Técnicas e Análise de Algoritmos

Grafos - Parte 02

Professor: **Jeremias Moreira Gomes**
E-mail: jeremias.gomes@idp.edu.br

Introdução

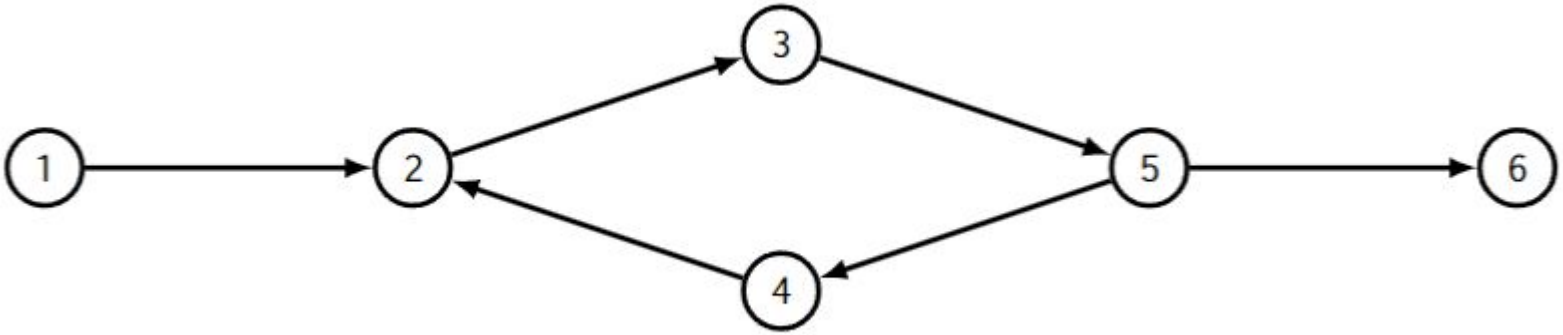
Recapitulação

- Grafo
  - O que é
  - Representação
  - Travessias
  - Componentes Conectadas

Detecção de Ciclos

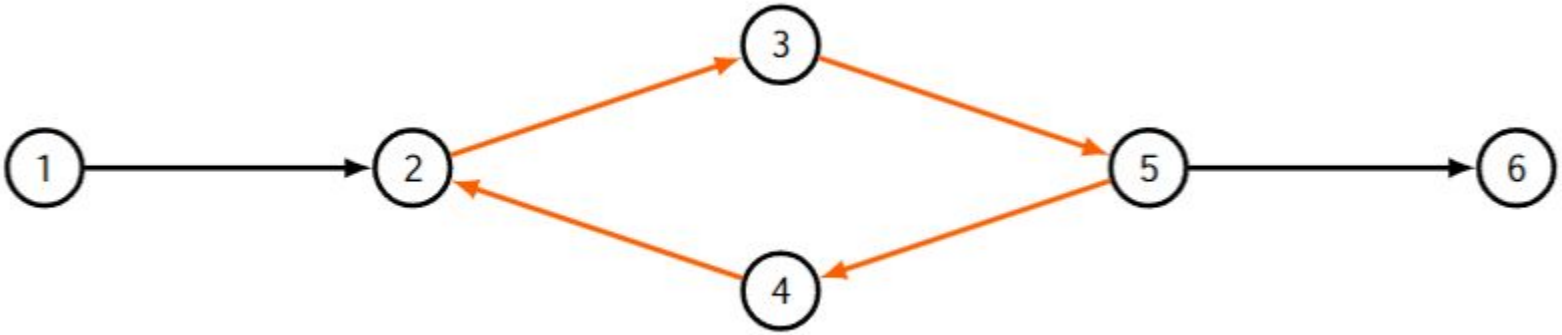
Ciclos

- Seja G um grafo direcionado. Um ciclo é um caminho, com três ou mais arestas distintas, cujos pontos de partida e de chegada são iguais



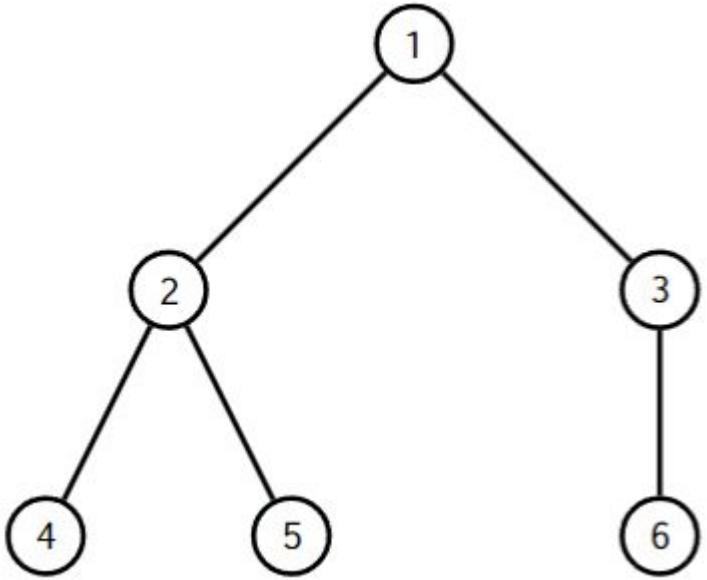
Ciclos

- Seja G um grafo direcionado. Um ciclo é um caminho, com três ou mais arestas distintas, cujos pontos de partida e de chegada são iguais



Grafos Acíclicos

- Um grafo é dito acíclico se não possui ciclos
  - Árvores são grafos acíclicos



Detecção de Ciclos

- Considere uma travessia por profundidade
  - Se, durante a travessia, um dos dos vizinhos v de u já foi visitado, e v não é o vértice p que descobriu u na travessia, então existe um ciclo que começa e termina em u e que passa por v

Detecção de Ciclos

- Implementação:
  - Mantenha o estado de “em visitação” e “visitados”
  - A DFS deve retornar verdadeiro caso um ciclo seja detectado
    - No início da DFS, marque o nó como “em visitacao” e ao final desative essa opção
    - Se uma DFS inicia com um nó já em visitação, então um ciclo foi detectado

Detecção de Ciclos

```
bool visitado[N + 1];
bool em_visitacao[N + 1];

bool detecta_ciclos()
{
    for (int u = 1; u <= N; u++) {
        if (dfs(u)) {
            return true;
        }
    }
    return false;

}
```

```
bool dfs(int u)
{
    if (em_visitacao[u]) return true;

    if (visitado[u]) return false;

    em_visitacao[u] = visitado[u] = true;

    for (auto v: G[u]) {
        if (dfs(v)) {
            return true;
        }
    }
    em_visitacao[u] = false;
    return false;
}
```

