AED2 - Aula 25
Busca em largura, caminhos mínimos não ponderados

Relembrando a busca genérica, mas usando um versão alternativa:

```
buscaGenerica(grafo G=(V,E), vértice s) {
    para v \in V
        marque v como não encontrado
    marque s como encontrado
    coloque s no conjunto de vértices ativos
    enquanto o conjunto de ativos não estiver vazio
        remova um vértice v dos ativos
        marque v como visitado
        coloque nos ativos todos os vizinhos de v
          ■ que ainda não foram visitados
}
```

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 do grafo,
    - já que toda aresta tem seus extremos em vértices.

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.

O comportamento da BFS está intimamente relacionado
- com a estrutura de dados fila (queue ou FIFO).

Pseudocódigo:

```
buscaLargura(grafo G=(V,E), vértice s) {
	para v \in V
		marque v como não encontrado
	marque s como encontrado
	seja Q uma fila inicializada com o vértice s
	enquanto Q != \empty
		remova um vértice 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
}
```

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 usarmos esse algoritmo para calcular distâncias.

Eficiência:

- O algoritmo leva tempo $O(n)$
    - para marcar todos os vértices do grafo 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.

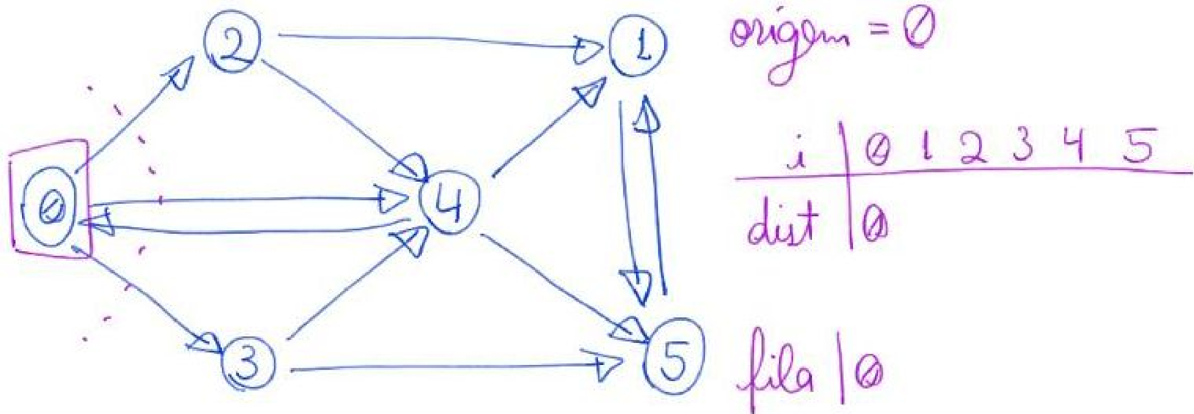
Cálculo de distâncias

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

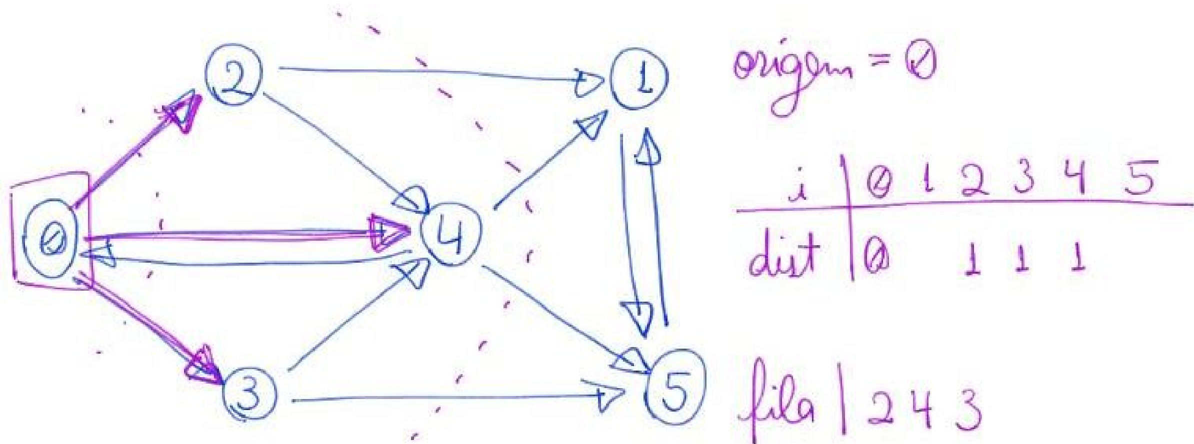
- ou, de modo equivalente, o número de vértices em P - 1.

