Introdução à Teoria dos Grafos

Laboratório de Programação Competitiva - 2020

Pedro Henrique Paiola (paiola@fc.unesp.br)
Giulia Moura Crusco (giulia@fc.unesp.br)
João Pedro Marin Comini (joaocomini@gmail.com)
Unesp Bauru

Baseado no material utilizado em Programação Competitiva I - 2019

Grafos

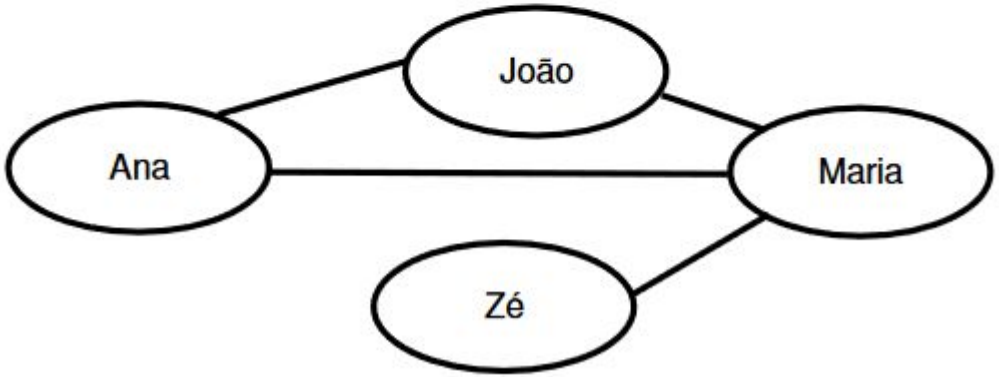
- Um grafo é uma abstração matemática que representa situações reais através de um diagrama, buscando representar a relação entre pares de elementos.
- Formalmente, um grafo G é um par (V, A) onde
    - V é um conjunto de vértices
    - A é um conjunto de pares (v, u) tal que v, u $\in$ V
- Vértice: representa um elemento em si
- Aresta: representa o relacionamento entre um par de elementos

Grafos

Exemplo de grafo: G = (V, E)

V = {Ana, João, Maria, Zé}

E = {(Ana,João), (Ana,Maria),(João,Maria),(Maria,Zé)}

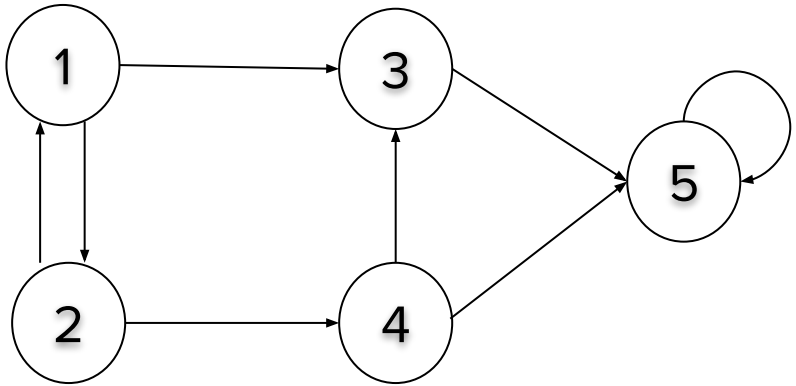


Grafo orientado

- Um grafo orientado, também chamado de grafo direcionado ou dígrafo, G é um par (V, A) onde V é um conjunto finito, não vazio, de vértices e A é uma relação binária em V, isto é, um conjunto finito de pares ordenados de vértices.
- Uma aresta (u, v) “sai” do vértice u e “entra” no vértice v, nesse caso, dizemos que o vértice v é adjacente ao vértice u.

Grafo orientado

- G = (V, A)
- V = {1, 2, 3, 4, 5}
- A = {(1, 2), (1, 3), (2, 1), (2, 4), (3, 5), (4, 3), (4, 5), (5,5)}



1 é adjacente à 2 e 2 é adjacente à 1

3 é adjacente à 1, mas 1 não é adjacente à 3

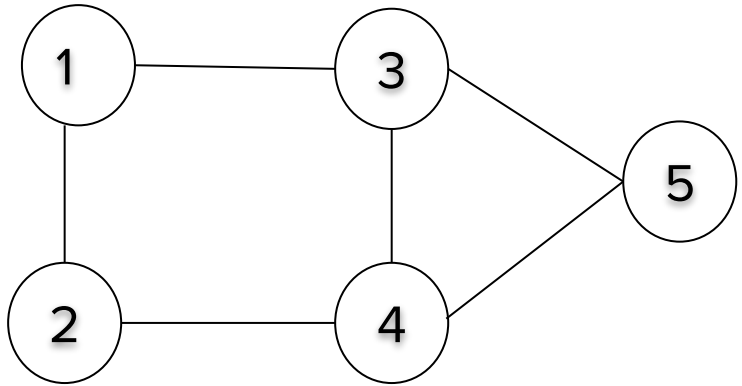
5 é adjacente a ele mesmo (laço)

Grafo não orientado

- Um grafo não orientado, ou não direcionado, G é um par (V, A) onde o conjunto de arestas A é um conjunto finito de pares não ordenados de vértices.
- Em outras palavras, nesse caso os pares (u, v) e (v, u) representam a mesma aresta.
- Se há uma aresta (u, v), então v é adjacente à u e u é adjacente à v.
- Laços não são permitidos

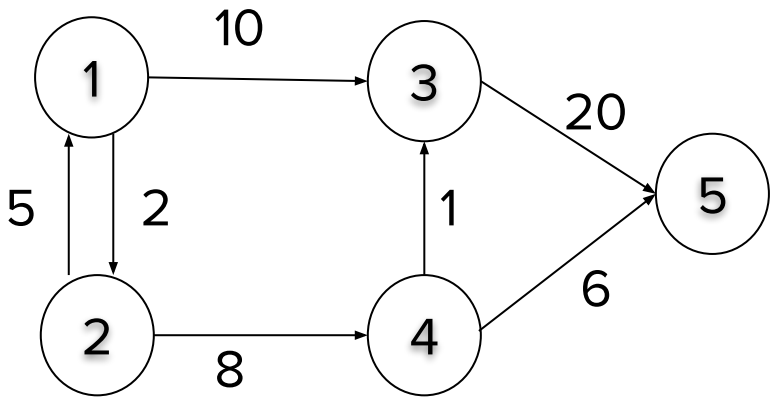
Grafo não orientado

- G = (V, A)
- V = {1, 2, 3, 4, 5}
- A = {(1, 2), (1, 3), (2, 4), (3, 4), (4, 5)}



Grafo ponderado (rotulado)

- Um grafo ponderado é um grafo (seja direcionado ou não) que possui pesos associados às arestas. Esses pesos podem representar, por exemplo, custos ou distâncias.



Grau de um vértice

- Em um grafo não direcionado:

  grau(v) = número de arestas que incidem em v

- Em um grafo direcionado:

  grau(v) = grau_entrada(v) + grau_saída(v)

  em que grau_entrada(v) = número de arestas que entram em v
  grau_saída(v) = número de arestas que saem de v

Grau de um vértice

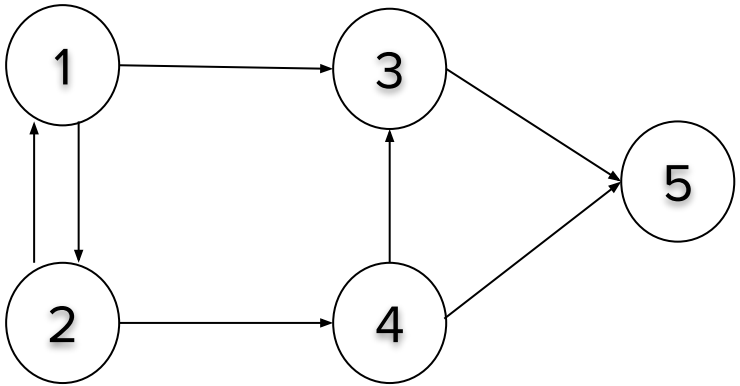
grau(1) = 1 + 2 = 3

grau(2) = 1 + 2 = 3

grau(3) = 2 + 1 = 3

grau(4) = 1 + 2 = 3

grau(5) = 2 + 0 = 2



Grau de um vértice

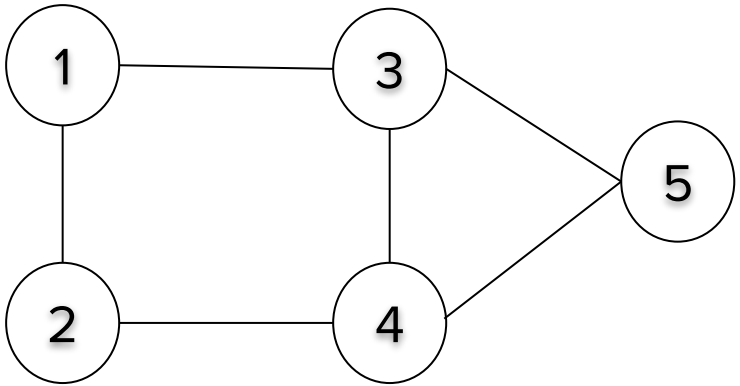
grau(1) = 2

grau(2) = 2

grau(3) = 3

grau(4) = 3

grau(5) = 2



Caminho entre vértices

- Um caminho é uma sequência de vértices conectados por arestas.
- O comprimento de um caminho é dado pela quantidade de arestas que formam este caminho

Caminho entre vértices

- Exemplos de caminhos

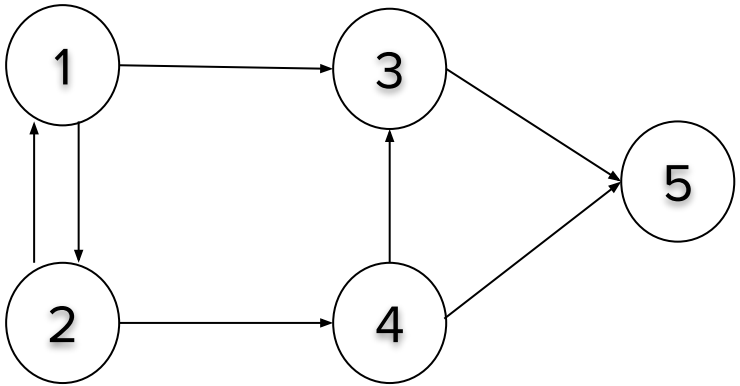
(1, 3, 5)

(2, 4, 3)

(1, 2, 4, 3, 5)

(1, 2, 4, 5)

Perceba que, de um vértice a outro, pode existir mais de um caminho possível

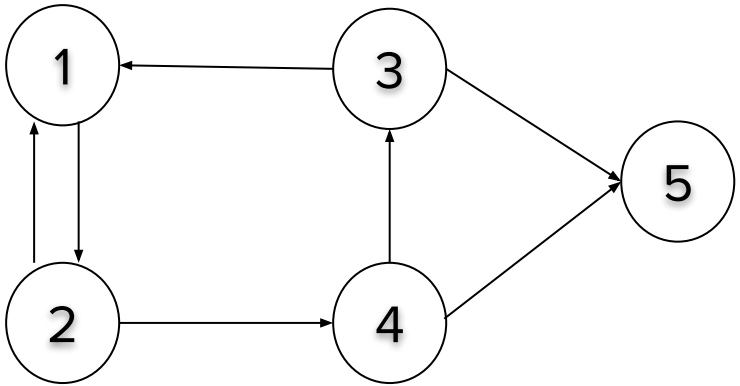


Caminho entre vértices

- Um caminho é **simples** se todos os vértices do caminho são distintos.
- Um caminho $(v_0, \ldots, v_k)$ forma um **ciclo** se $v_0 = v_k$
- Exemplo:

  (1, 2, 4, 3, 1)

- Um grafo sem ciclos é chamado **acíclico**.



Implementação

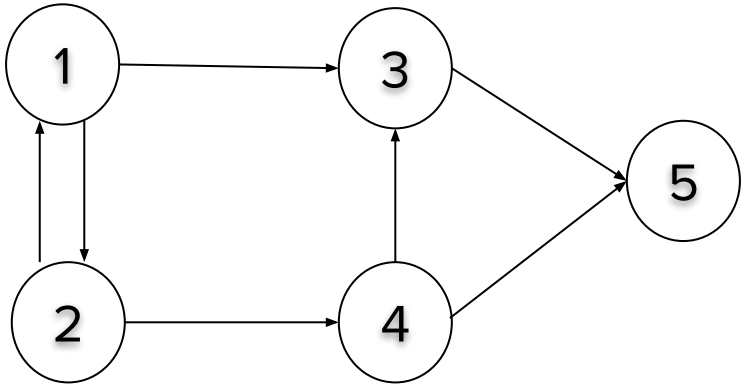
- Principal preocupação: como representar o conjunto de arestas?
- Duas formas usuais:
    - Matriz de adjacência
    - Lista de adjacência

Matriz de Adjacência

- Para um grafo de n vértices, usamos uma matriz M_{nxn} onde M[i][j] = 1 see existe uma aresta do vértice i para o vértice j.
- Para grafos não direcionados a matriz de adjacência é simétrica, ou seja, M[i][j] = M[j][i]
- Para grafos ponderados, ao invés de fazer M[i][j] = 1 caso exista a aresta (i, j), podemos fazer M[i][j] = p, sendo p o peso associado a essa aresta.

