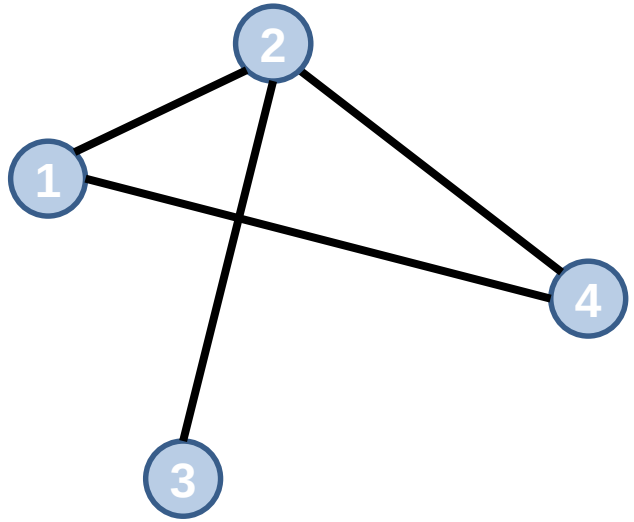
GRAFOS

Prof. André Backes | @progdescomplicada

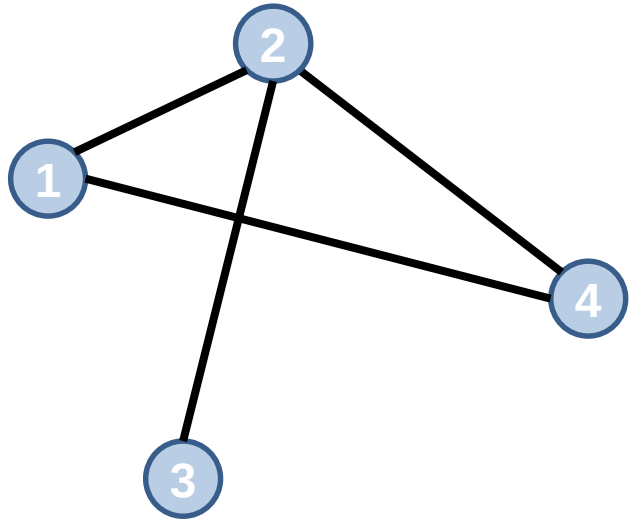
Definição

- Como representar um conjunto de objetos e as suas relações?
  - Diversos tipos de aplicações necessitam disso
  - Um grafo é um modelo matemático que representa as relações entre objetos de um determinado conjunto.



Definição

- Grafos em computação
  - Forma de solucionar problemas computáveis
  - Buscam o desenvolvimento de algoritmos mais eficientes
    - Qual a melhor rota da minha casa até o restaurante?
    - Duas pessoas tem amigos em comum?



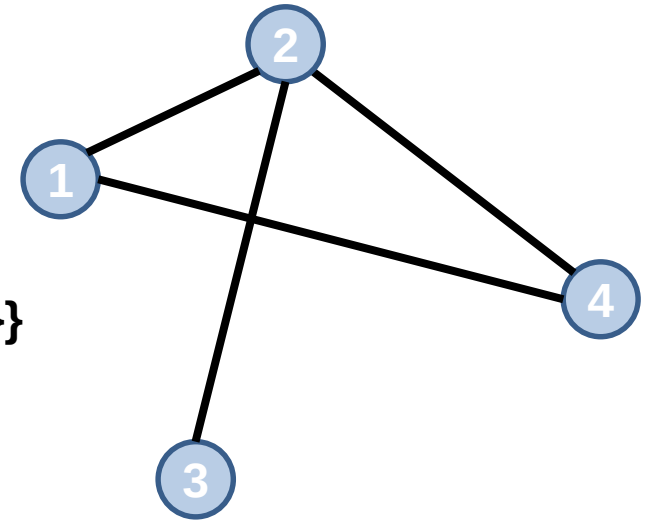
Definição

- Um grafo G(V,A) é definido por dois conjuntos
  - Conjunto V de vértices (não vazio)
    - Itens representados em um grafo;
  - Conjunto A de arestas
    - Utilizadas para conectar pares de vértices, usando um critério previamente estabelecido.

G(V,A)

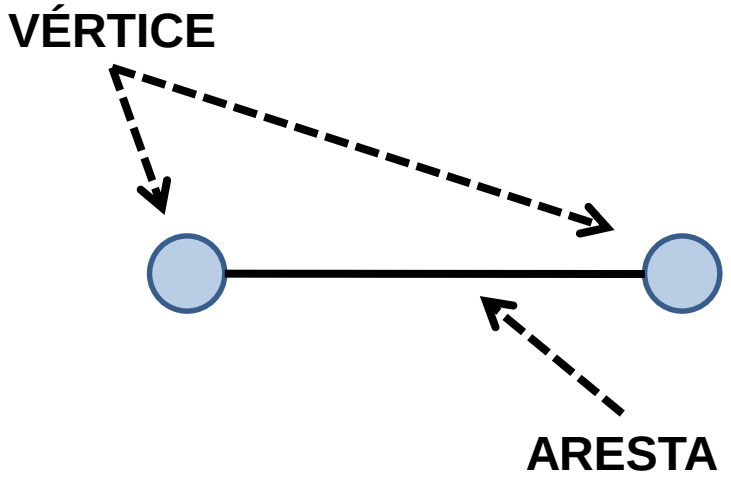
V = {1,2,3,4}

A = {{1,2},{1,4},{2,3},{2,4}}



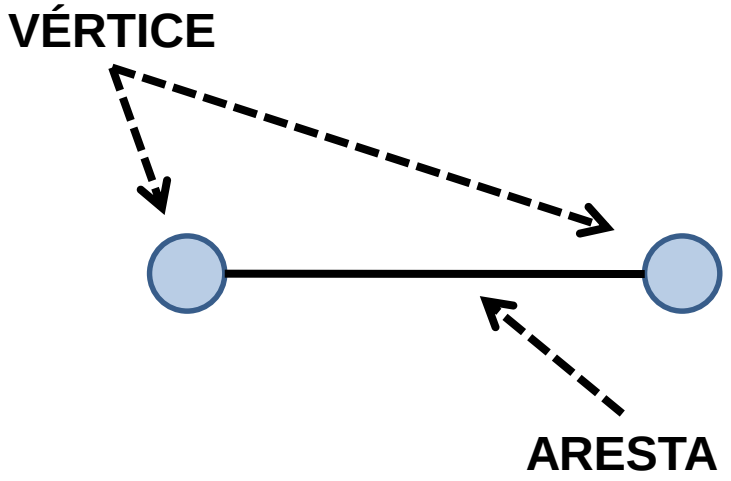
Conceitos básicos

- Vértice é cada um dos itens representados no grafo.
  - O seu significado depende da natureza do problema modelado
    - Pessoas, uma tarefa em um projeto, lugares em um mapa, etc.



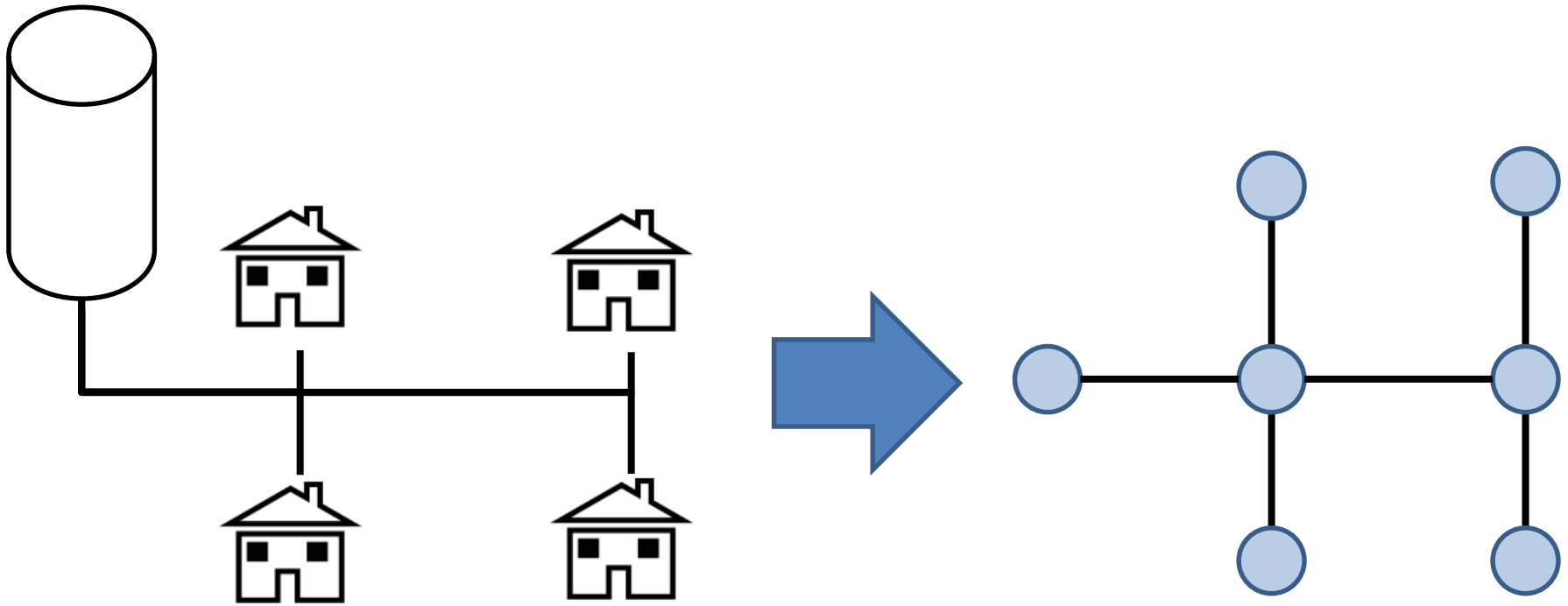
Conceitos básicos

- Aresta (ou arco) liga dois vértices
  - Diz qual a relação entre eles
  - Dois vértices são **adjacentes** se existir uma aresta ligando eles.
    - Pessoas (parentesco entre elas ou amizade), tarefas de um projeto (pré-requisito entre as tarefas), lugares de um mapa (estradas que existem ligando os lugares), etc.



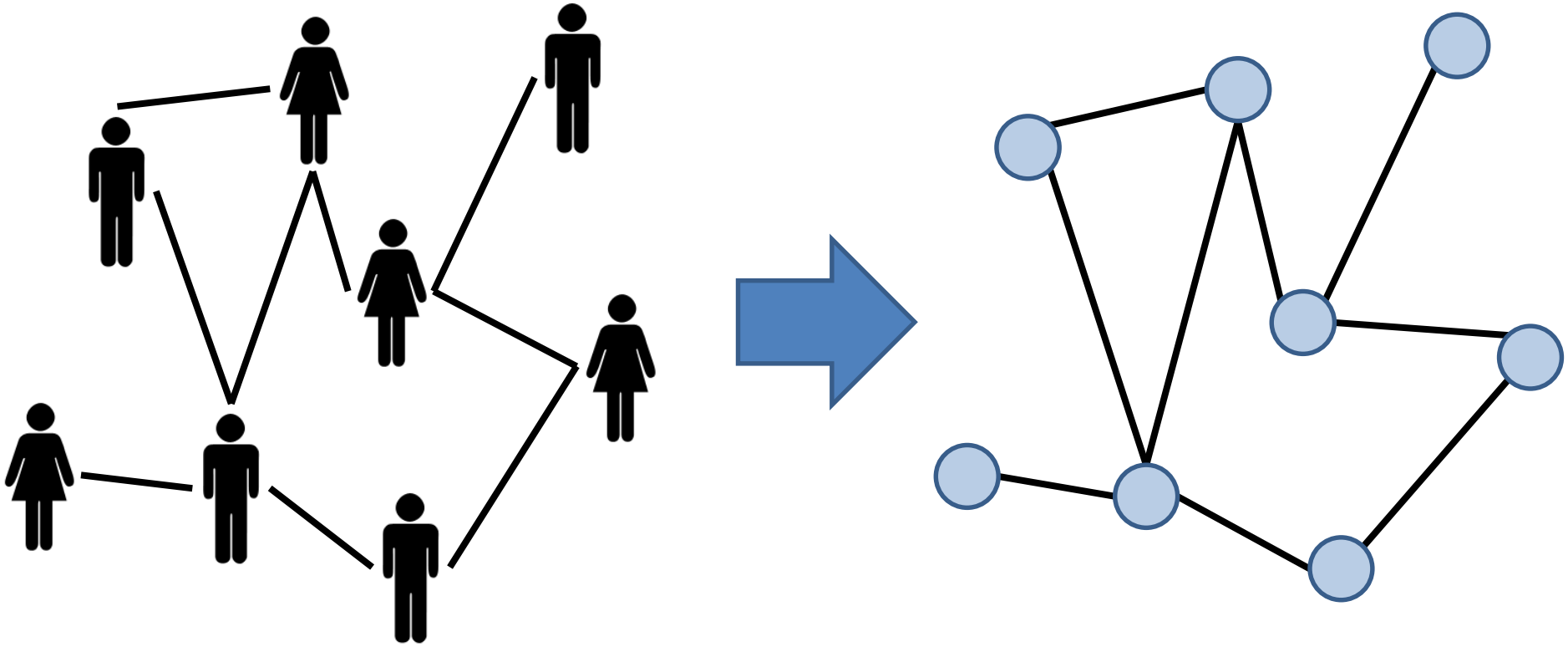
Conceitos básicos

- Praticamente qualquer objeto pode ser representado como um grafo.
  - Exemplo: sistema de distribuição de água



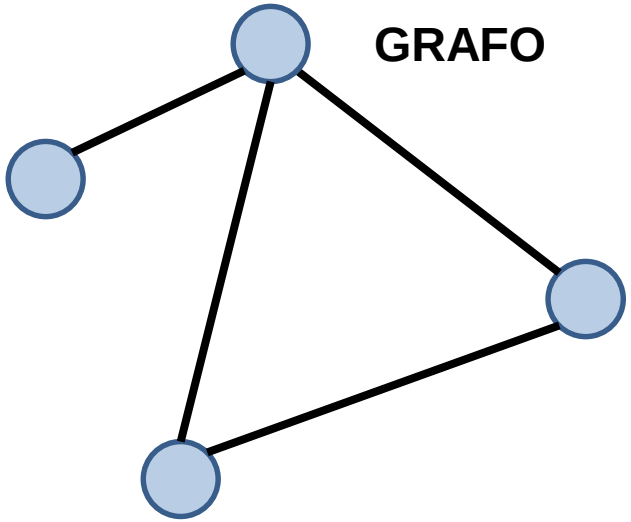
Conceitos básicos

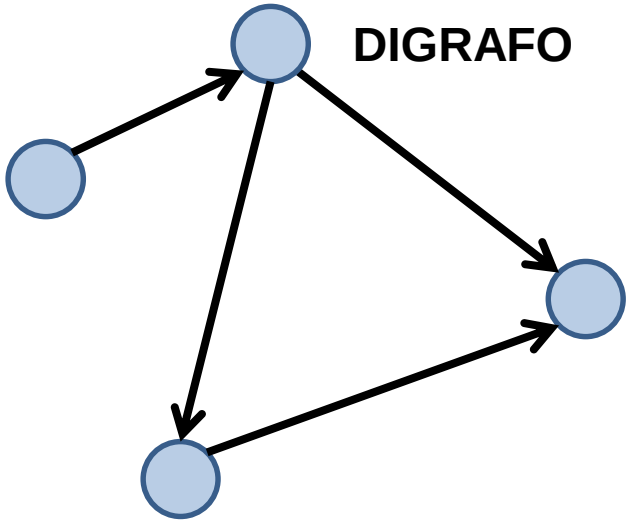
- Praticamente qualquer objeto pode ser representado como um grafo
  - Exemplo: rede social



Conceitos básicos

- As arestas podem ou não ter direção
  - Existe um orientação quanto ao sentido da aresta
    - Em um grafo direcionado ou **digrafo**, se uma aresta liga os vértices **A** a **B**, isso significa que podemos ir de **A** para **B**, mas não o contrário



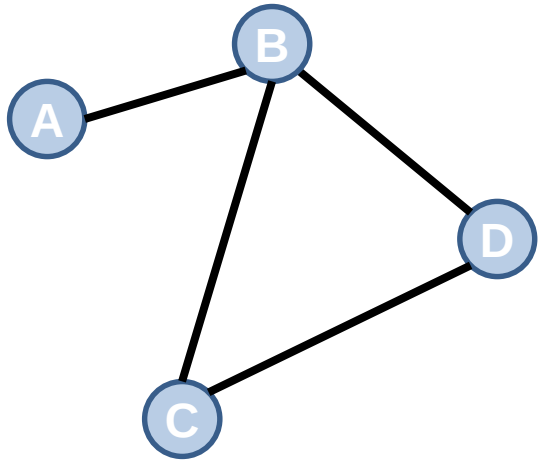


Conceitos básicos

- Grau
  - Indica o número de arestas que conectam um vértice do grafo a outros vértices
    - número de vizinhos que aquele vértice possui no grafo (que chegam ou partem dele)
  - No caso dos dígrafos, temos dois tipos de grau:
    - **grau de entrada**: número de arestas que chegam ao vértice;
    - **grau de saída**: número de arestas que partem do vértice.