Exemplo 1:

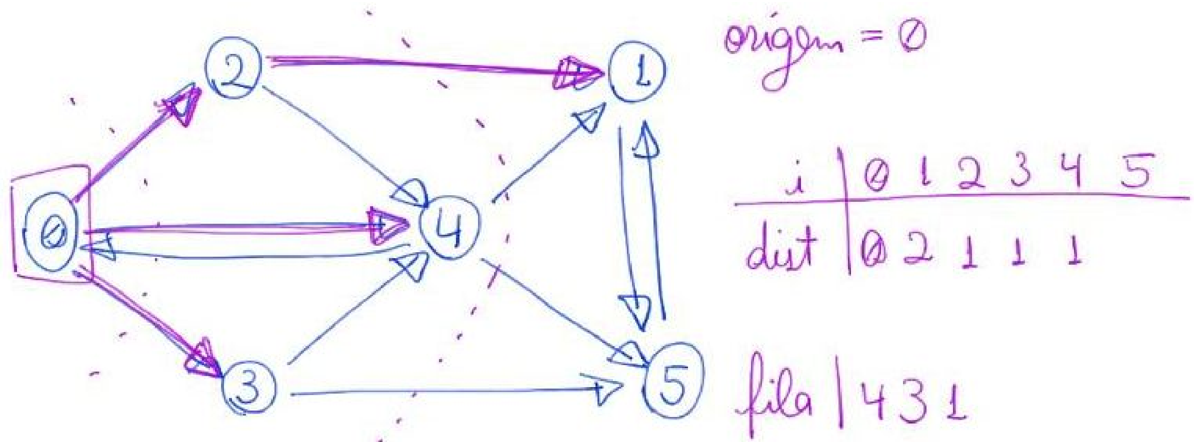
- No início apenas o vértice origem = 0 é alcançável e tem distância 0.



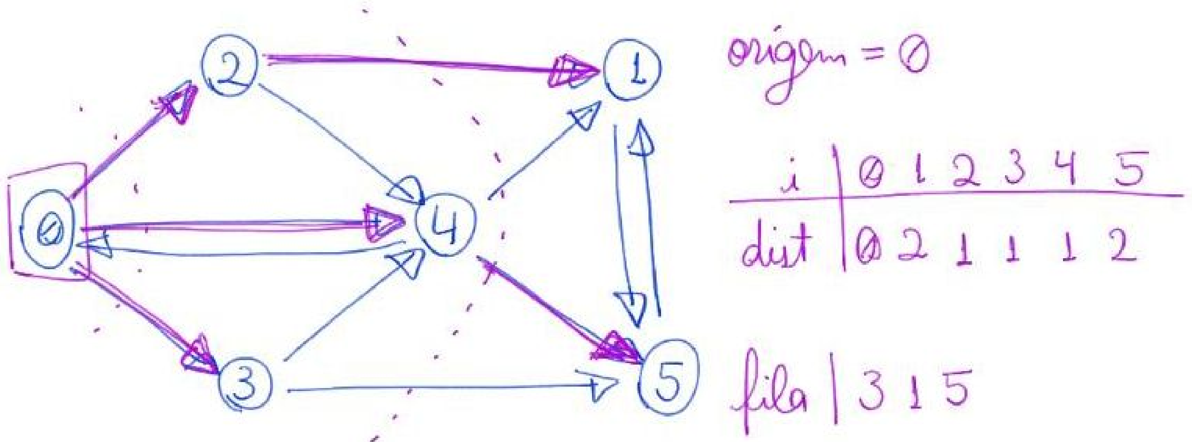
- Em cada iteração podemos encontrar novos vértices
    - e atualizar suas distâncias,
        - como sendo 1 a mais que a distância de quem o encontrou.



