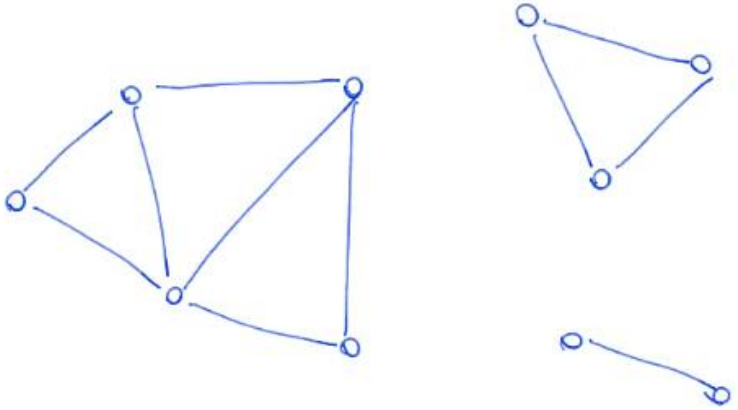
PAA - Aula 09
Grafos, Busca em Grafos e Busca em Largura

Grafos são uma estrutura matemática muito estudada

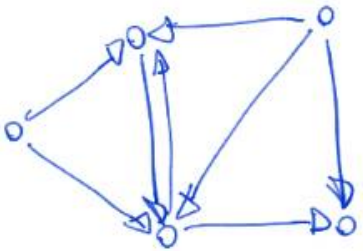
- e um Tipo Abstrato de Dado (TAD) usado para
    - representar relações entre elementos de um conjunto.
- Como todo TAD, precisa ser implementado por alguma estrutura de dados.

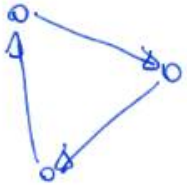
Grafos são formados por dois componentes:

- Um conjunto de vértices (ou nós) V, e um conjunto de pares de vértices E.
- Se estes pares são não ordenados
    - os chamamos de arestas e o grafo é dito não orientado.



- Se os pares são ordenados os chamamos arcos e o grafo é dito orientado (ou dirigido).





Em geral, grafos são representados compactamente como G = (V, E), e usamos

- n = |V| para indicar o número de vértices,
- m = |E| para indicar o número de arestas.

Grafos são relevantes tanto na matemática quanto na computação, pois

- conseguem modelar uma grande variedade de cenários, como:
    - Redes físicas (elétrica, comunicações, transportes),
    - redes conceituais (Web, sociais, lógicas, biológicas),
    - estruturas como listas encadeadas e árvores,
    - relações de dependência ou interação (grafo de filmes e atores),
    - mapas, etc.
- Quiz1: quem são os vértices e as arestas (ou arcos) em cada cenário?
    - Quais cenários são não-orientados e quais são orientados?

De modo mais geral, grafos modelam relações entre pares de um mesmo conjunto,

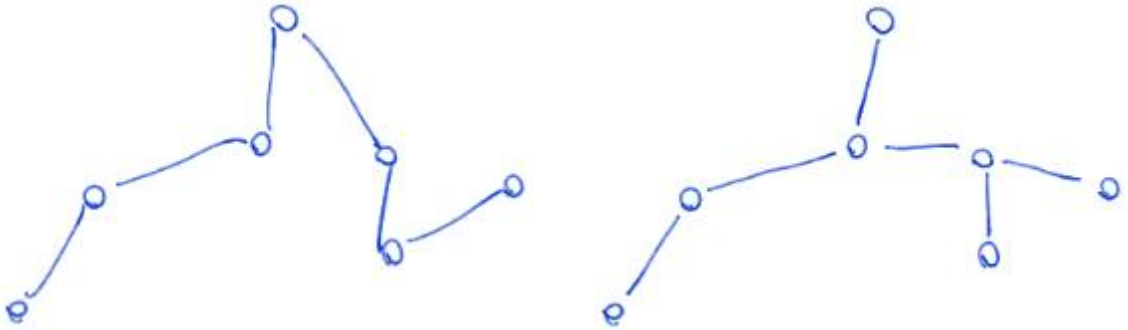
- ou relações entre pares de conjuntos relacionados,
    - o que abre uma imensa gama de possibilidades.

Grafos podem ser densos ou esparsos,

- o que diz respeito ao número de arestas que estes possuem.