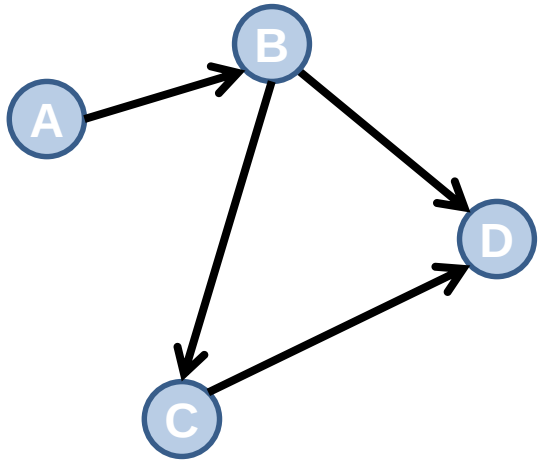
Conceitos básicos

GRAFO



Grau
G(A) = 1
G(B) = 3
G(C) = 2
G(D) = 2

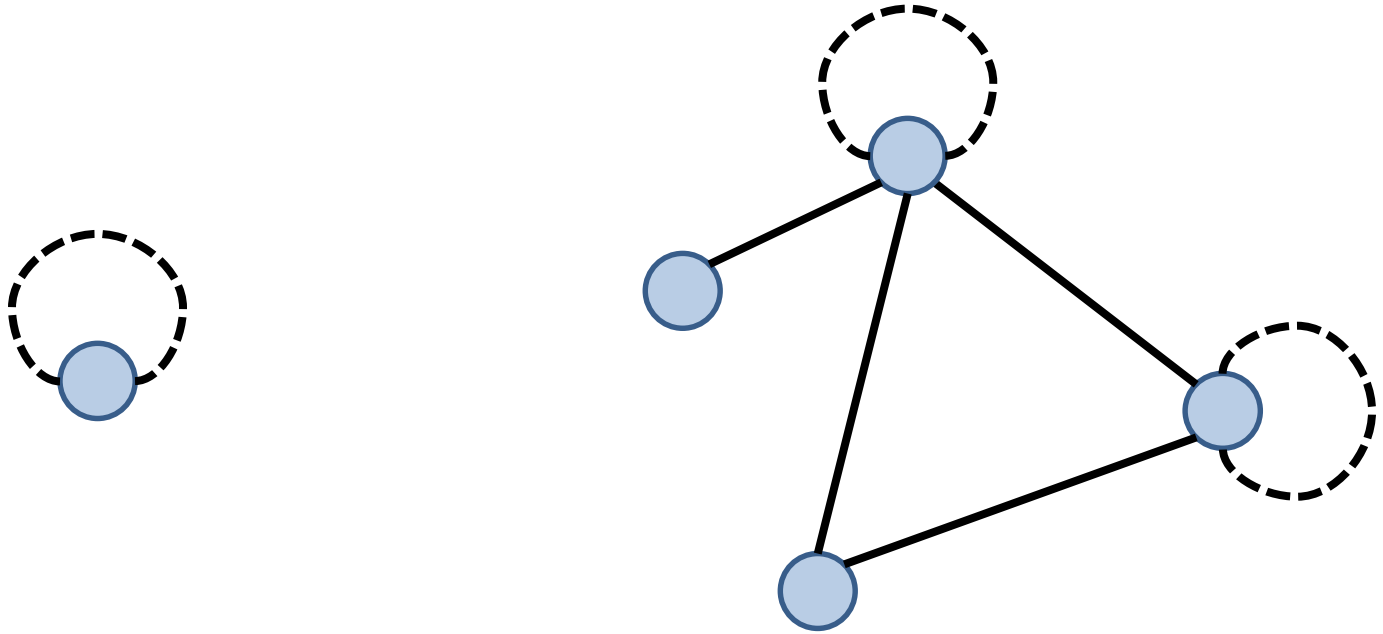
DIGRAFO



Grau Entrada	Grau Saída
G(A) = 0	G(A) = 1
G(B) = 1	G(B) = 2
G(C) = 1	G(C) = 1
G(D) = 2	G(D) = 0

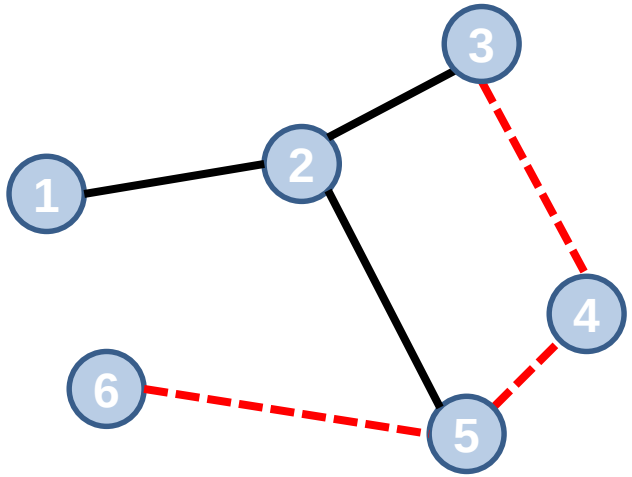
Conceitos básicos

- Laço
  - Uma aresta é chamada de laço se seu vértice de partida é o mesmo que o de chagada
    - A aresta conecta o vértice a ele mesmo

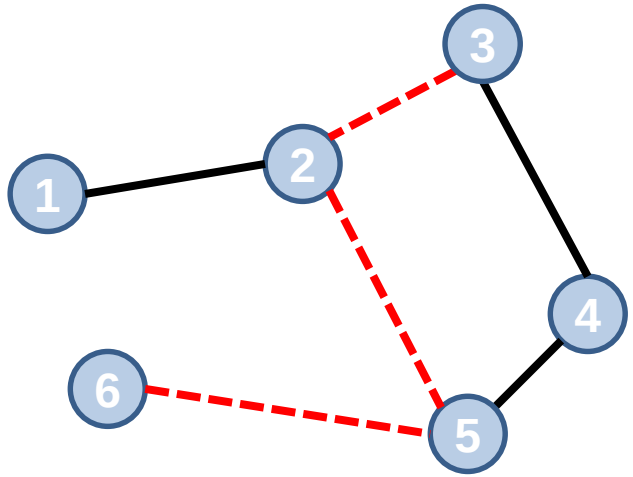


Conceitos básicos

- Caminho
  - Um caminho entre dois vértices é uma sequência de vértices onde cada vértice está conectado ao vértice seguinte por meio de uma aresta.



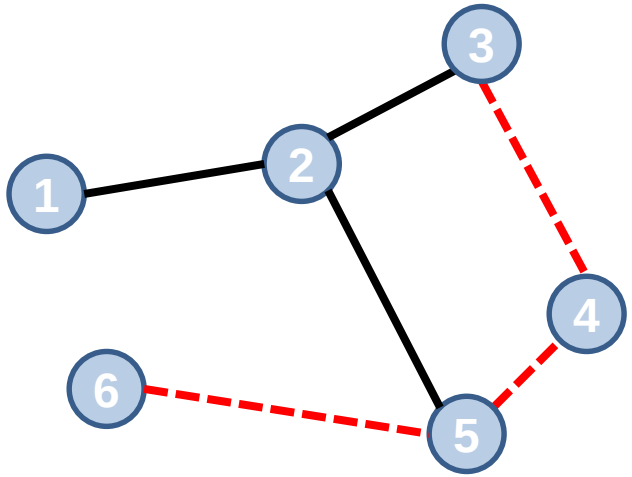
CAMINHO: 3-4-5-6



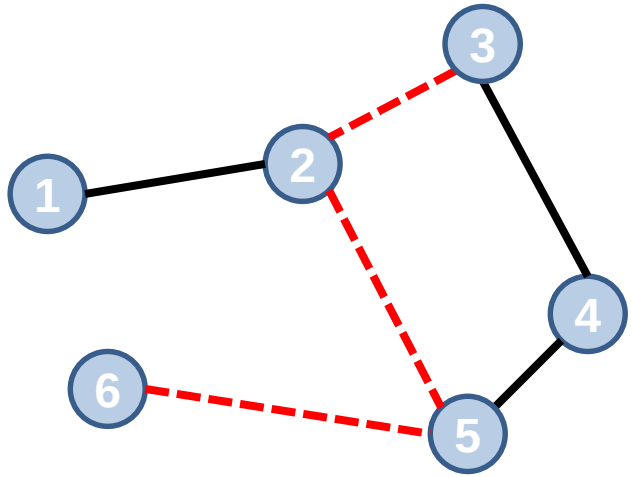
CAMINHO: 3-2-5-6

Conceitos básicos

- Caminho
  - Comprimento do caminho: número de vértices que precisamos percorrer de um vértice até o outro



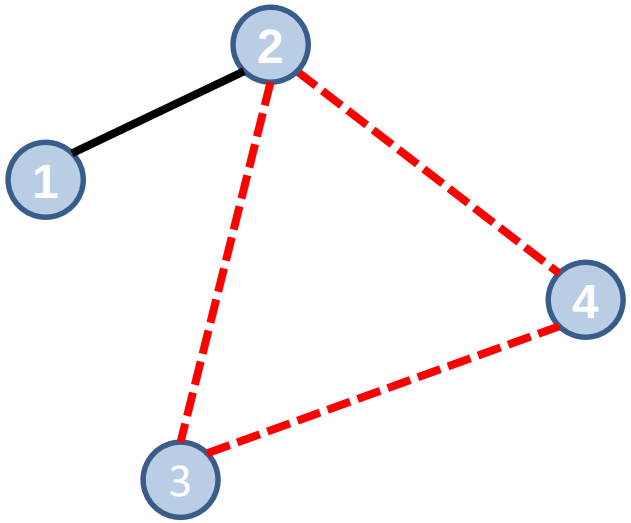
CAMINHO: 3-4-5-6



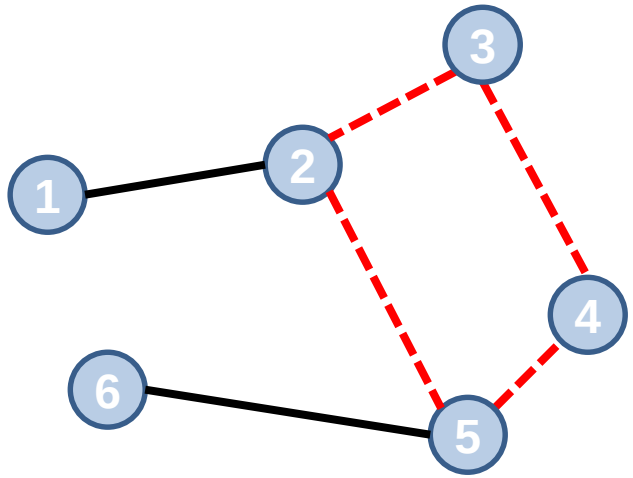
CAMINHO: 3-2-5-6

Conceitos básicos

- Ciclo
  - Caminho onde o vértice inicial e o final são o mesmo vértice.
    - Note que um ciclo é um caminho fechado sem vértices repetidos



CICLO: 2-3-4

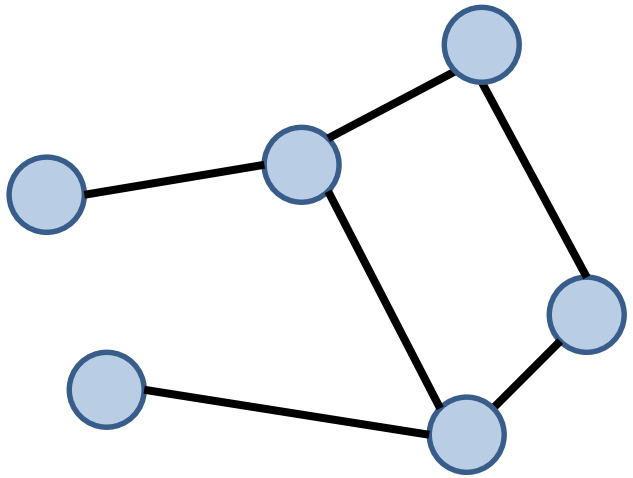


CICLO: 2-3-4-5

Tipos de Grafos

- Grafo trivial
  - Possui um único vértice e nenhuma aresta
- Grafo simples
  - Grafo não direcionado, sem laços e sem arestas paralelas (multigrafo)

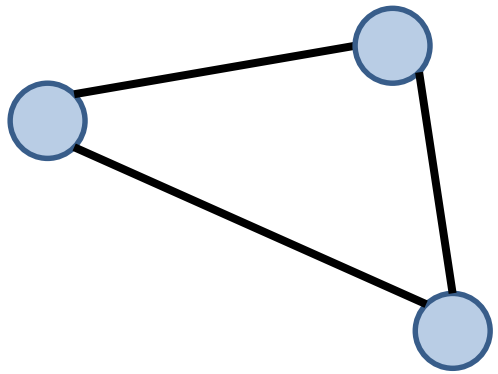
TRIVIAL

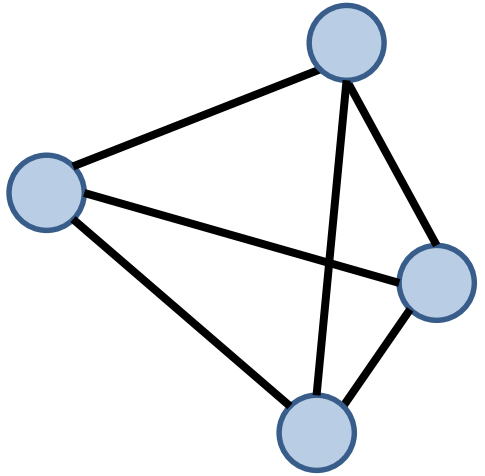


SIMPLES

Tipos de Grafos

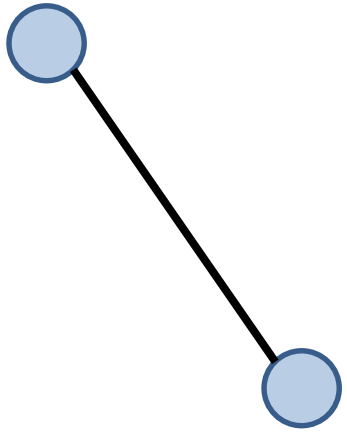
- Grafo completo
  - Grafo simples onde cada vértice se conecta a todos os outros vértices do grafo.



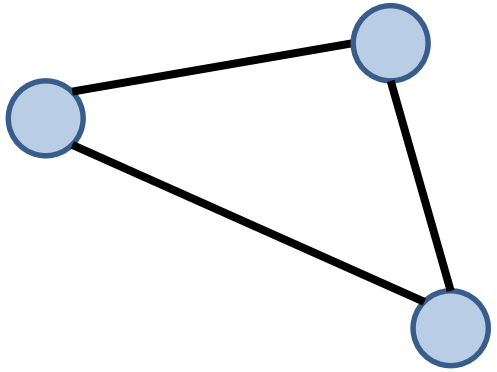


Tipos de Grafos

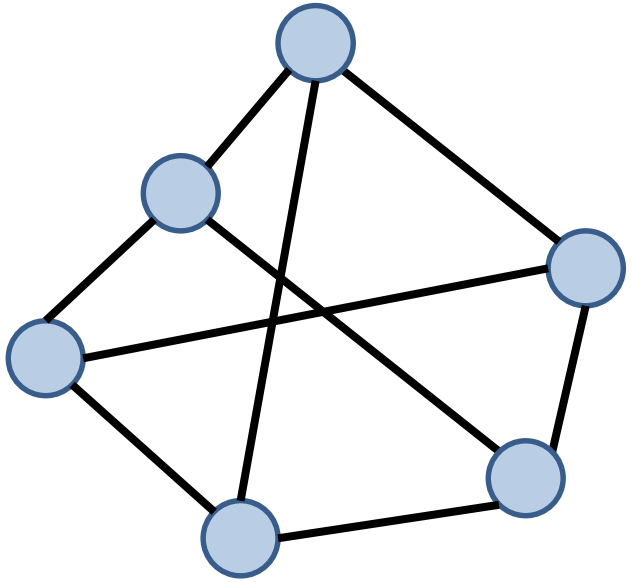
- Grafo regular
  - Grafo onde todos os seus vértices possuem o mesmo grau (número de arestas ligadas a ele)
  - Todo grafo completo é também regular



Grau = 1



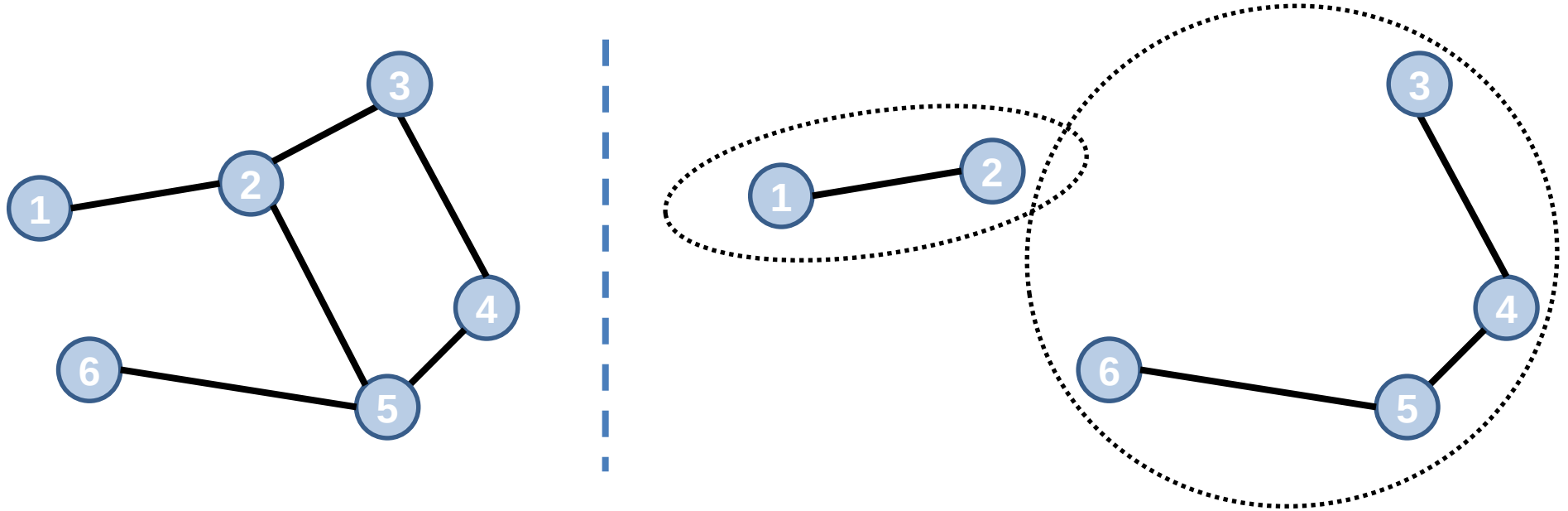
Grau = 2



Grau = 3

Tipos de Grafos

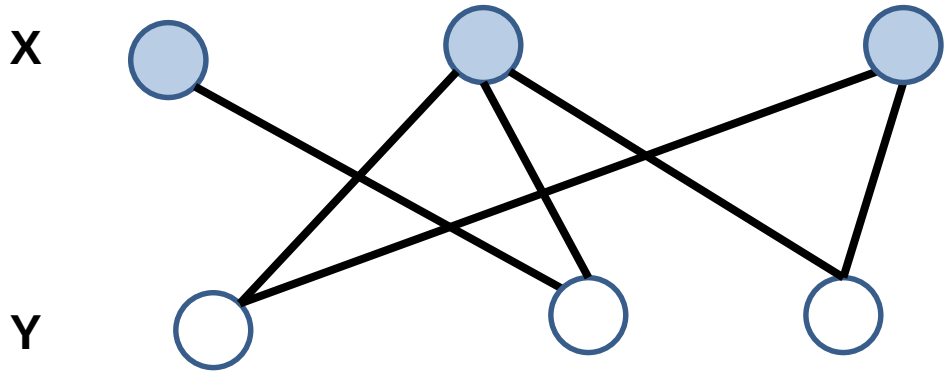
- Subgrafo
  - Gs(Vs,As) é um subgrafo de G(V,A) se o conjunto de vértices Vs for um subconjunto de V, Vs ⊆ V, e se o conjunto de arestas As for um subconjunto de A, As ⊆ A.