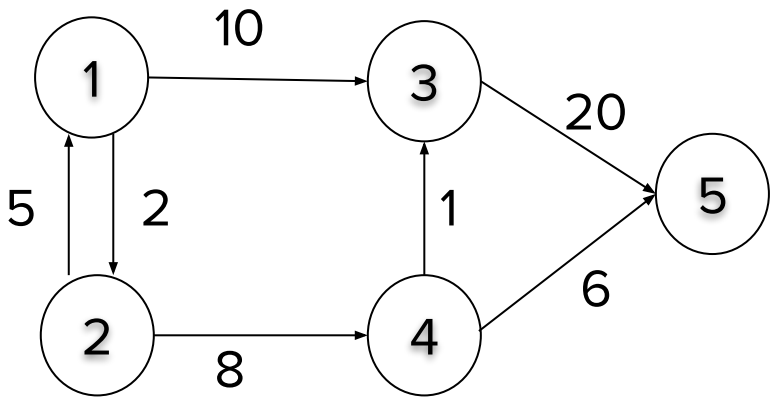
Matriz de Adjacência

	1	2	3	4	5
1	0	1	1	0	0
2	1	0	0	1	0
3	0	0	0	0	1
4	0	0	1	0	1
5	0	0	0	0	0



Matriz de Adjacência

	1	2	3	4	5
1	0	2	10	0	0
2	5	0	0	8	0
3	0	0	0	0	20
4	0	0	1	0	6
5	0	0	0	0	0



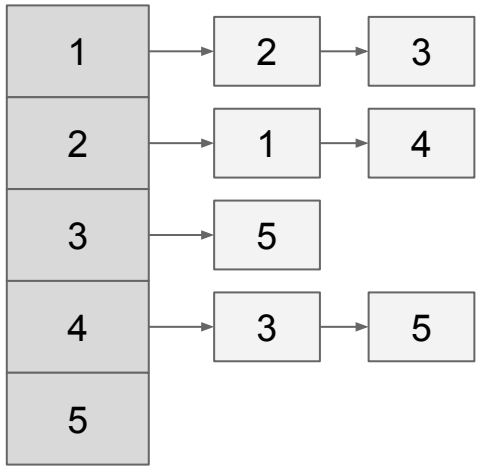
Matriz de Adjacência

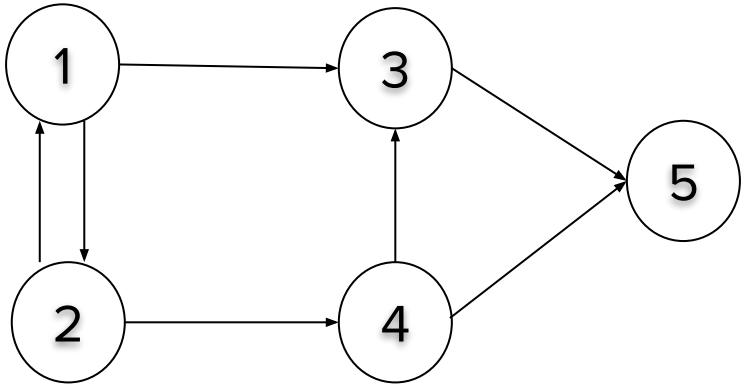
- Vantagens:
    - Verificar se existe uma aresta (i, j) pode ser feito em tempo constante.
    - Inserção ou remoção de arestas também podem ser realizadas com custo constante.
- Desvantagens:
    - Espaço necessário: $O(|V|^2)$
    - Acessar todos os nós adjacentes à um vértice v qualquer: $O(|V|)$

Lista de Adjacência

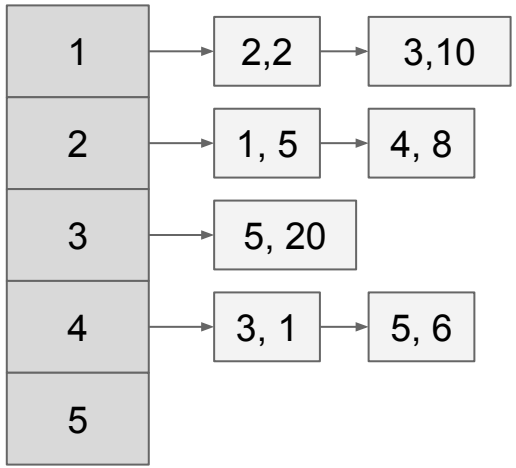
- Consiste em um vetor E com |V| entradas, uma para cada vértice do grafo, e cada entrada E[v] é uma lista de vértices adjacentes à v.

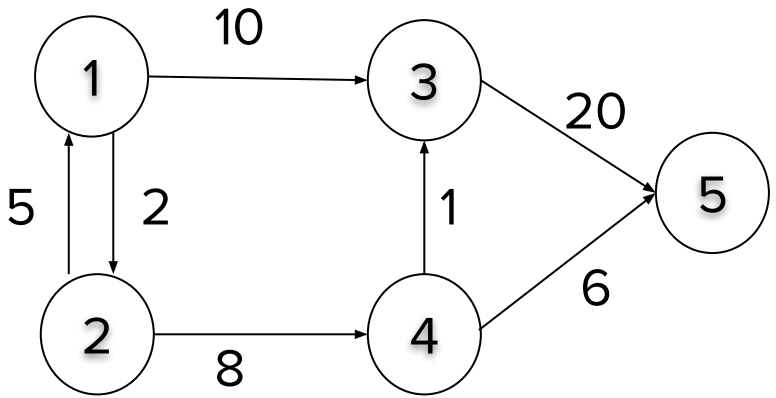
Lista de Adjacência



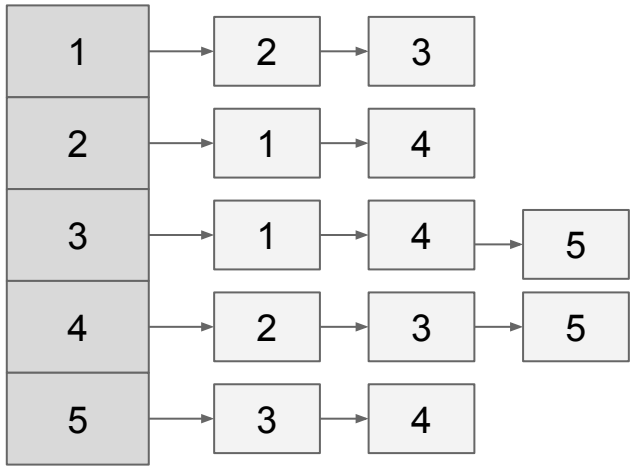


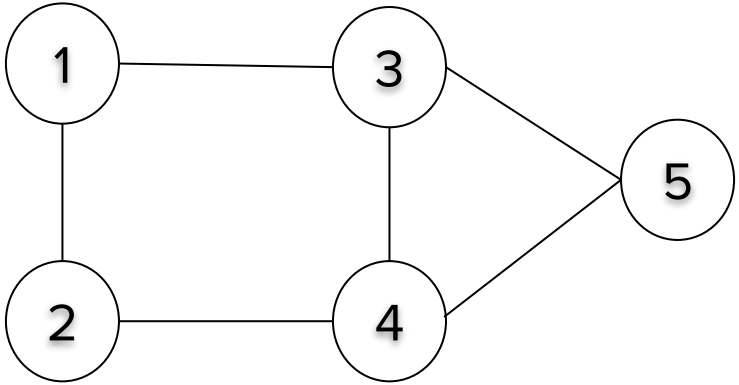
Lista de Adjacência





Lista de Adjacência





Lista de Adjacência

- Implementação

```
typedef struct{
    int v;    //vértice adjacente
    int w;    //peso
} TAdj;

vector<TAdj> adj[MAX_V]; //Lista de adjacência
int grau[MAX_V];         //número de arestas do vértice
```

Lista de Adjacência

- Implementação

```
void initGrafo(int qtdeVertices){
    memset(grau, 0, sizeof(int)*qtdeVertices);
    for(int i = 0; i < qtdeVertices; i++)
        adj[i].clear();
}
```

Lista de Adjacência

- Implementação

```
//Cria aresta de a para b, com peso w
void aresta(int a, int b, int w){
    TAdj aux;
     aux.v = b;
     aux.w = w;
     grau[a]++;
     adj[a].push_back(aux);
     //Se o grafo for não orientado, também adicionamos a aresta
     (b, a) com peso w
}
```

Busca em profundidade (DFS)

- Generalização da busca em profundidade em árvores.
- Na nossa implementação de exemplo, usaremos a busca em profundidade para armazenar as seguintes informações:
    - Ordem de acesso a cada nó (vamos aproveitar o vetor “visitado” para isso, que também poderia ser usado como um vetor booleano caso a ordem de visitação não importasse)
    - O nó “pai” de cada nó, armazenando em um vetor p. Por exemplo, se estamos visitando o nó v e através dele acessamos o nó u, então o p[u] = v

Busca em profundidade (DFS)

- Implementação

```
int visitado[MAX_V];
int p[MAX_V];
int ordemVis;

void initDfs(){
    memset(visitado, 0, sizeof(visitado));
    memset(p, -1, sizeof(p));
    ordemVis = 0;
}
```

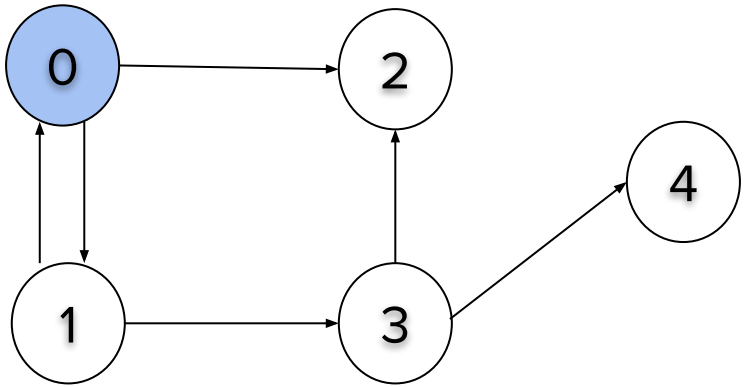
Busca em profundidade (DFS)

- Implementação

```
void dfs(int s){
    int t;
    visitado[s] = ++ordemVis;
    for(int i = 0; i < grau[s]; i++){
        t = adj[s][i].v;
        if (visitado[t] == 0){
            p[t] = s;
            dfs(t);
        }
    }
}
```

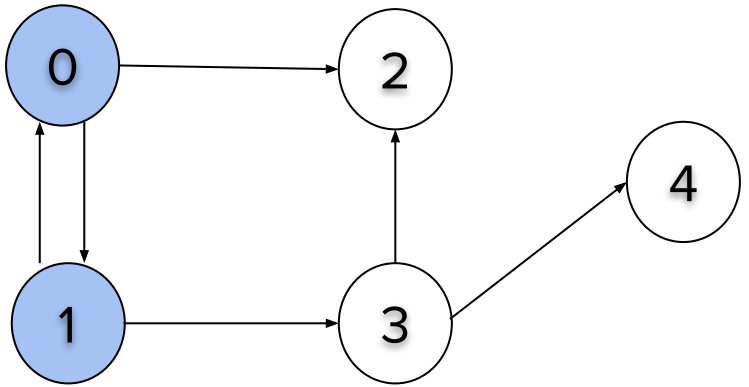
Busca em profundidade (DFS)

```
dfs(0):
    visitado: {1, 0, 0, 0, 0}
    p: {-1, -1, -1, -1, -1}
    chama dfs(1)

    OBS: após retornar do
dfs(1) ele chamaria dfs(2),
se o vértice 2 ainda não
estivesse marcado como
visitado
```



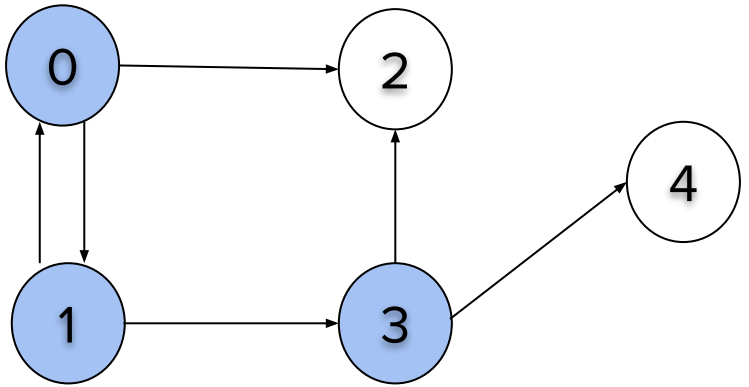
Busca em profundidade (DFS)

```
dfs(1):
    visitado: {1, 2, 0, 0, 0}
    p: {-1,  0, -1, -1, -1}
    chama dfs(3)
```



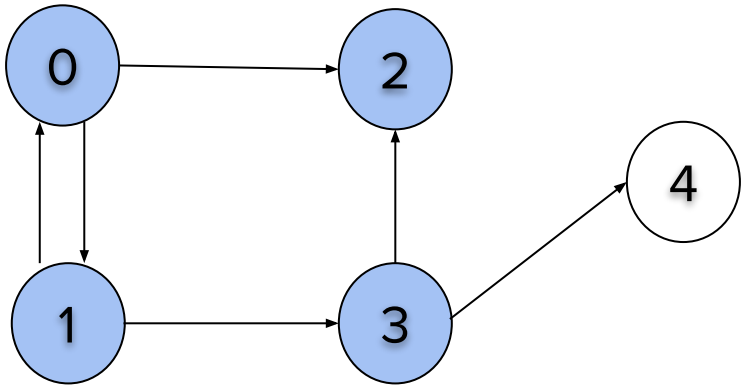
Busca em profundidade (DFS)

```
dfs(3):
    visitado: {1, 2, 0, 3, 0}
    p: {-1,  0, -1,  1, -1}
    chama dfs(2)
```



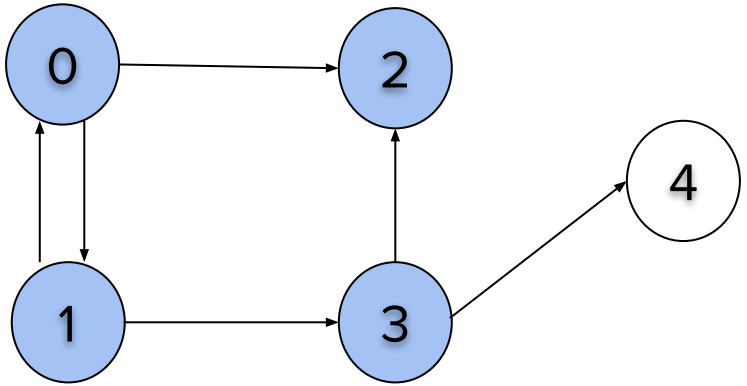
Busca em profundidade (DFS)

```
dfs(2):
    visitado: {1, 2, 4, 3, 0}
    p: {-1,  0,  3,  1, -1}
    nenhum vértice adjacente,
retorna de onde foi chamado
```



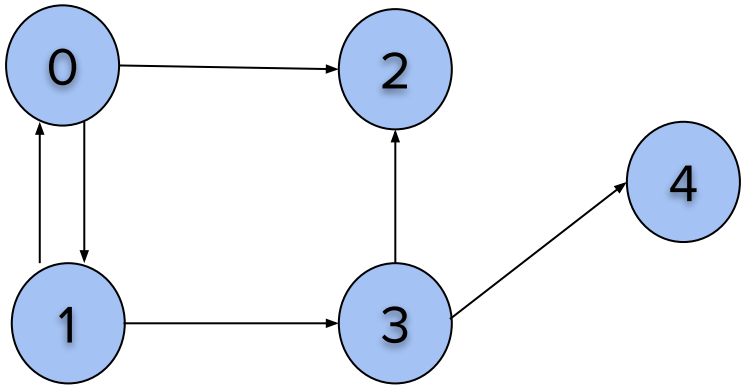
Busca em profundidade (DFS)

```
de volta em dfs(3):
    chama dfs(4)
```



Busca em profundidade (DFS)

```
dfs(4):
    visitado: {1, 2, 4, 3, 5}
    p: {-1,  0,  3,  1,  3}
    nenhum vértice adjacente,
retorna de onde foi chamado
```



Busca em largura (BFS)

- Da mesma forma que a DFS, a busca em largura é uma generalização da busca em largura aplicada em árvores.
- A busca em largura é feita por níveis.
- Vamos definir nível em um vértice v como sendo o comprimento do menor caminho do vértice inicial (por onde começamos a busca) até o vértice v.
- OBS: Perceba que, sendo assim, no final de uma busca em largura teremos o menor caminho (em número de arestas) do vértice inicial até cada vértice do grafo.

