AED2 - Aula 22
Busca em largura, cálculo de distâncias

Relembrando a busca genérica, usando um versão alternativa:

- marque todos os vértices como não encontrados.
- marque s como encontrado.
- coloque s no conjunto de vértices ativos,
    - i.e., encontrados mas não visitados.
- 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.

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 largura,
    - ou BFS (Breadth-First Search).
- A outra é a busca em profundidade,
    - ou DFS (Depth-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.

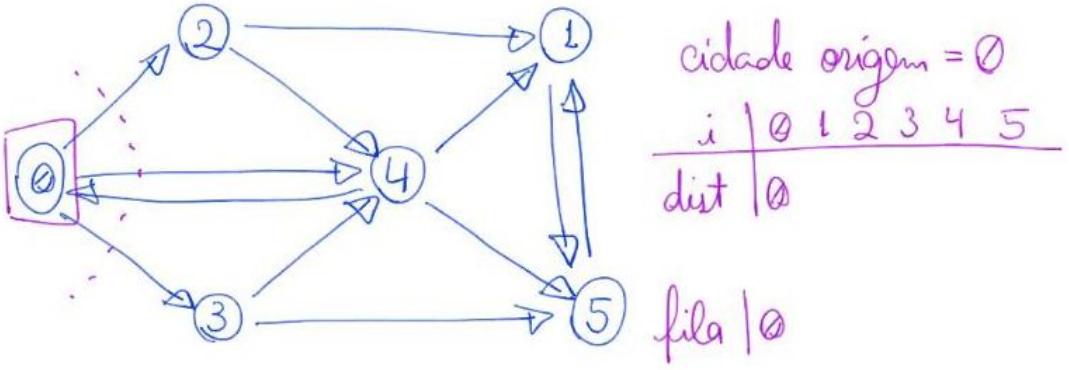
Eficiência:

- 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, o número de vértices e arestas
        - da componente 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, o algoritmo executa O(m_s) iterações do laço mais interno.

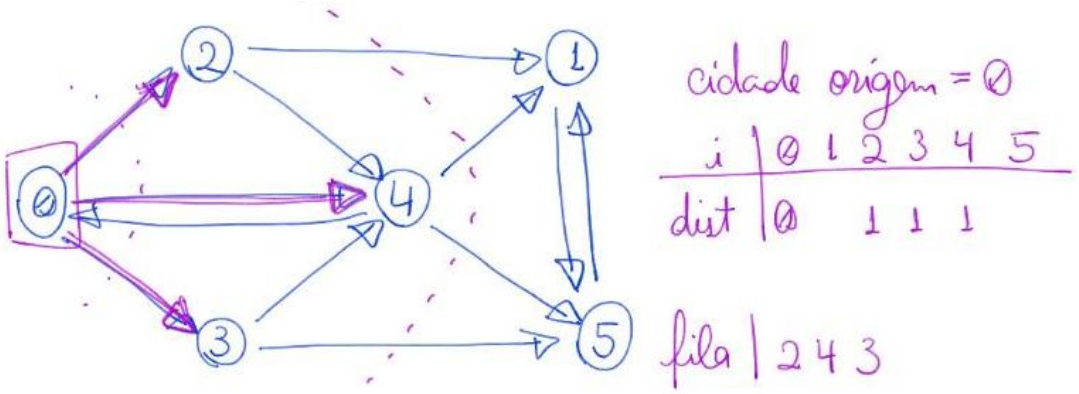
Cálculo de distâncias

Exemplo 1:

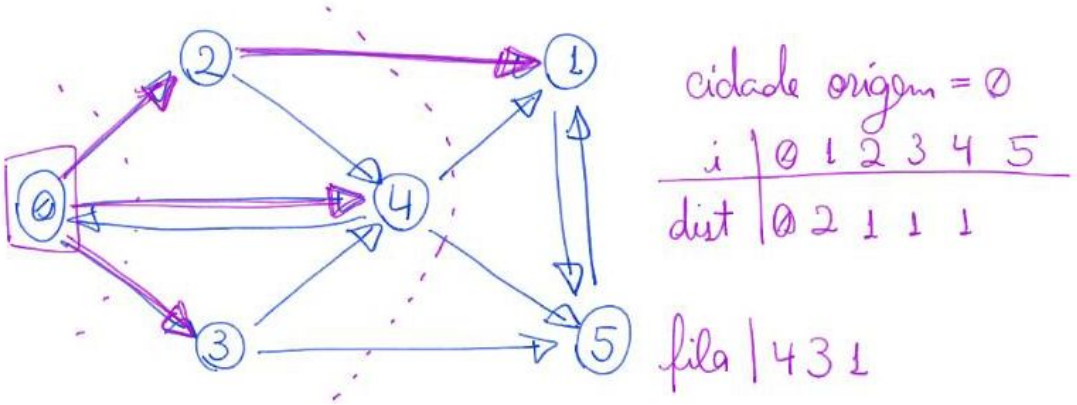
- No início apenas a cidade origem 0 é alcançável.

