

AED2 - Aula 21

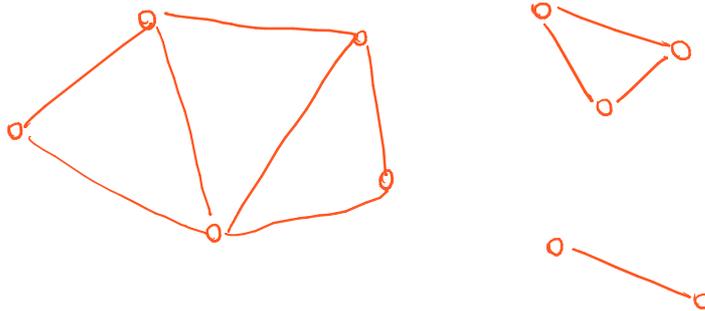
Grafos: tipos, implementação e construção aleatória

Grafos são uma estrutura matemática muito estudada

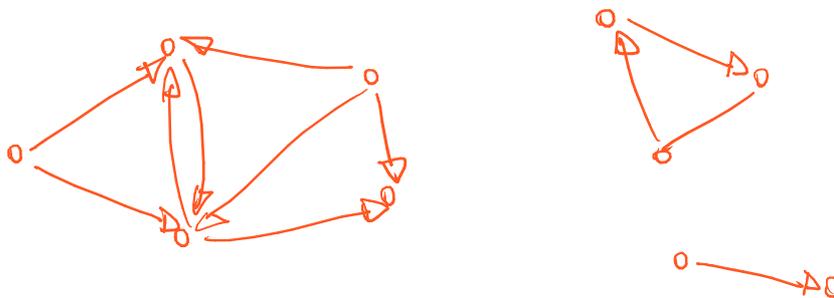
- e um Tipo Abstrato de Dado (TAD) usado para
 - representar relações entre elementos de um conjunto.
- Como todo TAD, precisa ser implementado por alguma estrutura de dados.
 - Vamos estudar algumas dessas estruturas.

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),
 - 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.

Densidade de grafos

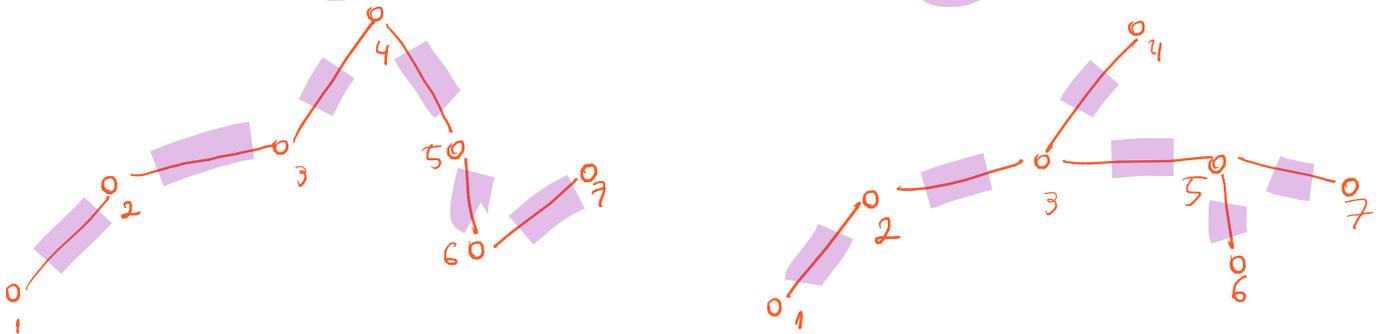
$$G = (V, E)$$

Grafos podem ser densos ou esparsos,

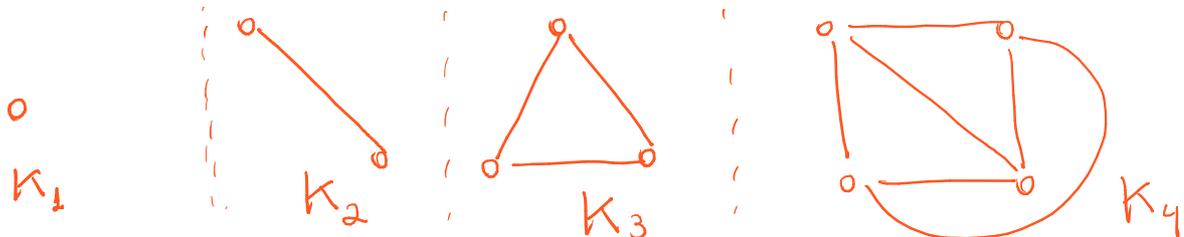
- o que diz respeito ao número de arestas que estes possuem.