GRAFO **SUBGRAFOS**

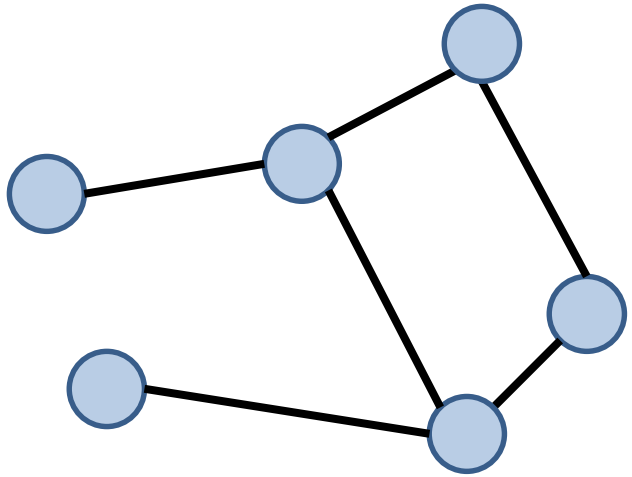
Tipos de Grafos

- Grafo bipartido
  - Um grafo G(V,A) onde o seu conjunto de vértices pode ser divididos em dois subconjuntos X e Y sem intersecção.
    - As arestas conectam apenas os vértices que estão em subconjuntos diferentes

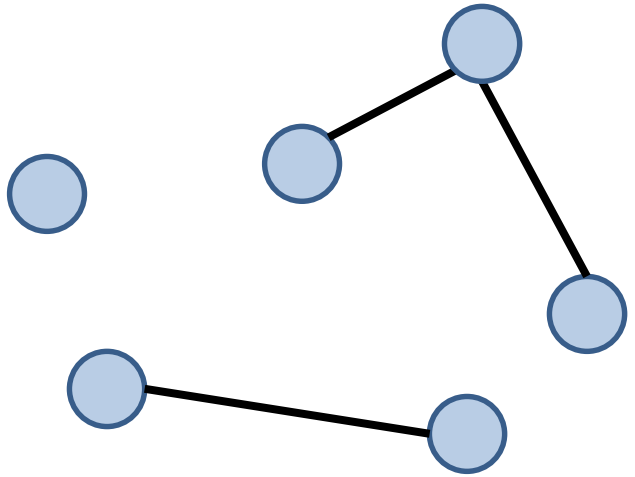


Tipos de Grafos

- Grafo conexo e desconexo
  - **Grafo conexo**: existe um caminho ligando quaisquer dois vértices.
  - Quando isso não acontece, temos um **grafo desconexo**



GRAFO CONEXO



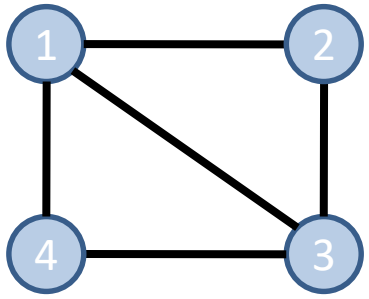
GRAFO DESCONEXO

Tipos de Grafos

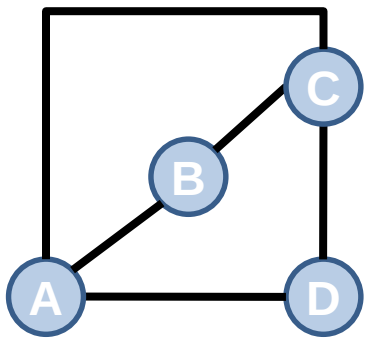
- Grafos isomorfos
  - Dois grafos, G1(V1,A1) e G2(V2,A2), são ditos **isomorfos** se existe uma função que faça o mapeamento de vértices e arestas de modo que os dois grafos se tornem coincidentes.
    - Em outras palavras, dois grafos são isomorfos se existe uma função ***f*** onde, para cada dois vértices **a** e **b** adjacentes no grafo G1, **f(a)** e **f(b)** também são adjacentes no grafo G2.

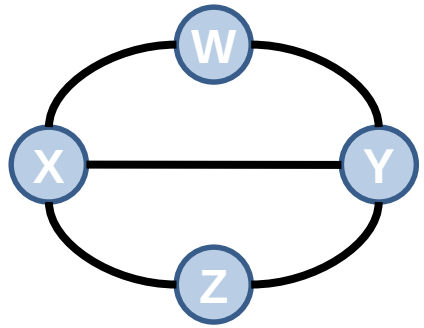
Tipos de Grafos

- Grafos isomorfos



Grau
f(1) = A
f(2) = B
f(3) = C
f(4) = D

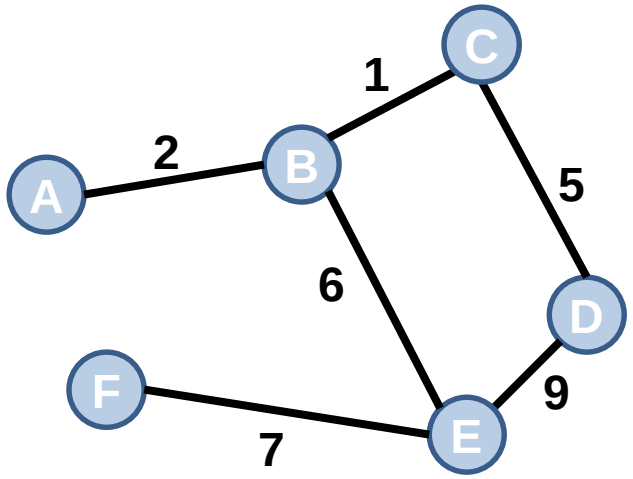




Grau
f(1) = X
f(2) = W
f(3) = Y
f(4) = Z

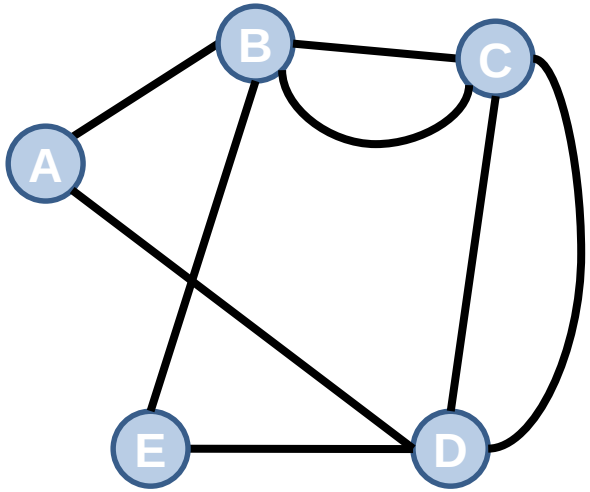
Tipos de Grafos

- Grafo ponderado
  - É um grafo que possui **pesos** (valor numérico) associados a cada uma de suas arestas.

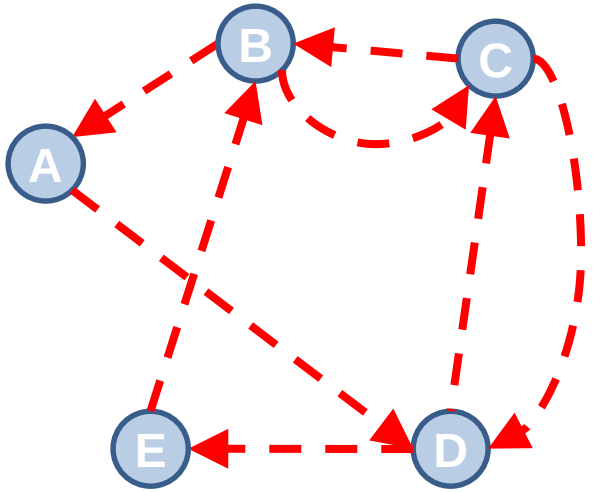


Tipos de Grafos

- Grafo Euleriano
  - Grafo que possui um **ciclo** que visita todas as suas arestas apenas uma vez, iniciando e terminando no mesmo vértice.



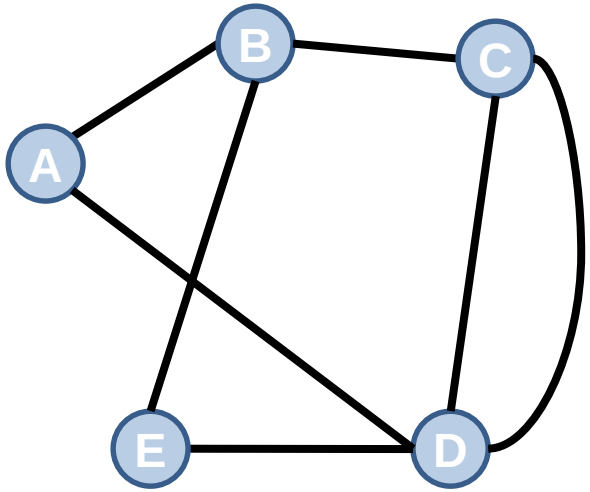
GRAFO EULERIANO



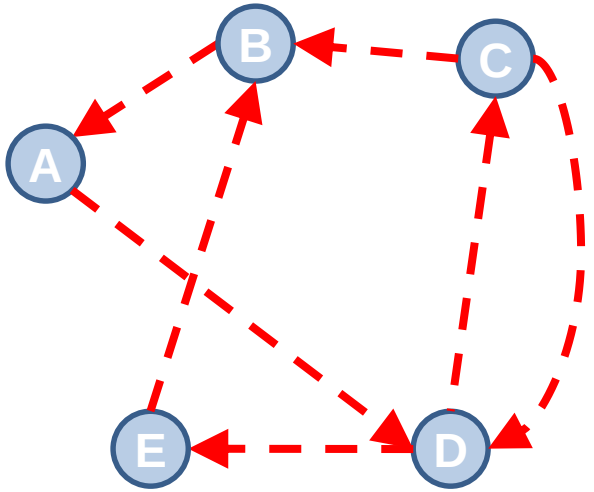
CICLO EULERIANO
C-D-C-B-A-D-E-B-C

Tipos de Grafos

- Grafo Semi-Euleriano
  - Grafo que possui um **caminho** aberto (não é um ciclo) que visita todas as suas arestas apenas uma vez.



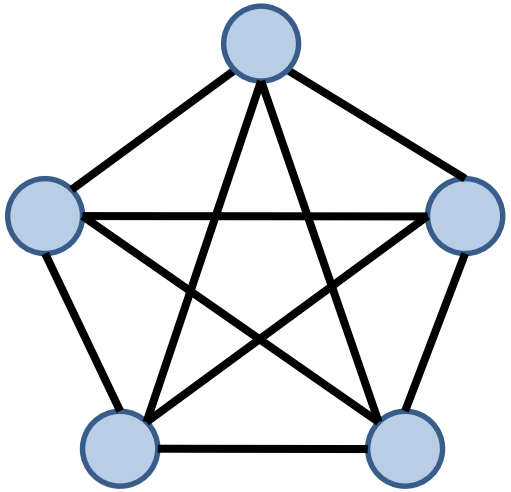
GRAFO SEMI-EULERIANO



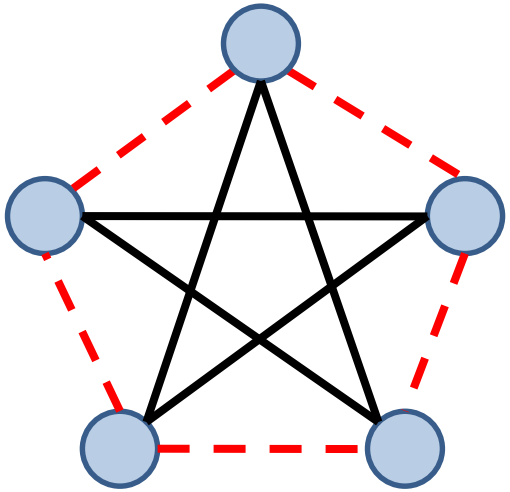
CAMINHO EULERIANO C-D-C-B-A-D-E-B

Tipos de Grafos

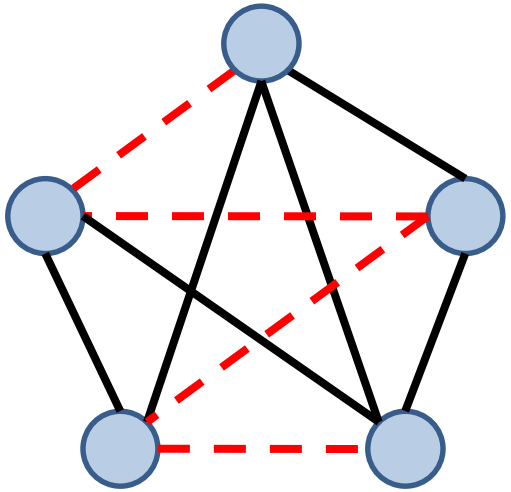
- Grafo Hamiltoniano
  - Grafo que possui um caminho que visita todos os seus vértices apenas uma vez.
  - Pode ser um ciclo



GRAFO HAMILTONIANO



CICLO HAMILTONIANO



CAMINHO HAMILTONIANO

Tipos de representação

- Como representar um grafo no computador?
  - Existem duas abordagens muito utilizadas:
    - Matriz de Adjacência
    - Lista de Adjacência
  - Qual a representação que deve ser utilizada?
    - Depende da aplicação!

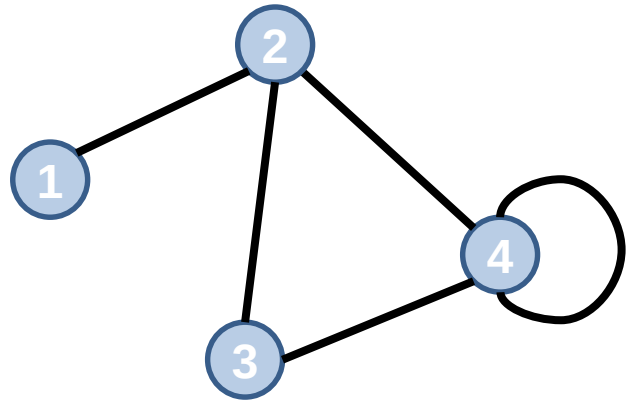
Tipos de representação

- Matriz de adjacência
  - Utiliza uma matriz **N x N** para armazenar o grafo, onde **N** é o número de vértices
    - Alto custo computacional, $O(N^2)$
  - Uma aresta é representada por uma marca na posição **(i , j)** da matriz
    - Aresta liga o vértice **i** ao **j**

Tipos de representação

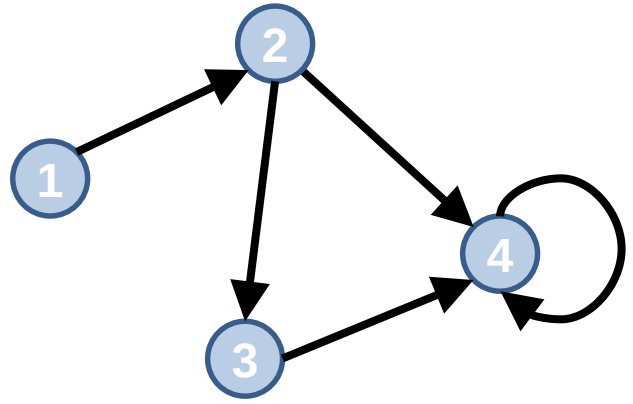
- Matriz de adjacência

GRAFO



	1	2	3	4
1	0	1	0	0
2	1	0	1	1
3	0	1	0	1
4	0	1	1	1

DIGRAFO



	1	2	3	4
1	0	1	0	0
2	0	0	1	1
3	0	0	0	1
4	0	0	0	1

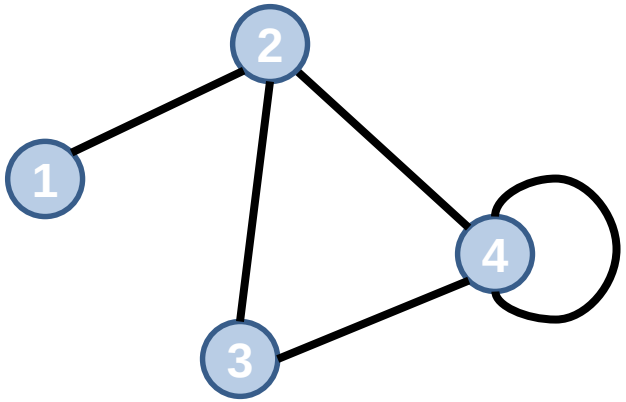
Tipos de representação

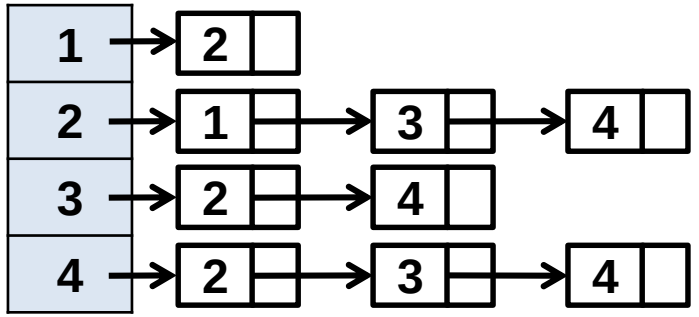
- Lista de adjacência
  - Utiliza uma lista de vértices para descrever as relações entre os vértices.
    - Um grafo contendo **N** vértices utiliza um array de ponteiros de tamanho **N** para armazenar os vértices do grafo
    - Para cada vértice é criada uma lista de arestas, onde cada posição da lista armazena o índice do vértice a qual aquele vértice se conecta