Busca em largura (BFS)

- Implementação

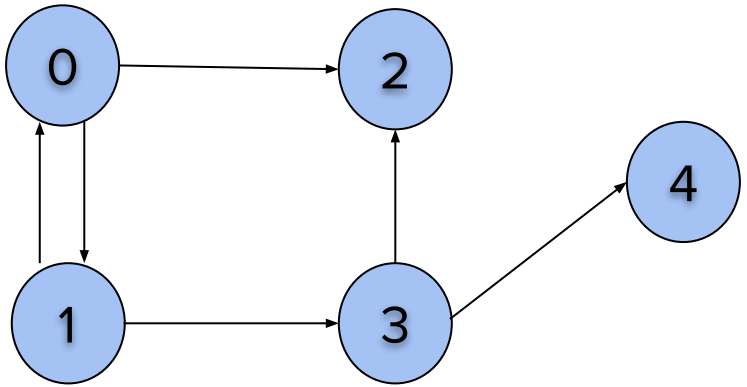
```
int d[MAX_V];        //armazena a distância do nó inicial até cada nó i

void bfs(int inicio){
        int s, t;
        queue<int> Q;
        memset(visitado, 0, sizeof(visitado));
        memset(p, -1, sizeof(p));
        d[inicio] = 0;
        visitado[inicio] = ++ordemVis;
        Q.push(inicio);
```

Busca em largura (BFS)

```
        while(!Q.empty()){
            s = Q.front();
            Q.pop();
            for(int i = 0; i < grau[s]; i++){
                t = adj[s][i];
                if (visitado[t] == 0){
                    visitado[t] = ++ordemVis;
                    d[t] = d[s] + 1;
                    p[t] = s;
                    Q.push(t);
                }
            }
        }
    }
```

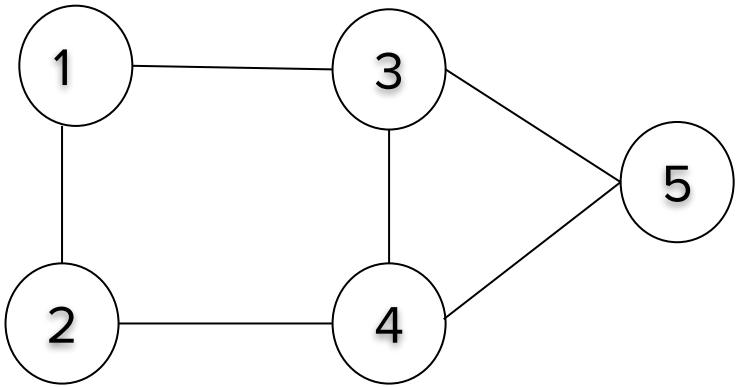
Busca em largura (BFS)

```
visited: {1, 2, 3, 4, 5}
p: {-1,  0, 0,  1, 3}
```

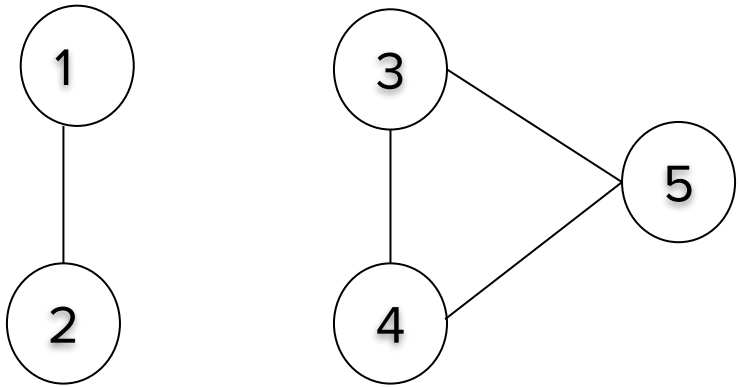


Conexidade

- Um grafo **não direcionado** G = (V, A) é conexo se existe um caminho em G entre todos os pares de vértices.



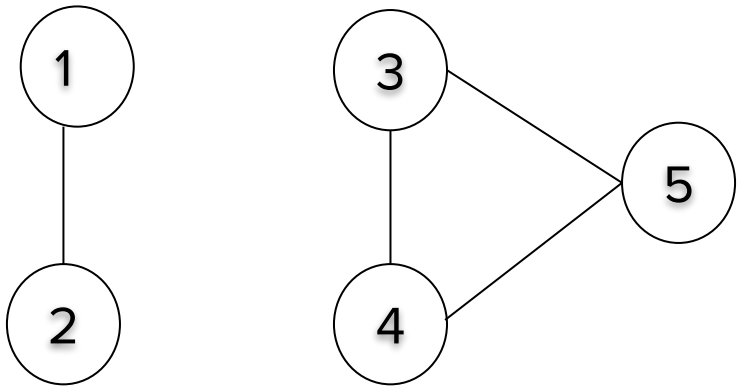
conexo



desconexo

Conexidade

- Um grafo G’ = (V’, A’) é um **subgrafo** de G = (V, A) see V’ ⊆ V e A’ ⊆ A.
- Um subgrafo conexo de G é chamado de **componente conexa** de G.
- O grafo a seguir, por exemplo, possui duas componentes conexas.

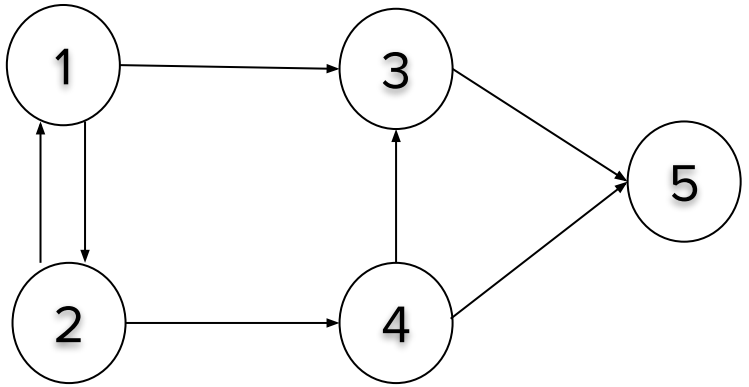


Conexidade

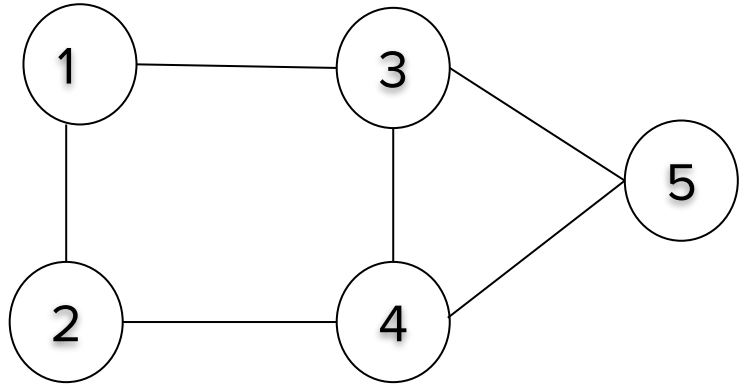
- Para grafos direcionados, definimos dois tipos de conexidade: forte e fraca.
- Um grafo direcionado é **fortemente conexo** se existir um caminho entre todos os pares de vértices do grafo.
- Um grafo direcionado é **fracamente conexo** se o seu grafo não direcionado subjacente (retirando a orientação das arestas) é conexo.

Conexidade

- Grafo:



- Grafo não direcionado subjacente:



Esse grafo não é fortemente conexo, mas é fracamente conexo.

Conexidade

- Como determinar se um **grafo não direcionado** é conexo?
    - Basta fazer um percurso no grafo (em profundidade ou em largura), a partir de qualquer nó, e verificar se todos os vértices foram visitados.
    - Se sim, esse grafo é conexo.
    - Caso contrário, não é, e os vértices visitados formam uma componente conexa.

Conexidade

- Como determinar se um **grafo direcionado** é fortemente conexo?
    - Deve-se fazer um percurso no grafo para cada vértice, e cada um desses percursos deve conseguir visitar todos os vértices do grafo.

Problema do Caminho Mínimo

- Imagine o seguinte problema: dado um mapa de cidades, contendo as distâncias entre as cidades, qual o menor caminho entre quaisquer cidades A e B?
- Esse problema pode ser modelado através de um grafo:
    - Cidades: vértices
    - Estradas entre cidades: arestas ponderadas com peso que indicam a distância entre as cidades

Problema do Caminho Mínimo

- Generalizando, o nosso problema é encontrar o caminho de menor custo em um grafo de um vértice A até um vértice B.
- Chamamos de custo de um caminho a soma dos pesos das arestas pertencentes a esse caminho.

Problema do Caminho Mínimo

- Existem alguns algoritmos que resolvem tal problema, vamos apresentar um dos mais conhecidos e utilizados, o algoritmo de **Dijkstra**.
- Restrição para esse algoritmo:
    - Todos os pesos devem ser não negativos.

Algoritmo de Dijkstra

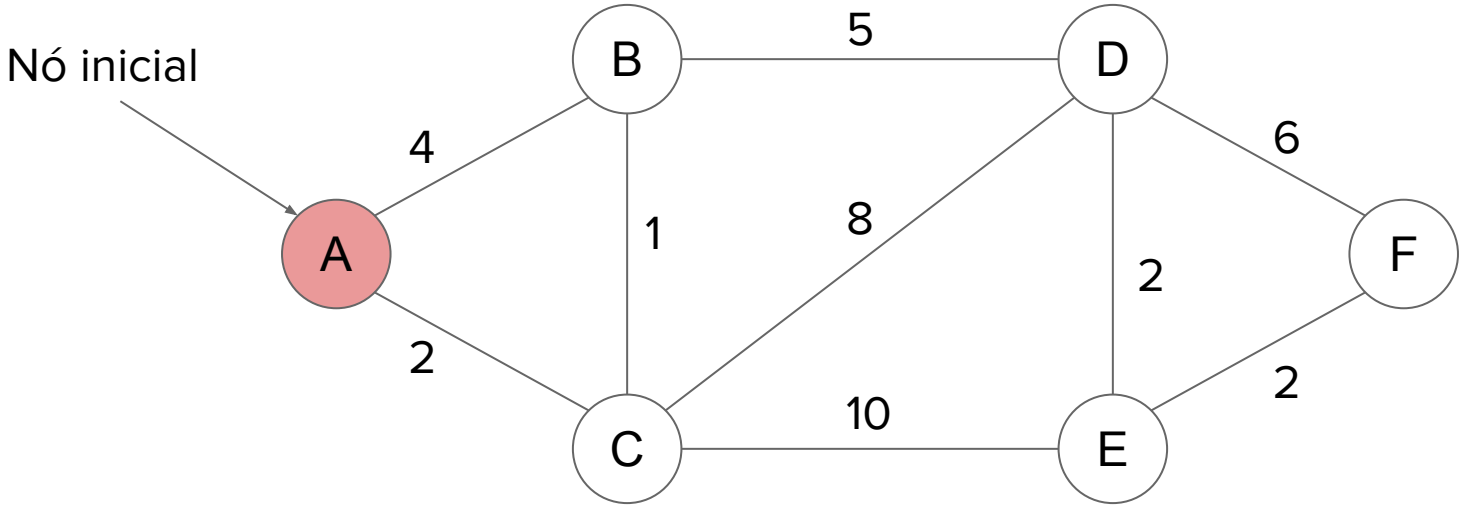
- Este algoritmo parte de uma estimativa inicial para o custo mínimo e vai, sucessivamente, ajustando esta estimativa. Ele considera que um vértice estará **fechado** quando já tiver sido obtido um caminho de custo mínimo do vértice tomado como origem da busca até ele. Caso contrário, ele é dito **aberto**.

Algoritmo de Dijkstra

- Algoritmo: Seja G(V, A) um grafo e **s** um vértice de G (tomado como origem):

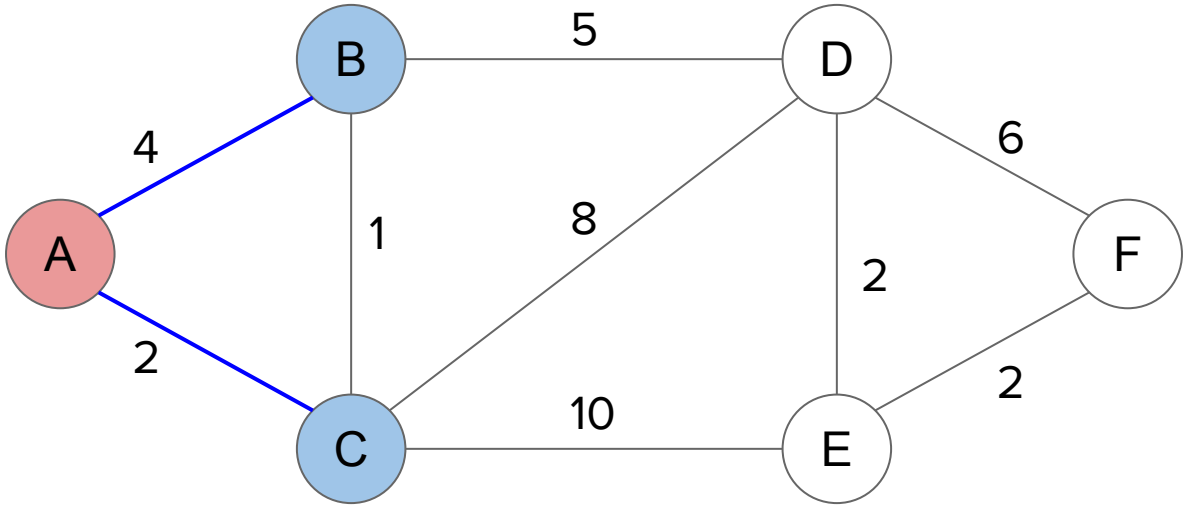
1. Atribua valor zero à estimativa de custo mínimo do vértice **s** e infinito às demais estimativas.
2. Enquanto houver vértice aberto:
    - Seja **k** um vértice ainda aberto cuja estimativa seja a menor entre todos os vértices abertos
    - Fecha o vértice **k**
    - Para todo o vértice **j** ainda aberto que seja adjacente à **k** faça:
        - Soma a estimativa do vértice **k** com o custo da aresta **(k, j)**
        - Caso essa estimativa seja melhor que a anterior para **j**, substitua-se e anote **k** como precedente (“pai”) de **j**