Caminhos Mínimos

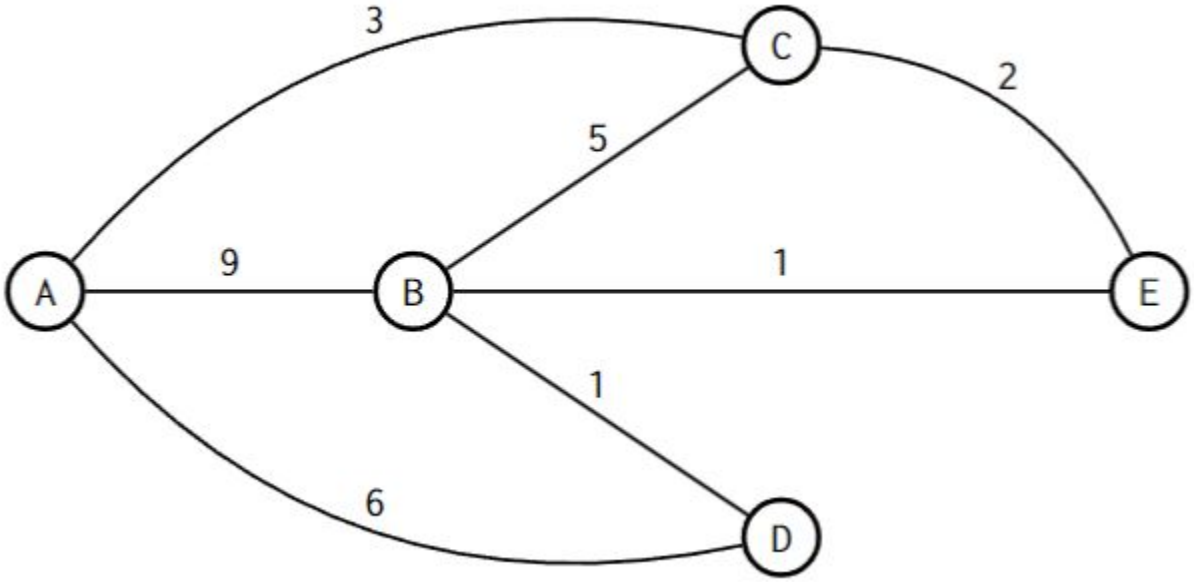
Caminho Mínimo

- Seja p um caminho entre os vértices u e v do grafo G, dizemos que p é um caminho mínimo de u a v se, para qualquer caminho q de u a v, vale que

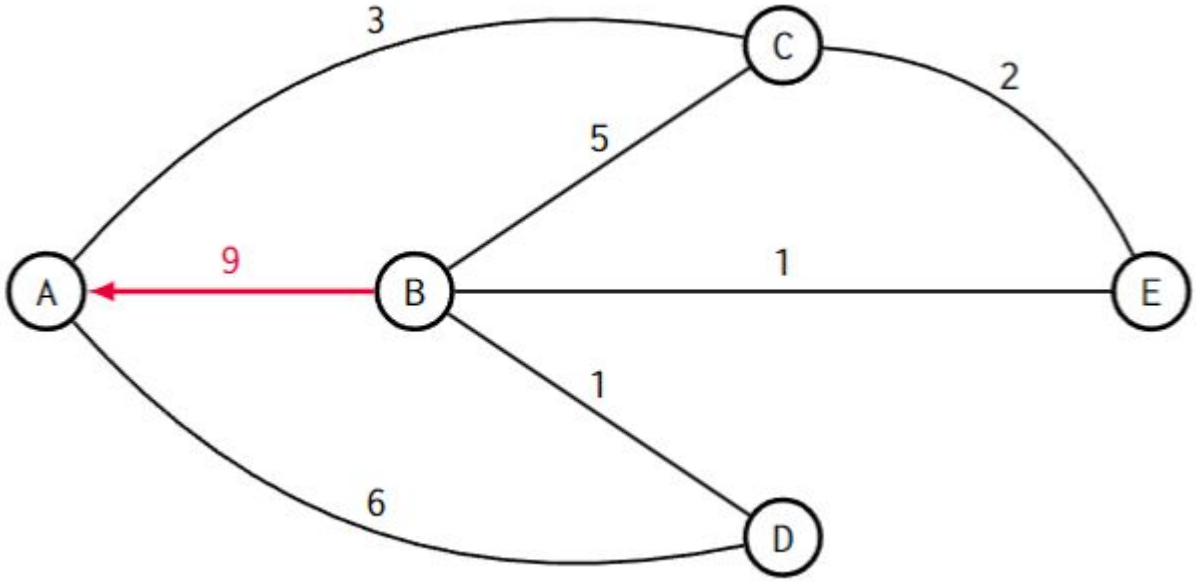
$$\sum_{e_i \in p} w(e_i) \leq \sum_{e_j \in q} w(e_j)$$

onde w(e) é o peso da aresta e.

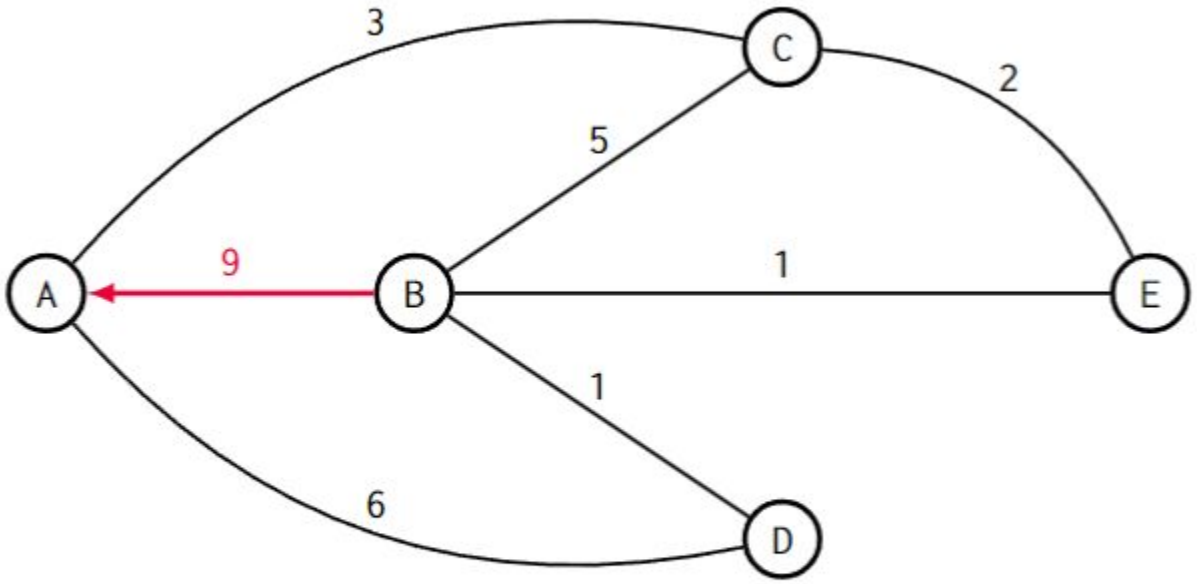
Caminho Mínimo - Exemplo (B, A)



Caminho Mínimo - Exemplo (B, A)