Tipos de representação

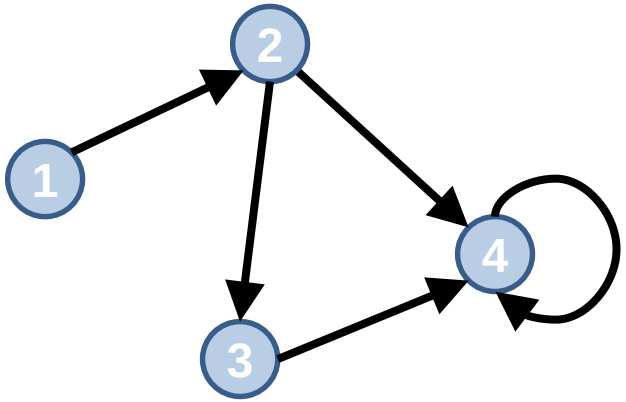
- Lista de adjacência

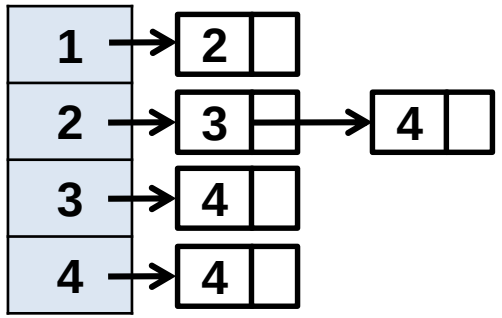
GRAFO





DIGRAFO





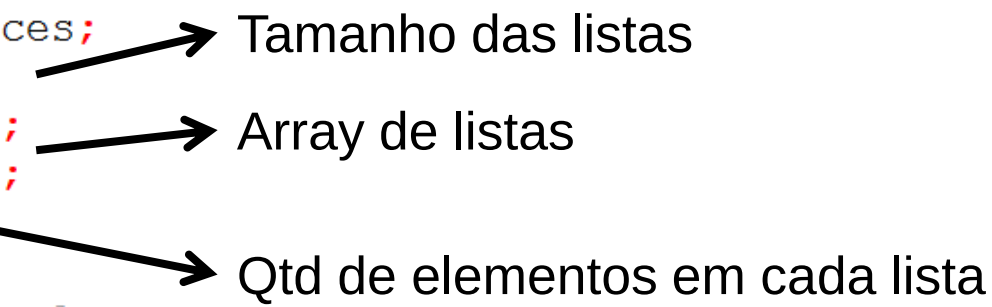
Tipos de representação

- Qual representação utilizar?
  - Lista de adjacência é mais indicada para um grafo que possui muitos vértices mas poucas arestas ligando esses vértices.
  - A medida que o número de arestas cresce e não havendo nenhuma outra informação associada a aresta (por exemplo, seu peso), o uso de uma matriz de adjacência se torna mais eficiente

TAD Grafo

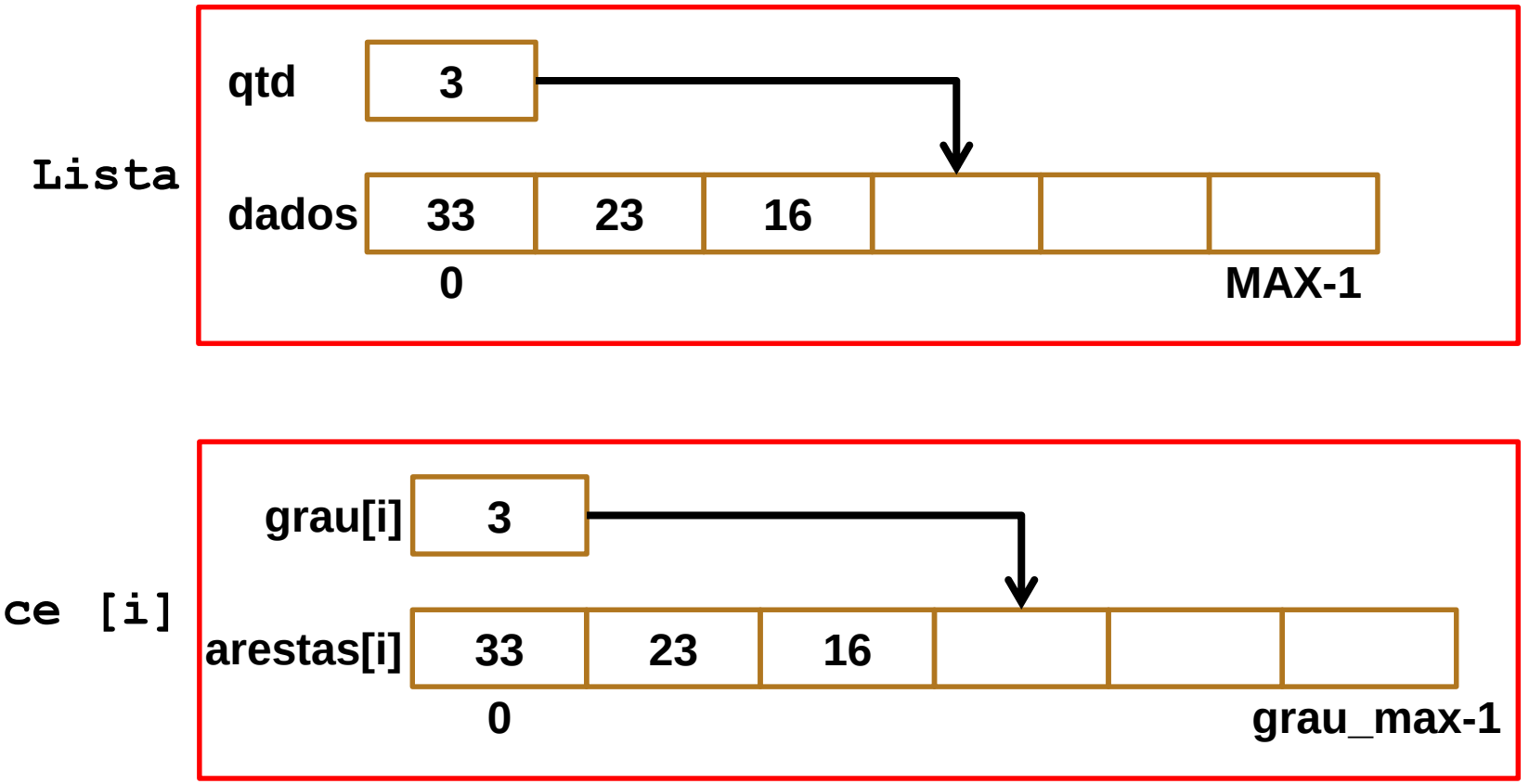
- Vamos usar uma **lista de adjacência**
  - Lista de arestas: lista sequencial

```
3   //Arquivo Grafo.h
4   typedef struct grafo Grafo;
5
6   //Arquivo Grafo.c
7   #include <stdio.h>
8   #include <stdlib.h>
9   #include "Grafo.h" //inclui os Protótipos
10  //Definição do tipo Grafo
11  struct grafo{
12      int eh_ponderado;
13      int nro_vertices;
14      int grau_max;
15      int** arestas;
16      float** pesos;
17      int* grau;
18  };
19  //Programa principal
20  Grafo *gr;
```



TAD Grafo

- Cada vértice funciona como uma lista estática



TAD Grafo

- Criando um grafo

```
5   "Criando um grafo"
6   //Arquivo Grafo.h
7   Grafo *cria_Grafo(int nro_vertices, int grau_max,
8                     int eh_ponderado);
9
10  //Programa principal
11  Grafo *gr;
12  gr = cria_Grafo(10, 7, 0);
13
```

TAD Grafo

- Criando um grafo

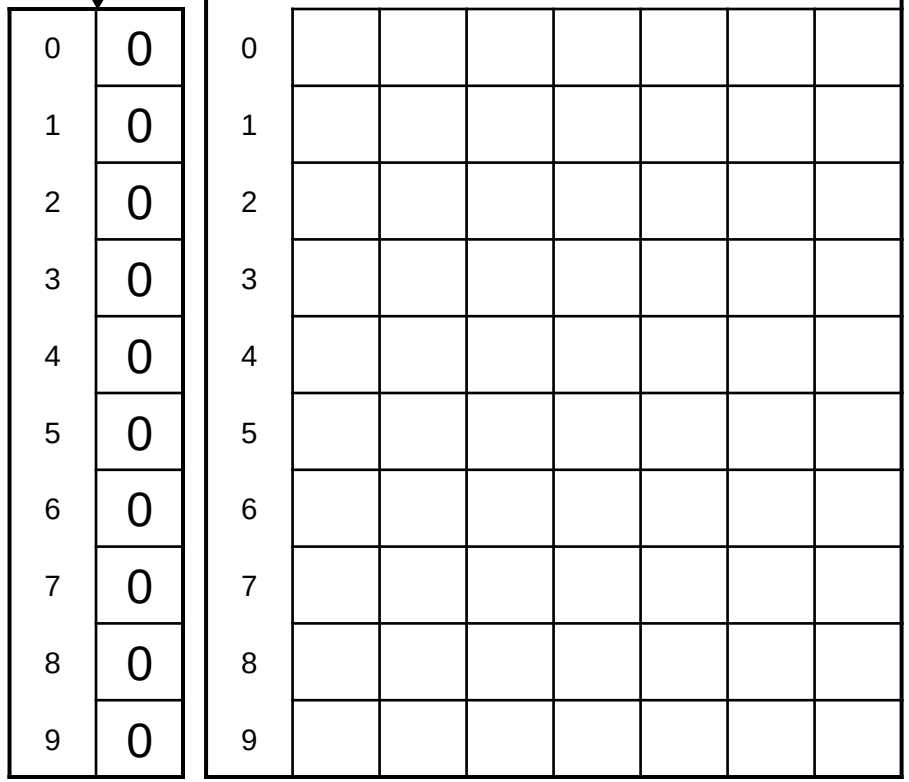
Cria matriz arestas

Cria matriz pesos

```
Grafo* cria_Grafo(int nro_vertices, int grau_max,
                  int eh_ponderado){
 Grafo *gr=(Grafo*) malloc(sizeof(struct grafo));
 if(gr != NULL){
  int i;
  gr->nro_vertices = nro_vertices;
  gr->grau_max = grau_max;
  gr->eh_ponderado = (eh_ponderado != 0)?1:0;
  gr->grau=(int*)calloc(nro_vertices,sizeof(int));
  gr->arestas=(int**)malloc(nro_vertices*sizeof(int*));
  for(i=0; i<nro_vertices; i++)
    gr->arestas[i]=(int*)malloc(grau_max*sizeof(int));
  if(gr->eh_ponderado){
    gr->pesos=(float**)malloc(nro_vertices*
                              sizeof(float*));
    for(i=0; i<nro_vertices; i++)
      gr->pesos[i]=(float*)malloc(grau_max*
                                  sizeof(float));
  }
 }
 return gr;
}
```

TAD Grafo

```
struct grafo{
    int eh_ponderado;
    int nro_vertices;
    int grau_max;
    int** arestas;
    float** pesos;
    int* grau;
};
Grafo *gr;
gr = cria_Grafo(10, 7, 0);
```



Cria um grafo de 10 vértices. Cada vértice se conecta com até outros 7 vértices

- **Matriz 10x7 para as arestas**
- **Vetor "grau" guarda o número de conexões de cada um dos 10 vértices**

TAD Grafo

- Liberando o grafo

```
 5    "Liberando um grafo"
 6    //Arquivo Grafo.h
 7    void libera_Grafo(Grafo* gr);
 8
 9    //Programa principal
10    Grafo *gr;
11    gr = cria_Grafo(10, 7, 0);
12    .
13    .
14    .
15    libera_Grafo(gr);
16
```

TAD Grafo

- Liberando o grafo

```
1  void libera_Grafo(Grafo* gr){
2      if(gr != NULL){
3          int i;
4          for(i=0; i<gr->nro_vertices; i++)
5              free(gr->arestas[i]);
6          free(gr->arestas);
7
8          if(gr->eh_ponderado){
9              for(i=0; i<gr->nro_vertices; i++)
10                 free(gr->pesos[i]);
11             free(gr->pesos);
12         }
13         free(gr->grau);
14         free(gr);
15     }
16 }
```

Libera matriz arestas

Libera matriz pesos

TAD Grafo

- Inserindo uma aresta

```
5     "Inserindo uma aresta no grafo"
6     //Arquivo Grafo.h
7     int insereAresta(Grafo* gr, int orig, int dest,
8                      int eh_digrafo, float peso);
9     //Programa principal
10    Grafo *gr;
11    gr = cria_Grafo(10, 7, 0);
12    insereAresta(gr, 0, 1, 0, 0);
13    insereAresta(gr, 1, 3, 0, 0);
14    .
15    .
16    .
17    libera_Grafo(gr);
18
```

TAD Grafo

- Inserindo uma aresta

```
1  int insereAresta(Grafo* gr, int orig, int dest,
2                   int eh_digrafo, float peso){
3      if(gr == NULL)
4          return 0;
5      if(orig < 0 || orig >= gr->nro_vertices)
6          return 0;
7      if(dest < 0 || dest >= gr->nro_vertices)
8          return 0;
9
10     gr->arestas[orig][gr->grau[orig]] = dest;
11     if(gr->eh_ponderado)
12         gr->pesos[orig][gr->grau[orig]] = peso;
13     gr->grau[orig]++;
14
15     if(eh_digrafo == 0)
16         insereAresta(gr,dest,orig,1,peso);
17     return 1;
18 }
```

Verifica se vértice existe

Insere no final da linha

Insere outra aresta se NÃO for digrafo

TAD Grafo

```
insereAresta(gr,0,1,0,0);
```

Antes da inserção

0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0

	0	1	2	3	4	5	6
0							
1							
2							
3							
4							
5							
6							
7							
8							
9							

Após a inserção

0	1
1	1
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0

	0	1	2	3	4	5	6
0	1						
1	0						
2							
3							
4							
5							
6							
7							
8							
9							

TAD Grafo

```
insereAresta(gr,0,1,0,0);
insereAresta(gr,1,3,0,0);
```

Antes da inserção

0	1
1	1
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0

	0	1	2	3	4	5	6
0	1						
1	0						
2							
3							
4							
5							
6							
7							
8							
9							

Após a inserção

0	1
1	2
2	0
3	1
4	0
5	0
6	0
7	0
8	0
9	0

	0	1	2	3	4	5	6
0	1						
1	0	3					
2							
3	1						
4							
5							
6							
7							
8							
9							

TAD Grafo

```
insereAresta(gr,0,1,0,0);
insereAresta(gr,1,3,0,0);
insereAresta(gr,3,2,0,0);
```

Antes da inserção

0	1
1	2
2	0
3	1
4	0
5	0
6	0
7	0
8	0
9	0

	0	1	2	3	4	5	6
0	1						
1	0	3					
2							
3	1						
4							
5							
6							
7							
8							
9							

Após a inserção

0	1
1	2
2	1
3	2
4	0
5	0
6	0
7	0
8	0
9	0

	0	1	2	3	4	5	6
0	1						
1	0	3					
2	3						
3	1	2					
4							
5							
6							
7							
8							
9							

TAD Grafo

```
insereAresta(gr,0,1,0,0);
insereAresta(gr,1,3,0,0);
insereAresta(gr,3,2,0,0);
insereAresta(gr,6,1,0,0);
```

Antes da inserção

0	1
1	2
2	1
3	2
4	0
5	0
6	0
7	0
8	0
9	0

	0	1	2	3	4	5	6
0	1						
1	0	3					
2	3						
3	1	2					
4							
5							
6							
7							
8							
9							

Após a inserção

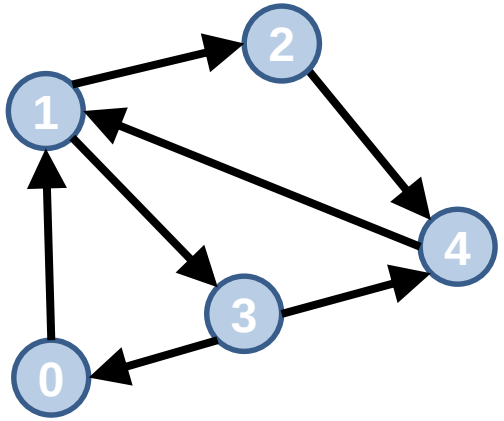
0	1
1	3
2	1
3	2
4	0
5	0
6	1
7	0
8	0
9	0

	0	1	2	3	4	5	6
0	1						
1	0	3	6				
2	3						
3	1	2					
4							
5							
6	1						
7							
8							
9							

BUSCAS E MENOR CAMINHO

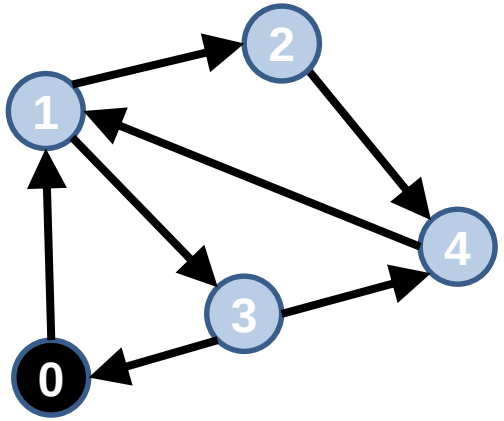
Busca em grafos

- Definição
  - Consiste em explorar o grafo de uma maneira bem específica.
  - Trata-se de um processo sistemático de como caminhar por seus vértices e arestas.



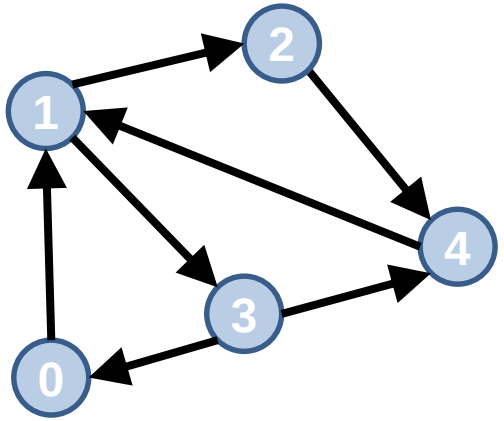
Busca em grafos

- De modo geral, as operações de busca dependem do vértice inicial
  - O ponto de partida é um aspecto bastante importante da própria busca.
    - Por exemplo, em uma busca pelo menor caminho, temos que saber qual é o ponto de partida desse caminho.



Busca em grafos

- Vários problemas em grafos podem ser resolvidos efetuando uma busca
- A busca pode precisar visitar todos ou apenas um subconjunto dos vértices.



Busca em grafos

- Existem vários tipos de busca que podemos realizar em um grafo. Os três principais:
  - Busca em profundidade
  - Busca em largura
  - Busca pelo menor caminho

Busca em largura

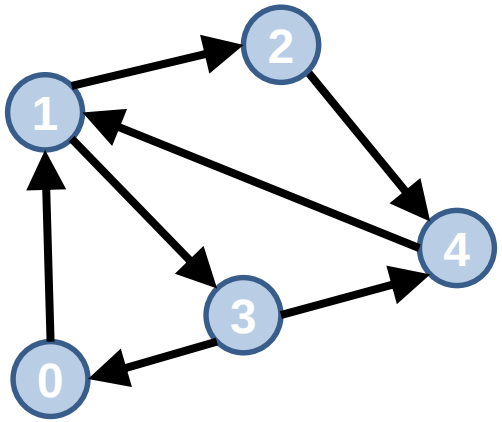
- Funcionamento
  - Partindo de um vértice inicial, a busca explora todos os vizinhos de um vértice. Em seguida, para cada vértice vizinho, ela repete esse processo, visitando os vértices ainda inexplorados
  - Em outras palavras, esse tipo de busca se inicia em um vértice e então visita todos os seus vizinhos antes de se aprofundar na busca. Esse processo continua até que
    - o alvo da busca seja encontrado
    - não existam mais vértices a serem visitados.