Algoritmo de Dijkstra



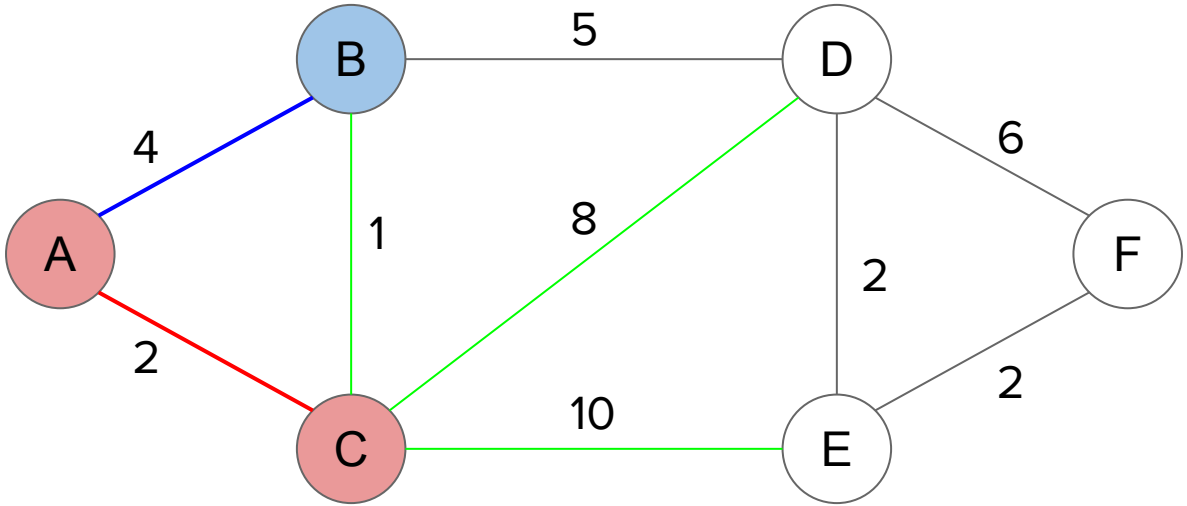
Vértices	**A**	**B**	**C**	**D**	**E**	**F**
Estimativas	0	∞	∞	∞	∞	∞
Precedentes	-	-	-	-	-	-

Algoritmo de Dijkstra



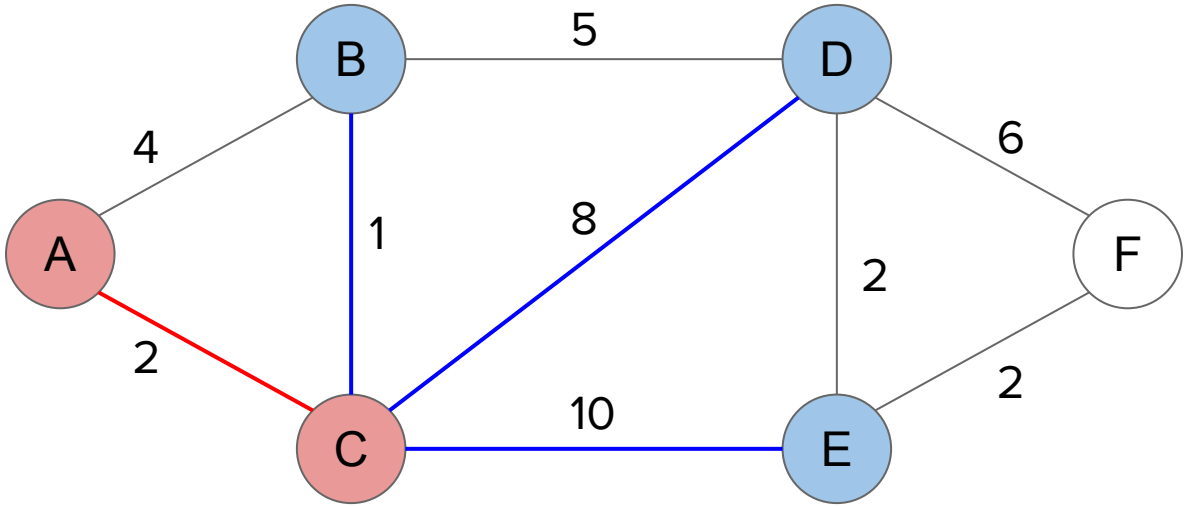
Vértices	**A**	**B**	**C**	**D**	**E**	**F**
Estimativas	0	4	2	∞	∞	∞
Precedentes	-	A	A	-	-	-

Algoritmo de Dijkstra



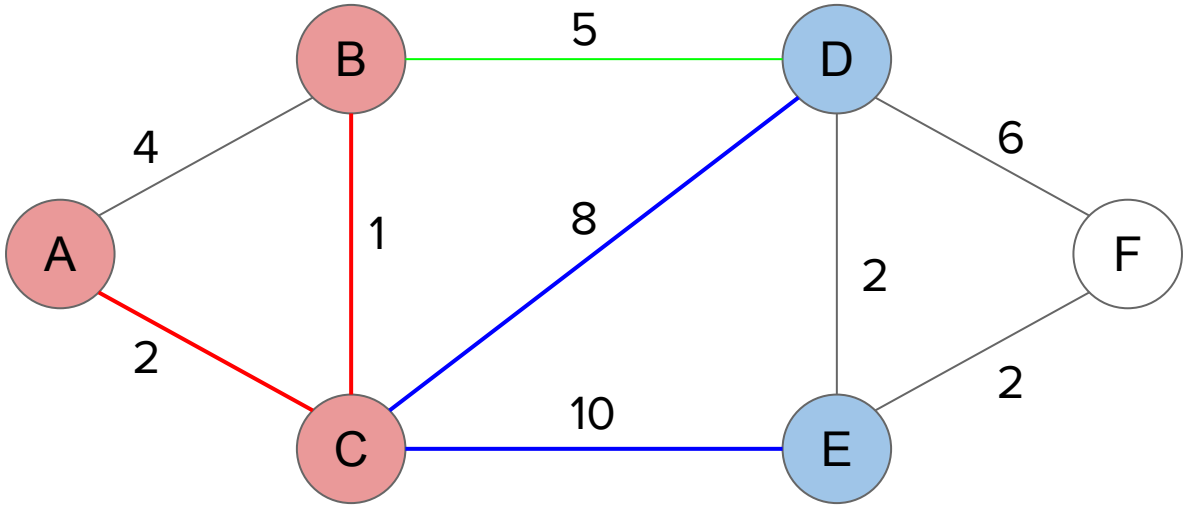
Vértices	A	B	C	D	E	F
Estimativas	0	4	2	∞	∞	∞
Precedentes	-	A	A	-	-	-

Algoritmo de Dijkstra



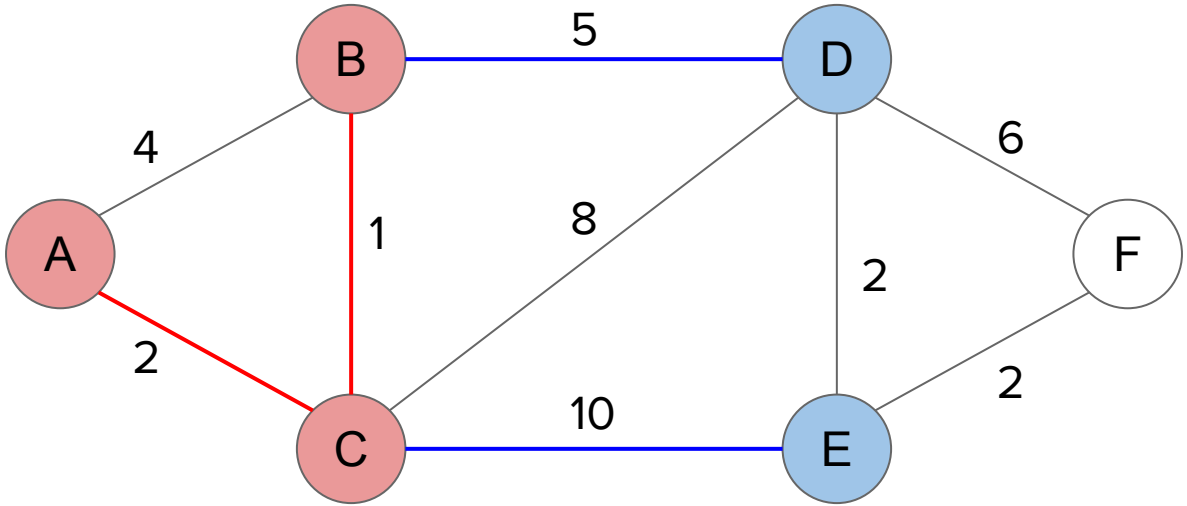
Vértices	A	B	C	D	E	F
Estimativas	0	3	2	10	12	∞
Precedentes	-	C	A	C	C	-

Algoritmo de Dijkstra



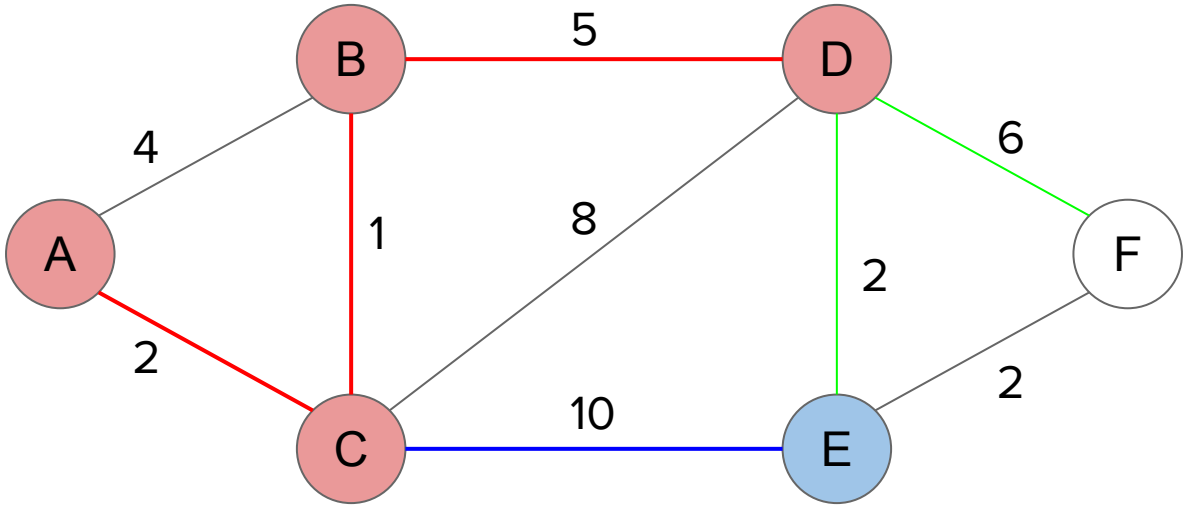
Vértices	A	B	C	D	E	F
Estimativas	0	3	2	10	12	∞
Precedentes	-	C	A	C	C	-

Algoritmo de Dijkstra



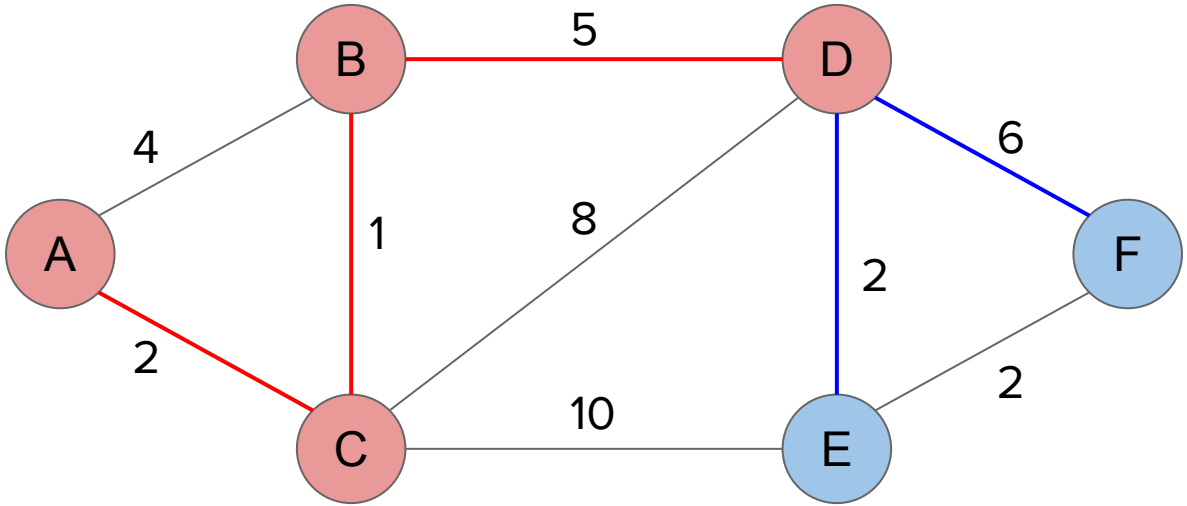
Vértices	A	B	C	D	E	F
Estimativas	0	3	2	8	12	∞
Precedentes	-	C	A	B	C	-

Algoritmo de Dijkstra



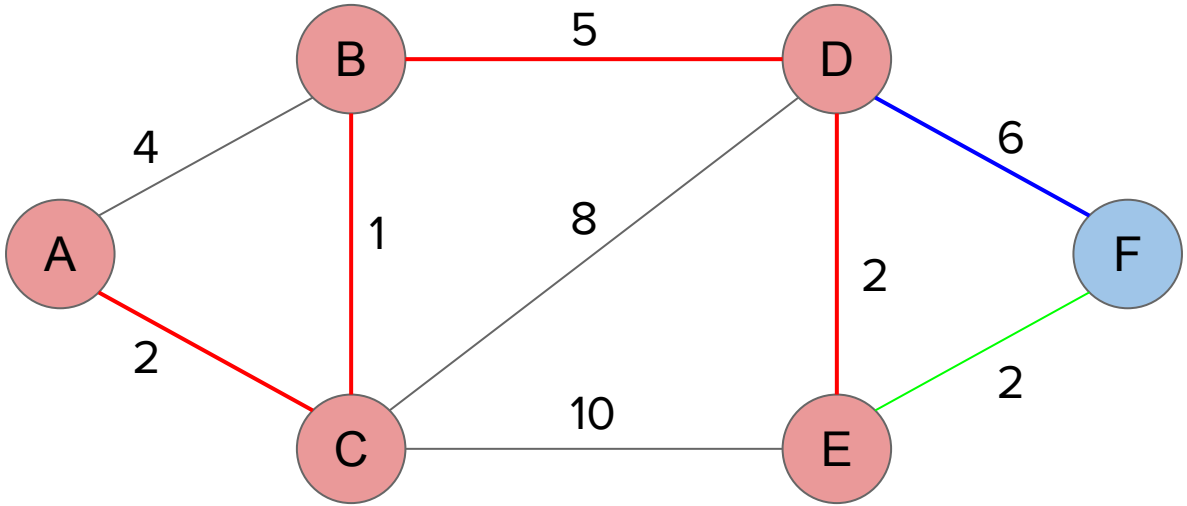
Vértices	A	B	C	D	E	F
Estimativas	0	3	2	8	12	∞
Precedentes	-	C	A	B	C	-