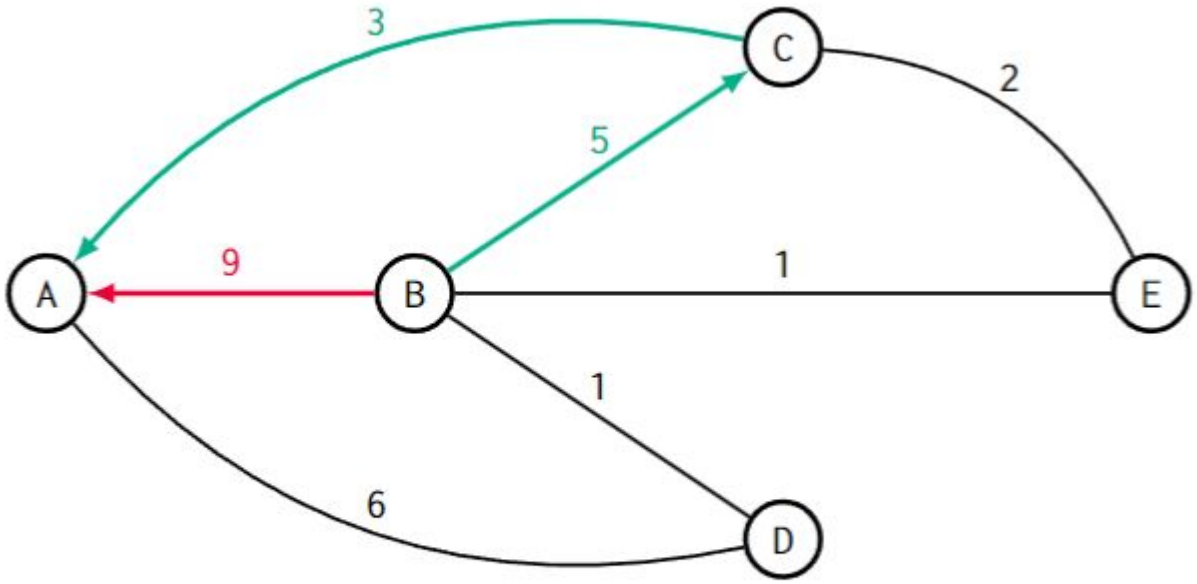


Caminho Mínimo - Exemplo (B, A)



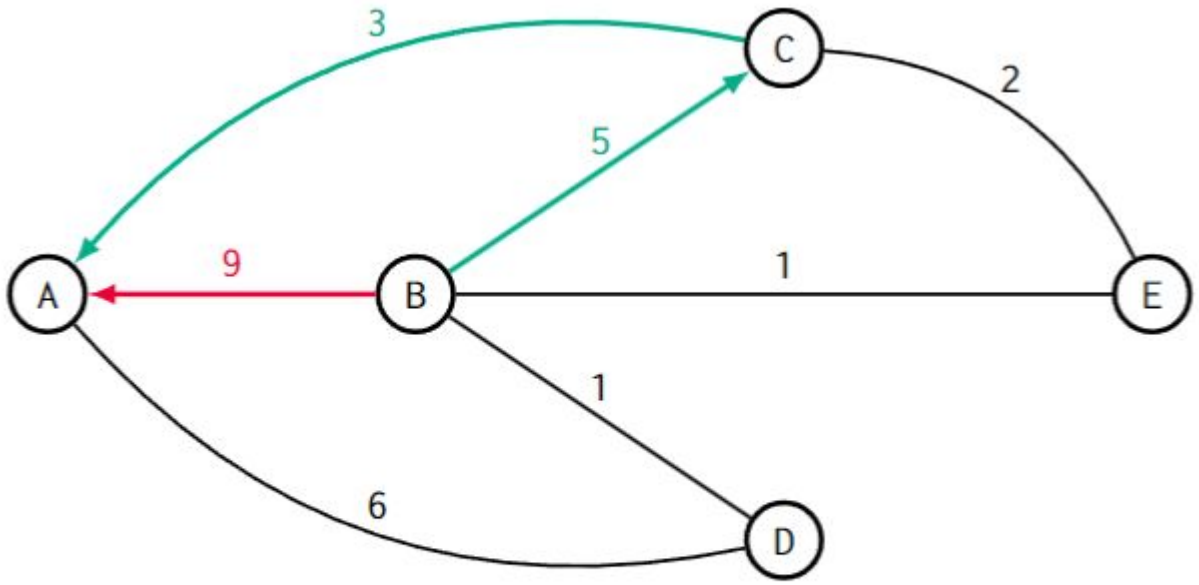
$$\sum_{e \in p_1} w(e) = 9$$

Caminho Mínimo - Exemplo (B, A)



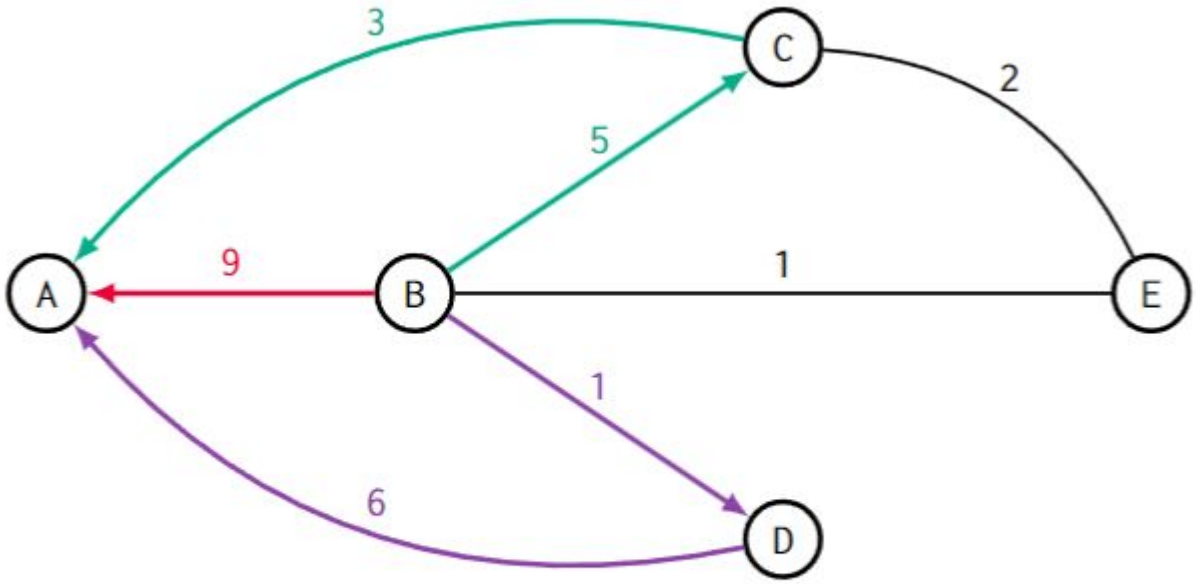
$$\sum_{e \in p_1} w(e) = 9$$

Caminho Mínimo - Exemplo (B, A)