Busca em largura

- Esse algoritmo faz uso do conceito de fila
  - O grafo é percorrido de maneira sistemática, primeiro marcando como “visitados” todos os vizinhos de um vértice e em seguida começa a visitar os vizinhos de cada vértice na ordem em que eles foram marcados.
  - Para realizar essa tarefa, uma fila é utilizada para administrar a visitação dos vértices
    - o primeiro vértice marcado (ou marcado a mais tempo) é o primeiro a ser visitado.

Busca em largura | Grafo para teste

```c
#include <stdio.h>
#include <stdlib.h>
#include "Grafo.h"
int main(){
    int eh_digrafo = 1;
    Grafo* gr = cria_Grafo(5, 5, 0);
    insereAresta(gr, 0, 1, eh_digrafo, 0);
    insereAresta(gr, 1, 3, eh_digrafo, 0);
    insereAresta(gr, 1, 2, eh_digrafo, 0);
    insereAresta(gr, 2, 4, eh_digrafo, 0);
    insereAresta(gr, 3, 0, eh_digrafo, 0);
    insereAresta(gr, 3, 4, eh_digrafo, 0);
    insereAresta(gr, 4, 1, eh_digrafo, 0);
    int vis[5];

    buscaLargura_Grafo(gr, 0, vis);

    libera_Grafo(gr);

    system("pause");
    return 0;
}
```



Busca em largura | Implementação

```
void buscaLargura_Grafo(Grafo *gr, int ini, int *visita
    int i, vert, NV, cont = 1, *fila, IF = 0, FF = 0;
    for(i=0; i<gr->nro_vertices; i++)
        visitado[i] = 0;
    NV = gr->nro_vertices;
    fila = (int*) malloc(NV * sizeof(int));
    FF++;
    fila[FF] = ini;
    visitado[ini] = cont;
    while(IF != FF){
        IF = (IF + 1) % NV;
        vert = fila[IF];
        cont++;
        for(i=0; i<gr->grau[vert]; i++){
            if(!visitado[gr->arestas[vert][i]]){
                FF = (FF + 1) % NV;
                fila[FF] = gr->arestas[vert][i];
                visitado[gr->arestas[vert][i]] = cont;
            }
        }
    }
    free(fila);
```

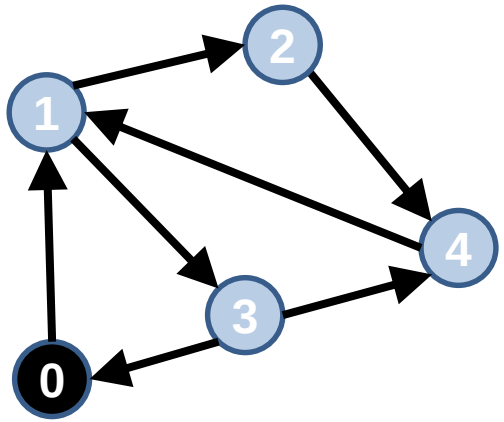
Marca vértices como NÃO visitados

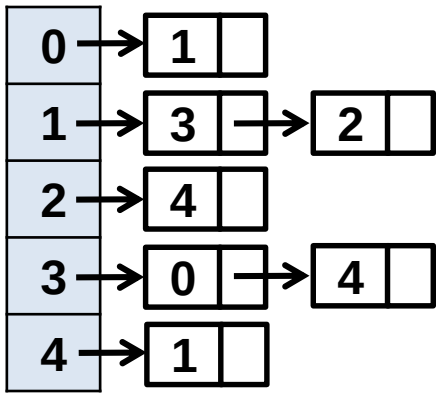
Cria fila. Visita e insere "ini" na fila

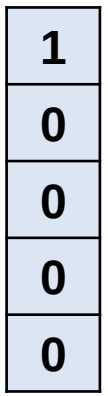
Pega primeiro da fila

Visita os vizinhos ainda não visitados e coloca na fila

Busca em largura | Passo a passo



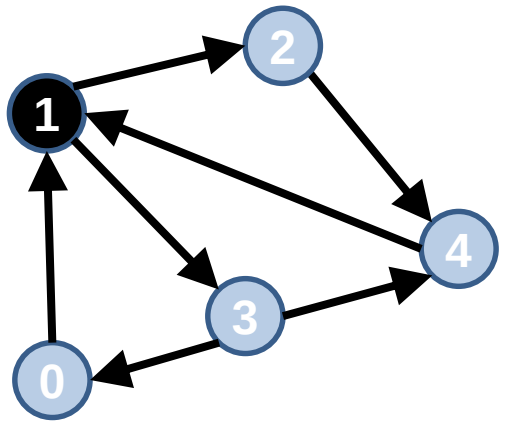


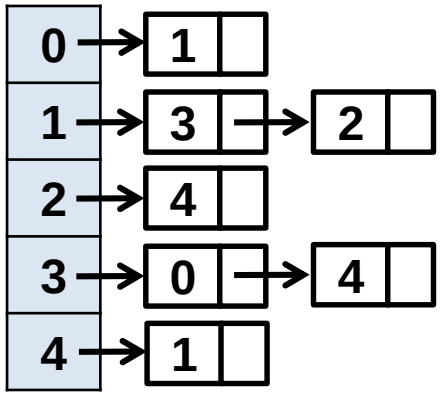


fila 0

Inicializa a busca em largura.
Visita e insere na fila o vértice 0.

Busca em largura | Passo a passo





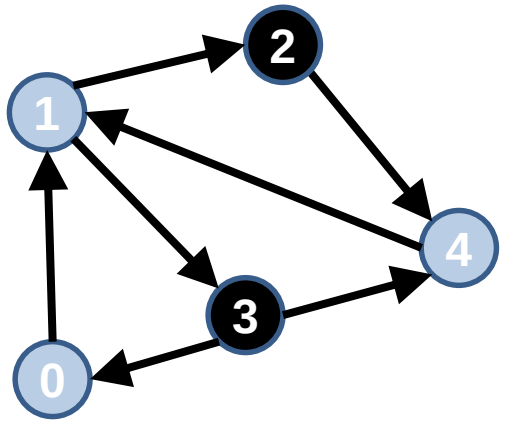
visitado

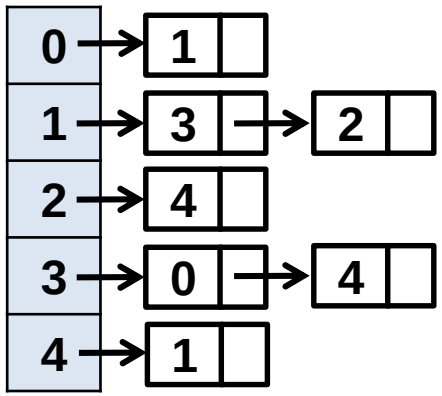
1
2
0
0
0

fila 1

Remove vértice 0 da fila. Insere na fila o vértice adjacente (1) que não foi visitado.

Busca em largura | Passo a passo





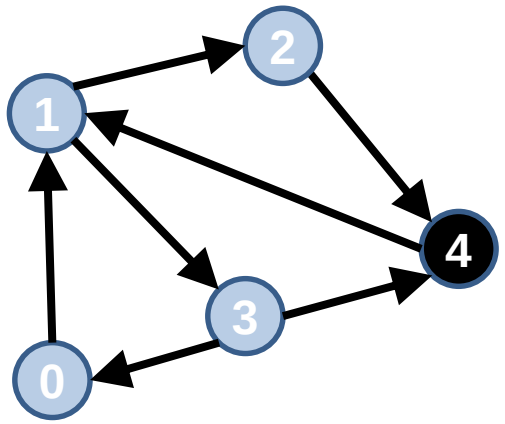
visitado

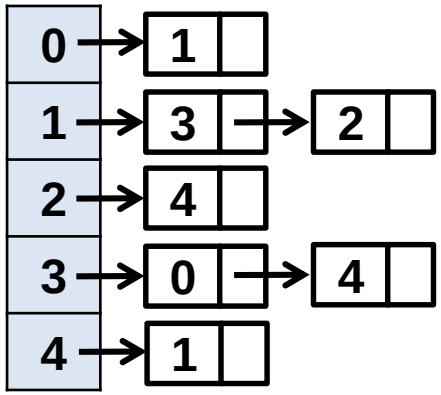
1
2
3
3
0

fila | 3 | 2 |

Remove vértice 1 da fila. Insere na fila os vértices adjacentes (3 e 2) que não foram visitados.

Busca em largura | Passo a passo





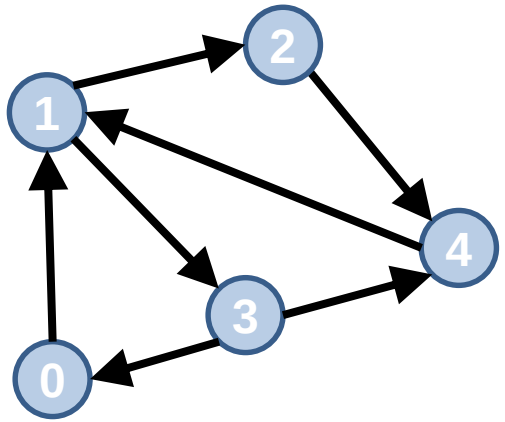
visitado

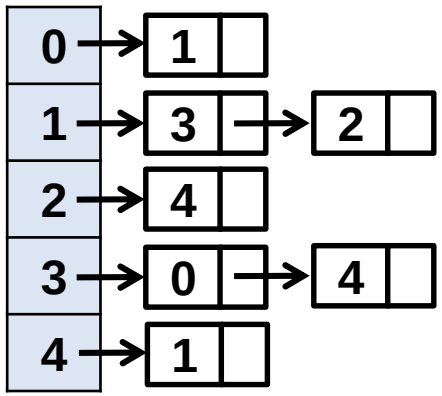
1
2
3
3
4

fila | 2 | 4 |

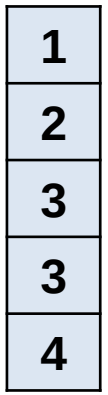
Remove vértice 3 da fila. Insere na fila o vértice adjacente (4) que não foi visitado.

Busca em largura | Passo a passo





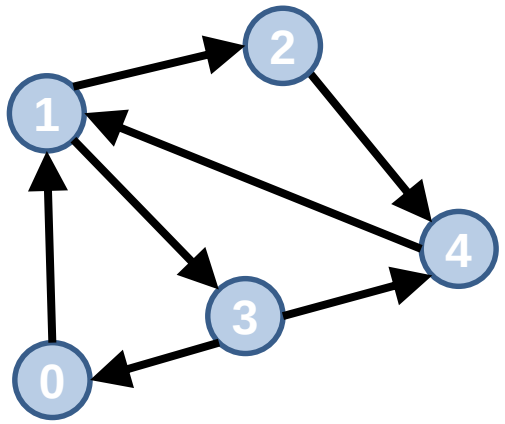
visitado

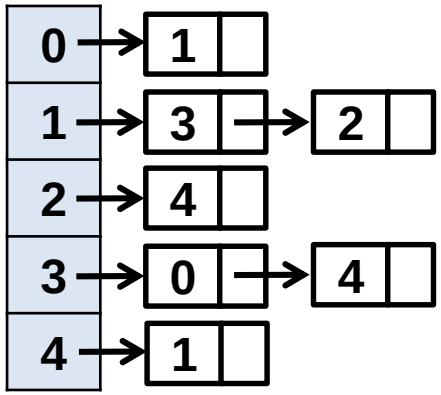


fila 4

Remove vértice 2 da fila. Vértices adjacentes já foram visitados.

Busca em largura | Passo a passo





visitado

fila

1
2
3
3
4

Remove vértice 4 da fila. Vértices adjacentes já foram visitados.

Fila vazia: fim da busca!

Busca em largura

- Complexidade
  - Considerando um grafo **G(V,A)**, onde **|V|** é o número de vértices e **|A|** é o número de arestas, a complexidade no pior caso é
    - custo de inserção e remoção em fila é constante
    - custo de enfileirar e remover todos os vértices uma vez ***O(|V|)***
    - custo de utilizar todas as arestas **|A|**
    - complexidade da busca no pior caso ***O(|V| + |A|)***

Busca em largura

- Aplicações
  - achar todos os vértices conectados a apenas um componente;
  - achar o menor caminho entre dois vértices;
  - testar se um grafo é bipartido;
  - roteamento: encontrar um número mínimo de hops em uma rede.
    - os hops são os vértices intermediários no caminho correspondente à conexão;
  - encontrar número mínimo de intermediários entre 2 pessoas.

Busca em Profundidade

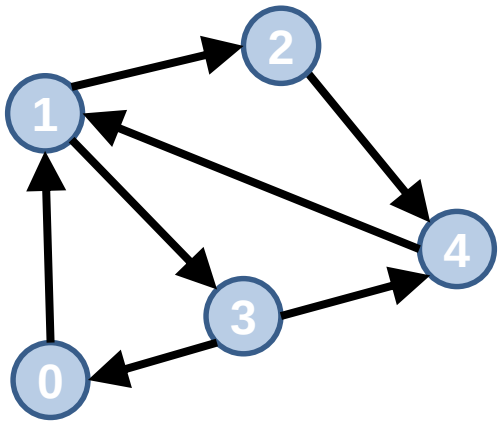
- Funcionamento
  - Partindo de um vértice inicial, a busca explora o máximo possível cada um dos vizinhos de um vértice antes de retroceder (*backtracking*)
  - Em outras palavras, esse tipo de busca se inicia em um vértice e se aprofunda nos vértices vizinhos deste até encontrar um dos dois casos:
    - o alvo da busca
    - um vértice sem vizinhos que possam ser visitados

Busca em Profundidade

- *Backtracking*
  - O grafo é percorrido de maneira sistemática até que a busca falhe, ou se encontre um vértice sem vizinhos
    - Nesse momento entra em funcionamento o mecanismo de *backtracking*: a busca retorna pelo mesmo caminho percorrido com o objetivo de encontrar um caminho alternativo.
  - Trata-se de um mecanismo usado em linguagens de programação como Prolog.

Busca em Profundidade | Grafo para teste

```c
#include <stdio.h>
#include <stdlib.h>
#include "Grafo.h"
int main(){
    int eh_digrafo = 1;
    Grafo* gr = cria_Grafo(5, 5, 0);
    insereAresta(gr, 0, 1, eh_digrafo, 0);
    insereAresta(gr, 1, 3, eh_digrafo, 0);
    insereAresta(gr, 1, 2, eh_digrafo, 0);
    insereAresta(gr, 2, 4, eh_digrafo, 0);
    insereAresta(gr, 3, 0, eh_digrafo, 0);
    insereAresta(gr, 3, 4, eh_digrafo, 0);
    insereAresta(gr, 4, 1, eh_digrafo, 0);
    int vis[5];

    buscaProfundidade_Grafo(gr, 0, vis);

    libera_Grafo(gr);

    system("pause");
    return 0;
}
```



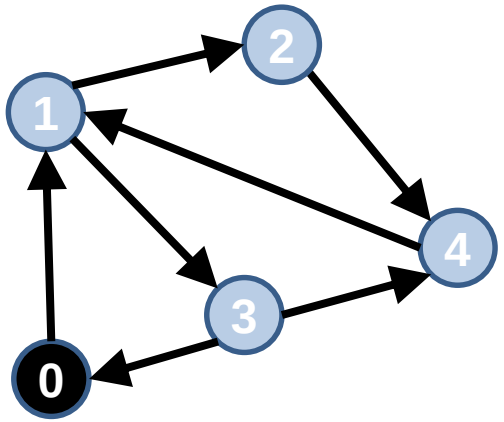
Busca em Profundidade | Implementação

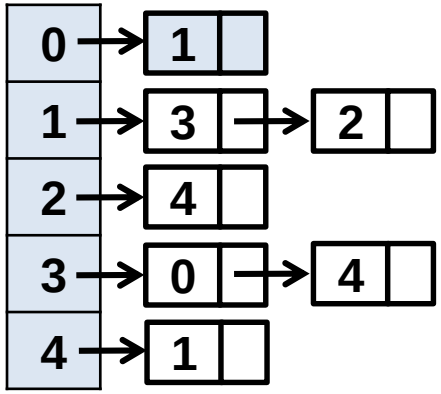
```c
//Função auxiliar: realiza o cálculo
void buscaProfundidade(Grafo *gr, int ini,
                       int *visitado, int cont){
    int i;
    visitado[ini] = cont;
    for(i=0; i<gr->grau[ini]; i++){
        if(!visitado[gr->arestas[ini][i]])
            buscaProfundidade(gr,gr->arestas[ini][i],
                              visitado,cont+1);
    }
}
//Função principal: faz a interface com o usuário
void buscaProfundidade_Grafo(Grafo *gr, int ini,
                             int *visitado){
    int i, cont = 1;
    for(i=0; i<gr->nro_vertices; i++)
        visitado[i] = 0;
    buscaProfundidade(gr,ini,visitado,cont);
}
```

Marca o vértice como visitado. Visita os vizinhos ainda não visitados

Marca vértices como NÃO visitados

Busca em profundidade | Passo a passo





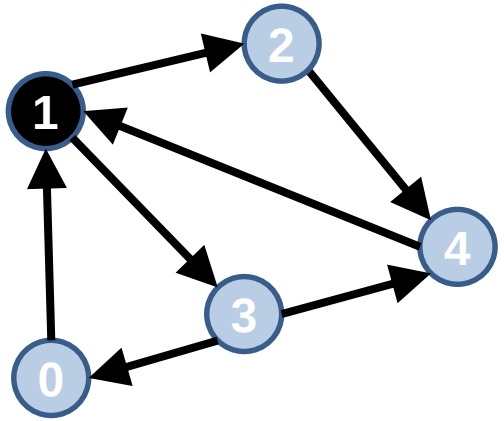
visitado

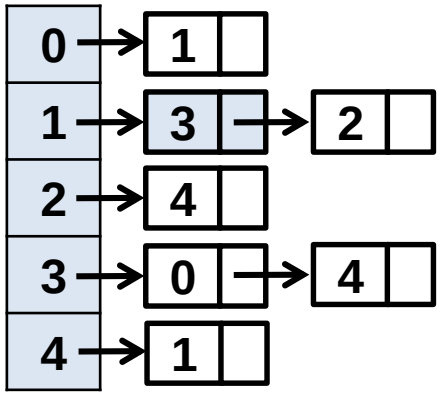
1
0
0
0
0

cont = 1

Inicia a busca com o vértice 0.
Marca o vértice 0 como visitado e executa a busca para o vértice adjacente (1)