Algoritmo de Dijkstra



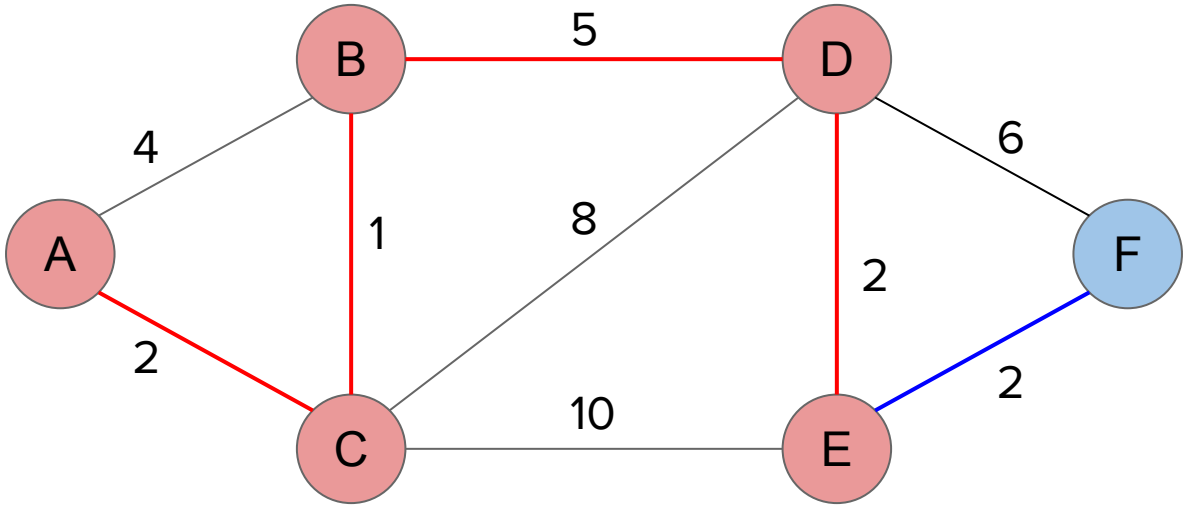
Vértices	A	B	C	D	E	F
Estimativas	0	3	2	8	10	14
Precedentes	-	C	A	B	D	D

Algoritmo de Dijkstra



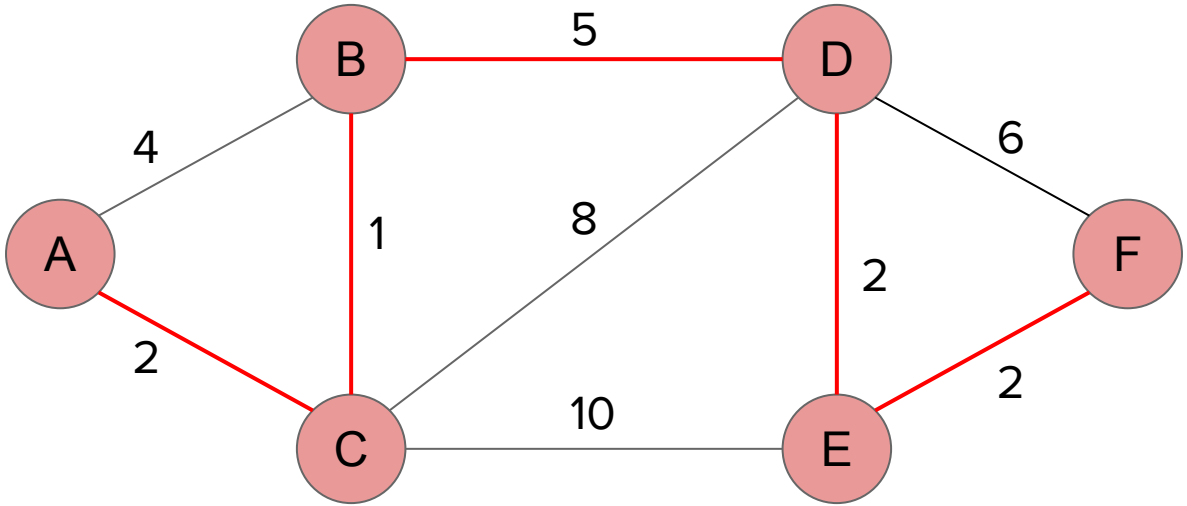
Vértices	A	B	C	D	E	F
Estimativas	0	3	2	8	10	14
Precedentes	-	C	A	B	D	D

Algoritmo de Dijkstra



Vértices	A	B	C	D	E	F
Estimativas	0	3	2	8	10	12
Precedentes	-	C	A	B	D	E

Algoritmo de Dijkstra



Vértices	**A**	**B**	**C**	**D**	**E**	**F**
Estimativas	0	3	2	8	10	12
Precedentes	-	C	A	B	D	E

Algoritmo de Dijkstra

```
int d[MAX_V];    //d[i] armazena a distância até o vértice i, e as
                 //estimativas durante as iterações
int p[MAX_V];    //armazena o predecessor de cada vértice

void dijkstra(int inicial, int vertices){
    priority_queue< pair<int, int> > heap; //distância, vértice
    int s, t, peso;

    for(int i = 0; i < vertices; i++)
        d[i] = INT_MAX;
    memset(p, -1, sizeof(p));

    heap.push(make_pair(d[inicial] = 0, inicial));
```

Algoritmo de Dijkstra

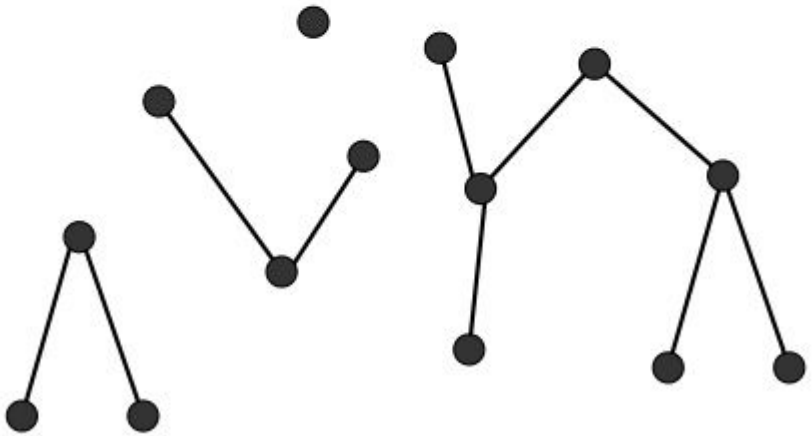
```
    while(!heap.empty()){
        s = heap.top().second;
        heap.pop();
        for(int i = 0; i < grau[s]; i++){
            t = adj[s][i].v;
            peso = adj[s][i].w;
            if (d[s] + peso < d[t]){
                d[t] = d[s] + peso;
                p[t] = s;
                heap.push(make_pair(-d[t], t));
            }
        }
    }
}
```

Algoritmo de Dijkstra

- Analisando a complexidade desse algoritmo de forma intuitiva, temos que (pensando no pior caso):
    - Todos os vértices são fechados: |V| operações
    - Cada vez que um vértice é fechado, é porque ele foi extraído de uma heap: custo O(1) => O(|V|)
    - Para cada vértice, todas as suas arestas são acessadas. No total, acessaremos |A| arestas => O(|V| + |A|)
    - Cada vez que uma aresta é acessada, podemos inserir um elemento na heap: custo O(log |V|) => O((|V| + |A|) * log |V|)
- Complexidade: **O((|V| + |A|) * log |V|)**

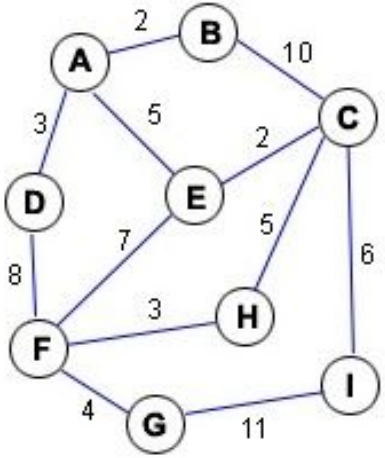
Árvore

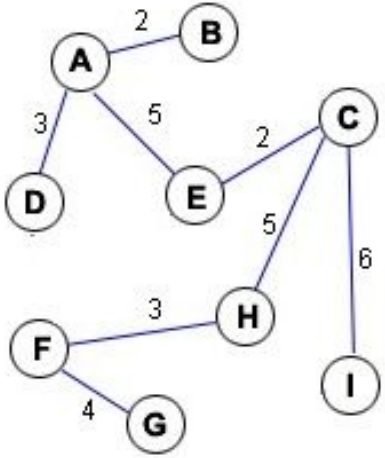
- Uma **árvore** pode ser definida como um grafo não orientado **conexo** e **acíclico.** Uma árvore de N vértices possui N-1 arestas.
- Uma **floresta** é um grafo não orientado **acíclico.** Cada componente conexa de uma floresta é uma árvore.



Árvore geradora

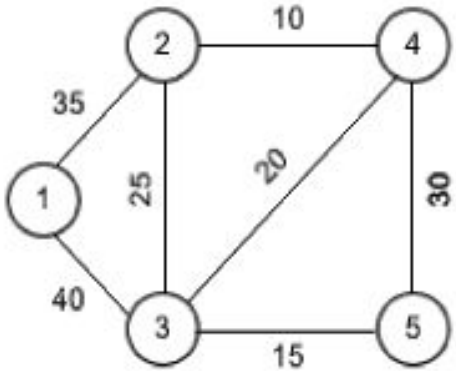
- Uma **árvore** G’ = (V’, A’) é uma árvore geradora de G = (V, A) se G’ é subgrafo de G e V’ = V.
- Observe que, ainda que G’ possivelmente não tenha todas as arestas de G, como G’ uma árvore, ainda existe um caminho que liga cada par de vértices de G’.

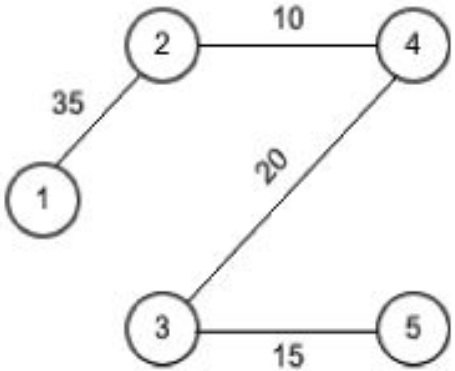




Árvore geradora mínima (MST)

- Vamos considerar que o **custo** de uma árvore geradora é a soma dos pesos das arestas que a compõe.
- Uma **árvore geradora mínima (MST)** de G é uma árvore geradora de G que tem custo mínimo.





Árvore geradora mínima (MST)

- Exemplo de problema prático: dado um conjunto de cidades e os custos para construir linhas de comunicação que interligam duas cidades, determine quais as linhas devem efetivamente ser construídas de forma que todas as cidades sejam interligadas com o menor custo possível.

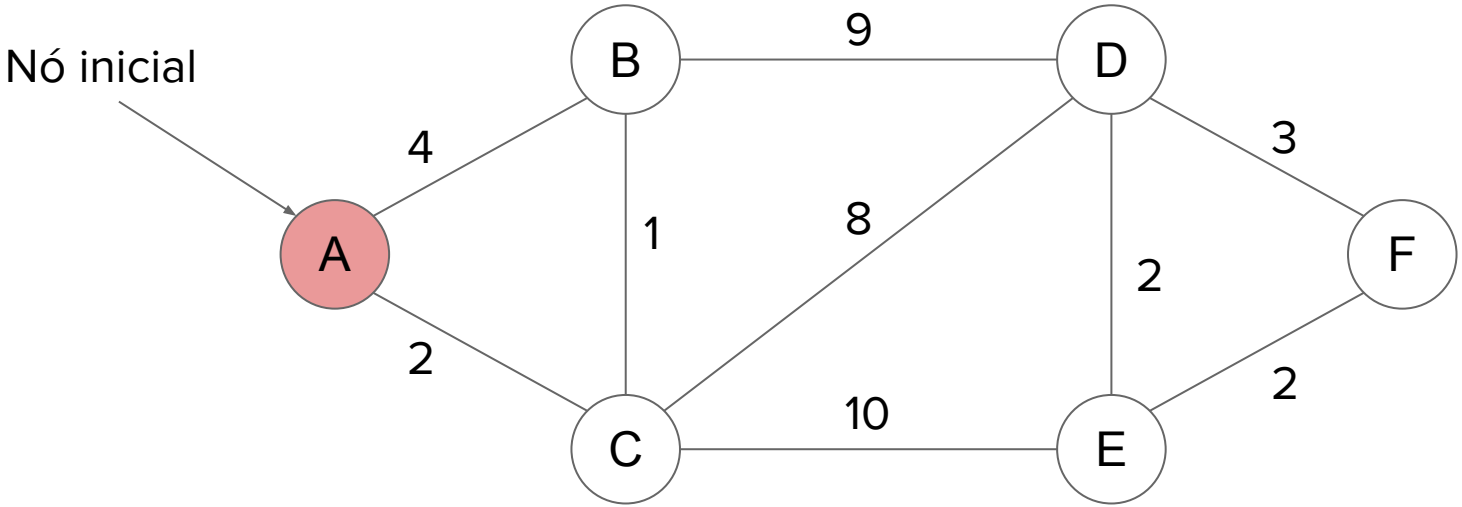
Árvore geradora mínima (MST)

- Dois algoritmos clássicos:
    - Algoritmo de **Prim**
    - Algoritmo de **Kruskal**
- Ambos são algoritmos gulosos, mas que levam a soluções ótimas.