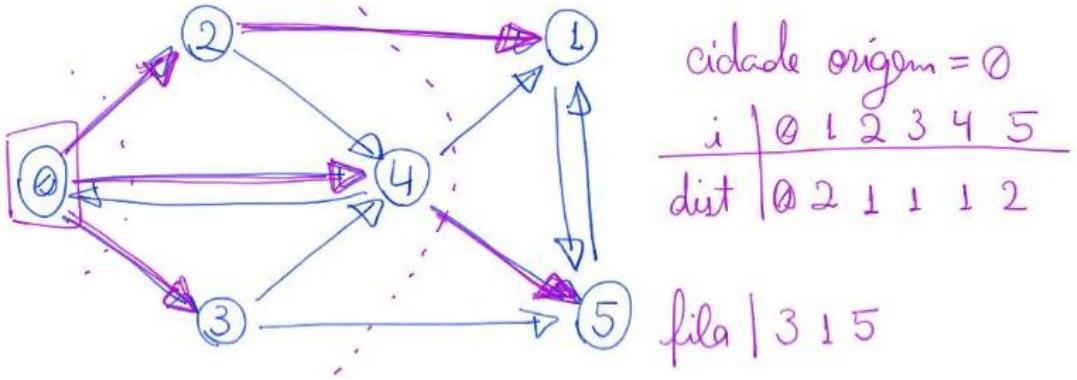


- Em cada iteração podemos encontrar novas cidades
    - e atualizar suas distâncias.



- Observe a importância de armazenar as cidades descobertas numa fila
    - para preservar a ordem de descoberta
        - e assim calcular corretamente as distâncias.

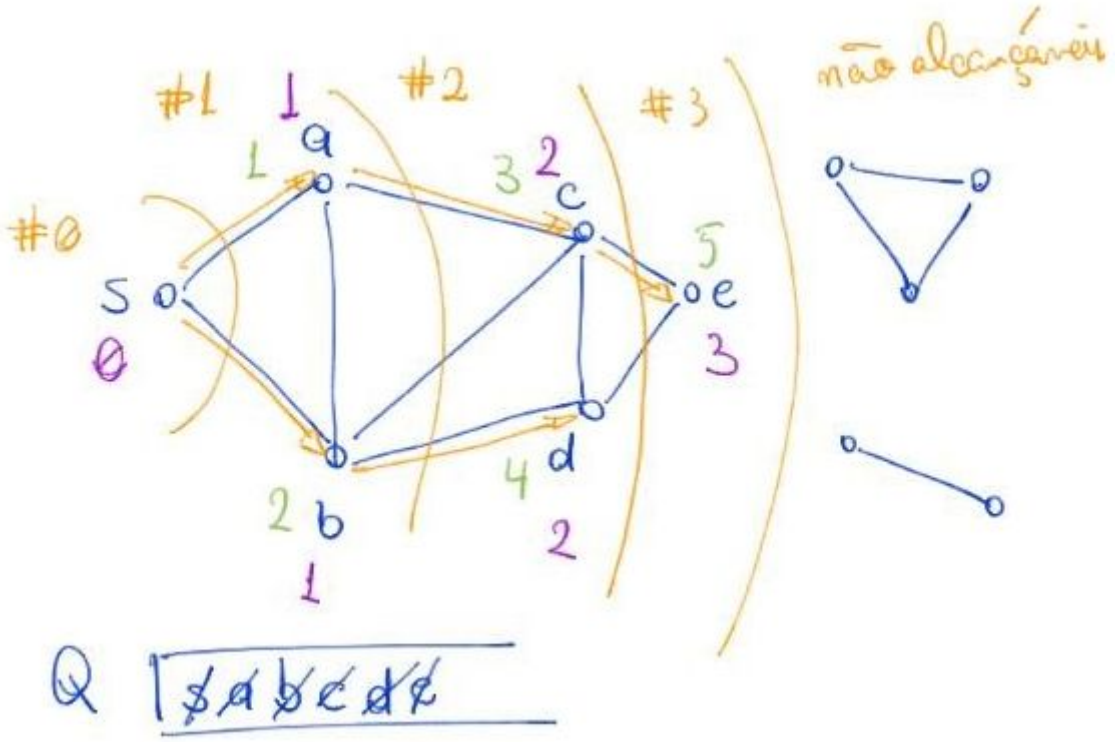




- Depois de alcançar todas as cidades podemos parar.

