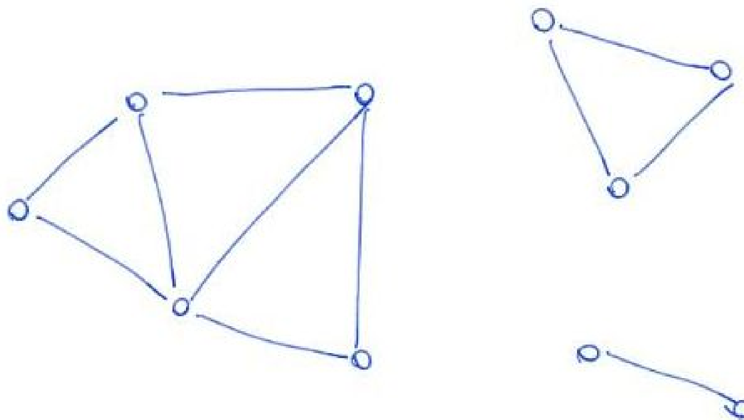
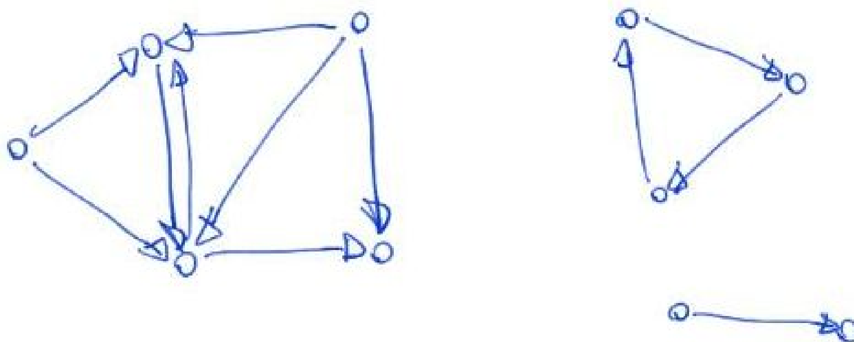
AED2 - Aula 21
Grafos, implementação,
construção aleatória e busca

Grafos são uma estrutura matemática muito estudada
- e um tipo abstrato de dados usado para
  - representar relações entre elementos de um conjunto.
- Como todo tipo abstrato de dados, precisa ser implementado
  - por alguma estrutura de dados.
- Vamos estudar algumas dessas estruturas.

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 de 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 e a própria Web),
    - redes conceituais (sociais, lógicas, biológicas),
    - estruturas como listas encadeadas e árvores,
    - relações de dependência ou interação (grafo de filmes e atores),
    - mapas.
- 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.

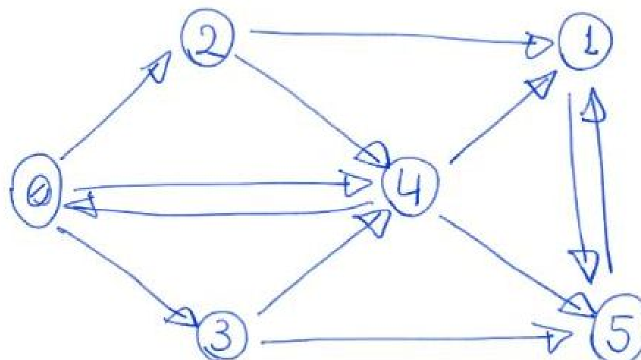
Um grafo não orientado, conexo e sem arestas múltiplas possui

- no mínimo $n - 1$ arestas, caso em que o grafo é uma árvore,
- no máximo $n (n - 1) / 2$ arestas, caso de um grafo completo.
    - Um grafo orientado completo tem $n (n - 1)$ arcos.

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.