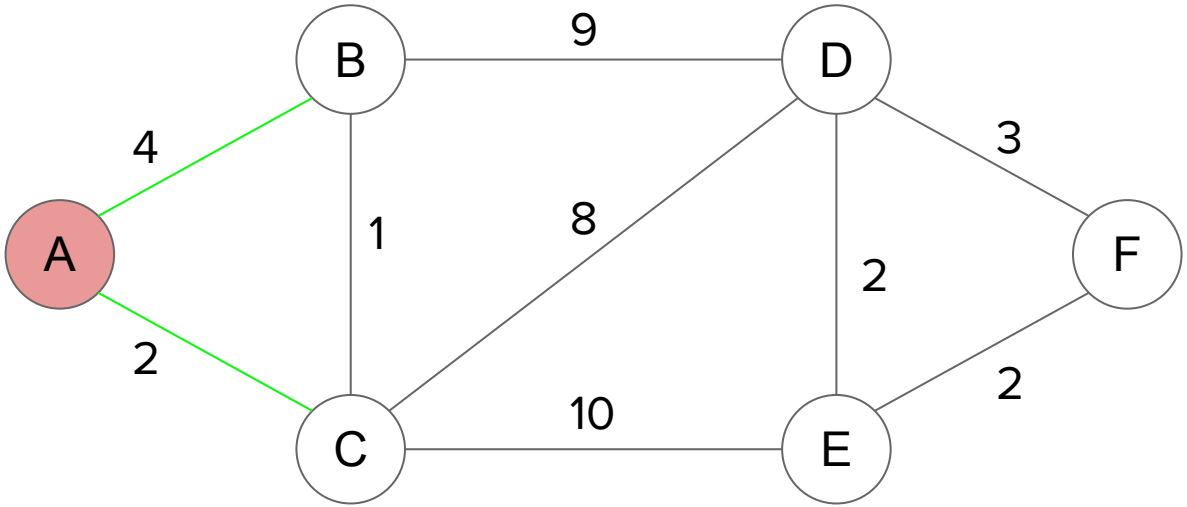
Algoritmo de Prim

- Escolha um vértice inicial e adicione na árvore
- Repita até que todos os vértices estejam na árvore:
    - Escolha a aresta de menor peso que seja incidente a um vértice já na árvore e que não forme ciclo.
    - Adicione essa aresta e o vértice correspondente a árvore