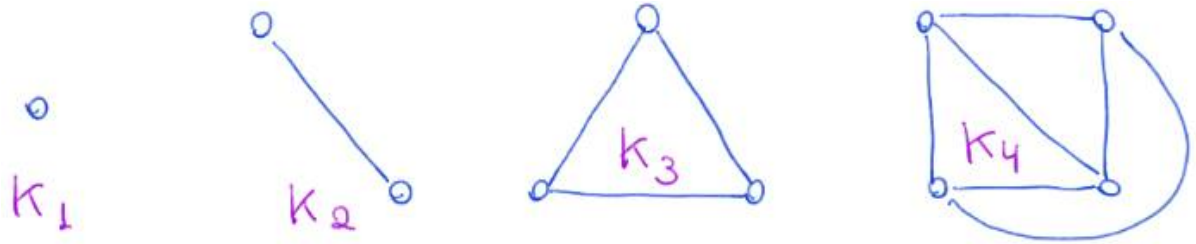
Um grafo não orientado, conexo e sem arestas múltiplas possui:

- No mínimo n - 1 arestas, caso em que o grafo é uma árvore

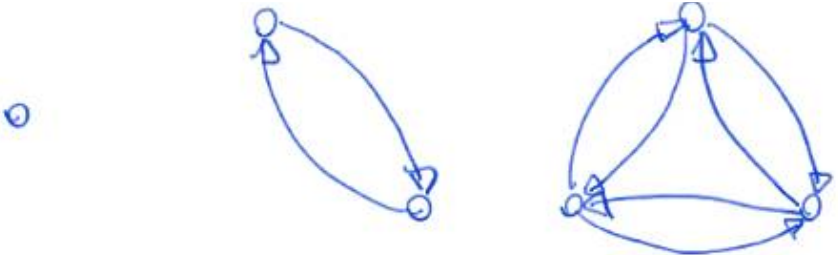


    - Quiz2: Por que? Dica: comece sem arestas e vá adicionando até ter um grafo conexo.

- No máximo (n escolhe 2) = n (n - 1) / 2 arestas, caso de um grafo completo.



    - Um grafo orientado completo tem n (n - 1) arcos,
        - i.e., o dobro do não orientado, pois entre cada par de vértices podem ter dois arcos em sentidos opostos.

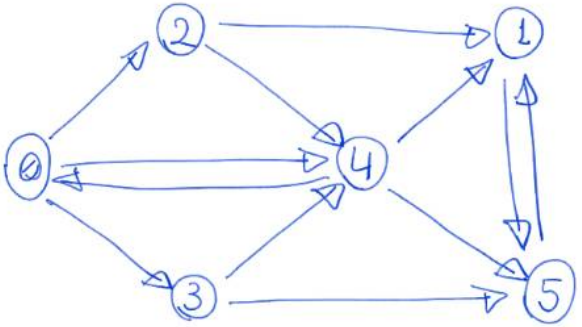


Assim, o número de arestas de um grafo varia entre O(n) até O(n^2).

- Dizemos que um grafo é esparso quando seu número de aresta
    - está próximo a n ou até n log n.
- Dizemos que ele é denso quando o número de arestas
    - está próximo de n^2 ou pelo menos superior n^3/2 = n * n^1/2.
- Embora, onde passa a linha exatamente seja arbitrário.

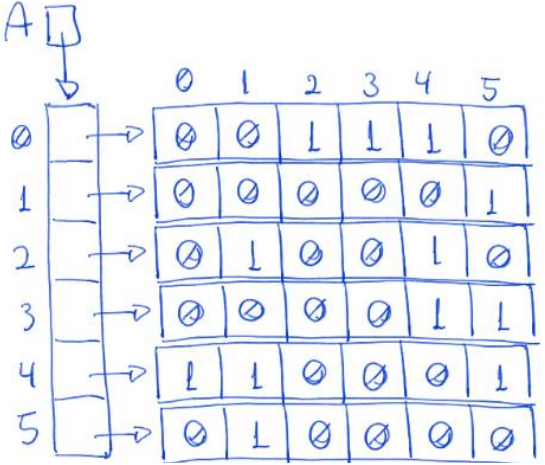
Existem duas implementações principais para grafos,

- i.e., duas estruturas de dados usadas para representá-los.
- Em ambas, os vértices são rotulados por inteiros não negativos.