Considere o seguinte grafo orientado



Existem duas implementações principais para grafos,

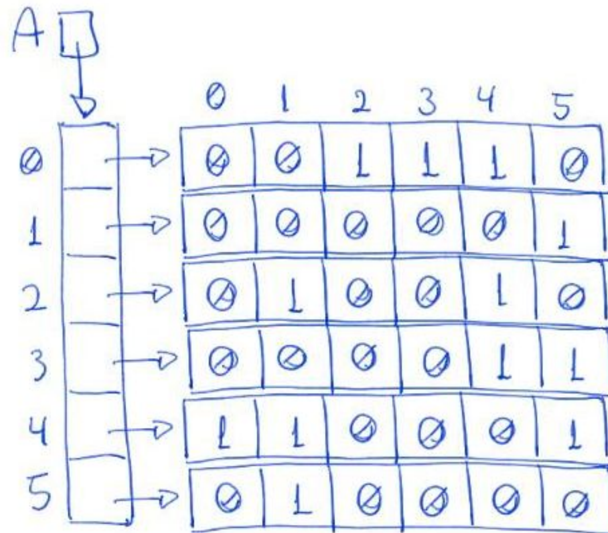
- i.e., duas estruturas de dados associadas a eles.

- Em ambas, os vértices são representados por inteiros não negativos.

Matriz de adjacência

Esta implementação utiliza uma matriz A de 0s e 1s com n linhas e n colunas

- e 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 nosso grafo não tem laços.
- Se o grafo não for orientado, a matriz é simétrica,
  - i.e., A[i][j] = A[j][i].

Interface para grafo implementado como matriz de adjacência:

```
typedef struct graph *Graph;
struct graph
{
   int **A;
   int n;
   int m;
};

Graph graphInit(int n);
void graphInsertArc(Graph G, int v, int w);
void graphInsertArcNotSafe(Graph G, int v, int w);
void graphRemoveArc(Graph G, int v, int w);
void graphShow(Graph G);
void graphPrint(Graph G);
Graph graphFree(Graph G);
```

Código de operações básicas para grafo implementado como matriz de adjacência:

```
#include <stdio.h>
#include <stdlib.h>

#include "grafosMatrizAdj.h"

/* REPRESENTAÇÃO POR MATRIZ DE ADJACÊNCIA: A função graphInit() constrói um grafo com
vértices 0 1 .. V-1 e nenhum arco. */
Graph graphInit(int n)
{
   int i, j;
   Graph G = malloc(sizeof *G);
   G->n = n;
   G->m = 0;
   G->A = malloc(n * sizeof(int *));
   for (i = 0; i < n; i++)
       G->A[i] = malloc(n * sizeof(int));
   for (i = 0; i < n; i++)
       for (j = 0; j < n; j++)
           G->A[i][j] = 0;
   return G;
}

/* REPRESENTAÇÃO POR MATRIZ DE ADJACÊNCIA: A função graphInsertArc() insere um arco v-w no
grafo G. A função supõe que v e w são distintos, positivos e menores que G->V. Se o grafo
já tem um arco v-w, a função não faz nada. */
void graphInsertArc(Graph G, int v, int w)
{
   if (G->A[v][w] == 0)
   {
       G->A[v][w] = 1;
       G->m++;
   }
}

void graphInsertArcNotSafe(Graph G, int v, int w)
{
   G->A[v][w] = 1;
   G->m++;
}

/* REPRESENTAÇÃO POR MATRIZ DE ADJACÊNCIA: A função graphRemoveArc() remove do grafo G o
arco v-w. A função supõe que v e w são distintos, positivos e menores que G->V. Se não
existe arco v-w, a função não faz nada. */
void graphRemoveArc(Graph G, int v, int w)
{
   if (G->A[v][w] == 1)
   {
```

```c
        G->A[v][w] = 0;
        G->m--;
    }
}

/* REPRESENTAÇÃO POR MATRIZ DE ADJACÊNCIA: A função graphShow() imprime, para cada vértice
v do grafo G, em uma linha, todos os vértices adjacentes a v. */
void graphShow(Graph G)
{
    int i, j;
    for (i = 0; i < G->n; i++)
    {
        printf("%2d:", i);

        for (j = 0; j < G->n; j++)
            if (G->A[i][j] == 1)
                printf(" %2d", j);
        printf("\n");
    }
}

void graphPrint(Graph G)
{
    int i, j;
    for (i = 0; i < G->n; i++)
    {
        for (j = 0; j < G->n; j++)
            if (G->A[i][j] == 1)
                printf("%2d ", j);
        printf("-1"); // digito para marcar fim de lista
        printf("\n");
    }
}

Graph graphFree(Graph G)
{
    int i;
    for (i = 0; i < G->n; i++)
    {
        free(G->A[i]);
        G->A[i] = NULL;
    }
    free(G->A);
    G->A = NULL;
    free(G);
    return NULL;
}
```