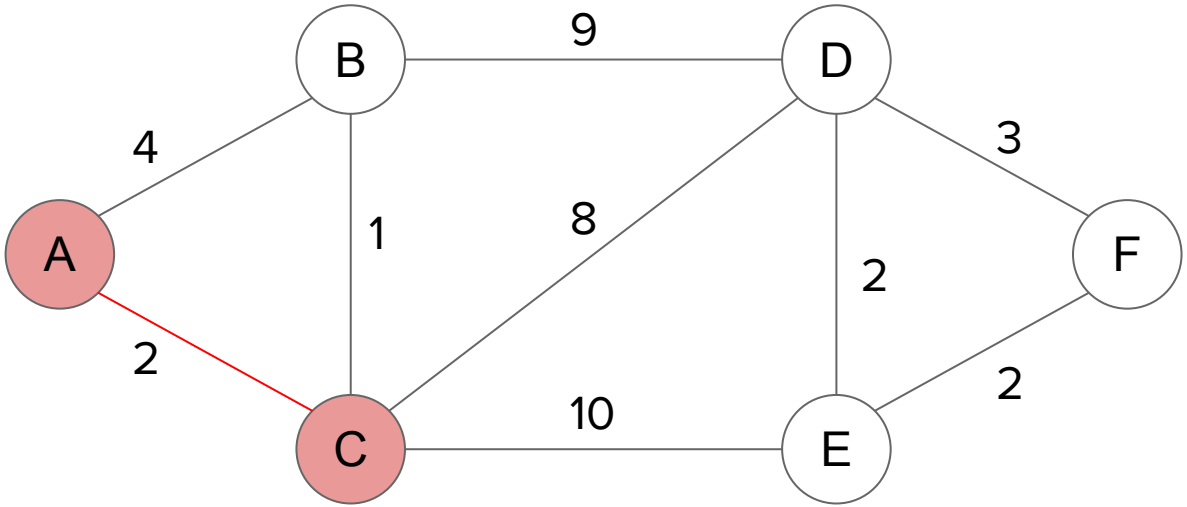
Algoritmo de Prim



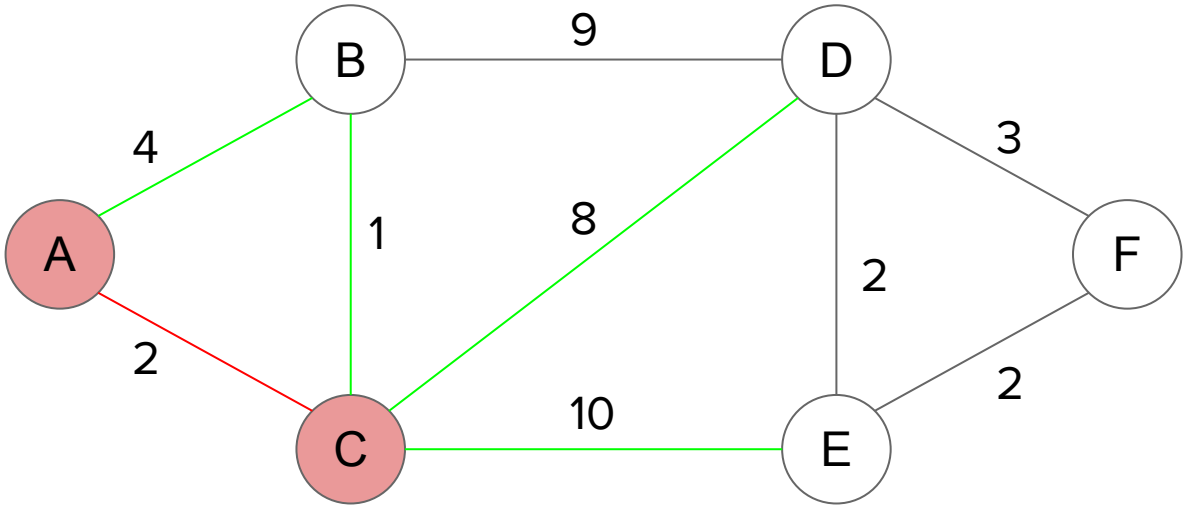
Algoritmo de Prim



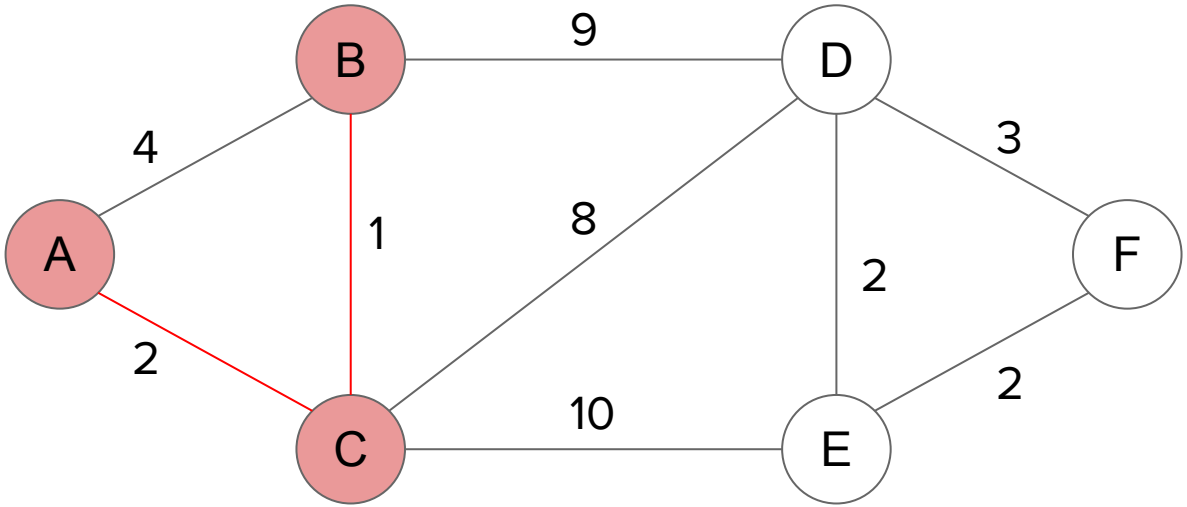
Algoritmo de Prim



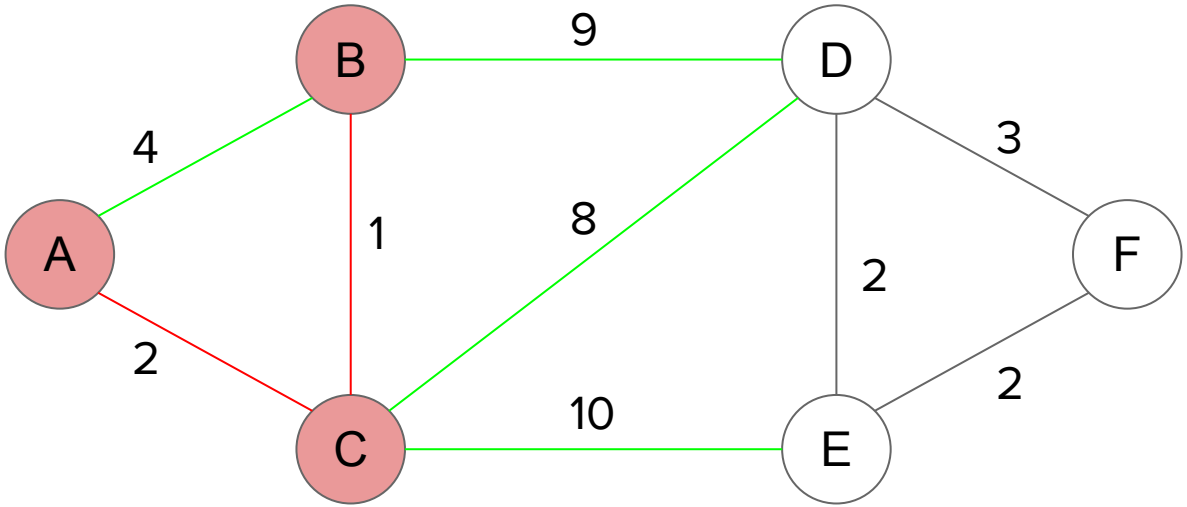
Algoritmo de Prim



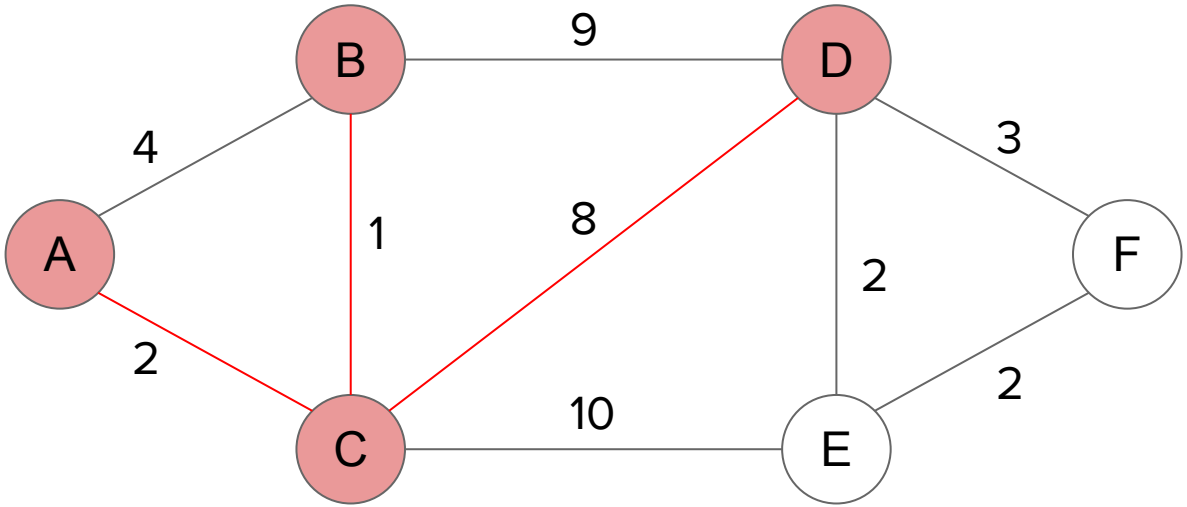
Algoritmo de Prim



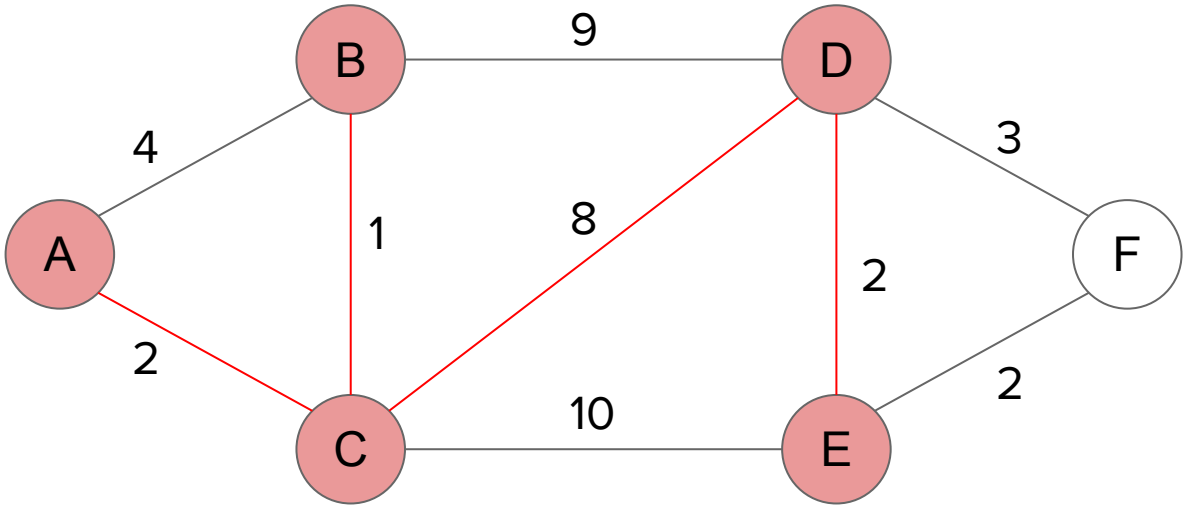
Algoritmo de Prim



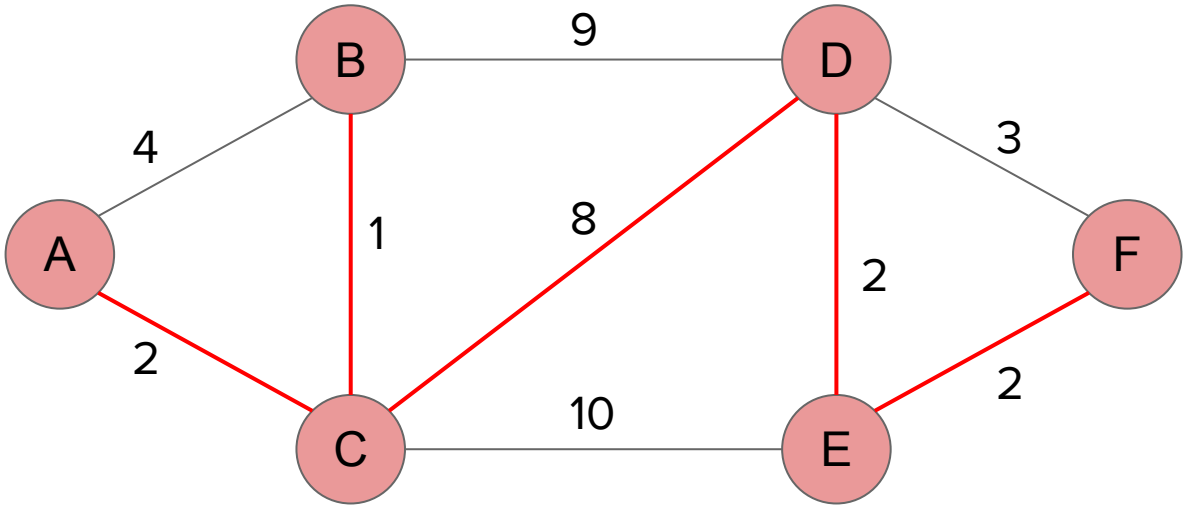
Algoritmo de Prim



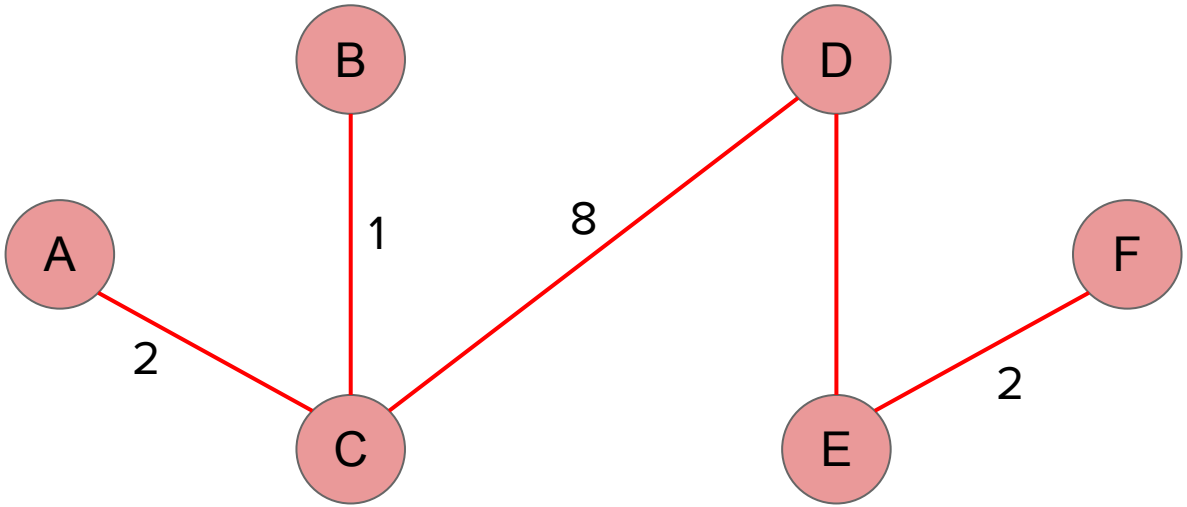
Algoritmo de Prim



Algoritmo de Prim



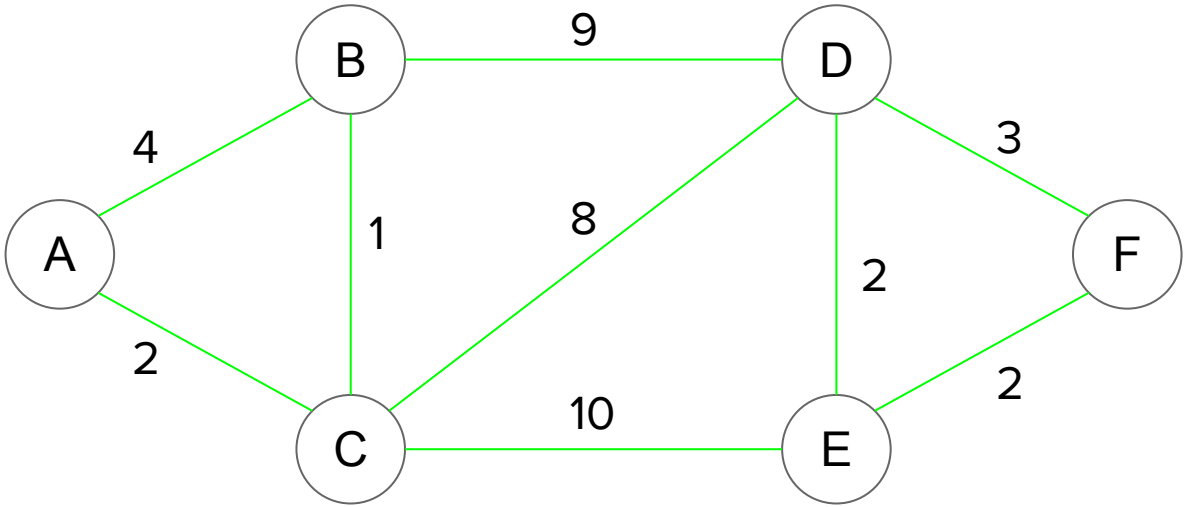
Algoritmo de Prim



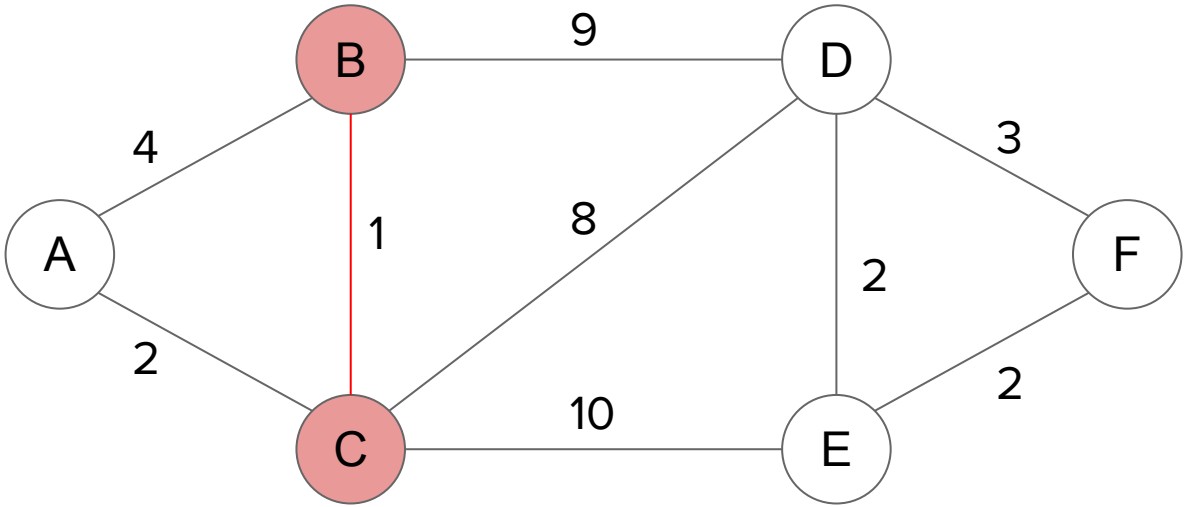
Algoritmo de Kruskal

- Repita n-1 vezes (sendo n o número de vértices do grafo):
    - Escolha a aresta de menor peso do grafo que ainda não esteja na floresta e que não forme ciclo.
    - Adicione essa aresta à floresta.

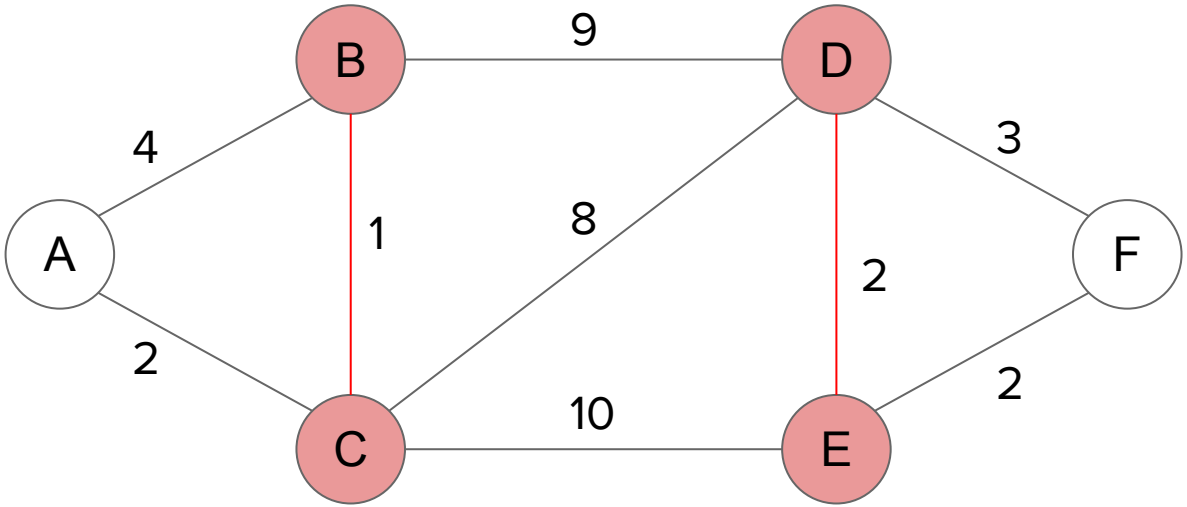
Algoritmo de Kruskal



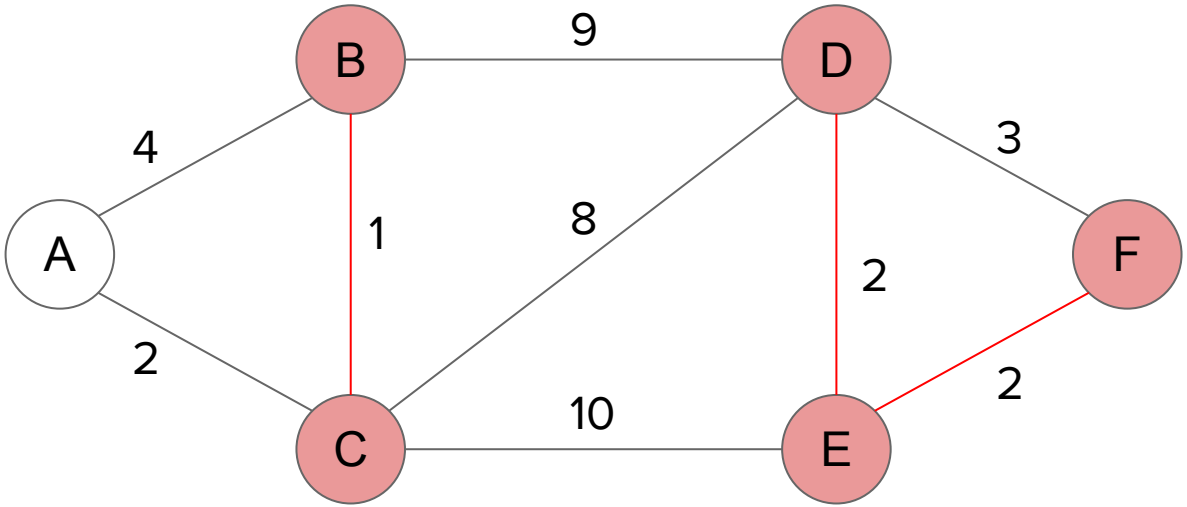
Algoritmo de Kruskal



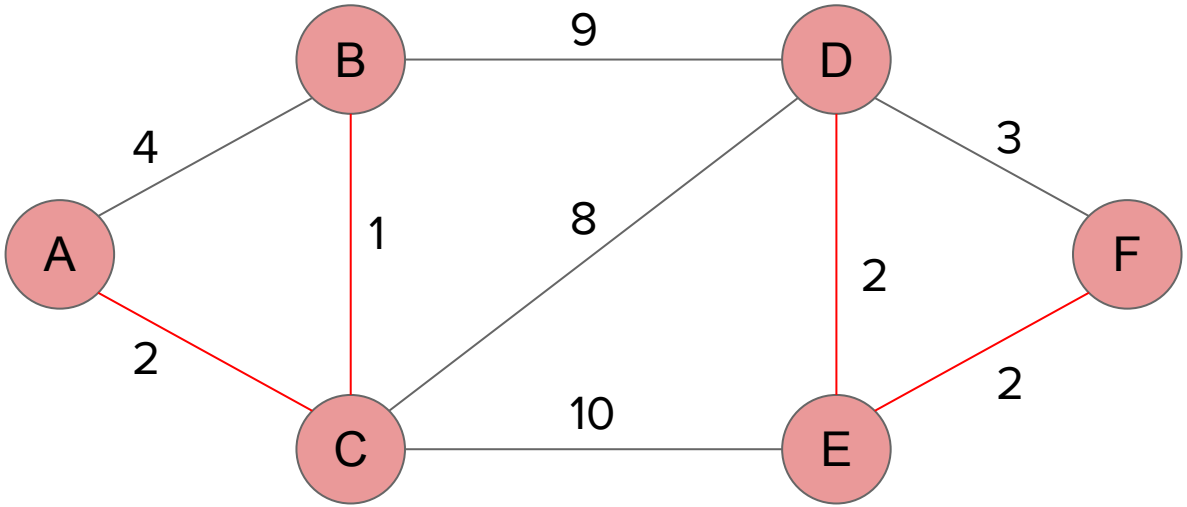
Algoritmo de Kruskal



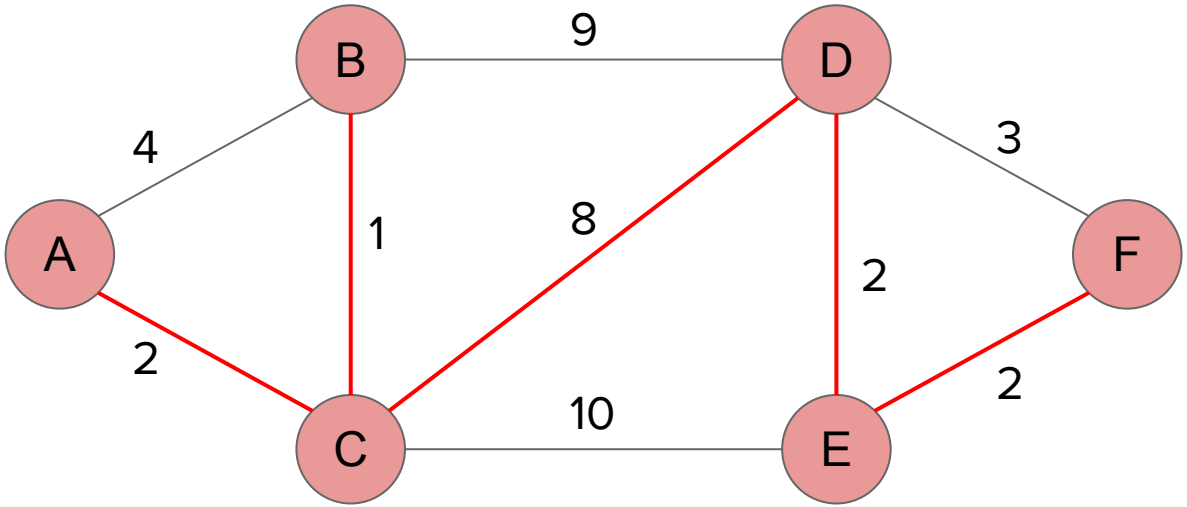
Algoritmo de Kruskal



Algoritmo de Kruskal



Algoritmo de Kruskal

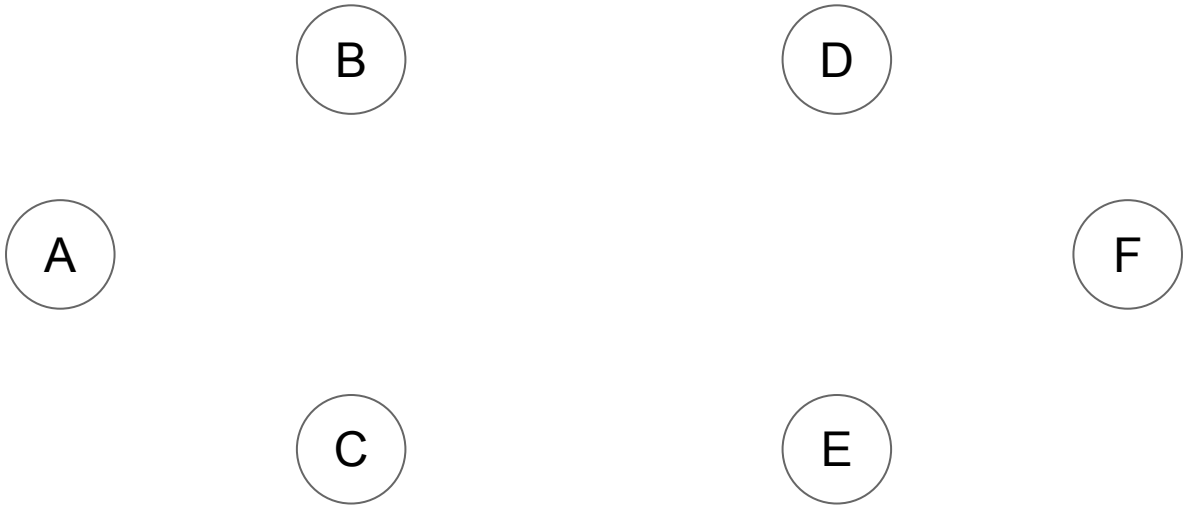


Disjoint Set (Union-find)

- Para a implementação do algoritmo de Kruskal, podemos utilizar uma estrutura de dados auxiliar conhecida como **disjoint set** ou **union-find** (devido as operações que podemos realizar com tal estrutura).
- Essa estrutura de dados é bastante eficiente e nos permite resolver uma série de problemas, por isso vamos apresentá-la rapidamente.