Matriz de adjacência é uma implementação que utiliza uma matriz A

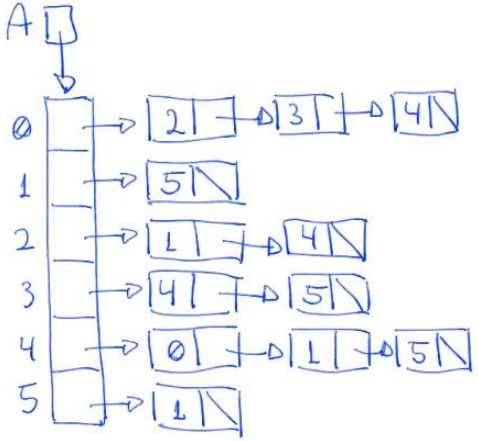
- de 0s e 1s com n linhas e n colunas sendo que o valor da célula A[i][j]
    - indica se existe uma aresta/arco entre os vértices i e j.



- Assim, a linha i da matriz A representa o leque de saída do vértice i
    - e a coluna j de A representa o leque de entrada do vértice j.
- A diagonal da matriz é preenchida por 0s, pois não temos auto-laços.
- Se o grafo não for orientado, a matriz é simétrica, i.e., A[i][j] = A[j][i].

Listas de adjacência é uma implementação que utiliza

- um vetor A de apontadores de vértices de tamanho n
    - e para cada vértice i temos uma lista ligada iniciada em A[i],
        - com os destinos das arestas que têm origem em i.



- Se o grafo não for orientado, dada uma aresta {i, j},
    - temos que j será inserido na lista de i
        - e i será inserido na lista de j.

Busca em grafos

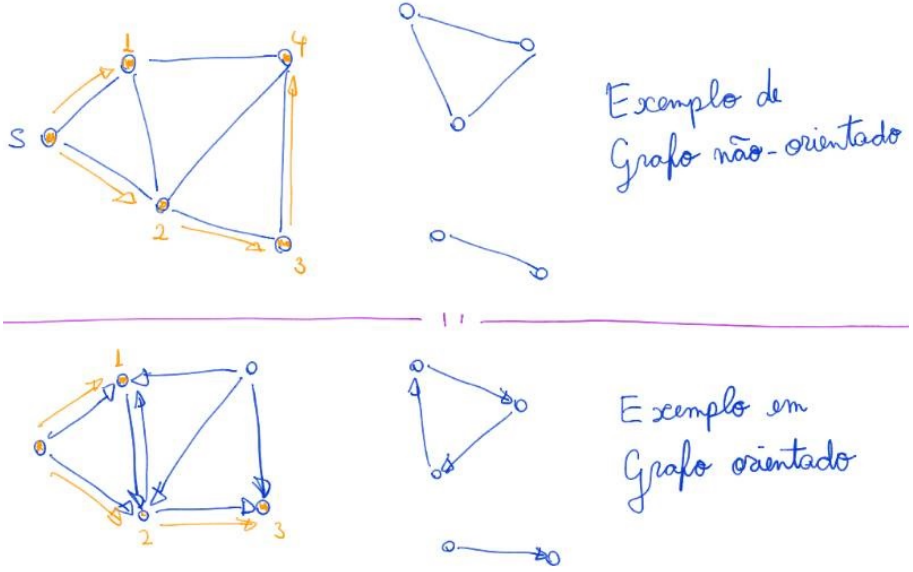
Busca em grafos é, possivelmente, a operação mais

- básica, importante e genérica para se fazer em um grafo.

Isso porque ela é fundamental para obter informações sobre conectividade,

  - i.e., determinar como e com quem se conecta cada vértice.
- Além disso, ela pode ser especializada em vários tipos de busca.

Exemplos de busca a partir de s:



A busca em grafos genérica encontra todos os vértices

- que podem ser alcançados a partir de um vértice inicial.
- A ordem em que se visita os vértices é arbitrária, e a única restrição
    - é ir sempre de um vértice encontrado para um não encontrado.
- Note que, em um grafo orientado, a busca respeita a orientação dos arcos.

Pseudocódigo:

```
buscaGenerica(grafo G=(V,E), vértice s):
    para v \in V
        marque v como não encontrado
    marque s como encontrado
    enquanto existir uma aresta (u, v) com u encontrado e v não encontrado
        marque v como encontrado
```

Corretude: Ao fim do algoritmo, um vértice v foi encontrado

- se, e somente se, existe um caminho em G de s até v.

Demonstração: (→) Vamos provar a ida por indução no número de iterações.

O caso base vale porque no início o único vértice encontrado é o próprio s

- e é conhecido um caminho trivial de s para s.

Nossa hipótese de indução é que a afirmação é verdadeira

- até a iteração anterior ao algoritmo encontrar o vértice v.
- Mais precisamente, supomos que é conhecido um caminho de s
    - até qualquer vértice encontrado antes do início da iteração
        - em que o algoritmo encontra v,
    - e que estes caminhos só usam vértices encontrados.