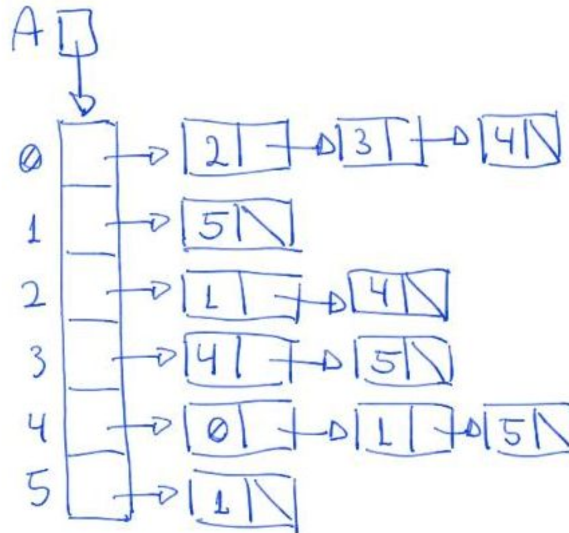
- Qual a eficiência das operações?

- Algo muda se o grafo for denso ou esparso?

Lista de adjacências

Esta implementação utiliza um vetor A de apontadores de vértices de tamanho n

- e para cada vértice i temos uma lista encadeada indicada por 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.

Interface para grafo implementado como lista de adjacência:

```
typedef struct node *link;
struct node
{
   int index;
   link next;
};

typedef struct graph *Graph;
struct graph
{
   link *A;
   int n;
   int m;
};

Graph graphInit(int n);
void graphInsertArc(Graph G, int v, int w);
void graphInsertArcNotSafe(Graph G, int v, int w);
void graphShow(Graph G);
void graphPrint(Graph G);
```

```
Graph graphFree(Graph G);
```

Código de operações básicas para grafo implementado como lista de adjacência:

```
#include <stdio.h>
#include <stdlib.h>

#include "grafosListaAdj.h"

/* REPRESENTAÇÃO POR LISTA DE ADJACÊNCIA: A função graphInit() constrói um grafo com
vértices 0 1 .. V-1 e nenhum arco. */
Graph graphInit(int n)
{
   int i;
   Graph G = malloc(sizeof *G);
   G->n = n;
   G->m = 0;
   G->A = malloc(n * sizeof(link));
   for (i = 0; i < n; i++)
       G->A[i] = NULL;
   return G;
}

/* REPRESENTAÇÃO POR LISTA DE ADJACÊNCIA: A função graphInsertArc() insere um arco v-w no
grafo G. A função supõe que v e w são distintos, positivos e menores que G->V. Se o grafo
já tem um arco v-w, a função não faz nada. */
void graphInsertArc(Graph G, int v, int w)
{
   link p;
   for (p = G->A[v]; p != NULL; p = p->next)
       if (p->index == w)
           return;
   p = malloc(sizeof(struct node));
   p->index = w;
   p->next = G->A[v];
   G->A[v] = p;
   G->m++;
}

void graphInsertArcNotSafe(Graph G, int v, int w)
{
   link p;
   p = malloc(sizeof(struct node));
   p->index = w;
   p->next = G->A[v];
   G->A[v] = p;
   G->m++;
}
```

```c
/* REPRESENTAÇÃO POR LISTA DE ADJACÊNCIA: A função graphShow() imprime, para cada vértice v
do grafo G, em uma linha, todos os vértices adjacentes a v. */
void graphShow(Graph G)
{
   int i;
   link p;
   for (i = 0; i < G->n; i++)
   {
       printf("%2d:", i);
       for (p = G->A[i]; p != NULL; p = p->next)
           printf(" %2d", p->index);
       printf("\n");
   }
}

void graphPrint(Graph G)
{
   int i;
   link p;
   for (i = 0; i < G->n; i++)
   {
       for (p = G->A[i]; p != NULL; p = p->next)
           printf("%2d ", p->index);
       printf("-1"); // digito para marcar fim de lista
       printf("\n");
   }
}

Graph graphFree(Graph G)
{
   int i;
   link p;
   for (i = 0; i < G->n; i++)
   {
       p = G->A[i];
       while (p != NULL)
       {
           G->A[i] = p;
           p = p->next;
           free(G->A[i]);
       }
       G->A[i] = NULL;
   }
   free(G->A);
   G->A = NULL;
   free(G);
   return NULL;
}
```