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

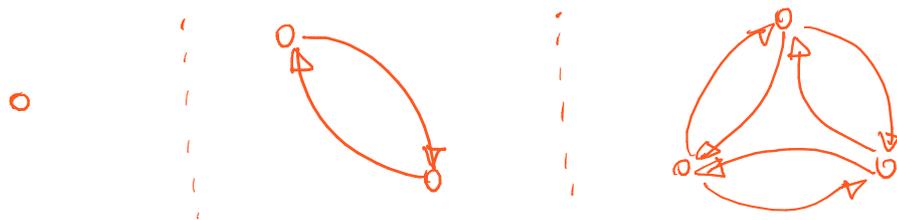
- No mínimo $n - 1$ arestas, caso em que o grafo é uma árvore



- Quiz2: Por que? Dica: comece sem arestas e vá adicionando até ter um grafo conexo.
- No máximo $(n \text{ escolhe } 2) = n(n - 1) / 2$ 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.

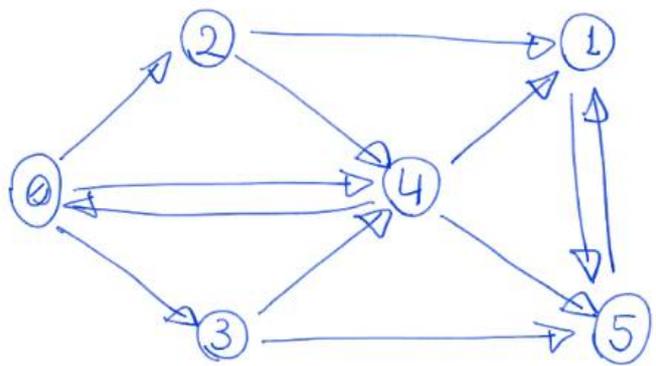


Assim, o número de arestas de um grafo varia entre $O(n)$ até $O(n^2)$.

- 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 $n^{3/2} = n * n^{1/2} = n \sqrt{n}$
- Embora, onde passa a linha exatamente seja arbitrário.

Existem duas implementações principais para grafos,

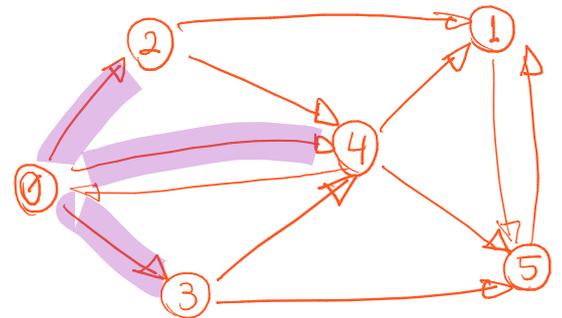
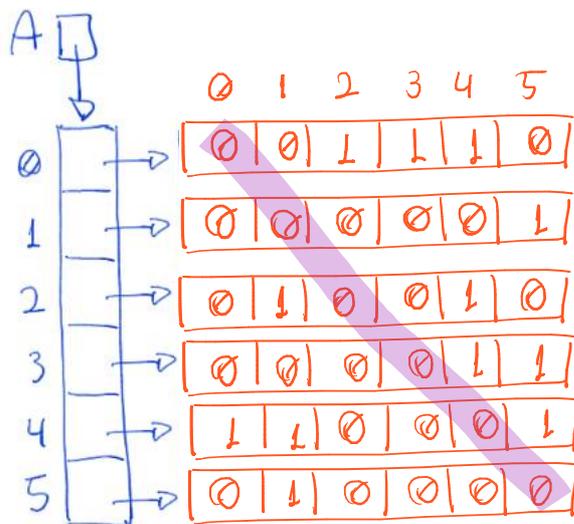
- i.e., duas estruturas de dados usadas para representá-los.
- Em ambas, os vértices são rotulados por inteiros não negativos.



Matriz de adjacência

Esta implementação 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., $A[i][j] = A[j][i]$.