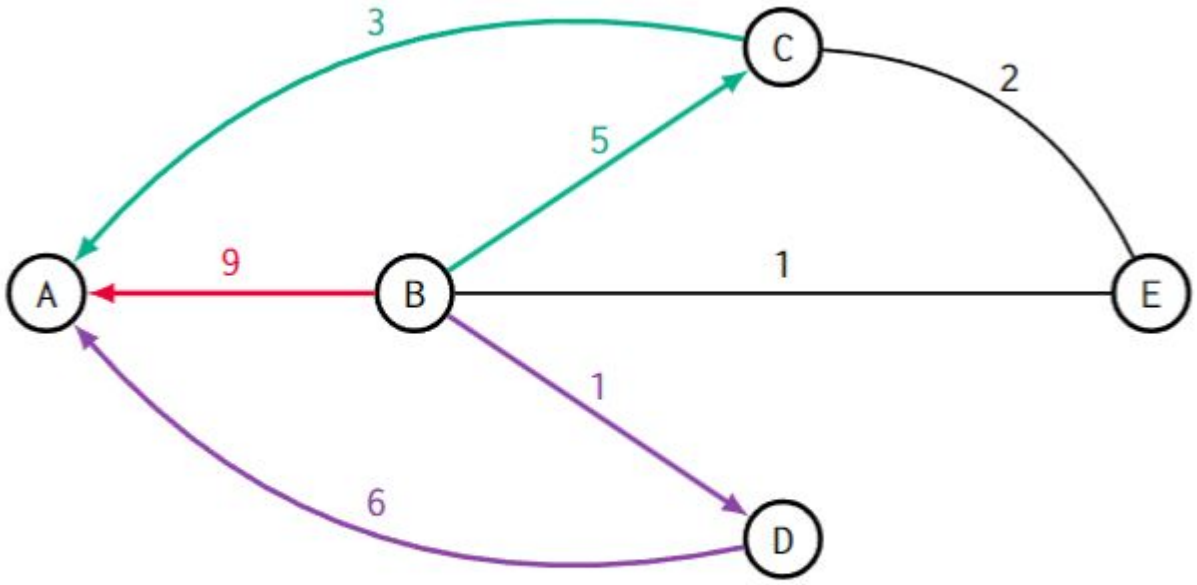
$$\sum_{e \in p_1} w(e) = 9 \qquad \sum_{e \in p_2} w(e) = 8$$

Caminho Mínimo - Exemplo (B, A)



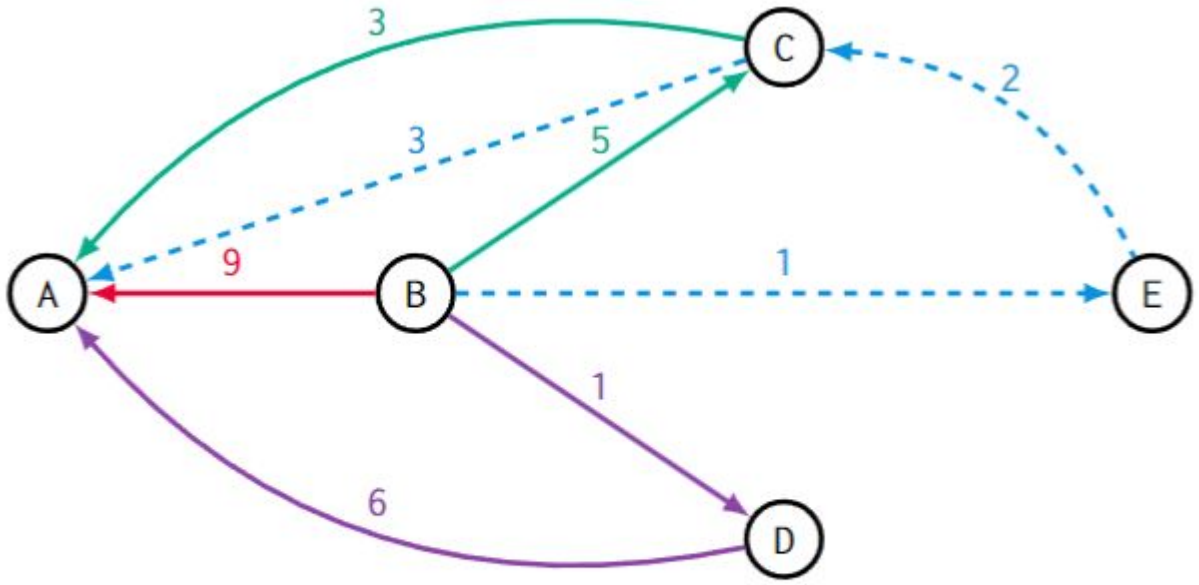
$$\sum_{e \in p_1} w(e) = 9 \qquad \sum_{e \in p_2} w(e) = 8$$

Caminho Mínimo - Exemplo (B, A)



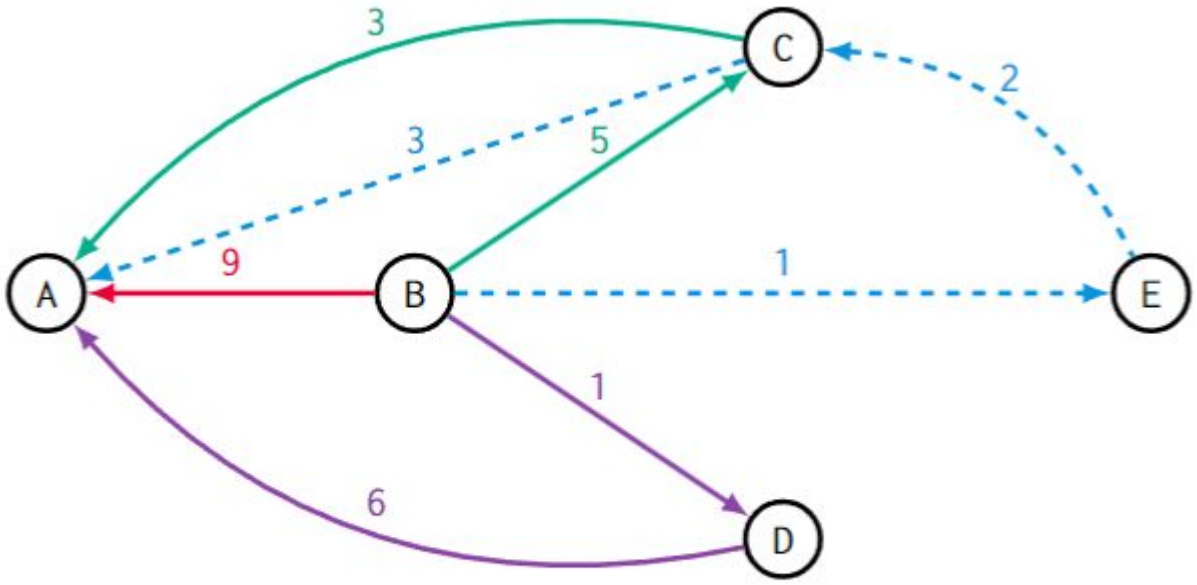
$$\sum_{e \in p_1} w(e) = 9 \qquad \sum_{e \in p_2} w(e) = 8 \qquad \sum_{e \in p_3} w(e) = 7$$

Caminho Mínimo - Exemplo (B, A)



$$\sum_{e \in p_1} w(e) = 9 \qquad \sum_{e \in p_2} w(e) = 8 \qquad \sum_{e \in p_3} w(e) = 7$$

Caminho Mínimo - Exemplo (B, A)