- Qual a eficiência das operações?
    - Algo muda se o grafo for denso ou esparso?

Comparação entre as estruturas de dados

Matriz de adjacência:

- Vantagens
    - Acessar um elemento A[i][j] qualquer leva tempo constante.
    - Economia de espaço quando a rede é densa,
        - pois é possível operar sobre uma matriz de bits.
- Desvantagens
    - Ocupa espaço proporcional a n^2, ainda que a rede seja esparsa,
        - resultando na maioria dos elementos da matriz iguais a zero.
    - Visitar todos os nós para os quais um nó i tem conexão,
        - leva tempo proporcional a n, ainda que i tenha poucos vizinhos.
    - O mesmo vale para visitar todos os nós que tem conexão para i.

Lista de adjacências:

- Vantagens
    - Economia de memória quando a rede é esparsa,
        - pois ocupa espaço proporcional a n + m,
            - sendo n o número de nós e m o número de arestas.
    - Visitar todos os nós para os quais um nó i tem conexão,
        - leva tempo proporcional ao número de vizinhos de i.
- Desvantagens
    - Verificar se um nó i tem conexão para um nó j
        - leva tempo linear no número de vizinhos do nó i.
    - Quando a rede é densa, a ordem de grandeza
        - tanto da memória quanto do tempo serão quadráticos.
    - A memória ocupada por conexão é maior que na matriz.
    - Verificar quais nós tem conexão para um nó j
        - exige percorrer todas as listas.
        - Para contornar essa limitação, podemos usar listas ortogonais.

Grafos aleatórios

Podemos construir grafos aleatórios (na verdade, pseudoaleatórios),

- o que é muito útil para utilizar em testes de algoritmos, por exemplo.

Nossa primeira função constrói grafos aleatórios com exatamente m arcos.

```
/* A função randV() devolve um vértice aleatório do grafo G. Vamos supor que G->V <=
RAND_MAX. */
```

```c
int randV(Graph G)
{
   double r;
   r = rand() / (RAND_MAX + 1.0);
   return r * G->n;
}

/* Esta função constrói um grafo aleatório com vértices 0..V-1 e exatamente A arcos. A
função supõe que A <= V*(V-1). Se A for próximo de V*(V-1), a função pode consumir muito
tempo. (Código inspirado no Programa 17.7 de Sedgewick.) */
Graph graphRand1(int n, int m)
{
   Graph G = graphInit(n);
   while (G->m < m)
   {
       int v = randV(G);
       int w = randV(G);
       if (v != w)
           graphInsertArc(G, v, w);
   }
   return G;
}
```

- Ela é particularmente útil para construir grafos esparços grandes,
  - mas tende a ficar ineficiente se usada para construir grafos densos.
    - Por que?