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



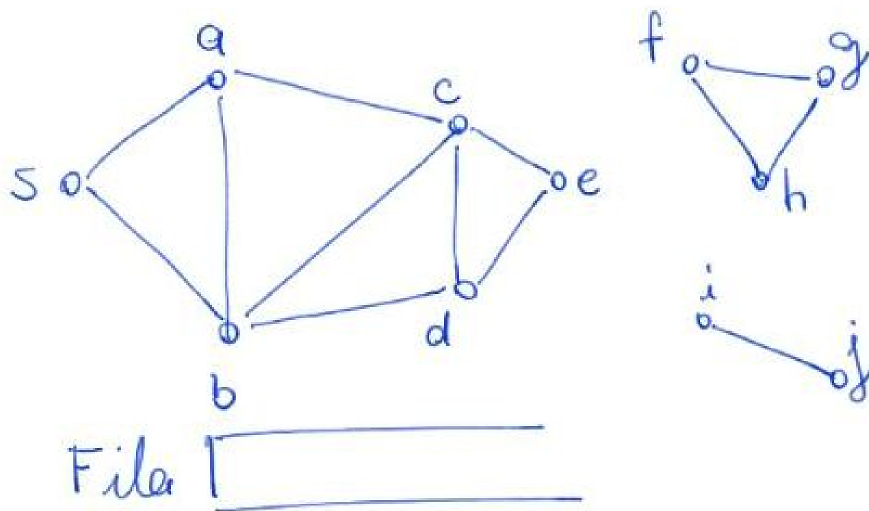
- Por exemplo, se usássemos uma pilha, primeiro encontraríamos
    - o caminho que vai até 5 passando por 1, que tem comprimento 3.

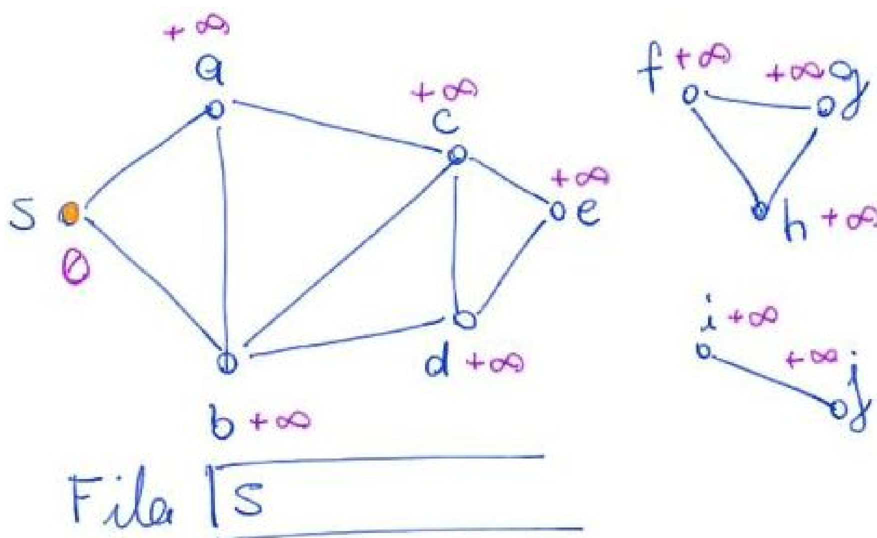


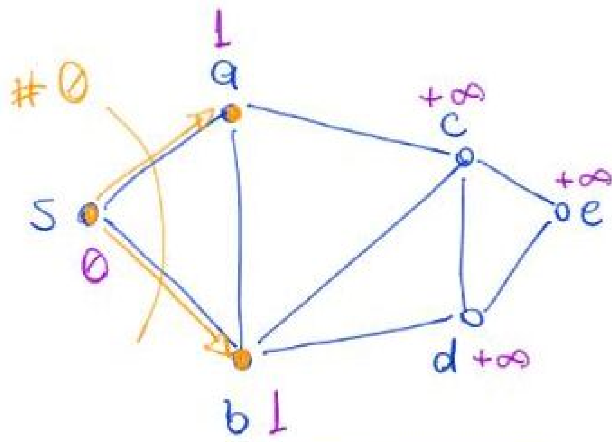
- Depois de alcançar todas os vértices,
    - ou quando a fila ficar vazia, podemos parar.

