

## Busca em largura, caminhos mínimos em grafos não ponderados

Relembrando a busca genérica, mas usando um versão alternativa:

busca Genérica ( $G = (V, E)$ ,  $s$ )  
 marque todo  $v \in V$  como  $\tilde{n}$  visitado  
 - coloque  $s$  no conj. dos vértices ativos  
 → enquanto ativos  $\neq \emptyset$ :  
   gera um vértice  $v$  dos ativos  
 → se  $v$  ainda não foi visitado  
   marque  $v$  como visitado  
   para cada aresta  $(v, w)$ :  
     se  $w$  ainda  $\tilde{n}$  foi visitado  
       coloque  $w$  no conj. dos ativos

Pense no conjunto de vértices ativos,

- como os vértices encontrados mas não visitados.

Observe que o algoritmo anterior

- não para antes de considerar todas as arestas alcançáveis a partir de  $s$ ,
  - já que todo vértice visitado tem suas arestas analisadas.

Existem dois tipos de busca em grafo que são muito eficientes

- e cumprem funções bastante diferentes,
  - embora ambas sejam especializações da busca genérica.
- Uma delas é a busca em profundidade ou DFS (Depth-First Search),
  - que já estudamos exaustivamente.
- A outra é a busca em largura ou BFS (Breadth-First Search).

Hoje vamos nos aprofundar na BFS,

- que explora o grafo em camadas a partir de um vértice inicial  $s$ .
- Por isso, ela é particularmente útil
  - para calcular a distância não ponderada entre vértices.

A BFS explora o grafo em camadas a partir de um vértice inicial  $s$ ,

- e esse comportamento está intimamente relacionado
  - com a estrutura de dados fila (queue ou FIFO).

Pseudocódigo:

buscaLargura( $G=(V,E), s$ )

marque todo  $v \in V$  como ã encontrado  $O(n)$

→ marque  $s$  como encontrado

coloque  $s$  na fila  $Q$

$O(m_s)$  enquanto  $Q \neq \emptyset$ :

remova  $v$  do início de  $Q$

para cada aresta  $(v,w)$

(se  $w$  não foi encontrado

marque  $w$  como encontrado ←

insira  $w$  no final de  $Q$  ←

buscaGenérica( $G=(V,E), s$ )

marque todo  $v \in V$  como ã visitado

- coloque  $s$  no conj. dos vértices ativos

→ enquanto ativos  $\neq \emptyset$ :

remova um vértice  $v$  dos ativos

→ se  $v$  ainda não foi visitado ←

marque  $v$  como visitado

para cada aresta  $(v,w)$ :

(se  $w$  ainda ã foi visitado

coloque  $w$  no conj. dos ativos

Comparando a BFS com a busca genérica, três diferenças se destacam:

- Marcamos explicitamente quando um vértice é encontrado.
- Não marcamos quando um vértice é visitado.
- Não verificamos se o vértice saindo da fila já foi visitado.

Notem que, os conceitos de encontrado e visitado valem

- mesmo que o algoritmo não os registre explicitamente.
- Assim, o que justifica as mudanças anteriores?

Corretude:

- O algoritmo encontra todos os vértices alcançáveis a partir de  $s$ .
  - Esse resultado segue da corretude do algoritmo de busca genérica,
    - já que a busca em largura é um caso particular daquela.
- Além disso, o algoritmo de busca em largura
  - explora o grafo em camadas centradas em  $s$ ,
- mas isso vamos mostrar quando formos calcular distâncias.