Nossa segunda função constrói grafos aleatórios com m arcos em média,

- sendo mais indicada para gerar grafos densos.

```c
Graph graphRand2(int n, int m)
{
   double prob = (double)m / n / (n - 1);
   Graph G = graphInit(n);
   for (int v = 0; v < n; v++)
       for (int w = 0; w < n; w++)
           if (v != w)
               if (rand() < prob * (RAND_MAX + 1.0))
                   graphInsertArc(G, v, w);
   return G;
}
```

- Qual a eficiência de tempo desta função?
  - E qual a vantagem da seguinte variante?

```c
Graph graphRand2_1(int n, int m)
{
   double prob = (double)m / n / (n - 1);
   Graph G = graphInit(n);
   for (int v = 0; v < n; v++)
       for (int w = 0; w < n; w++)
```

```
            if (v != w)
                if (rand() < prob * (RAND_MAX + 1.0))
                    graphInsertArcNotSafe(G, v, w);
    return G;
}
```

Busca em Grafos

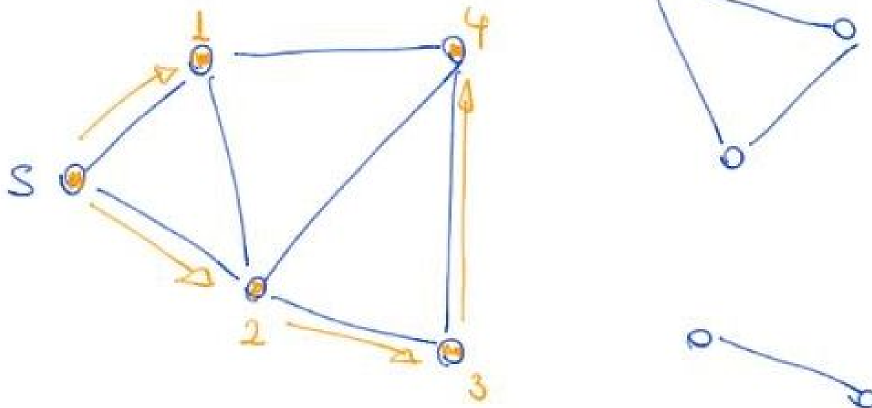
Busca em grafos é, possivelmente, a operação
- mais básica
- importante
- e genérica

para se fazer em um grafo.

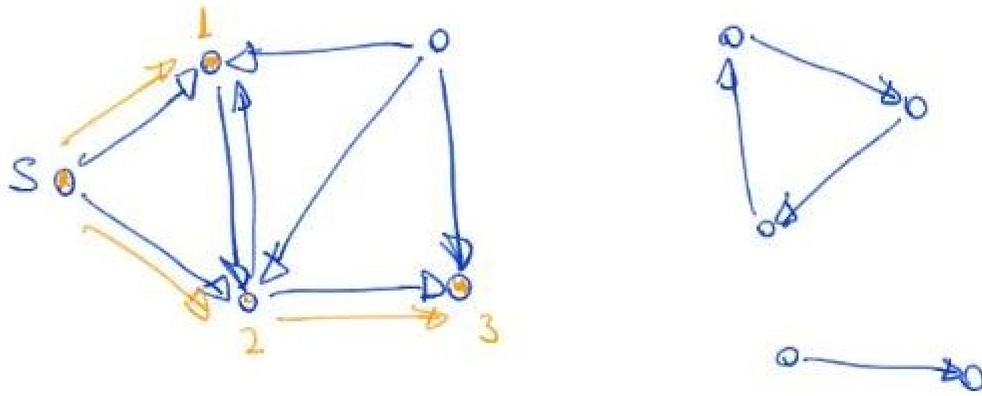
Isso porque ela é fundamental para
- obter informações sobre conectividade,
    - i.e., determinar como e com quem se conecta cada vértice.
- Além disso, ela pode ser especializada em vários tipos de busca.

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.

Exemplos de busca a partir de s
- em grafo não orientado



- e em grafo orientado



    - 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.

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 para enquanto existe 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.

Eficiência:
- Em cada iteração do algoritmo, vamos sempre
    - de um vértice encontrado para um não encontrado,
        - nunca visitando um vértice
            - ou percorrendo um arco mais que uma vez.
- Isso sugere que a eficiência do algoritmo é proporcional
    - ao número de vértices e arestas do grafo.
- Note que, se o grafo não for orientado
    - consideramos cada aresta até duas vezes.
- Vamos analisar mais detalhadamente a eficiência da busca
    - em suas subsequentes implementações específicas.