Busca em profundidade | Passo a passo





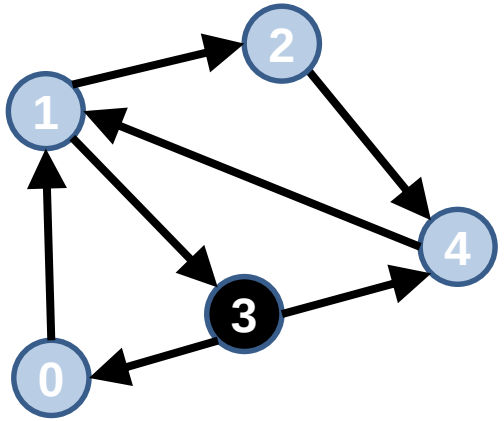
visitado

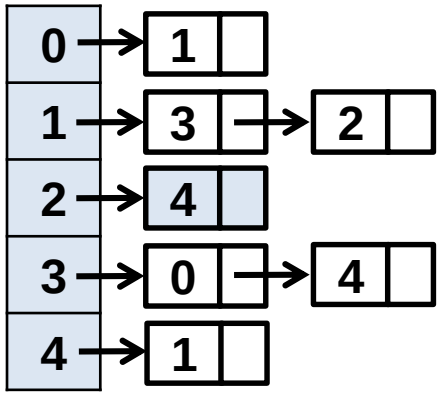
1
2
0
0
0

cont = 2

Marca o vértice 1 como visitado e executa a busca para o primeiro vértice adjacente (3)

Busca em profundidade | Passo a passo





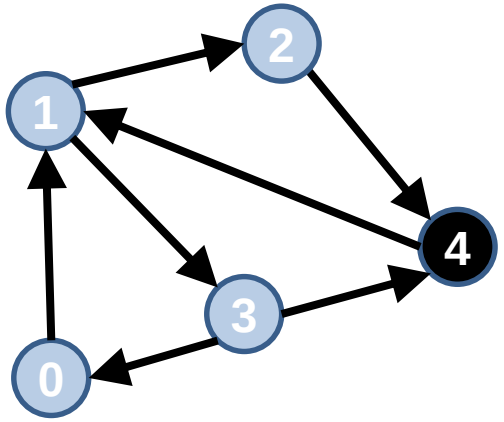
visitado

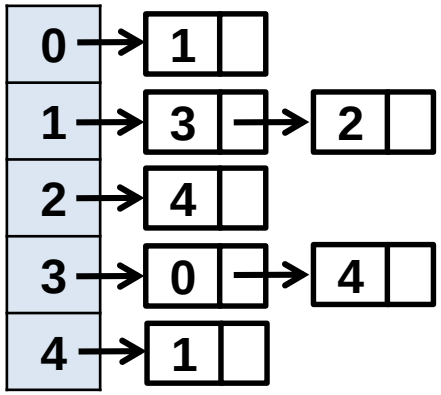
1
2
0
3
0

cont = 3

Marca o vértice 3 como visitado e executa a busca para o primeiro vértice adjacente não visitado (4)

Busca em profundidade | Passo a passo





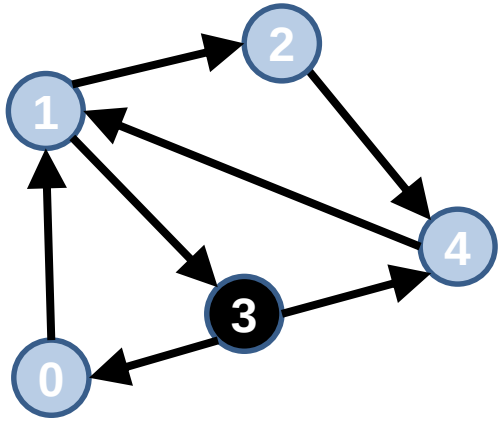
visitado

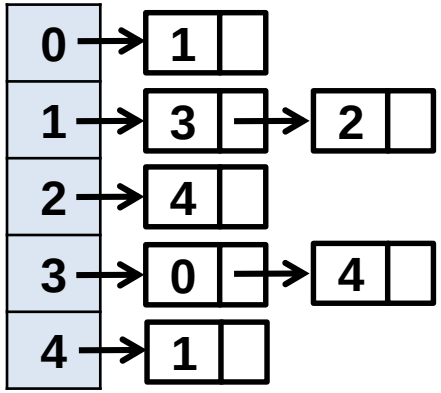
1
2
0
3
4

cont = 4

Marca o vértice 4 como visitado. Todos os vértice adjacentes já foram visitados. Volta para o vértice 3

Busca em profundidade | Passo a passo





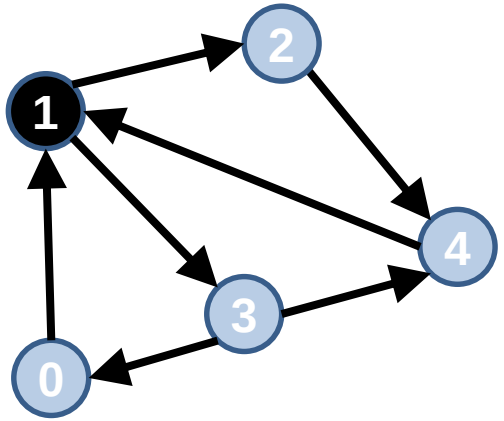
visitado

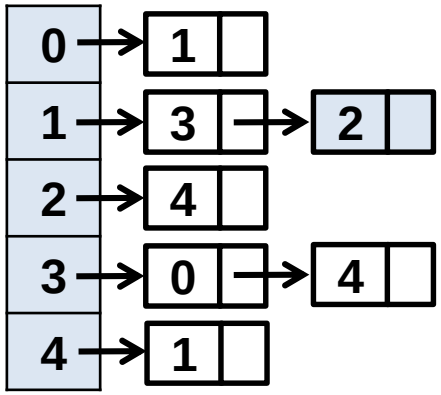
1
2
0
3
4

cont = 3

Todos os vértice adjacentes ao vértice 3 já foram visitados.
Volta para o vértice 1

Busca em profundidade | Passo a passo





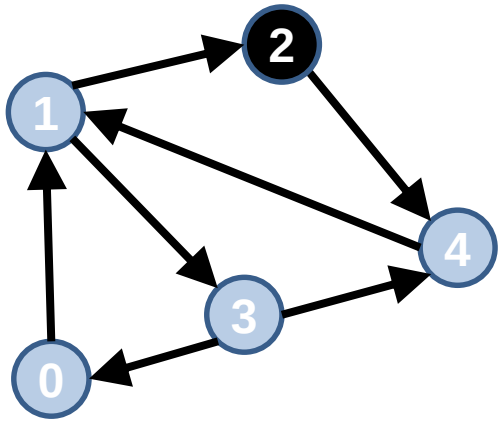
visitado

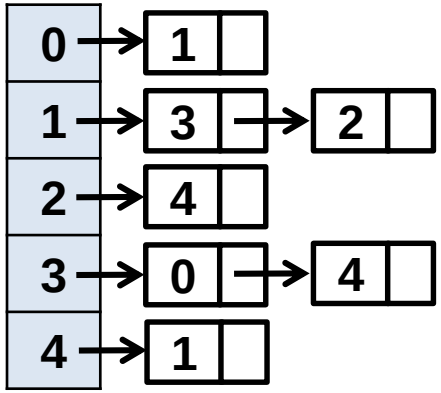
1
2
0
3
4

cont = 2

Executa a busca para o segundo vértice adjacente (2)

Busca em profundidade | Passo a passo





visitado

1
2
3
3
4

cont = 3

Marca o vértice 2 como visitado.
A partir desse ponto o algoritmo apenas volta na recursão (todos os vértices já foram visitados) e finaliza a busca

Busca em Profundidade

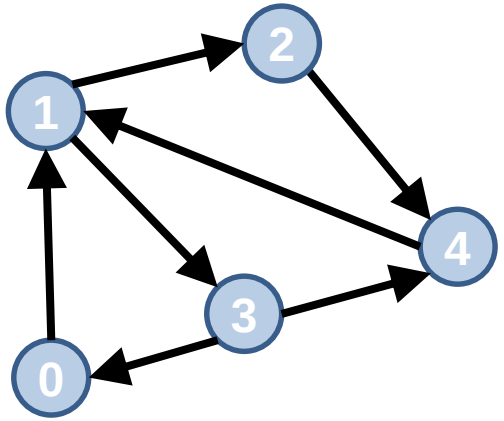
- Complexidade
  - Considerando um grafo **G(V,A)**, onde **|V|** é o número de vértices e **|A|** é o número de arestas, a complexidade no pior caso é
    - custo de ir para cada vértice é proporcional a **|V|**
    - custo de transitar em cada aresta é proporcional **|A|**
    - complexidade da busca no pior caso ***O(|V| + |A|)***

Busca em Profundidade

- Aplicações
  - encontrar componentes conectados e fortemente conectados;
  - ordenação topológica de um grafo;
  - procurar a saída de um labirinto;
  - verificar se um grafo é completamente conexo
    - por exemplo, a rede de computadores esta funcionando direito ou não;
  - implementar a ferramenta de preenchimento
    - balde de pintura do Photoshop

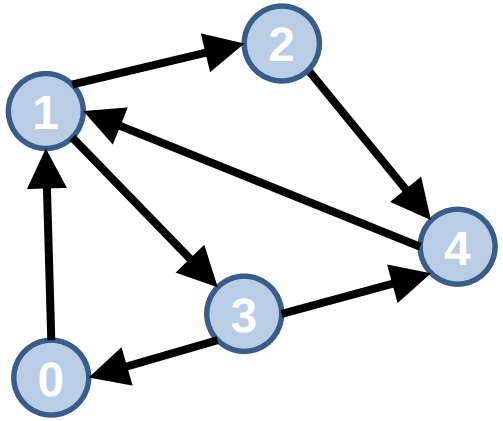
Busca pelo menor caminho

- O menor caminho entre dois vértices é a aresta que os conecta.
- No entanto, é muito comum não existir uma aresta conectando dois vértices
  - Os vértices 0 e 4 não são adjacentes



Busca pelo menor caminho

- Caminho
  - Dois vértices que não são adjacentes podem ser conectados por uma sequência de arestas
    - (0,1),(1,2),(2,4)
- Menor caminho
  - Menor sequência de arestas que liga os dois vértices



Busca pelo menor caminho

- Menor caminho
  - Caminho mais curto ou caminho geodésico
  - Caminho que apresenta o menor comprimento dentre todos os possíveis caminhos que conectam esses vértices
    - O comprimento pode ser o número de arestas que conectam os dois vértices ou a soma dos pesos das arestas que compõem esse caminho (grafo ponderado)

Busca pelo menor caminho

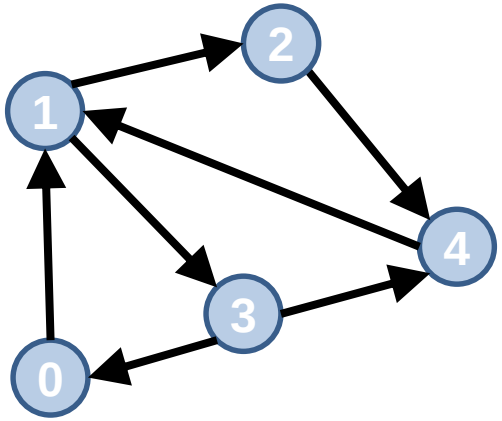
- Uma das maneiras de achar o menor caminho é utilizando o algoritmo de Dijkstra
  - Talvez o mais conhecido algoritmo
  - Trabalha com grafos e digrafos, ponderados ou não.
    - No caso de um grafo ponderado, as arestas não podem ter pesos negativos.

Busca pelo menor caminho

- Funcionamento
  - Partindo de um vértice inicial, o algoritmo de Dijkstra calcula a menor distância deste vértice a todos os demais (desde que exista um caminho entre eles)

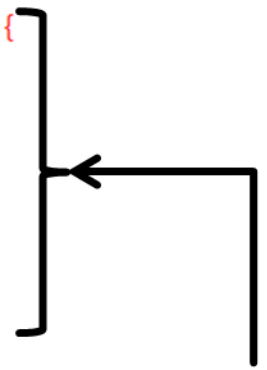
Busca pelo menor caminho | Grafo para teste

```c
#include <stdio.h>
#include <stdlib.h>
#include "Grafo.h"
int main(){
    int eh_digrafo = 1;
    Grafo* gr = cria_Grafo(5, 5, 0);
    insereAresta(gr, 0, 1, eh_digrafo, 0);
    insereAresta(gr, 1, 3, eh_digrafo, 0);
    insereAresta(gr, 1, 2, eh_digrafo, 0);
    insereAresta(gr, 2, 4, eh_digrafo, 0);
    insereAresta(gr, 3, 0, eh_digrafo, 0);
    insereAresta(gr, 3, 4, eh_digrafo, 0);
    insereAresta(gr, 4, 1, eh_digrafo, 0);
    int ant[5];
    float dist[5];
    menorCaminho_Grafo(gr, 0, ant, dist);

    libera_Grafo(gr);

    system("pause");
    return 0;
}
```



Busca pelo menor caminho | Implementação

```
1  int procuraMenorDistancia(float *dist, int *visitado,
2                            int NV){
3      int i, menor = -1, primeiro = 1;
4      for(i=0; i < NV; i++){
5          if(dist[i] >= 0 && visitado[i] == 0){
6              if(primeiro){
7                  menor = i;
8                  primeiro = 0;
9              }else{
10                 if(dist[menor] > dist[i])
11                     menor = i;
12             }
13         }
14     }
15     return menor;
16 }
```



Procura vértice com menor distância e que não tenha sido visitado

Busca pelo menor caminho | Implementação

Cria vetor auxiliar. Inicializa distâncias e anteriores

Procura vértice com menor distância e marca como visitado

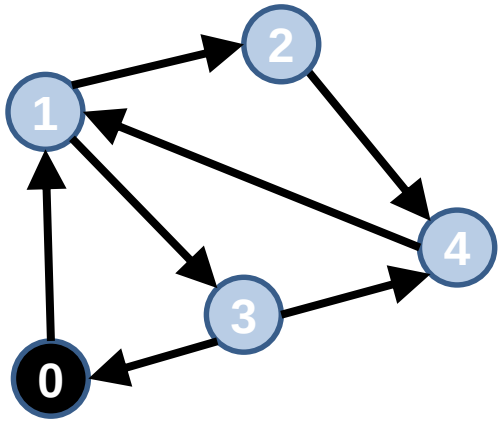
```
void menorCaminho_Grafo(Grafo *gr, int ini,
                        int *ant, float *dist){
    int i, cont, NV, ind, *visitado, u;
    cont = NV = gr->nro_vertices;
    visitado = (int*) malloc(NV * sizeof(int));
    for(i=0; i < NV; i++){
        ant[i] = -1;
        dist[i] = -1;
        visitado[i] = 0;
    }
    dist[ini] = 0;
    while(cont > 0){
        u = procuraMenorDistancia(dist, visitado, NV);
        if(u == -1)
            break;

        visitado[u] = 1;
        cont--;
        //CONTINUA...
    }
    free(visitado);
}
```

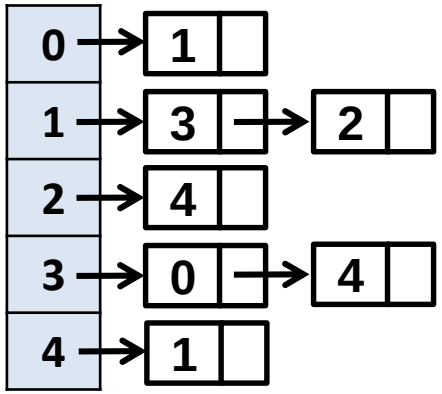
Busca pelo menor caminho | Implementação

Atualizar distâncias dos vizinhos

```
for(i=0; i<gr->grau[u]; i++){ Para cada vértice vizinho
    ind = gr->arestas[u][i];
    if(dist[ind] < 0){
        dist[ind] = dist[u] + 1;
        //ou peso da aresta
        //dist[ind] = dist[u] + gr->pesos[u][i];
        ant[ind] = u;
    }else{
        if(dist[ind] > dist[u] + 1){
        //if(dist[ind] > dist[u] + gr->pesos[u][i])
            dist[ind] = dist[u] + 1;
            //ou peso da aresta
            //dist[ind] = dist[u] + gr->pesos[u][i];
            ant[ind] = u;
        }
    }
}
```

Busca pelo menor caminho | Passo a passo

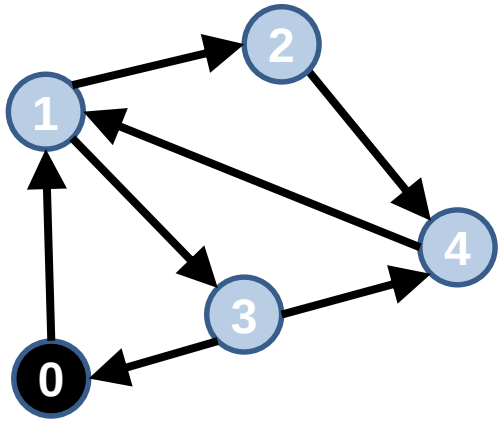


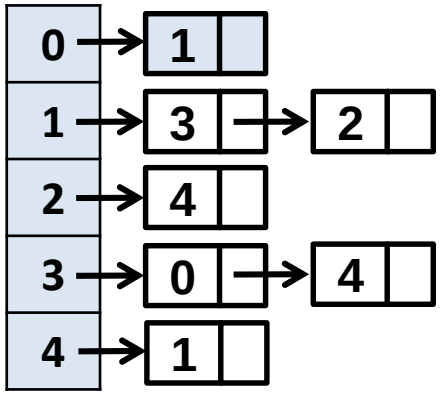


dist	ant	visitado
0	-1	0
-1	-1	0
-1	-1	0
-1	-1	0
-1	-1	0

Inicia o cálculo com o vértice 0. Atribui distância ZERO a ele (início). O restante dos vértice recebem distância -1