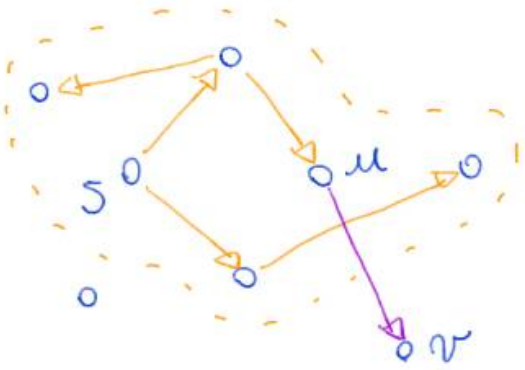
Desenvolvemos o passo assim: Na iteração em que o algoritmo encontra v,

- ele o faz através de uma aresta (u, v), sendo que u já estava encontrado.

- Pela hipótese de indução, sabemos que existe um caminho de s até u.

- Concatenando o caminho de s até u com a aresta (u, v),
    - temos um caminho de s até v.

- Note que o caminho de s até u não passa por v
    - porque ele só usa vértices encontrados previamente.

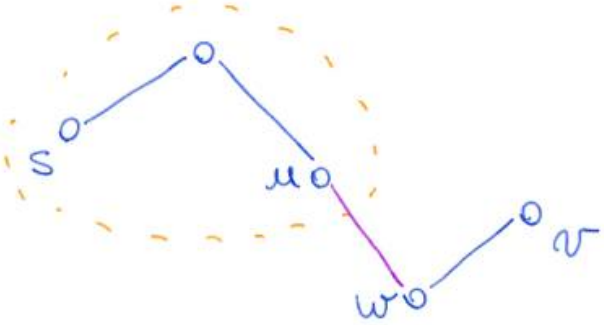


(←) Para provar a volta supomos, por contradição,

- que existe um caminho de s até v, mas v não foi encontrado.
    - Lembramos que, neste tipo de prova queremos chegar a um absurdo.

Começamos percorrendo o caminho de s até v,

- mas paramos ao encontrar a primeira aresta
    - que leva de um vértice encontrado para um vértice não encontrado.
- Note que, tal aresta deve existir, pois
    - o caminho começa com um vértice encontrado (s)
        - e termina com um não encontrado (v).



- Digamos que esta aresta é (u, w),
    - sendo que u pode ser o próprio s e w pode ser o próprio v.

De posse da aresta (u, w) consideramos o comportamento do algoritmo

- e notamos que ele não pára enquanto existir uma aresta do tipo de (u, w),
    - ou seja, que tem origem encontrada e destino não encontrado.
- Portanto, temos um absurdo e concluímos a demonstração. Q.E.D.

Busca em largura

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

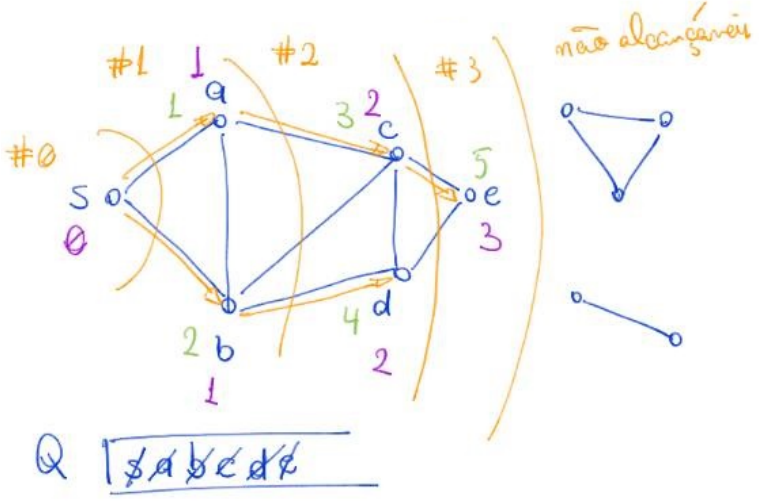
Hoje vamos nos aprofundar na BFS,
- que explora o grafo em camadas a partir de um vértice inicial s.
- Por isso, ela é particularmente útil
    - para calcular a distância não ponderada entre vértices.
- Esse comportamento da BFS está intimamente relacionado
    - com a estrutura de dados fila (queue ou FIFO).

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.