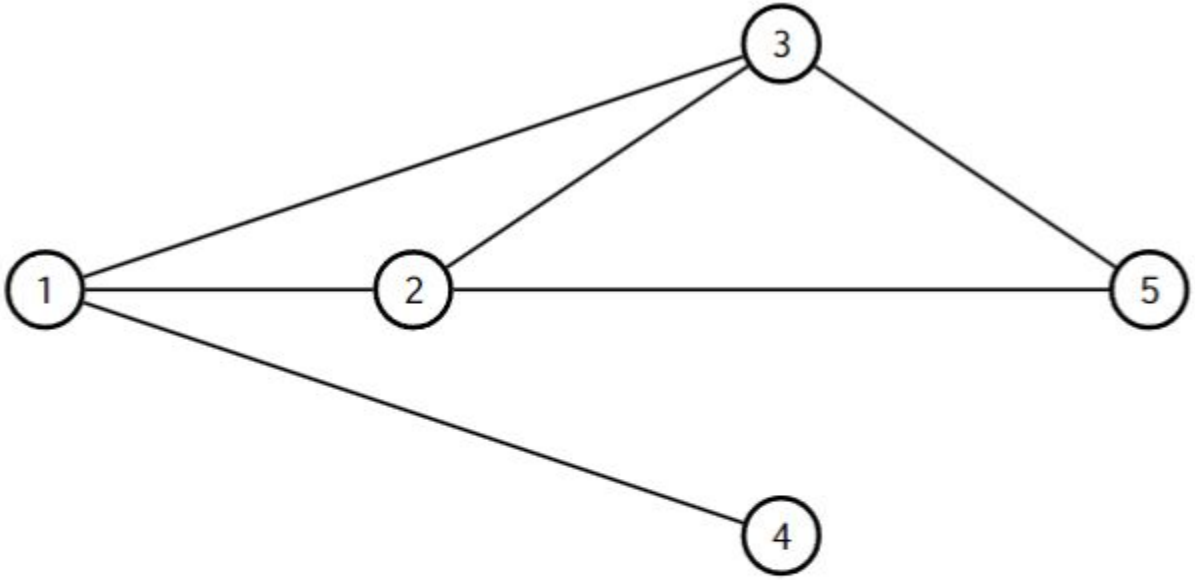


$$\sum_{e \in p_1} w(e) = 9 \qquad \sum_{e \in p_2} w(e) = 8 \qquad \sum_{e \in p_3} w(e) = 7 \qquad \sum_{e \in p_3} w(e) = 6$$

Caminhos Mínimos em Grafos não Ponderados

- Se G é não-ponderado, o custo de um caminho pode ser medido em arestas
- Isto equivale a considerar que todas as arestas de G tem peso 1
- Neste caso, uma BFS pode determinar a distância mínima entre o vértice de partida s e todos os vértices de G
- A BFS também poder ser usada em grafos cujas arestas tem o mesmo peso

Caminhos Mínimos em Grafos não Pon



	1	2	3	4	5
$\text{dist}(u, \text{E})$	∞	∞	∞	∞	∞

Caminhos Mínimos em Grafos não Pondera

```
const int oo = 0x3f3f3f3f;
vector<int>dist(N + 1, oo);
void bfs_dists(int u) {
    queue<int> fila;
    fila.push(u);
    dist[u] = 0;
    while (!fila.empty()) {
        u = fila.front();
        fila.pop();
        for (auto v: G[u]) {
            if (dist[v] == oo) {
                fila.push(v);
                dist[v] = dist[u] + 1;
            }
        }
    }
}
```

Caminhos Mínimos em Grafos Ponderados

- Se G é ponderado, existem alguns algoritmos para determinar a distância mínima
  - Se G **não** tem arestas com pesos negativos, utiliza-se o **algoritmo de Dijkstra** (mais importante e mais utilizado)
  - Se G possui arestas negativas, é utilizado o **algoritmo de Bellman-Ford** (caso mais geral)
  - Para calcula as distâncias mínimas entre todos os pares de vértices de G, existe o **algoritmo de Floyd-Warshall**

Algoritmo de Dijkstra

- Desenvolvido por Edsger Wybe Dijkstra
- Computa o caminho mínimo de todos os vértices de G(V, E) a um dado nó s
- **Processa apenas grafos com arestas não-negativas**
- Eficiência: cada aresta é processada uma única vez
- Complexidade: O(E + V log V)

Algoritmo de Dijkstra - Pseudocódigo

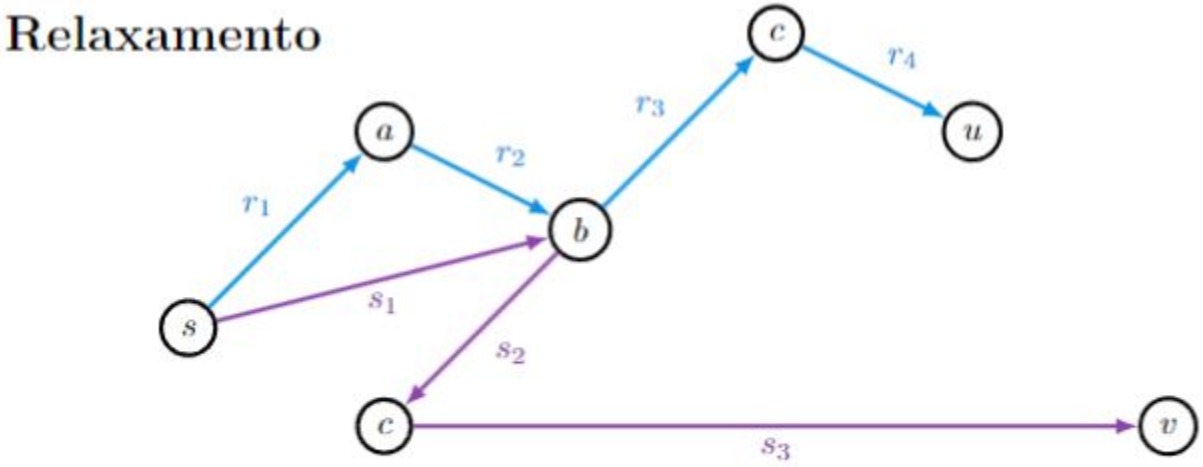
- Entrada: um grafo ponderado G(V, E) e um vértice s ∈ V
- Saída: um vetor d tal que d[u] é a distância mínima entre s e u

1. Faça d[s] = 0, d[u] = ∞ se u ≠ s e seja U = V
2. Enquanto U ≠ ∅:
   - Seja u ∈ U o vértice mais próximo de s em U
   - Relaxe as distâncias usando as arestas que partem de u
   - Remova u de U
3. Retorne d

Algoritmo de Dijkstra

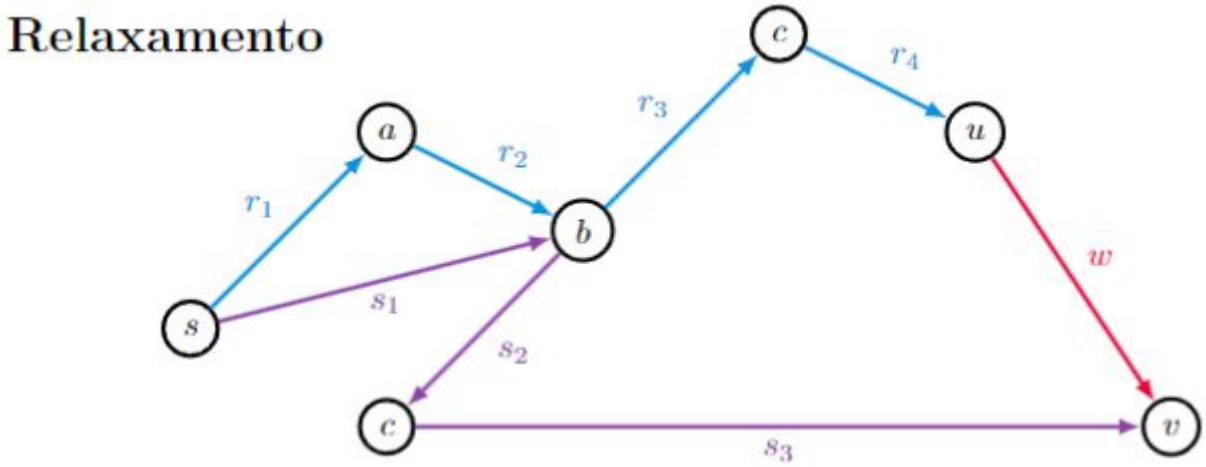
- Esse algoritmo usa uma ideia chamada de **relaxamento**, onde as distâncias mínimas vão sendo atualizadas a medida em que o algoritmo verifica novos vértices, de acordo com a sua proximidade