Eficiência:

- O algoritmo leva tempo  $O(n)$  para marcar os vértices como não encontrados.
- O restante do algoritmo leva tempo  $O(n_s + m_s)$ ,
  - sendo  $n_s$  e  $m_s$ , respectivamente, os números de vértices e arestas
    - da componente conexa que contém o vértice  $s$ .
- Isso porque, em cada iteração do laço principal,
  - um vértice é removido da fila.
  - Logo, esse laço é executado  $O(n_s)$  vezes.
- Como cada vértice é colocado apenas uma vez na fila,
  - pois nunca inserimos vértices já encontrados,
  - cada aresta é visitada no máximo uma vez,
    - na iteração em que seu vértice origem sai da fila.
- Portanto, no total o algoritmo executa  $O(m_s)$  iterações do laço mais interno.

## Cálculo de distâncias

*gráfico não ponderado*

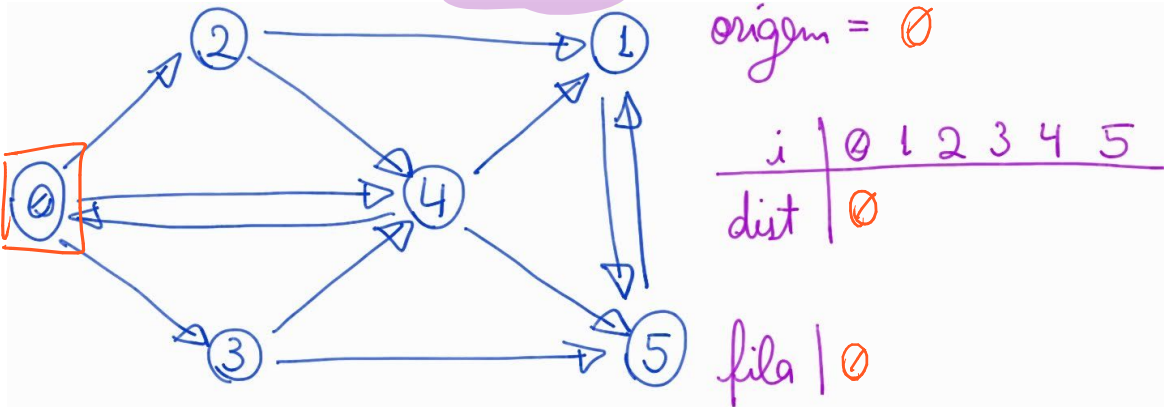
O comprimento de um caminho  $P$  é o número de arestas em  $P$ ,

- ou, de modo equivalente, o número de vértices em  $P - 1$ .

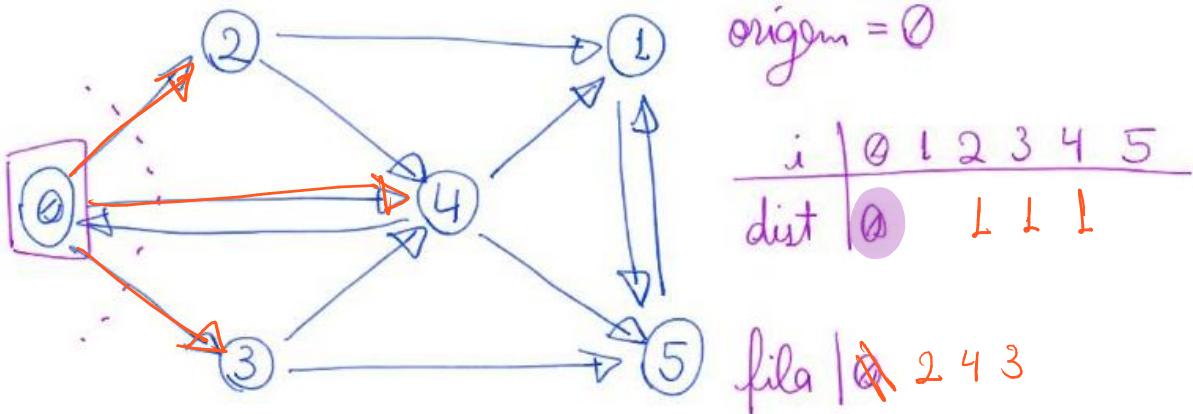


Exemplo 1:

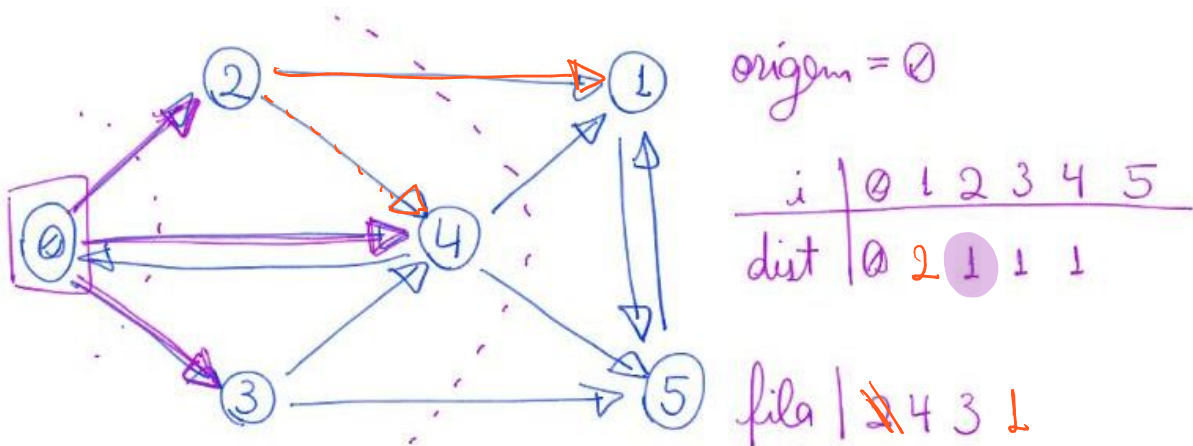
- No início apenas o vértice origem = 0 é alcançável e tem distância 0.



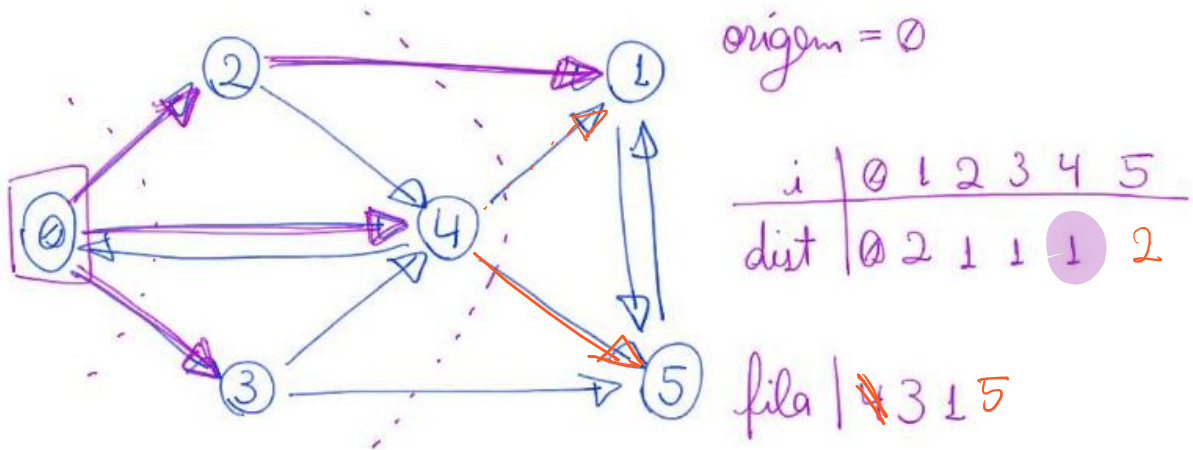
- Em cada iteração podemos encontrar novos vértices
  - e atualizar suas distâncias,
    - como sendo 1 a mais que a distância de quem o encontrou.