Interface para grafo implementado como matriz de adjacência:

```
typedef (struct grafo *)Grafo;
```

```
struct grafo {
    int **A;
    int n; // número de vértices
    int m; // número de arestas/arcos
};
```

grafos MatrizAdj.h

```
Grafo inicializaGrafo(int n);
```

```
void insereArcoGrafo(Grafo G, int v, int w);
```

```
void insereArcoNaoSeguraGrafo(Grafo G, int v, int w);
```

```
void removeArcoGrafo(Grafo G, int v, int w);
```

```
void mostraGrafo(Grafo G);
```

```
void imprimeGrafo(Grafo G);
```

```
Grafo liberaGrafo(Grafo G);
```

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

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

// Constrói um grafo com vértices 0 1 .. n-1 e nenhum arco.

```
Grafo inicializaGrafo(int n) {
```

```
    int i, j;
```

```
    Grafo 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;
```

```
}
```

/* Insere arco v-w no grafo G, supondo que v e w são inteiros distintos entre 0 e n-1. Se grafo já tem arco v-w, não faz nada. */

```
void insereArcoGrafo(Grafo G, int v, int w) {
```

```
    if (G->A[v][w] == 0) {
```

```
        G->A[v][w] = 1;
```

```
        G->m++;
```

```
    }
```

```
}
```

// Versão da insereArcoGrafo() que não testa se v-w já está presente

```
void insereArcoNaoSeguraGrafo(Grafo G, int v, int w) {
```

```
    G->A[v][w] = 1;
```

```
    G->m++;
```

```
}
```

/* Remove arco v-w do grafo G, supondo que v e w são inteiros distintos entre 0 e n-1. Se não existe arco v-w, não faz nada. */

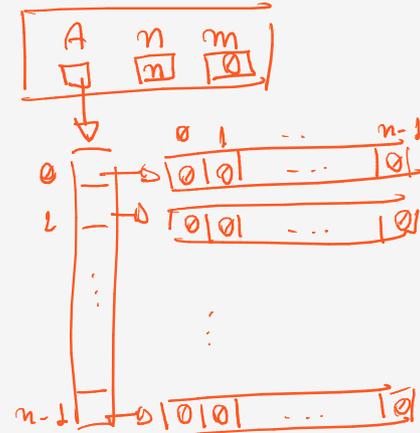
```
void removeArcoGrafo(Grafo G, int v, int w) {
```

```
    if (G->A[v][w] == 1) {
```

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

```
        G->m--;
```

$O(1)$



$O(n^2)$

```

}
}

// Imprime, para cada vértice v, os vértices adjacentes a v.

```

```

void mostraGrafo(Grafo 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");

```

```

        }

```

```

    }

```

```

// Versão da mostraGrafo() com impressão mais limpa

```

```

void imprimeGrafo(Grafo 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"); // sentinela para marcar fim de lista

```

```

            printf("\n");

```

```

        }

```

```

    }

```

```

// Libera a memória alocada para o grafo G e devolve NULL.

```

```

Grafo LiberaGrafo(Grafo 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;

```

```

}

```

$O(n^2)$

para cada vértice

— todos os possíveis vizinhos

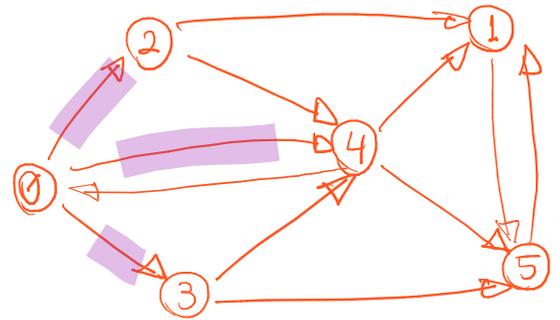
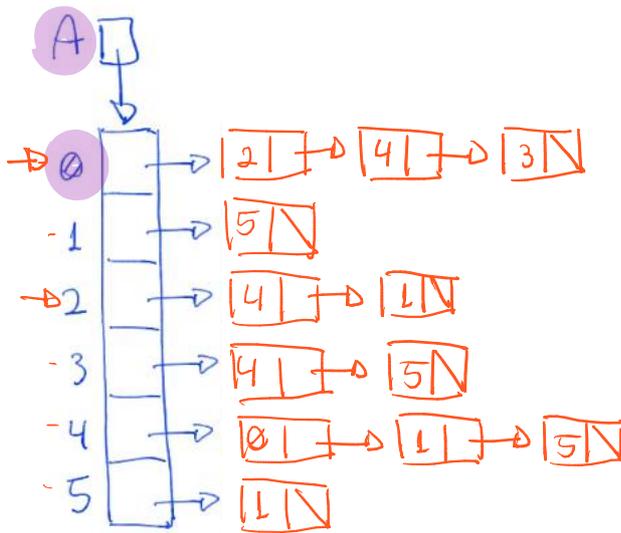
$O(n)$

- Quiz3: Qual a eficiência das operações?
 - Algo muda se o grafo for denso ou esparsos?