Algoritmo de Dijkstra

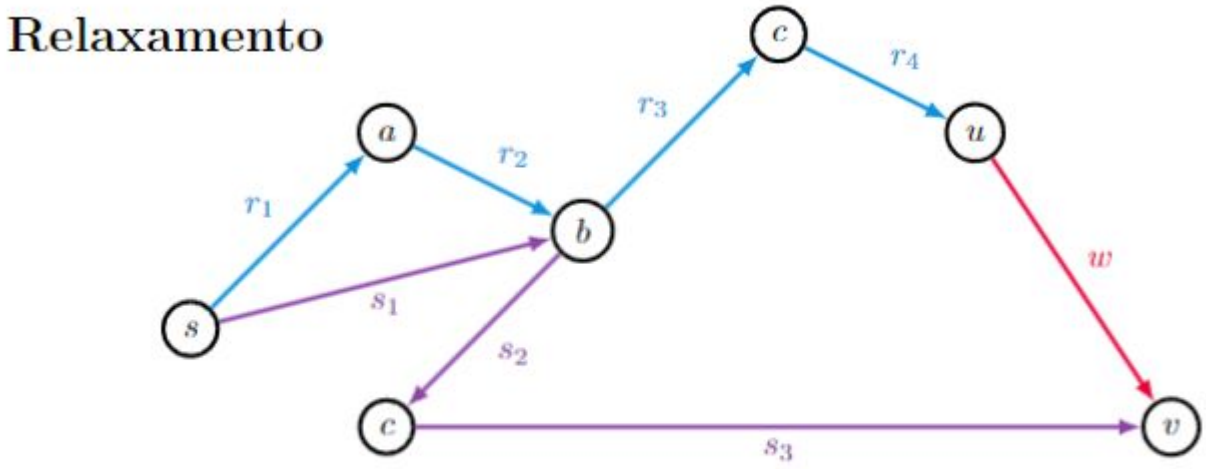


$$dist(s,u) = \sum_{i=1}^{4} r_i \qquad dist(s,v) = \sum_{j=1}^{3} s_i$$

Algoritmo de Dijkstra



Algoritmo de Dijkstra

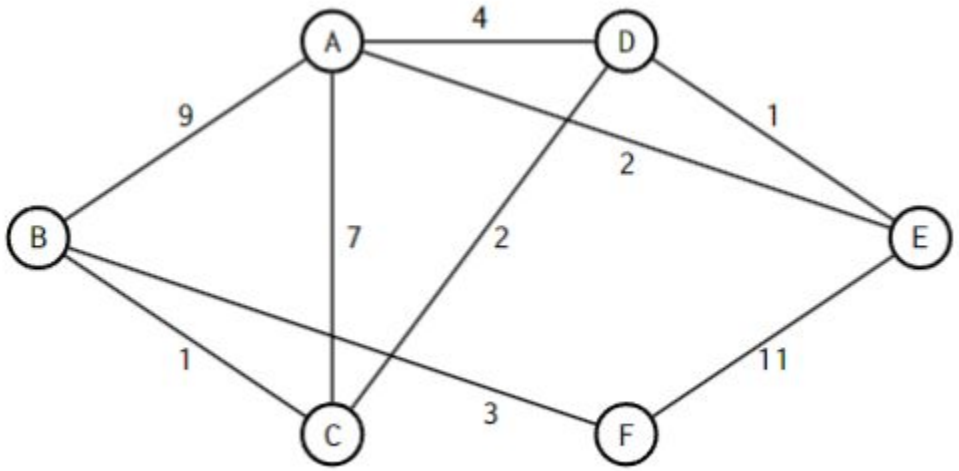


Se $dist(s,u) + w < dist(s,v)$, faça $dist(s,v) = dist(s,u) + w$

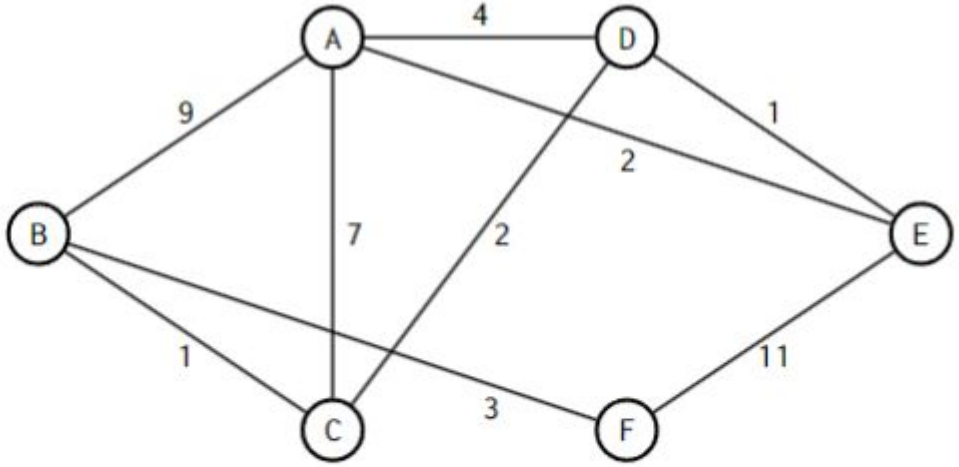
Algoritmo de Dijkstra

- Na implementação do algoritmo, é mantido um vetor de distâncias, e um conjunto de vértices
  - O vetor é inicializado com distâncias infinitas e vão reduzindo ao longo da execução
  - O conjunto de vértices tem seus elementos removidos à medida que estes vão sendo visitados