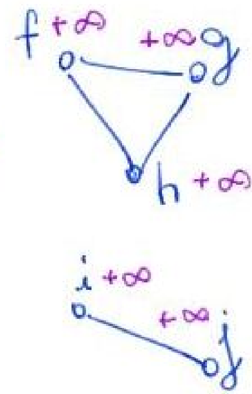
Exemplo 2: camadas em laranja e distâncias em roxo.

- Observem o momento em que uma camada é concluída.

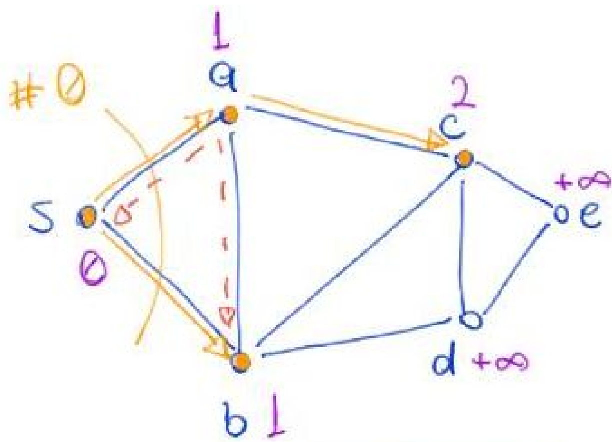


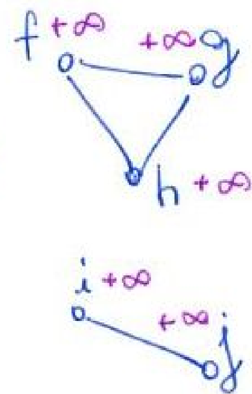




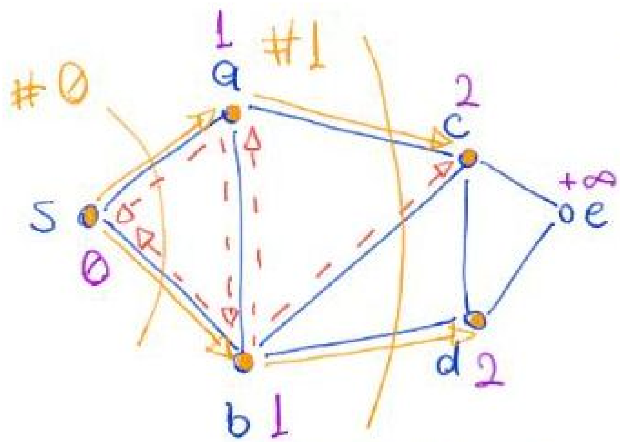


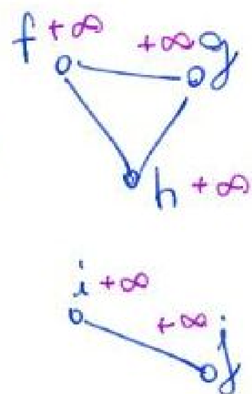
Fila ~~s~~ a b



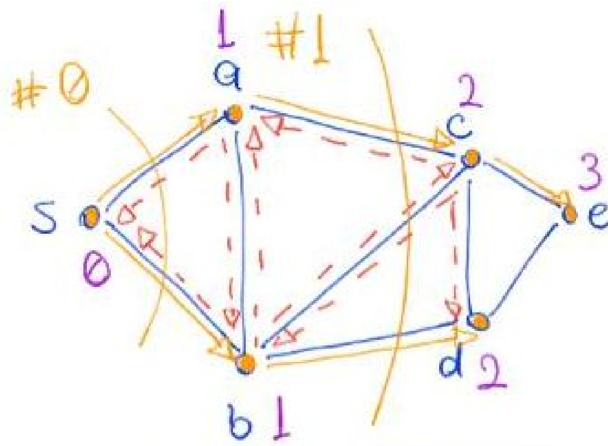


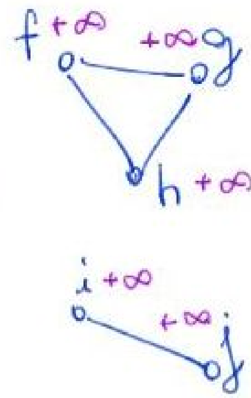
Fila ~~s~~ ~~a~~ b c



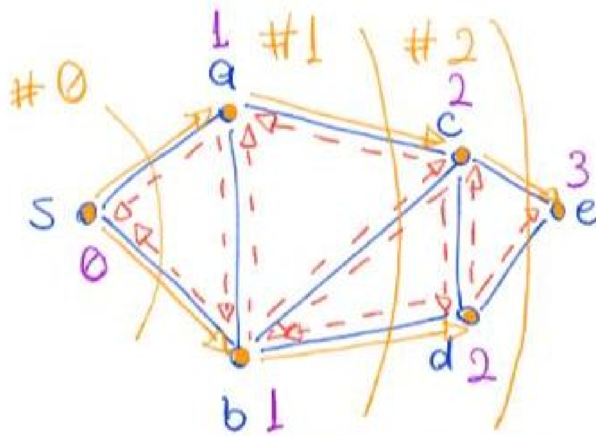


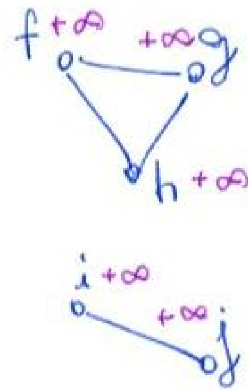
Fila ~~s~~ ~~a~~ ~~b~~ c d



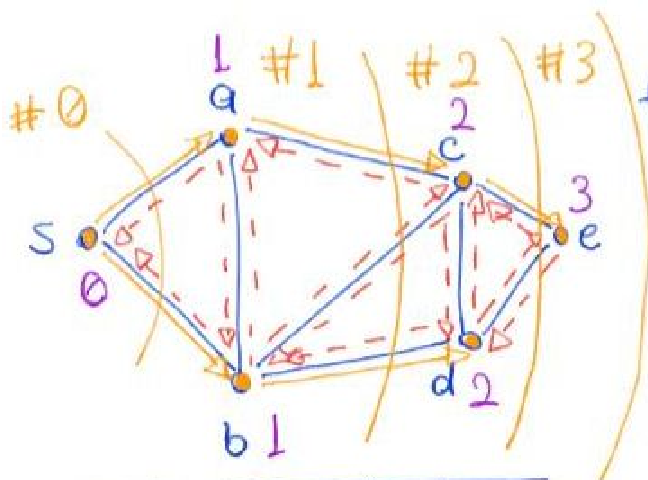


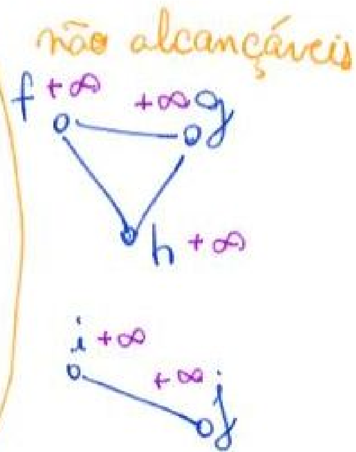
Fila ~~s~~ ~~a~~ ~~b~~ ~~c~~ d e





Fila ~~s~~ ~~a~~ ~~b~~ ~~c~~ ~~d~~ e





Fila ~~s~~ ~~a~~ ~~b~~ ~~c~~ ~~d~~ ~~e~~

Pseudocódigo:

```
distancias(grafo G=(V,E), vértice s) {
	para v \in V
		marque v como não encontrado
		dist[v] = +\inf
	marque s como encontrado
	dist[s] = 0
	seja Q uma fila inicializada com o vértice s
	enquanto Q != \empty
		remova um vértice 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
				dist[w] = dist[v] + 1
```

Vamos mostrar que um vértice qualquer v

- tem dist[v] = k se, e somente se, ele está na camada k,
  - ou seja, o caminho mais curto de s até v tem comprimento k.

A prova segue por indução no número de camadas, ou seja,

- queremos mostrar que para todo vértice v da camada k temos dist[v] = k.