Pseudocódigo:

```
buscaLargura(grafo G=(V,E), vértice s):
        marque todo v \in 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
```

Exemplo:



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:

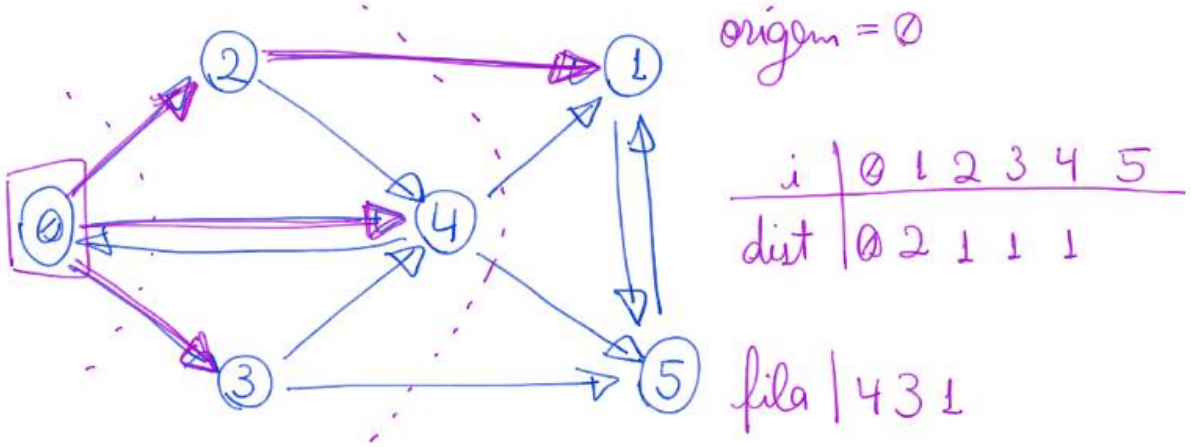
- O algoritmo leva tempo O(n) para marcar os vértices como não encontrados.
- O restante do algoritmo leva tempo O(n_s + m_s),
    - sendo n_s e m_s, respectivamente, os números de vértices e arestas
        - da componente conexa que contém o vértice s.
- Isso porque, em cada iteração do laço principal,
    - 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

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

- ou, de modo equivalente, o número de vértices em P - 1.
- A distância entre dois vértices é o comprimento
    - de um menor caminho entre eles.

Exemplo:



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 nosso algoritmo calcula corretamente

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

Caso base: No início da primeira iteração apenas s foi encontrado e dist[s] = 0.

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

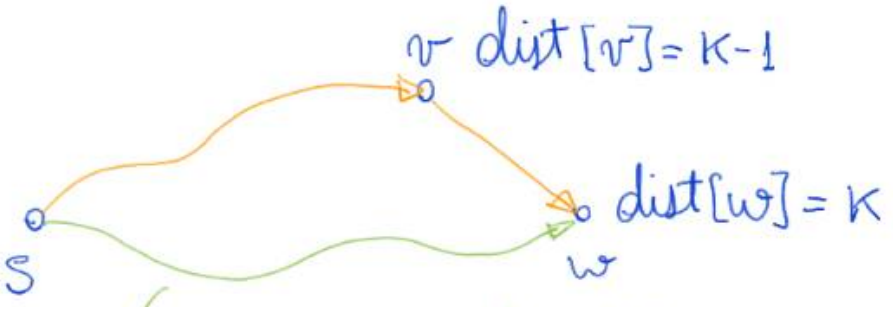
- temos que dist[v] é a distância de s até v.

- Além disso, os vértices são encontrados em ordem crescente de 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 dist[v] = k.
- Ao analisar os vizinhos de v encontramos, pela primeira vez,
  - o vértice w e atribuímos dist[w] = k + 1.



Nosso objetivo é mostrar que k + 1 é a distância de s até w,

- e que todo vértice com distância k já foi encontrado.

- 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?

Se existir um caminho de comprimento menor que k+1 até w,
    - então existe um vértice u com distância menor que k.
- Pela H.I., u foi encontrado antes de v e, pelo comportamento da fila,
    - u foi visitado antes de v.
- Assim, w teria sido encontrado antes da iteração atual (contradição).

Agora resta mostrar que todo vértice com distância até k já foi encontrado.
- Pela H.I., sabemos que todo vértice
    - com distância menor que k foi encontrado.
- Mas, pelo comportamento da fila, sabemos que esses vértices
    - foram visitados antes de v.
- Assim, todo vértice com distância menor que k+1 foi encontrado antes de w.

Portanto, confirmamos a hipótese de que os vértices
- são encontrados em ordem crescente de distância,
    - e a prova está concluída.