Algoritmo de Dijkstra



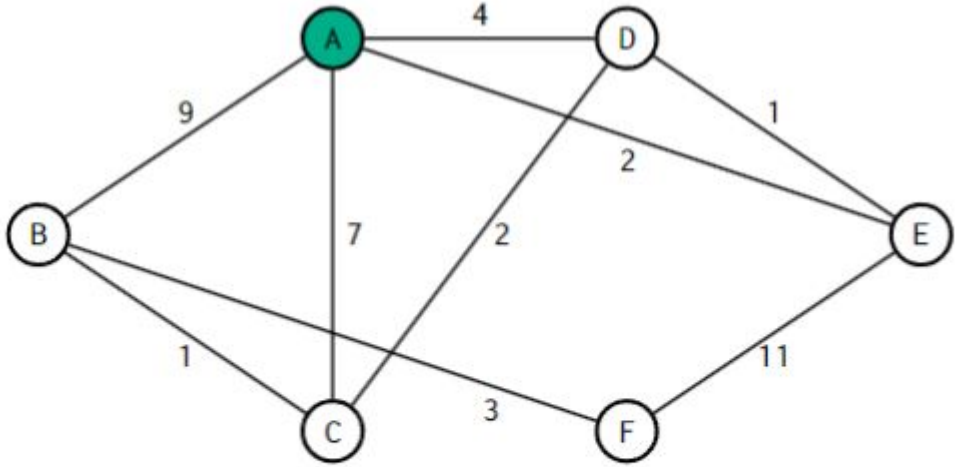
Algoritmo de Dijkstra



	A	B	C	D	E	F
$\text{dist}(u, \text{A})$	0	∞	∞	∞	∞	∞

$U = \{ \text{A}, \text{B}, \text{C}, \text{D}, \text{E}, \text{F} \}$

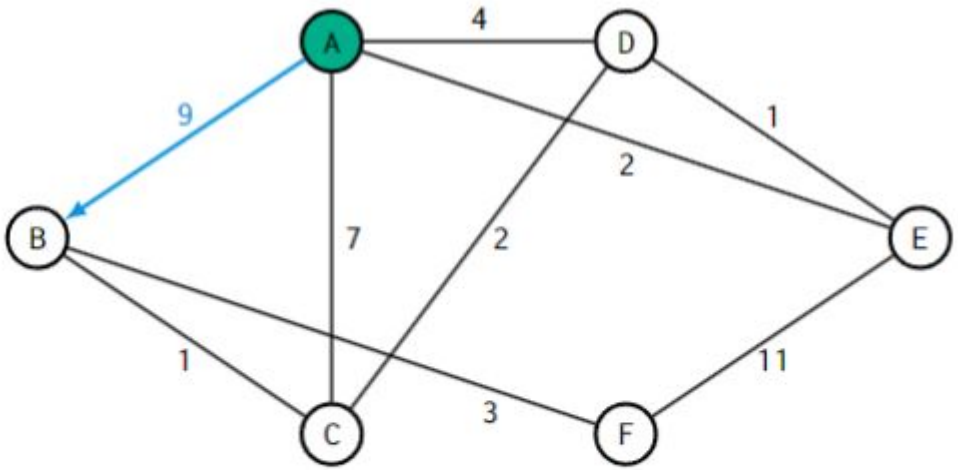
Algoritmo de Dijkstra



	A	B	C	D	E	F
$\text{dist}(u, \text{A})$	0	∞	∞	∞	∞	∞

$U = \{ \text{B}, \text{C}, \text{D}, \text{E}, \text{F} \}$

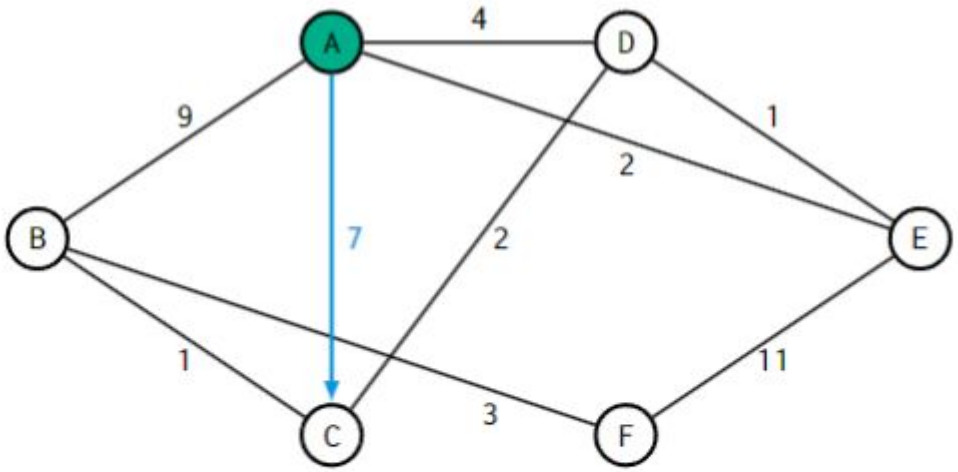
Algoritmo de Dijkstra



	A	B	C	D	E	F
$\text{dist}(u, \text{A})$	0	**9**	∞	∞	∞	∞

$U = \{ \text{B}, \text{C}, \text{D}, \text{E}, \text{F} \}$

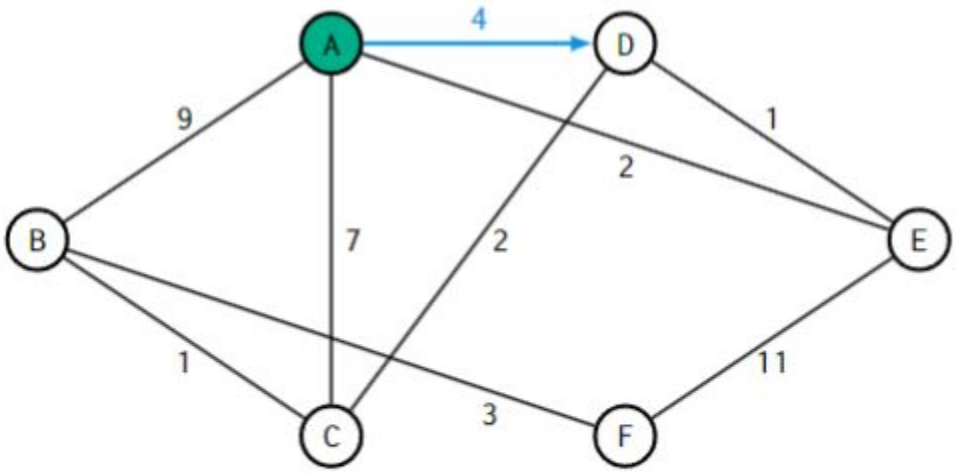
Algoritmo de Dijkstra



	A	B	C	D	E	F
$\text{dist}(u, \text{A})$	0	9	**7**	∞	∞	∞

$U = \{ \text{B}, \text{C}, \text{D}, \text{E}, \text{F} \}$

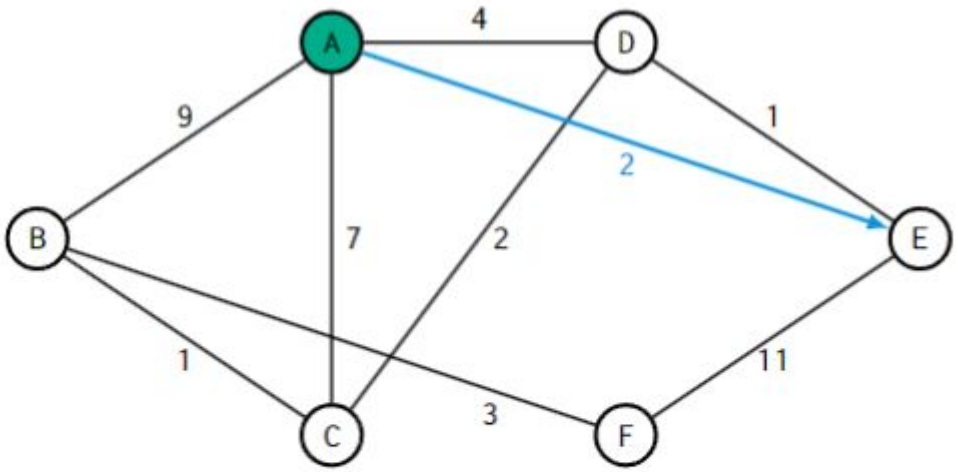