Listas de adjacência

Esta implementação 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.

Interface para grafo implementado como listas de adjacência:

```
typedef struct noh Noh;
struct noh {
    int rotulo;
    Noh *prox;
};

typedef (struct grafo *)Grafo;
struct grafo {
    (Noh **)A;
    -int n; // número de vértices
    -int m; // número de arestas/arcos
};

- Grafo inicializaGrafo(int n);
- void insereArcoGrafo(Grafo G, int v, int w);
- void insereArcoNaoSeguraGrafo(Grafo G, int v, int w);
- void mostraGrafo(Grafo G);
- void imprimeGrafo(Grafo G);
- void imprimeArquivoGrafo(Grafo G, FILE *saida);
- Grafo liberaGrafo(Grafo G);
```

grafos Listas Adj. h

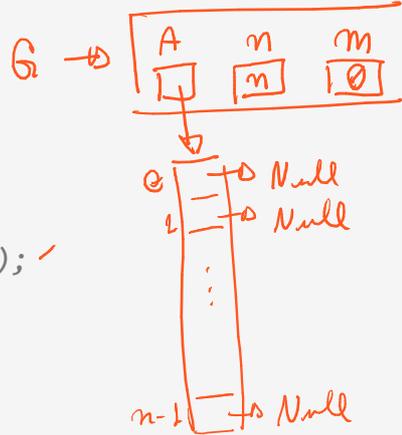
Código de operações básicas para grafo implementado como listas de adjacência:

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

// Constrói um grafo com vértices 0 1 .. n-1 e nenhum arco.

```
Grafo inicializaGrafo(int n) {
```

```
    int i;
    Grafo G = malloc(sizeof *G);
    G->n = n;
    G->m = 0;
    G->A = malloc(n * sizeof(Noh *));
    for (i = 0; i < n; i++)
        G->A[i] = NULL;
    return G;
}
```



$O(n)$

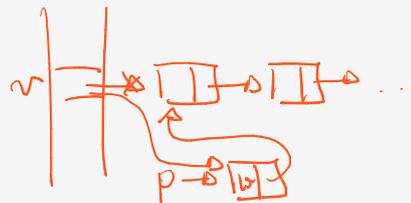
/* Insere arco v-w no grafo G, supondo que v e w são inteiros distintos entre 0 e n-1. Se grafo já tem arco v-w, não faz nada. */

```
void insereArcoGrafo(Grafo G, int v, int w) {
```

```
    Noh *p;
    for (p = G->A[v]; p != NULL; p = p->prox)
        if (p->rotulo == w)
            return;
}
```

$O(|S(v)|)$

```
    p = malloc(sizeof(Noh));
    p->rotulo = w;
    p->prox = G->A[v];
    G->A[v] = p;
    G->m++;
}
```



// Versão da insereArcoGrafo() que não testa se v-w já está presente

```
void insereArcoNaoSeguraGrafo(Grafo G, int v, int w) {
```

```
    Noh *p;
    p = malloc(sizeof(Noh));
    p->rotulo = w;
    p->prox = G->A[v];
    G->A[v] = p;
    G->m++;
}
```

$O(1)$

```
/* Imprime no arquivo saida, para cada vértice v, os vértices adjacentes a v. */
```

```
void imprimeArquivoGrafo(Grafo G, FILE *saida) {  
    int i;  
    Noh *p;  
    - fprintf(saida, "%d %d\n", G->n, G->m);  
    O(n) - for (i = 0; i < G->n; i++) {  
        for (p = G->A[i]; p != NULL; p = p->prox) ) O(|S(i)|)  
            fprintf(saida, "%2d ", p->rotulo);  
        fprintf(saida, "-1"); // sentinela para marcar fim de lista  
        fprintf(saida, "\n");  
    }  
}
```

} O(m+m)

```
// Libera a memória alocada para o grafo G e devolve NULL.
```

```
Grafo LiberaGrafo(Grafo G) {
```

```
    int i;  
    Noh *p;  
    -> for (i = 0; i < G->n; i++) {  
        p = G->A[i];  
        while (p != NULL) {  
            G->A[i] = p;  
            -> p = p->prox;  
            free(G->A[i]);  
        }  
        G->A[i] = NULL;  
    }  
    - free(G->A);  
    G->A = NULL;  
    - free(G);  
    return NULL;  
}
```

} O(m+m)

- Quiz4: Qual a eficiência das operações?
 - Algo muda se o grafo for denso ou esparso? ✓
 - Compare a versão segura e não segura da função `insereArcoGrafo()`.