- Observe a importância de armazenar os vértices encontrados em uma fila
  - para preservar a ordem de descoberta
    - e assim calcular corretamente as distâncias.



- Por exemplo, se usássemos uma pilha, primeiro encontraríamos
  - o caminho que vai até 5 passando por 1, que tem comprimento 3.

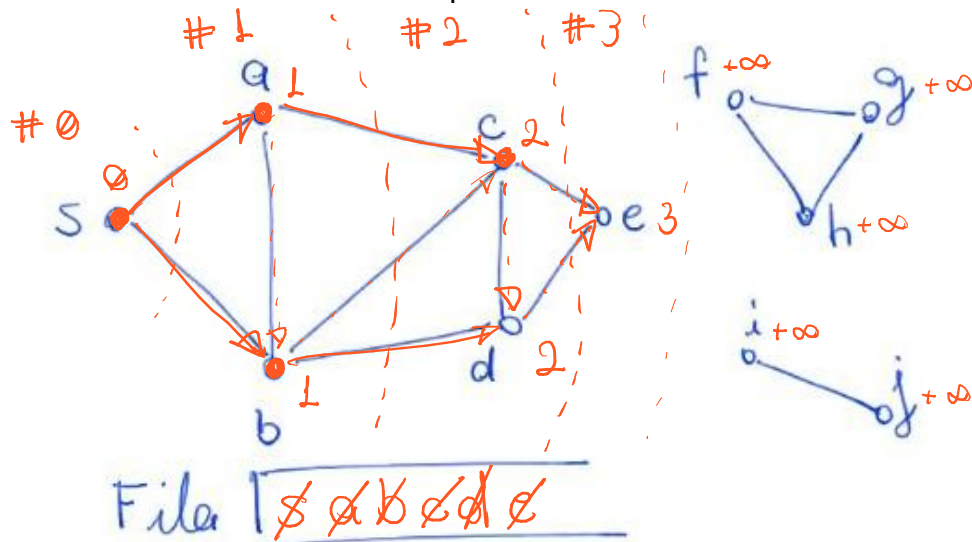


- Depois de alcançar todos os vértices,
  - ou quando a fila ficar vazia, podemos parar.

Pseudocódigo:

distâncias  $(G=(V,E), s)$   
 marque todo  $v \in V$  com  $dist[v] = +\infty$   
 $dist[s] = 0$   
 coloque  $s$  na fila  $Q$   
 enquanto  $Q \neq \emptyset$ :  
   remova  $v$  do início de  $Q$   
   para cada aresta  $(v,w)$   
     se  $dist[w] = +\infty$   
        $dist[w] = dist[v] + 1$   
       insira  $w$  no final de  $Q$

Exemplo 2: Observem o momento em que uma camada é concluída.



Suponha que  $s$  é a origem

Vamos mostrar que nosso algoritmo calcula corretamente

- a distância não ponderada de  $s$  até cada vértice  $v$ .
  - Para tanto, vamos usar indução matemática no número de iterações.

*comp. do menor caminho*

H.I.: Para todo vértice  $v$  encontrado até o início da iteração atual

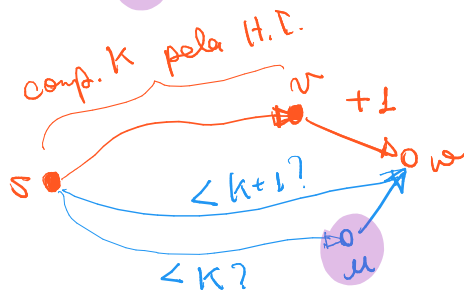
- temos que  $\text{dist}[v]$  é a distância de  $s$  até  $v$ .