Busca pelo menor caminho | Passo a passo

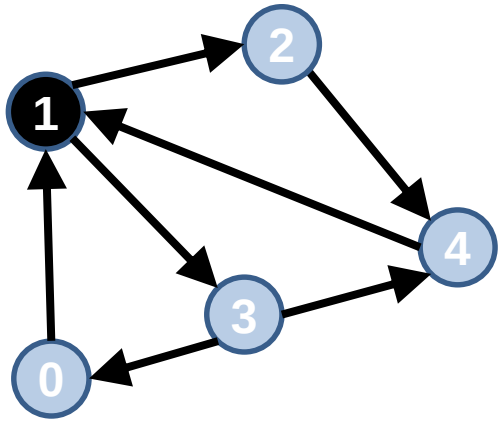


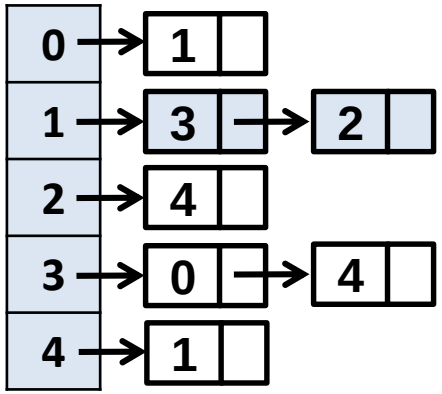


dist	ant	visitado
0	-1	1
1	0	0
-1	-1	0
-1	-1	0
-1	-1	0

Recupera vértice com menor distância ainda não visitado e o marca como visitado: vértice 0. Verifica e atualiza (se necessário) dist e ant do vértice adjacente (1)

Busca pelo menor caminho | Passo a passo

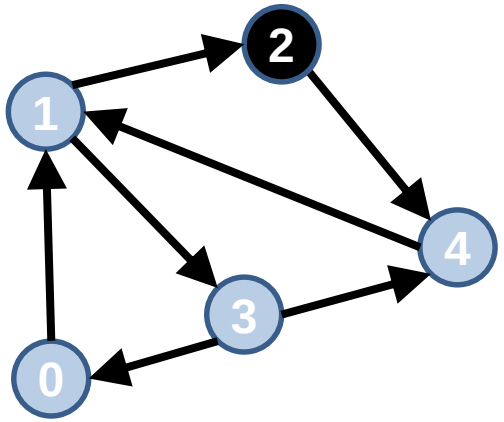


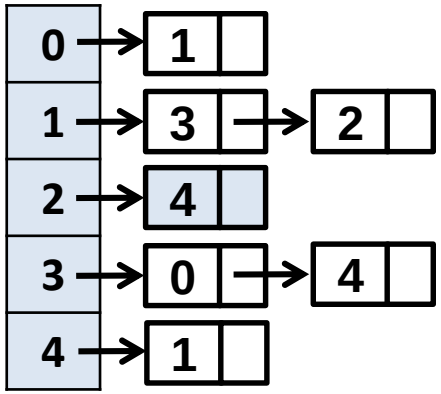


dist	ant	visitado
0	-1	1
1	0	1
2	1	0
2	1	0
-1	-1	0

Recupera vértice com menor distância ainda não visitado e o marca como visitado: vértice 1. Verifica e atualiza (se necessário) dist e ant dos vértices adjacentes (2 e 3)

Busca pelo menor caminho | Passo a passo

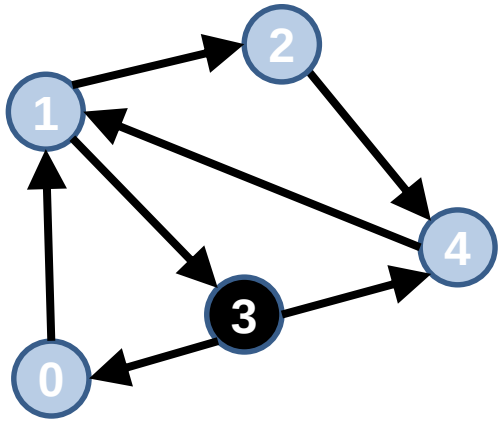


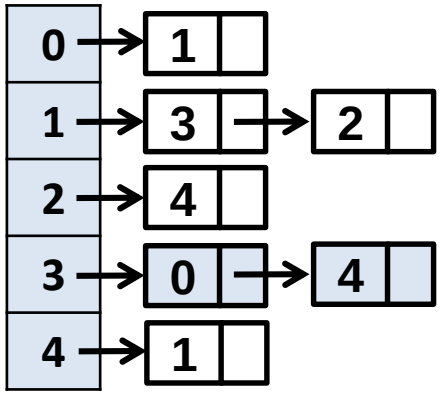


dist	ant	visitado
0	-1	1
1	0	1
2	1	1
2	1	0
3	2	0

Recupera vértice com menor distância ainda não visitado e o marca como visitado: vértice 2. Verifica e atualiza (se necessário) dist e ant do vértice adjacente (4)

Busca pelo menor caminho | Passo a passo

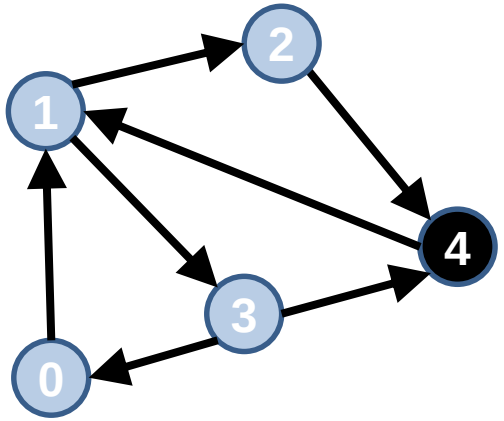


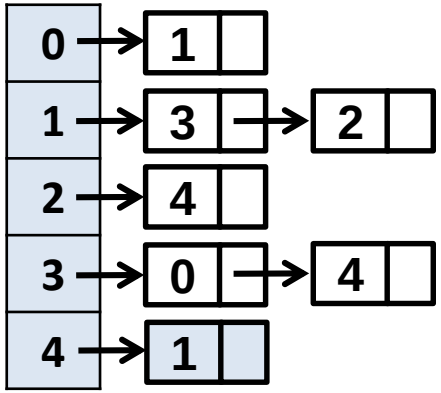


dist	ant	visitado
0	-1	1
1	0	1
2	1	1
2	1	1
3	2	0

Recupera vértice com menor distância ainda não visitado e o marca como visitado: vértice 3. Verifica e atualiza (se necessário) dist e ant dos vértices adjacentes (0 e 4)

Busca pelo menor caminho | Passo a passo





dist	ant	visitado
0	-1	1
1	0	1
2	1	1
2	1	1
3	2	1

Recupera vértice com menor distância ainda não visitado e o marca como visitado: vértice 4. Verifica e atualiza (se necessário) dist e ant do vértice adjacente (1)

Todos os vértices já foram visitados. Cálculo do menor caminho chegou ao fim.

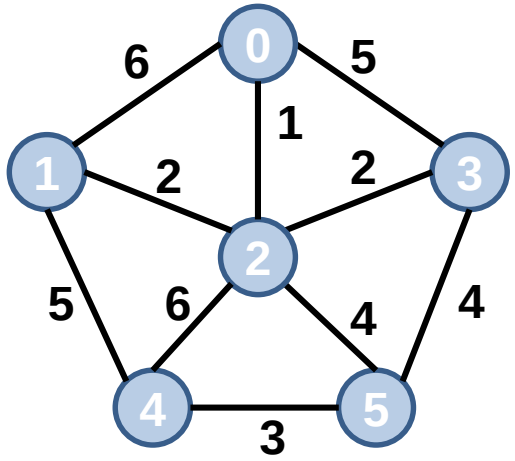
Busca pelo menor caminho

- Aplicações
  - para achar o grau de separação entre duas pessoas em uma rede social;
  - para achar um trajeto em um mapa rodoviário;
  - para programar robôs explorar áreas;
  - em algoritmos de roteamento.

ÁRVORE GERADORA MÍNIMA

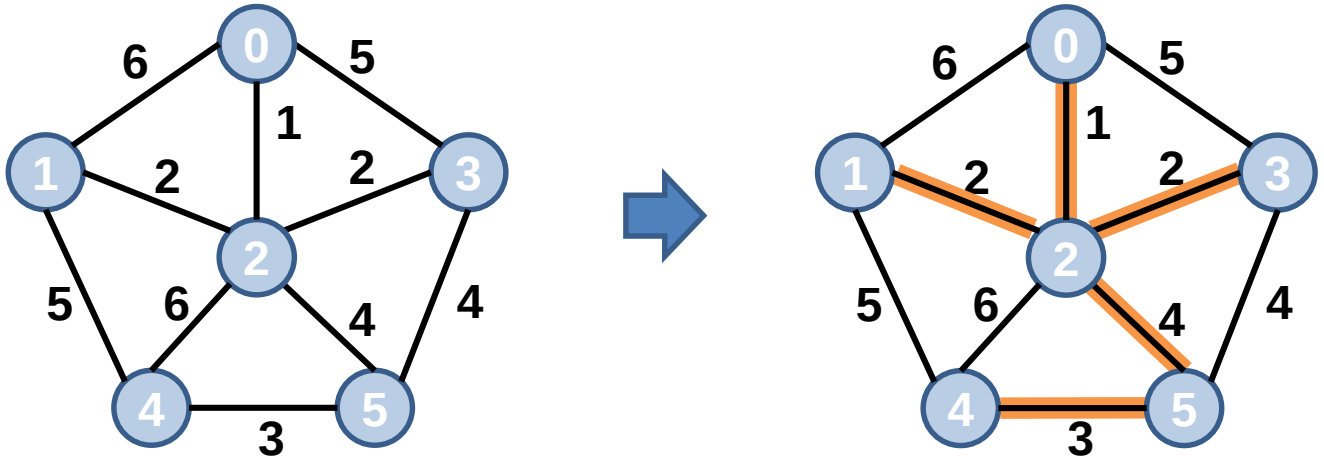
Árvore Geradora Mínima

- Uma árvore geradora (do inglês, *spanning tree*) é um subgrafo que contenha todos os vértices do grafo original e um conjunto de arestas que permita conectar todos esses vértices na forma de uma árvore.
- É a menor estrutura que conecta todos os vértices



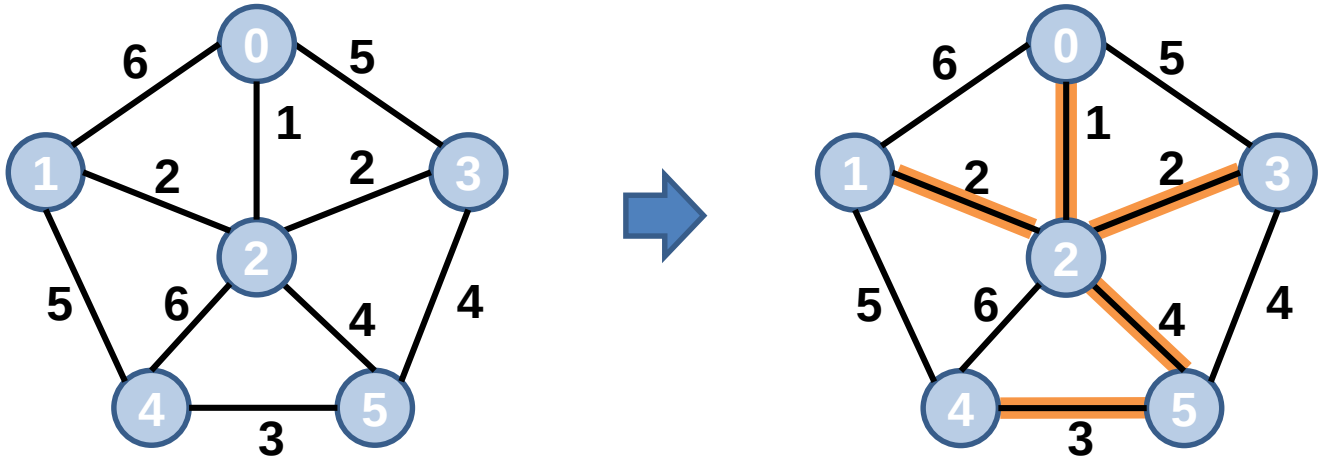
Árvore Geradora Mínima

- Dado um grafo ***G(V,A)***, a árvore geradora possui
  - todos os vértices ***V***
  - um total de arestas igual a **|V|-1** *(*o número de vértices menos um)



Árvore Geradora Mínima

- Se o grafo é ponderado (arestas com peso), podemos querer encontrar a árvore geradora mínima
  - Do inglês, *minimum spanning tree*
  - Procura o conjunto de arestas de menor custo



Árvore Geradora Mínima

- Condição para existir uma árvore geradora mínima
  - Para quaisquer dois vértices distintos, sempre deve existir um caminho que os une
  - Como todos os vértices estão conectados, calcular a árvore geradora não depende do vértice inicial
- Portanto, o grafo deve ser
  - Não-direcionado
  - Conexo
  - Ponderado

Árvore Geradora Mínima

- Aplicações
  - transporte aéreo: mapa de conexões de vôo
  - transporte terrestre: infra-estrutura das rodovias com o menor uso de material;
  - redes de computadores: conectar uma série de computadores com a menor quantidade de fibra ótica possível
  - redes elétricas e telefônicas: unir um conjunto de localidades com menor gasto
  - circuitos integrados
  - análise de clusters
  - armazenamento de informações

Árvore Geradora Mínima

- O problema pode ser resolvido usando uma estratégia gulosa que constrói a árvore incrementalmente
- Existem dois algoritmos clássicos para obter soluções ótimas
  - Algoritmo de Prim
  - Algoritmo de Kruskal
- A diferença entre eles está na regra usada para encontrar a aresta que fará parte da árvore

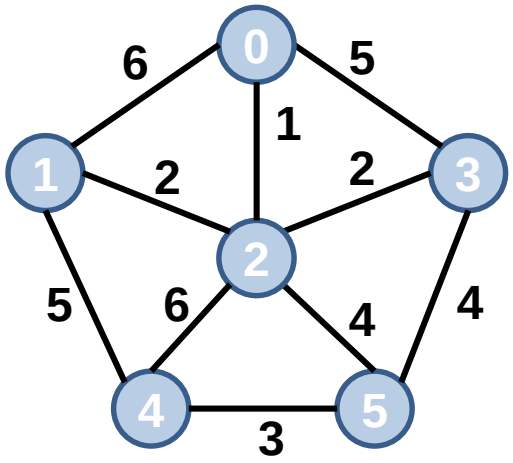
ALGORITMO DE PRIM

Algoritmo de Prim

- Funcionamento
  - Considera um vértice inicialmente na árvore
  - A cada iteração, o algoritmo procura a aresta de **menor peso** que conecte um vértice da árvore a outro que ainda não esteja na árvore.
  - Esse vértice é adicionado a árvore e o processo se repete.
  - Esse processo continua até que
    - Todos os vértices façam parte da árvore
    - Não se pode encontrar uma aresta que satisfaça essa condição (grafo desconexo)

Algoritmo de Prim | Grafo para teste

```c
#include <stdio.h>
#include <stdlib.h>
#include "Grafo.h"
int main(){
    int eh_digrafo = 0;
    Grafo* gr = cria_Grafo(6, 6, 1);
    insereAresta(gr, 0, 1, eh_digrafo, 6);
    insereAresta(gr, 0, 2, eh_digrafo, 1);
    insereAresta(gr, 0, 3, eh_digrafo, 5);
    insereAresta(gr, 1, 2, eh_digrafo, 2);
    insereAresta(gr, 1, 4, eh_digrafo, 5);
    insereAresta(gr, 2, 3, eh_digrafo, 2);
    insereAresta(gr, 2, 4, eh_digrafo, 6);
    insereAresta(gr, 2, 5, eh_digrafo, 4);
    insereAresta(gr, 3, 5, eh_digrafo, 4);
    insereAresta(gr, 4, 5, eh_digrafo, 3);

    int i, pai[6];

    arvoreGeradoraMinimaPRIM_Grafo(gr, 0, pai);

    libera_Grafo(gr);

    return 0;
}
```



Algoritmo de Prim | Algoritmo

Vértices não tem pai, menos orig

Procura menor aresta ligando um vértice que está na árvore a outro fora da árvore

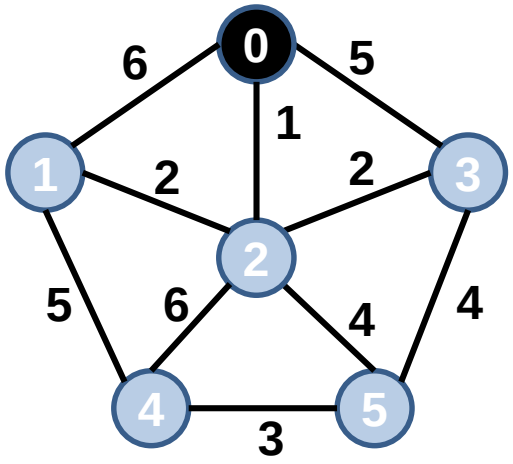
```
void algPRIM(Grafo *gr, int orig, int *pai){
    int i, j, dest, primeiro, NV = gr->nro_vertices;
    double menorPeso;
    for(i=0; i < NV; i++)
      pai[i] = -1;// sem pai
    pai[orig] = orig;
    while(1){
        primeiro = 1;
        //percorre todos os vértices
        for(i=0; i < NV; i++){
            //achou vértices já visitado
            if(pai[i] != -1){
            // percorre os vizinhos do vértice visitado
                for(j=0; j<gr->grau[i]; j++){
                    //procurar menor peso: continua
                }
            }
        }
        if(primeiro == 1)
            break;

        pai[dest] = orig;
}
```

Algoritmo de Prim | Algoritmo

```
1  //continuação
2  //achou vértice vizinho não visitado
3  if(pai[gr->arestas[i][j]] == -1){
4      if(primeiro){//procura aresta de menor custo
5          menorPeso = gr->pesos[i][j];
6          orig = i;
7          dest = gr->arestas[i][j];
8          primeiro = 0;
9      }else{
10         if(menorPeso > gr->pesos[i][j]){
11             menorPeso = gr->pesos[i][j];
12             orig = i;
13             dest = gr->arestas[i][j];
14         }
15     }
16 }
```

Algoritmo de Prim | Passo a passo

Passo 1

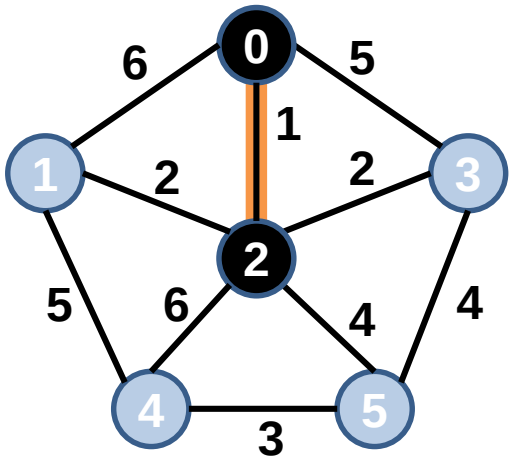


pai
0
-1
-1
-1
-1
-1

Inicia o cálculo com o vértice 0. Atribui seu próprio índice como pai. O restante dos vértices recebem pai igual a -1 (sem pai).