Comparação entre as estruturas de dados

Matriz de adjacência:

- Vantagens
 - Acessar um elemento $A[i][j]$ qualquer leva tempo constante. $\rightarrow O(1)$
 - Economia de espaço quando o grafo é denso,
 - pois é possível operar sobre uma matriz de bits.
- Desvantagens
 - Ocupa espaço proporcional a n^2 , ainda que o grafo seja esparso,
 - 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. $\rightarrow O(n)$
 - O mesmo vale para visitar todos os nós que tem conexão para i .

Listas de adjacência:

- Vantagens
 - Economia de memória quando o grafo é esparso,
 - pois ocupa espaço proporcional a $n + m$, $\rightarrow O(n+m)$
 - 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 . $\rightarrow O(|S(i)|)$
- Desvantagens
 - Verificar se um nó i tem conexão para um nó j
 - leva tempo linear no número de vizinhos do nó i . $\rightarrow O(|S(i)|)$
 - Quando o grafo é denso, a ordem de grandeza
 - tanto da memória quanto do tempo serão quadráticos,
 - e 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 testar nossos algoritmos, por exemplo.

Existem duas maneiras principais de gerar grafos aleatórios.

Nossa primeira função constrói grafos aleatórios com exatamente m arcos.

```
/* Devolve um vértice aleatório do grafo G, supondo que  $G \rightarrow n \leq$ 
RAND_MAX. */
int verticeAleatorio(Grafo G) {
    double r;
    r = (double)rand() / ((double)RAND_MAX + 1.0);
    return (int)(r * G->n);
}

/* Constrói um grafo aleatório com vértices 0..n-1
e exatamente m arcos, supondo que  $m \leq n*(n-1)$ . (Código inspirado no
Programa 17.7 de Sedgewick.) */
Grafo grafoAleatorio1(int n, int m) {
    Grafo G = inicializaGrafo(n);
    while (G->m < m) {
        int v = verticeAleatorio(G);
        int w = verticeAleatorio(G);
        if (v != w)
            insereArcoGrafo(G, v, w);
    }
    return G;
}
```

$r \in [0, 1)$

$\in [0, n)$

\hookrightarrow # máx. de arcos

- Ela é particularmente útil para construir grafos esparsos grandes,
 - mas tende a ficar ineficiente se usada para construir grafos densos.
- Quiz5: Por que esse comportamento?
 - Qual a eficiência de melhor e pior caso dessa função?

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

- sendo mais indicada para gerar grafos densos.

/ Constrói um grafo aleatório com vértices $0..n-1$*

e número de arcos esperado igual a m , supondo que $m \leq n(n-1)$. */*

Grafo **grafoAleatorio2**(int n, int m) {

int v, w;

double prob = (double)m / n / (n - 1);

Grafo G = **inicializaGrafo**(n);

for (v = 0; v < n; v++)

for (w = 0; w < n; w++)

if (v != w)

if (rand() < prob * (RAND_MAX + 1.0))

insereArcoGrafo(G, v, w);

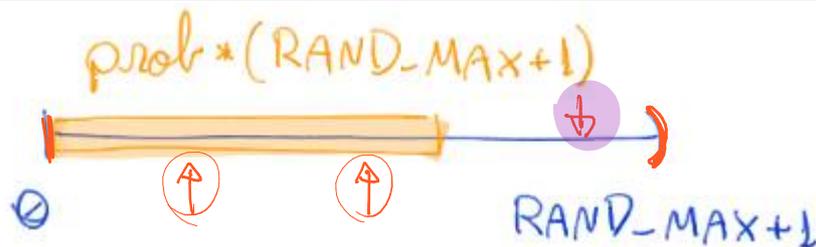
return G;

}

não de arcos

$$\left[\text{prob} = \frac{m}{n(n-1)} \right]$$

$O(n^2)$



- Quiz6: Qual a eficiência de tempo desta função?
 - E qual a vantagem da seguinte variante?

Grafo **grafoAleatorio2_1**(int n, int m) {

int v, w;

double prob = (double)m / n / (n - 1);

Grafo G = **inicializaGrafo**(n);

for (v = 0; v < n; v++)

for (w = 0; w < n; w++)

if (v != w)

if (rand() < prob * (RAND_MAX + 1.0))

insereArcoNaoSeguraGrafo(G, v, w);

return G;

}