*Além disso, os vértices são encontrados em ordem crescente de distância*

Caso base: No início da primeira iteração apenas  $s$  foi encontrado e  $\text{dist}[s] = 0$ , que é a menor distância

Passo: Queremos mostrar que a H.I. continua válida no final da iteração atual

- depois que encontramos novos vértices.
- Suponha que na iteração atual o algoritmo visita o vértice  $v$ ,
  - que tem distância  $\text{dist}[v] = k$ .
- Ao analisar os vizinhos de  $v$  encontramos, pela primeira vez,
  - o vértice  $w$  e atribuímos  $\text{dist}[w] = k + 1$ .



- Nosso objetivo é mostrar que  $k + 1$  é a distância de  $s$  até  $w$ .
- Pela hipótese de indução,
  - temos um caminho de  $s$  até  $v$  com comprimento  $k$ .
- Portanto, existe um caminho de  $s$  até  $w$  com comprimento  $k + 1$ .
- Mas, como garantir que esse é um caminho mínimo de  $s$  a  $w$ ?
  - Por que não pode haver um caminho de comprimento menor?

*fortalecer a H.I.*

*Se existir  $u$  e/ aresta  $(u, w)$  e  $\text{dist}[u] < k$*

*então, pela H.I.,  $u$  teria sido encontrado antes de  $v$*

*e, pelo func. do alg.,  $u$  teria sido visitado antes de  $v$ .*

*Portanto,  $w$  teria sido encontrado antes da iteração atual (contradição).*

*Assim, concluímos que  $u$  não existe e que distância de  $s$  até  $w$  é  $k+1$ .*

*Caso  $w$  tem distância  $k+1$ , precisamos mostrar que todo vértice e/ distância  $< k+1$  já foi encontrado.*

*Pela H.I., no início da iteração todo vértice havia sido encontrado em ordem crescente de distância.*

*pela H.I.*

*Como  $v$  tem distância  $k$ , sabemos que todo vértice e/ distância  $< k$  foi encontrado antes de  $v$ . Pelo funcionamento do alg., sabemos que todo vértice e/ distância  $< k$  foi visitado antes de  $v$ .*

*Assim, todo vértice e/ distância  $< k+1$  foi encontrado antes de  $w$  ou  $u$  encontrado.*

Código cálculo de distâncias com grafo implementado por listas de adjacência.

```

int *distancias(Grafo G, int origem) {
    int v, w, *dist;
    Fila *fila;
    Noh *p;
    - dist = malloc(G->n * sizeof(int));
    - fila = criaFila(G->n);
    /* inicializa todos como não encontrados, exceto pela origem */
    for (v = 0; v < G->n; v++)
        dist[v] = -1;
    - dist[origem] = 0;
    - insereFila(fila, origem);
    while (!filaVazia(fila)) {
        v = removeFila(fila);
        /* para cada vizinho de v que ainda não foi encontrado */
        p = G->A[v];
        while (p != NULL) {
            w = p->rotulo;
            -> if (dist[w] == -1) {
                /* calcule a distância do vizinho e o coloque na fila */
                dist[w] = dist[v] + 1;
                - insereFila(fila, w);
            }
            p = p->prox;
        }
    }
    - fila = liberaFila(fila);
    return dist;
}

```

$O(n)$

$O(m_s)$

$O(|\delta(v)|)$

$\sum_{v \in m_s} |\delta(v)| = m_s$

- Qual a eficiência deste algoritmo?
  - $O(n + n_s + m_s)$ , sendo  $s$  o vértice origem. Por que?

Quiz1: Considerem o grafo de uma grande rede social,

- com mais ou menos  $10^9$  vértices
  - e  $10^3$  arestas por vértice (grau médio dos vértices).
- Supondo que o grafo da rede social é conexo,
  - compare a eficiência de um algoritmo de cálculo de distâncias
  - que usa matriz de adjacência com um que usa listas de adjacência.
- Dica: lembre que, no caso da matriz de adjacência,
  - o algoritmo leva tempo  $O(n)$  para avaliar os vizinhos de cada vértice.