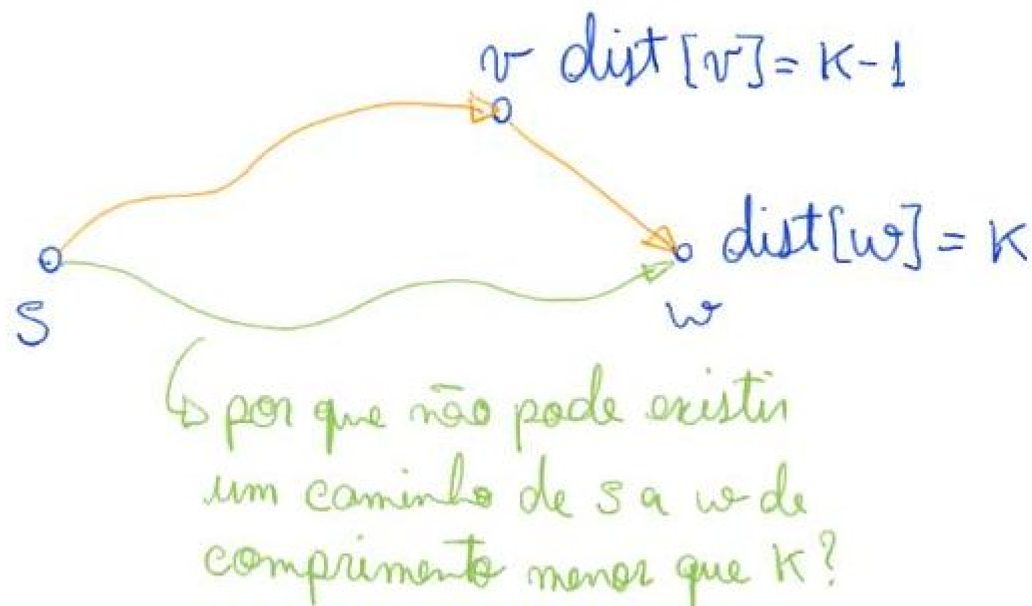
Caso base: Temos apenas o vértice s na camada 0 e dist[s] = 0.

H.I.: Para todo vértice v de uma camada k' < k temos dist[v] = k',

- i.e., quando estamos preenchendo a camada k,
  - as camadas anteriores já foram completa e corretamente preenchidas.
- Além disso, todo vértice da camada i
  - é visitado antes dos vértices da camada i + 1, para todo i.

Passo:

- Considere a iteração em que o algoritmo encontra pela primeira vez
  - um vértice w e atribui dist[w] = k para ele.
- Certamente o último vértice que o algoritmo removeu da fila,
  - i.e., o vértice sendo visitado nesta iteração,
    - é um vértice v com dist[v] = k - 1,
      - já que dist[w] = dist[v] + 1.
- Pela H.I., v está na camada k - 1.
  - Portanto o caminho mais curto até v tem comprimento k - 1
    - e existe um caminho de comprimento k até w, por construção.



- Resta mostrar que não existe um caminho
  - de comprimento menor que k até w.
- Note que, se fosse esse o caso,
  - algum vértice u de uma camada k’’ < k - 1 teria que ser vizinho de w.
- Mas, pela H.I., teríamos dist[u] = k’’ < k - 1,
  - o que significa que u já foi visitado,
    - uma vez que estamos visitando v, que é da camada k - 1.
- Como w ainda não havia sido encontrado,
  - sabemos que não pode haver este vértice u.
- Portanto, o algoritmo calcula corretamente a distância até w.

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 a fila
    /* inicializa todos como não encontrados, exceto pela origem */
    for (v = 0; v < G->n; v++)
        dist[v] = -1;
    dist[origem] = 0;
    insereFila(fila, origem); // colocando origem na fila
    /* enquanto a fila dos ativos (encontrados mas não visitados)
    não estiver vazia */
```

```
    while (!filaVazia(fila)) {
        v = removeFila(fila); // remova o mais antigo da 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;
}
```

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

Quiz: 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).
- Compare a eficiência de um algoritmo de cálculo de distâncias
    - que usa matriz de adjacência com um que usa listas de adjacência.

Funções para ler grafos

- Compare a eficiência das seguintes funções de leitura.

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);
    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;
}
```