Exemplo 2:

- Legenda:
    - camadas em laranja,
    - ordem de descoberta dos vértices em verde,
    - distâncias em roxo.



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'.

Passo:

- considere a iteração em que o algoritmo encontra um vértice w
    - e atribui dist[w] = k para ele.
- Certamente o último vértice que o algoritmo removeu da fila
    - foi 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 existe um caminho de comprimento k até w.
- Resta mostrar que não existe um caminho
    - de comprimento menor que k até w.
- Note que, se fosse esse o caso, pela H.I. teríamos
    - dist[w] = k' para algum k' < k.
- Como w ainda não havia sido encontrado,
    - sabemos que esse não é o caso.
- Portanto, o algoritmo calcula corretamente a distância até w.

Código cálculo de distâncias com grafo implementado por matriz de adjacência

```
int *distancias(Graph G, int s)
{
   int i, y;
   int *dist;
   Fila *q;

   dist = malloc(G->n * sizeof(int));
   /* inicializa a fila */
   q = criaFila(G->n);
   /* inicializa todos como não encontrados, exceto pelo x */
   for (i = 0; i < G->n; i++)
       dist[i] = -1;
   dist[s] = 0;
   /* colocando x na fila */
   insereFila(q, s);
   /* enquanto a fila dos ativos (encontrados
   mas não visitados) não estiver vazia */
   while (!filaVazia(q))
   {
       /* remova o mais antigo da fila */
       y = removeFila(q);
       /* para cada vizinho deste que ainda não foi encontrado */
       for (i = 0; i < G->n; i++)
           if (G->A[y][i] == 1 && dist[i] == -1)
```

```
            {
                /* calcule a distancia do vizinho
                e o coloque na fila */
                dist[i] = dist[y] + 1;
                insereFila(q, i);
            }
    }
    q = liberaFila(q);
    return dist;
}
```

- Qual a eficiência deste algoritmo?

Código cálculo de distâncias com grafo implementado por lista de adjacência

```
int *distancias(Graph G, int s)
{
    int i, y;
    int *dist;
    Fila *q;
    link p;

    dist = malloc(G->n * sizeof(int));
    /* inicializa a fila */
    q = criaFila(G->n);
    /* inicializa todos como não encontrados, exceto pelo x */
    for (i = 0; i < G->n; i++)
        dist[i] = -1;
    dist[s] = 0;
    /* colocando x na fila */
    insereFila(q, s);
    /* enquanto a fila dos ativos
    (encontrados mas não visitados)
    não estiver vazia */
    while (!filaVazia(q))
    {
        /* remova o mais antigo da fila */
        y = removeFila(q);
        /* para cada vizinho deste que ainda não foi encontrado */
        p = G->A[y];
        while (p != NULL)
        {
            i = p->index;
            if (dist[i] == -1)
            {
                /* calcule a distancia do vizinho
                e o coloque na fila */
                dist[i] = dist[y] + 1;
                insereFila(q, i);
            }
```

```
            p = p->next;
        }
    }
    q = liberaFila(q);
    return dist;
}
```

- Qual a eficiência deste algoritmo?

Funções para ler grafos

- Compare a eficiência das seguintes funções de leitura.

Função auxiliar para ler de arquivo grafo representado por matriz binária

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

Função auxiliar para ler de arquivo grafo gerado por graphPrint

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

Função auxiliar para ler de arquivo grafo gerado por graphShow

```
Graph readGraphShow(FILE *entrada)
{
   int n, m, v, w, tam;
   Graph G;
   char *str, *aux;
   fscanf(entrada, "%d %d\n", &n, &m);
   G = graphInit(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);
           graphInsertArcNotSafe(G, v, w);
           aux = strtok(NULL, " \n");
       }
   }
   free(str);
   return G;
}
```