar

Se você marcar `visited` só quando *retira* da fila, o mesmo vértice pode ser adicionado **múltiplas vezes**, causando $O(E)$ duplicatas e loop em grafos densos.

Correto: marcar *imediatamente* ao enfileirar.

Erro 2: Usar pilha em vez de fila

Pilha (Stack) vira DFS, não BFS. A ordem de visita muda completamente e o caminho mínimo **não** é garantido.

Erro 3: Grafos desconexos

BFS a partir de um vértice só visita seu componente. Se o grafo for desconexo, é preciso iterar sobre *todos* os vértices e chamar BFS para os não visitados.

Dica: Grafos com pesos

BFS assume arestas de custo 1. Para pesos diferentes use:

- **Dijkstra**: pesos positivos
- **0-1 BFS** (deque): pesos 0 ou 1
- **Bellman-Ford**: pesos negativos

BFS vs DFS: Quando Usar Cada Um?

Critério	**BFS**	**DFS**
Estrutura auxiliar	Fila	Pilha / Recursão
Caminho mínimo (sem peso)	**Sim**	**Não**
Detecta ciclos	Sim	**Sim (mais fácil)**
Ordenação topológica	Não	**Sim**
Componentes fortemente conexos	Não	**Sim (Kosaraju)**
Uso de memória	$O(V)$ fila (pode ser grande)	$O(V)$ pilha
Ideal para grafos largos	**Ruim**	**Melhor**
Ideal para grafos rasos	**Melhor**	**Ruim**

Use BFS quando o objetivo é **distância mínima**. *Use DFS para* **exploração profunda** *(topologia, SCCs, backtracking).*

1. **[Fácil]** Dado o grafo: $0-1$, $0-2$, $1-3$, $2-3$, $3-4$. Execute BFS a partir do vértice 0 e mostre o estado da fila a cada passo. Qual a distância de 0 a 4?
2. **[Médio]** Implemente em Java um método que verifica se um grafo não-direcionado é **bipartido** usando BFS. (Dica: tente colorir os vértices com 2 cores durante a BFS.)
3. **[Médio]** Dado um grid $N \times M$ com células livres (`'.'`) e obstáculos (`'#'`), encontre o **menor caminho** entre as células `(0,0)` e `(N-1,M-1)`. Retorne -1 se não houver caminho.
4. **[Difícil]** Em um grid com células com valores 0 ou 1, use **0-1 BFS** (deque) para encontrar o menor custo de `(0,0)` a `(N-1,M-1)`, onde mover para célula 0 custa 0 e para célula 1 custa 1.

Resumo da Aula

O que aprendemos

- BFS explora em **camadas** usando uma **Fila**
- Garante **caminho mínimo** em grafos não ponderados
- Complexidade $O(V + E)$ em tempo e $O(V)$ em espaço
- Aplicável a grafos e **grids** (labirintos)
- **Multi-Source BFS**: fontes múltiplas simultaneamente
- Marcar ao **enfileirar** é fundamental

Próximas aulas

- Dijkstra: BFS com pesos
- A*: BFS com heurística
- Bellman-Ford: pesos negativos

Lembre-se

"BFS = fila. DFS = pilha. Caminho mínimo = BFS."

Prof. Aléssio Miranda Júnior - alessio@cefetmg.br

CEFET-MG - Campus Timóteo