Algoritmo de Prim | Passo a passo

Passo 2

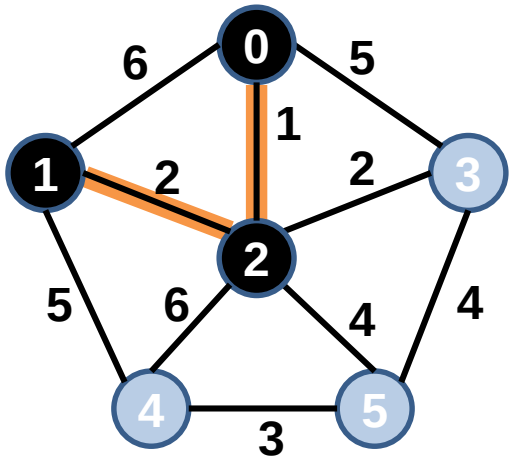


pai
0
-1
0
-1
-1
-1

Procura nos vértices com pai por um vértice sem pai e com menor peso: vértice 2.
Atribui vértice 0 como pai do vértice 2.

Algoritmo de Prim | Passo a passo

Passo 3

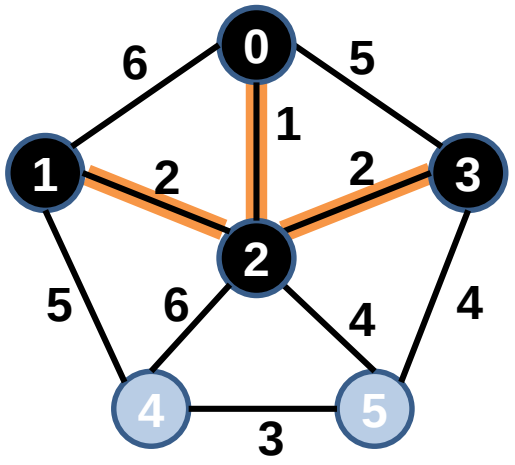


pai
0
2
0
-1
-1
-1

Procura nos vértices com pai por um vértice sem pai e com menor peso: vértice 1.
Atribui vértice 2 como pai do vértice 1.

Algoritmo de Prim | Passo a passo

Passo 4

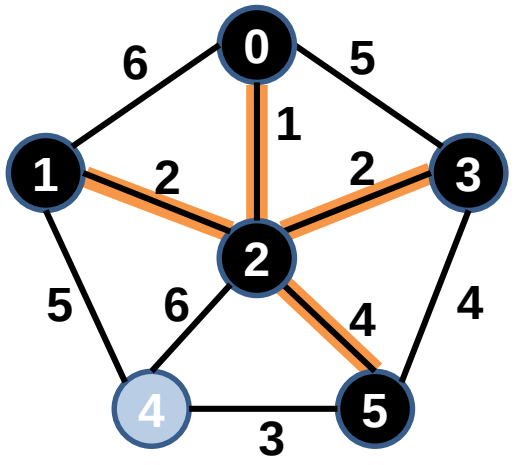


pai
0
2
0
2
-1
-1

Procura nos vértices com pai por um vértice sem pai e com menor peso: vértice 3.
Atribui vértice 2 como pai do vértice 3.

Algoritmo de Prim | Passo a passo

Passo 5

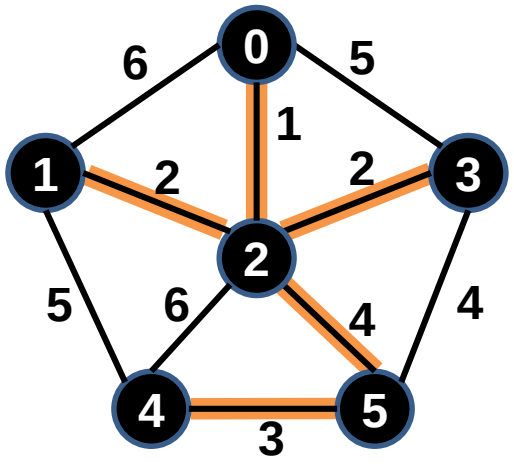


pai
0
2
0
2
-1
2

Procura nos vértices com pai por um vértice sem pai e com menor peso: vértice 5.
Atribui vértice 2 como pai do vértice 5.

Algoritmo de Prim | Passo a passo

Passo 6



pai
0
2
0
2
5
2

Procura nos vértices com pai por um vértice sem pai e com menor peso : vértice 4.
Atribui vértice 5 como pai do vértice 4.

Fim do cálculo.

Algoritmo de Prim | Complexidade

- Considerando um grafo **G(V,A)**, onde **|V|** é o número de vértices e **|A|** é o número de arestas, a complexidade no pior caso é ***O(|V|*|A|).*** Como **|A|** é proporcional a **$|V|^2$,** seu custo é ***$O(|V|^3)$***

- A eficiência depende da forma usada para procurar a aresta de menor peso. Usando uma fila de prioridade o custo pode ser reduzido para ***O(|A|log|V|)***

ALGORITMO DE KRUSKAL

Algoritmo de Kruskal

- O algoritmo de Prim se inicia com um vértice e cresce uma única árvore a partir dele

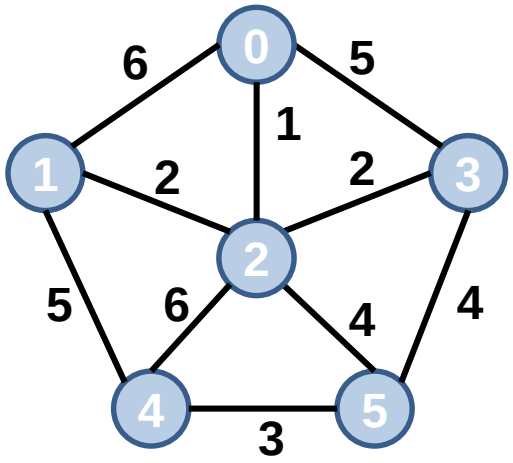
- O algoritmo de Kruskal constrói uma floresta (várias árvores) ao longo do tempo, e que são unidas ao final do processo

Algoritmo de Kruskal

- Funcionamento
  - Considera cada vértice como uma árvore independente (floresta)
  - A cada iteração, o algoritmo procura a aresta de **menor peso** que conecta duas árvores diferentes
  - Os vértices das árvores selecionadas passam a fazer parte de uma mesma árvore
  - Esse processo continua até que
    - Todos os vértices façam parte da árvore
    - Não se pode encontrar uma aresta que satisfaça essa condição (grafo desconexo)

Algoritmo de Kruskal | Grafo para teste

```c
#include <stdio.h>
#include <stdlib.h>
#include "Grafo.h"
int main(){
    int eh_digrafo = 0;
    Grafo* gr = cria_Grafo(6, 6, 1);
    insereAresta(gr, 0, 1, eh_digrafo, 6);
    insereAresta(gr, 0, 2, eh_digrafo, 1);
    insereAresta(gr, 0, 3, eh_digrafo, 5);
    insereAresta(gr, 1, 2, eh_digrafo, 2);
    insereAresta(gr, 1, 4, eh_digrafo, 5);
    insereAresta(gr, 2, 3, eh_digrafo, 2);
    insereAresta(gr, 2, 4, eh_digrafo, 6);
    insereAresta(gr, 2, 5, eh_digrafo, 4);
    insereAresta(gr, 3, 5, eh_digrafo, 4);
    insereAresta(gr, 4, 5, eh_digrafo, 3);

    int i, pai[6];

    arvoreGeradoraMinimaKruskal_Grafo(gr, 0, pai);

    libera_Grafo(gr);

    return 0;
}
```



Algoritmo de Kruskal | Algoritmo

Cada vértice é uma árvore, sem pai

Procura menor aresta ligando árvores diferentes

Une as duas árvores da aresta selecionada

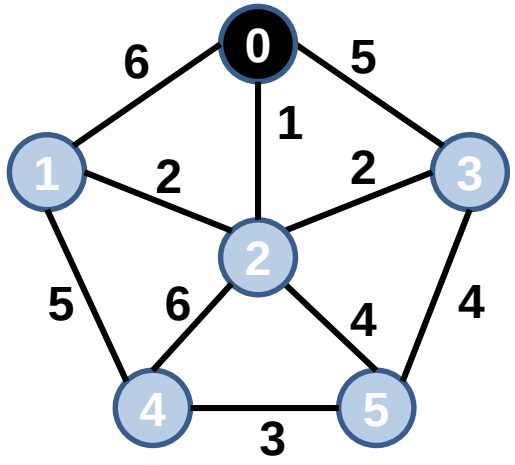
```
1  void algKruskal(Grafo *gr, int orig, int *pai){
2      int i,j,dest,primeiro,NV = gr->nro_vertices;
3      double menorPeso;
4      int *arv = (int*) malloc(NV * sizeof(int));
5      for(i=0; i < NV; i++){
6          arv[i] = i;
7          pai[i] = -1;// sem pai
8      }
9      pai[orig] = orig;
10     while(1){
11         primeiro = 1;
12         for(i=0; i < NV; i++){//percorre os vértices
13             for(j=0; j<gr->grau[i]; j++){ //arestas
14                 //procura vértice menor peso: continua
15             }
16         }
17         if(primeiro == 1) break;
18         if(pai[orig] == -1) pai[orig] = dest;
19         else pai[dest] = orig;
20
21         for(i=0; i < NV; i++)
22             if(arv[i] == arv[dest])
23                 arv[i] = arv[orig];
24     }
25     free(arv);
26 }
```

Algoritmo de Kruskal | Algoritmo

```
1  //continuação
2  //procura aresta de menor custo
3  if(arv[i] != arv[gr->arestas[i][j]]){
4      if(primeiro){
5          menorPeso = gr->pesos[i][j];
6          orig = i;
7          dest = gr->arestas[i][j];
8          primeiro = 0;
9      }else{
10         if(menorPeso > gr->pesos[i][j]){
11             menorPeso = gr->pesos[i][j];
12             orig = i;
13             dest = gr->arestas[i][j];
14         }
15     }
16 }
17
18
```

Algoritmo de Kruskal | Passo a passo

Passo 1

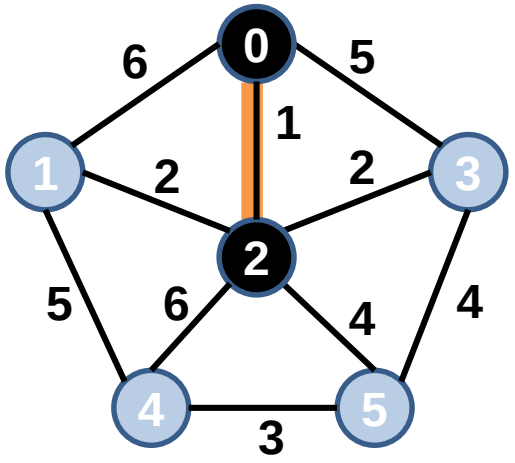


pai	arv
0	0
-1	1
-1	2
-1	3
-1	4
-1	5

Inicia o cálculo com o vértice 0. Atribui seu próprio índice como pai. O restante dos vértice recebem pai igual a -1 (sem pai). Inicializa a árvore com o índice do vértice.

Algoritmo de Kruskal | Passo a passo

Passo 2

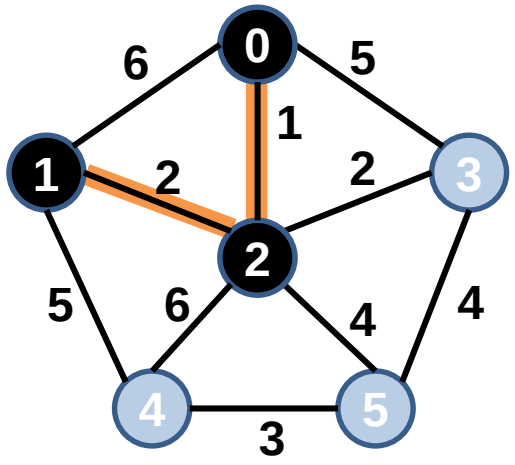


pai	arv
0	0
-1	1
0	0
-1	3
-1	4
-1	5

Procura a aresta com menor peso conectando vértices com árvores diferentes: vértices 0 e 2.
Atribui vértice 0 como pai do vértice 2.
Todos que possuem árvore igual ao vértice 2 passam a ter árvore igual ao vértice 0.

Algoritmo de Kruskal | Passo a passo

Passo 3

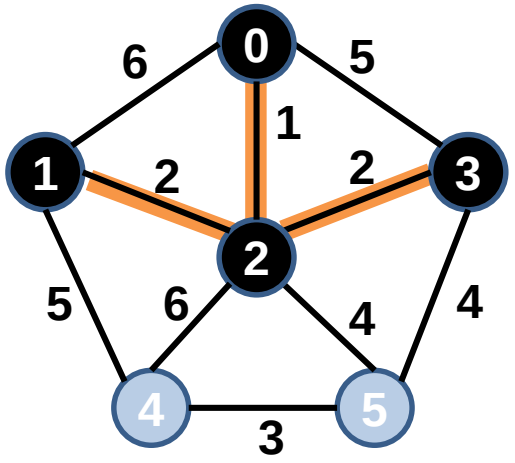


pai	arv
0	1
2	1
0	1
-1	3
-1	4
-1	5

Procura a aresta com menor peso conectando vértices com árvores diferentes: vértices 1 e 2.
Atribui vértice 2 como pai do vértice 1.
Todos que possuem árvore igual ao vértice 2 passam a ter árvore igual ao vértice 1.

Algoritmo de Kruskal | Passo a passo

Passo 4

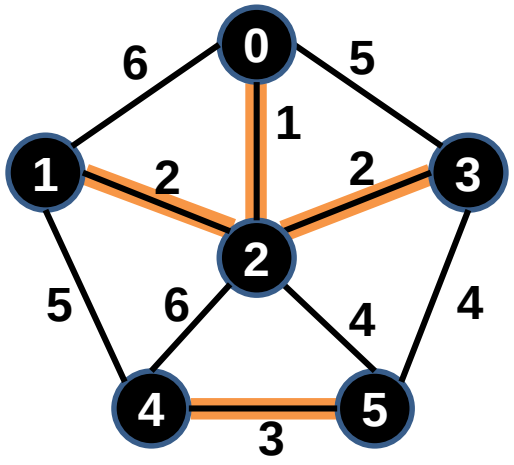


pai	arv
0	1
2	1
0	1
2	1
-1	4
-1	5

Procura a aresta com menor peso conectando vértices com árvores diferentes: vértices 2 e 3.
Atribui vértice 2 como pai do vértice 3.
Todos que possuem árvore igual ao vértice 3 passam a ter árvore igual ao vértice 2.

Algoritmo de Kruskal | Passo a passo

Passo 5

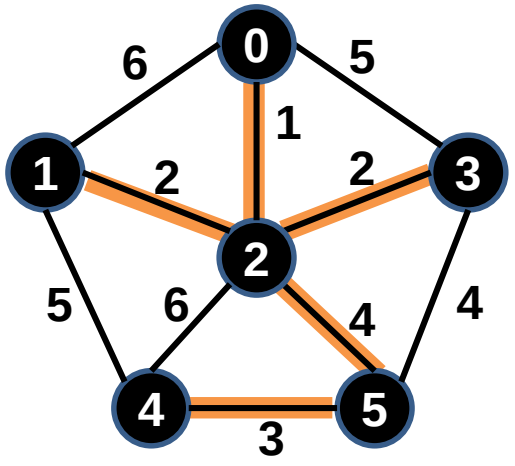


pai	arv
0	1
2	1
0	1
2	1
5	4
-1	4

Procura a aresta com menor peso conectando vértices com árvores diferentes: vértices 4 e 5.
Atribui vértice 5 como pai do vértice 4.
Todos que possuem árvore igual ao vértice 5 passam a ter árvore igual ao vértice 4.

Algoritmo de Kruskal | Passo a passo

Passo 6



pai	arv
0	1
2	1
0	1
2	1
5	1
2	1

Procura a aresta com menor peso conectando vértices com árvores diferentes: vértices 2 e 5.
Atribui vértice 2 como pai do vértice 5.
Todos que possuem árvore igual ao vértice 5 passam a ter árvore igual ao vértice 2.

Fim do cálculo

Algoritmo de Kruskal | Complexidade

- Considerando um grafo **G(V,A)**, onde **|V|** é o número de vértices e **|A|** é o número de arestas, a complexidade no pior caso é ***O(|V|*|A|).*** Como **|A|** é proporcional a **$|V|^2$,** seu custo é ***$O(|V|^3)$***

- A eficiência depende da forma usada para procurar a aresta de menor peso. Usando uma estrutura de dados **união-busca** (*Union&Find*) o custo pode ser reduzido para ***O(|A|log|V|)***

Material Complementar | Vídeo Aulas

- Aula 56: Grafos – Definição:
  - youtu.be/gJvSmrxekDo
- Aula 57: Grafos – Propriedades:
  - youtu.be/qvSbkbUkZjo
- Aula 58: Grafos – Tipos de Grafos (Parte 1):
  - youtu.be/5saF2Dg6sIc
- Aula 59: Grafos – Tipos de Grafos (Parte 2):
  - youtu.be/LsLK04bWgy4
- Aula 60: Grafos – Representação de Grafos (Parte 1):
  - youtu.be/k9DJn-COtKg
- Aula 61: Grafos – Representação de Grafos (Parte 2):
  - youtu.be/-dAxrWDufa8

Material Complementar | Vídeo Aulas

- Aula 62: Grafos – Busca em Grafos:
  - youtu.be/iN6PWvga5IQ
- Aula 63: Grafos – Busca em Profundidade:
  - youtu.be/pJ3ilnhXWCQ
- Aula 64: Grafos – Busca em Largura:
  - youtu.be/jWoP1fTTDzE
- Aula 65: Grafos – Busca pelo Menor Caminho:
  - youtu.be/5y8dch2uHR4

Material Complementar | Vídeo Aulas

- Aula 112: Grafo - Árvore Geradora Mínima:
  - https://www.youtube.com/watch?v=eHC2tjQPX3A
- Aula 113: Grafos – Algoritmo de Prim:
  - https://www.youtube.com/watch?v=bBq_Cu5doy0
- Aula 114: Grafos – Algoritmo de Kruskal:
  - https://www.youtube.com/watch?v=EzMHc5xW6Pc

Material Complementar | GitHub

- https://github.com/arbackes