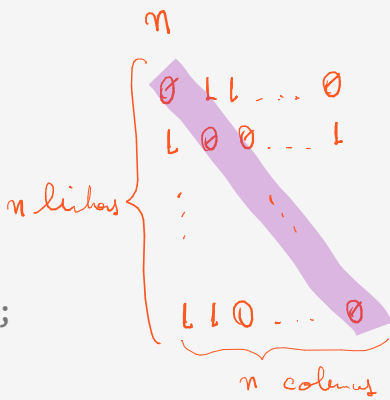


## Funções para ler grafos

Quiz2: Compare a eficiência das seguintes funções para leitura de grafos.

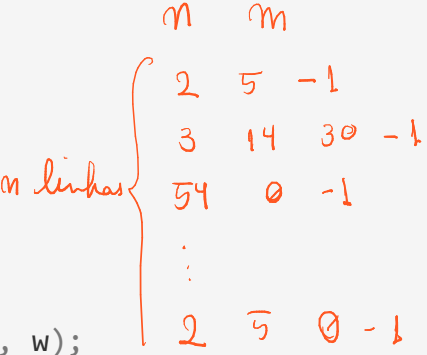
Função auxiliar para ler de arquivo grafo representado por matriz binária

```
Grafo lerGrafoMatriz(FILE *entrada) {  
    int n, v, w, value;  
    Grafo G;  
    fscanf(entrada, "%d\n", &n);  
    G = inicializaGrafo(n);  
    for (v = 0; v < G->n; v++)  
        for (w = 0; w < G->n; w++) {  
            fscanf(entrada, "%d", &value);  
            if (value == 1)  
                insereArcoNaoSeguraGrafo(G, v, w);  
        }  
    return G;  
}
```



Função auxiliar para ler de arquivo grafo em listas gerado por imprimeGrafo

```
Grafo lerGrafoImpresso(FILE *entrada) {  
    int n, m, v, w;  
    Grafo G;  
    fscanf(entrada, "%d %d\n", &n, &m);  
    G = inicializaGrafo(n);  
    for (v = 0; v < G->n; v++) {  
        fscanf(entrada, "%d", &w);  
        while (w != -1) {  
            insereArcoNaoSeguraGrafo(G, v, w);  
            fscanf(entrada, "%d", &w);  
        }  
    }  
    return G;  
}
```



Função auxiliar para ler de arquivo grafo em listas gerado por mostraGrafo

```

Grafo lerGrafoMostra(FILE *entrada) {
    int n, m, v, w, tam;
    Grafo G;
    char *str, *aux;
    fscanf(entrada, "%d %d\n", &n, &m);
    G = inicializaGrafo(n);
    tam = ((G->n * ((int)log10((double)G->n) + 1)) + 3) *
    sizeof(char); // Quiz3: o que justifica cada termo dessa expressão?
    str = malloc(tam);
    for (v = 0; v < G->n; v++) {
        fgets(str, tam, entrada);
        aux = strtok(str, ":");
        aux = strtok(NULL, "\n");
        while (aux != NULL) {
            w = atoi(aux);
            insereArcoNaoSeguraGrafo(G, v, w);
            aux = strtok(NULL, "\n");
        }
    }
    free(str);
    return G;
}

```

lê a entrada até encontrar \n ou fim de arquivo ou depois de ler tam-1

m linhas

0	:	1	2	5	4
1	:	0	13	25	
2	:	4	0		
...	:				
n-1	:	42	1		

Uma outra maneira comum para armazenar grafos em arquivos é

- uma linha com dois números n e m,
  - indicando o número de vértices e de arestas, respectivamente,
- seguida por m linhas, cada uma com um par u e v em {0, ..., n-1},
  - com cada par indicando um arco do vértice u para o vértice v.
- Quiz4: implementem um leitor de grafos que utiliza esse padrão.

m linhas

1	2	?
42	3	?
57	100	?
...		

0 → 2  
(1, 2)