Disjoint Set (Union-find)

- Um disjoint set é uma estrutura de dados que considera um **conjunto** de elementos **particionados** em vários **subconjuntos disjuntos**.
- Inicialmente, cada conjunto contém precisamente um elemento.

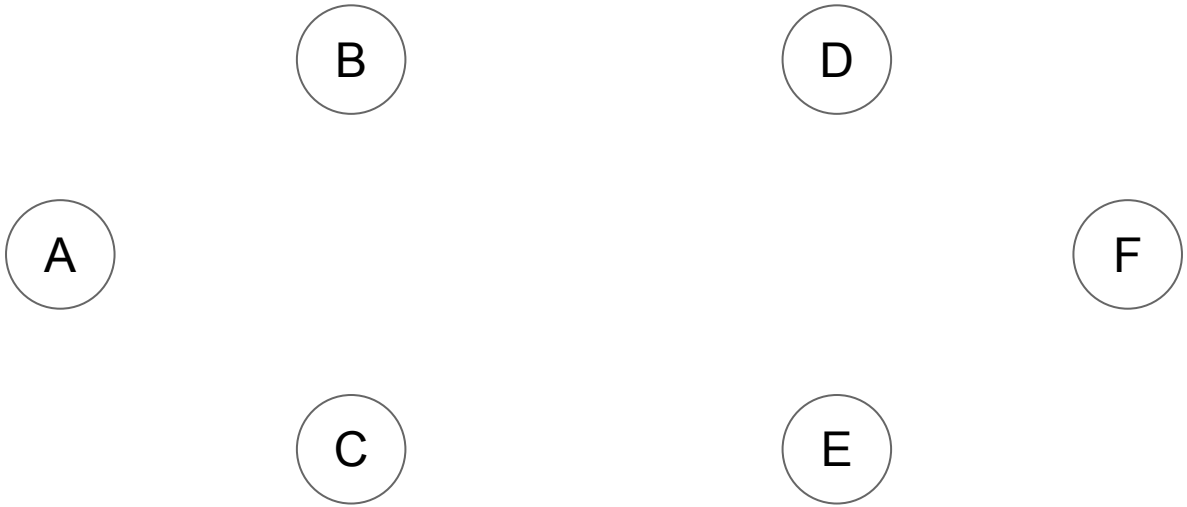


Disjoint Set (Union-find)

- Podemos fazer basicamente duas operações em um disjoint set:
    - union(x, y): une os conjuntos aos quais x e y pertencem
    - find(x): determina a qual conjunto x pertence
- A partir da função find(x), também é comum criarmos a função same(x, y), que verifica se os elementos x e y pertencem ao mesmo conjunto.

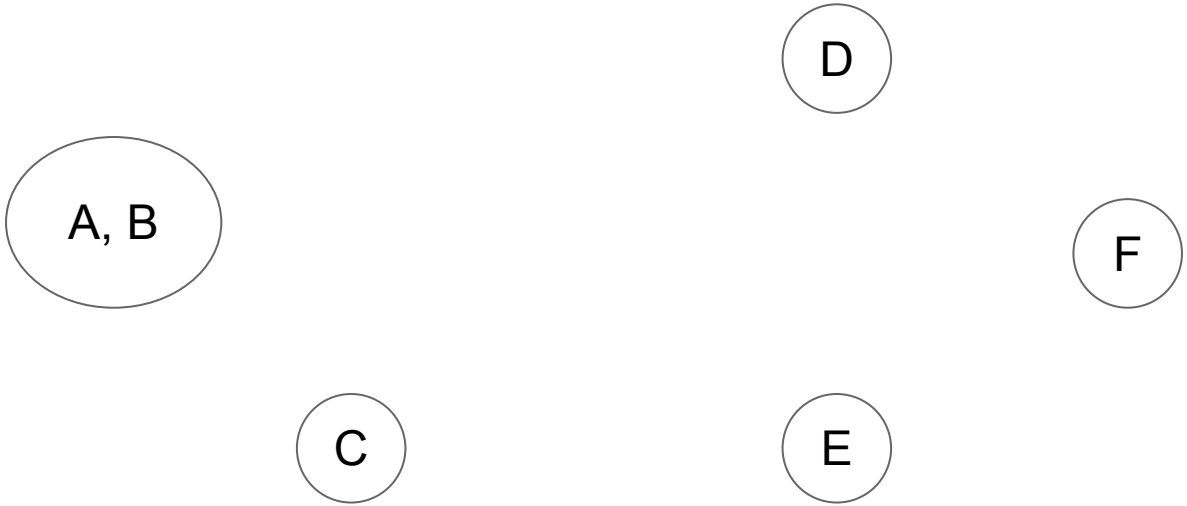
Disjoint Set (Union-find)

same(A, B) => False



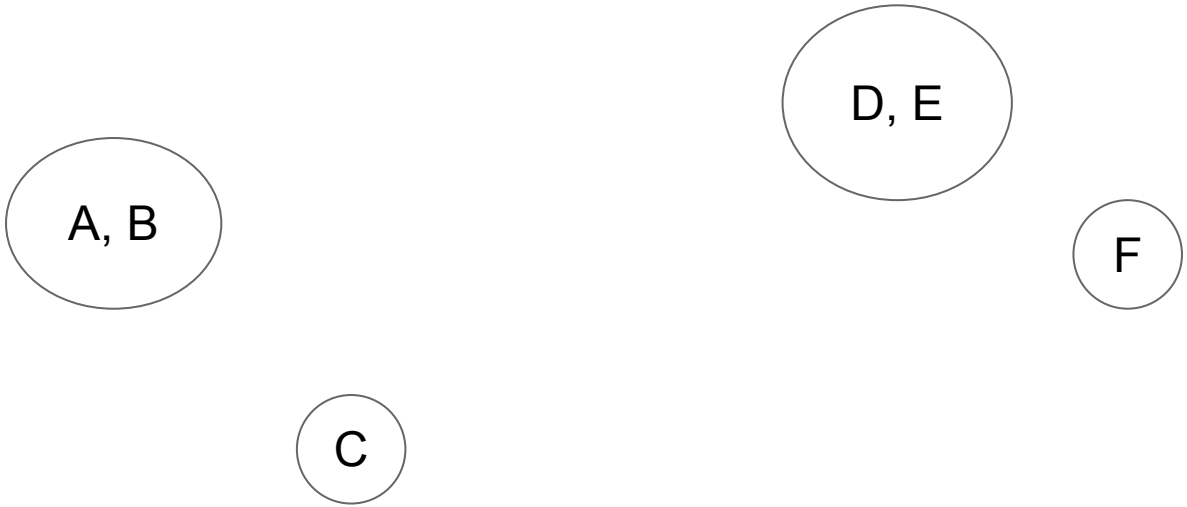
Disjoint Set (Union-find)

union(A, B)



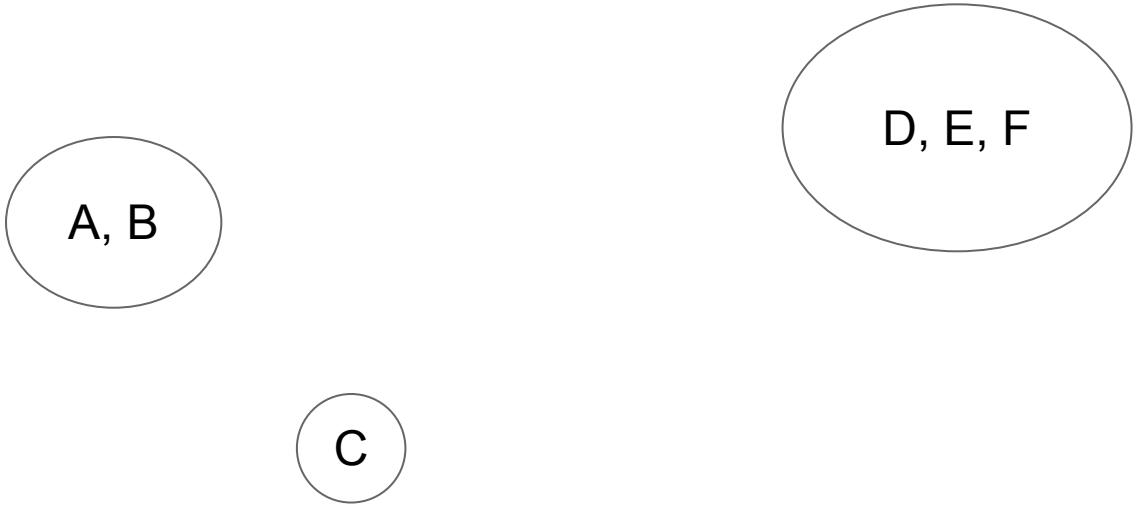
Disjoint Set (Union-find)

union(D, E)



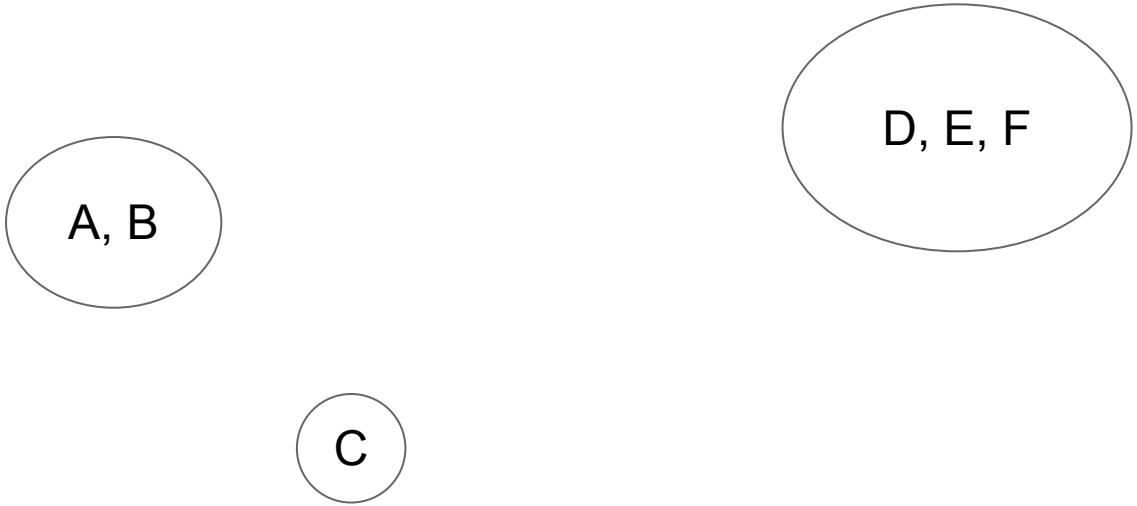
Disjoint Set (Union-find)

union(D, F)



Disjoint Set (Union-find)

same(A, B) => True
same(E, F) => True
same(A, D) => False



Disjoint Set (Union-find)

- Implementação

```
int pai[MAX_N];
int tam[MAX_N];

void initDS(int n){
    for(int i = 0; i < n; i++){
        pai[i] = i;
        tam[i] = 1;
    }
}
```

Disjoint Set (Union-find)

```
int find(int x){
    if (pai[x] == x)
        return x;
    return find(pai[x]);
}

int same(int x, int y){
    return (find(x) == find(y));
}
```

Disjoint Set (Union-find)

```
void union(int u, int v) {
    int a = find(u);
    int b = find(v);
    if (tam[a] < tam[b])
        swap(a, b);
    pai[b] = a;
    tam[a] += tam[b];
}
```

Disjoint Set (Union-find)

- Exemplo de problema: dado um conjunto de pessoas (numeradas de 1 a N), será realizada M operações que podem ser de dois tipos:
  - U x y (essa operação determina que x e y são da mesma família)
  - S x y (consulta se x e y são da mesma família ou não, se sim, deve-se imprimir “Y” na tela, caso contrário, “N”).

```
5 4
U 1 2
U 1 3
S 2 3        => Y
S 2 4        => N
S 5 4        => N
```

Disjoint Set (Union-find)

```
...

int main()
{
    int n, m, i, x, y;
    char c;
    cin >> n >> m;
    initDS(n);
```

Disjoint Set (Union-find)

```
    for(i = 0; i < m; i++)
    {
        cin >> c >> x >> y;
        if (c == 'U')
            union(x, y);
        else
            cout << (same(x, y) ? 'Y' : 'N') << endl;
    }
}
```

Referências

Aulas de Estrutura de Dados II da Profª Drª Marcia Aparecida Zanoli Meira e Silva.
Biblioteca de códigos de Thiago Alexandre Domingues de Souza.
Matemática Discreta e Suas Aplicações. Kenneth H. Rosen.
https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/trees.html
https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/mst.html
https://files.johnjq.com/slides/summer/union-find.pdf
http://professor.ufabc.edu.br/~leticia.bueno/classes/aed2/materiais/unionfind.pdf
https://www.ime.usp.br/~pf/mac0122-2002/aulas/union-find.html
http://wiki.icmc.usp.br/images/b/b6/SCC211Cap10.pdf
https://www.geeksforgeeks.org/union-find/

Referências

https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/graphs.html

http://www.inf.ufsc.br/grafos/definicoes/definicao.html

https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/shortestpaths.html

https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/cheapestpaths.html

http://professor.ufabc.edu.br/~leticia.bueno/classes/aa/materiais/caminhominimo.pdf

http://www.inf.ufsc.br/grafos/temas/custo-minimo/dijkstra.html

Referências

https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/dijkstra.html
http://www.deinf.ufma.br/~portela/ed211_Dijkstra.pdf
http://www.facom.ufu.br/~madriana/ED2/6-AlgDijkstra.pdf
http://www.lcad.icmc.usp.br/~jbatista/scc210/Aula_Grafos1.pdf
http://www.lcad.icmc.usp.br/~jbatista/scc210/Aula_Grafos2.pdf
http://www4.pucsp.br/~jarakaki/grafos/Aula2.pdf
https://miltonborba.org/Algf/Grafos.htm
https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/graphs.html
https://www.obm.org.br/content/uploads/2017/01/Nivel1_grafos_bruno.pdf
http://www.inf.ufsc.br/grafos/definicoes/definicao.html