Algoritmo de Dijkstra



	A	B	C	D	E	F
dist(u, A)	0	9	7	**4**	∞	∞

$U = \{ \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{F} \}$

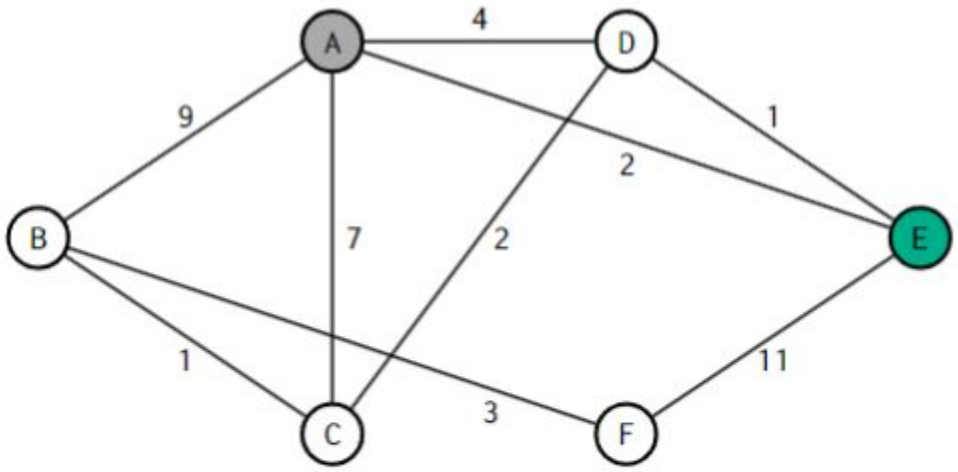
Algoritmo de Dijkstra



	A	B	C	D	E	F
dist(u, A)	0	9	7	4	**2**	∞

$U = \{$ B, C, D, E, F $\}$

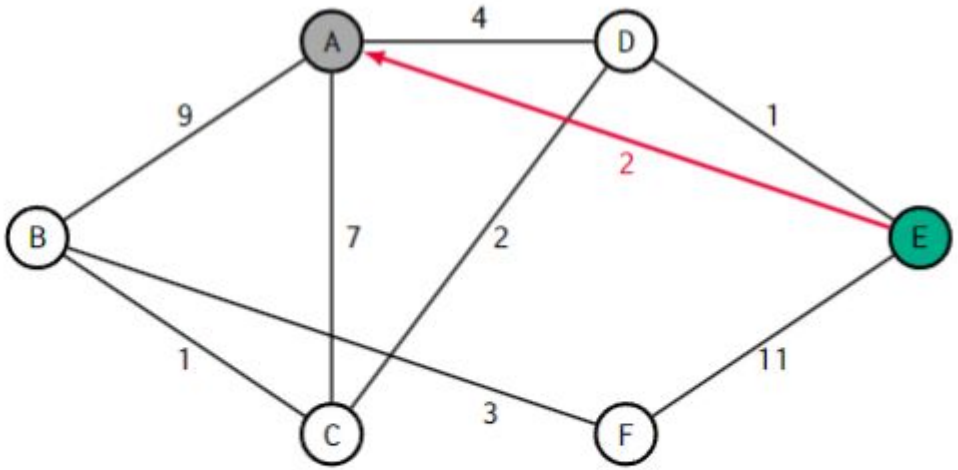
Algoritmo de Dijkstra



	A	B	C	D	E	F
$\text{dist}(u, \text{A})$	0	9	7	4	2	∞

$U = \{ \text{B}, \text{C}, \text{D}, \text{F} \}$

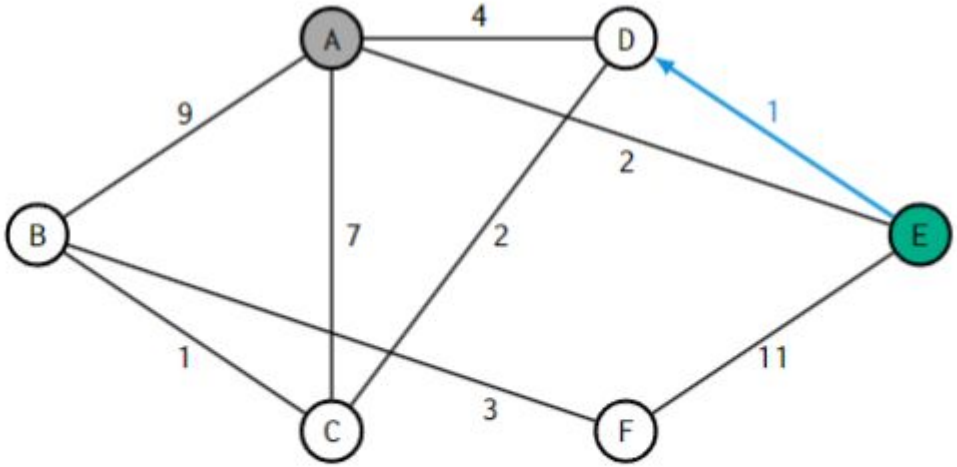
Algoritmo de Dijkstra



	A	B	C	D	E	F
$\text{dist}(u, \text{A})$	0	9	7	4	2	∞

$U = \{ \text{B}, \text{C}, \text{D}, \text{F} \}$

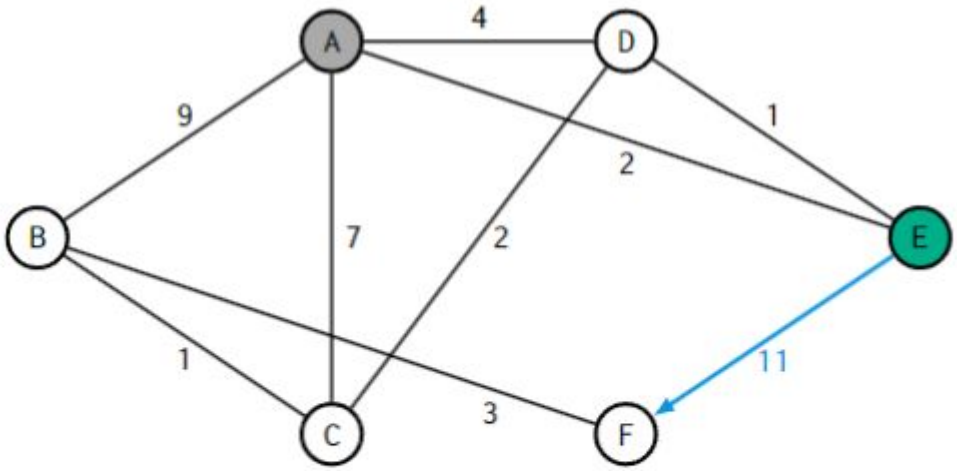
Algoritmo de Dijkstra



	A	B	C	D	E	F
$\text{dist}(u, \text{A})$	0	9	7	**3**	2	∞

$U = \{ \text{B}, \text{C}, \text{D}, \text{F} \}$

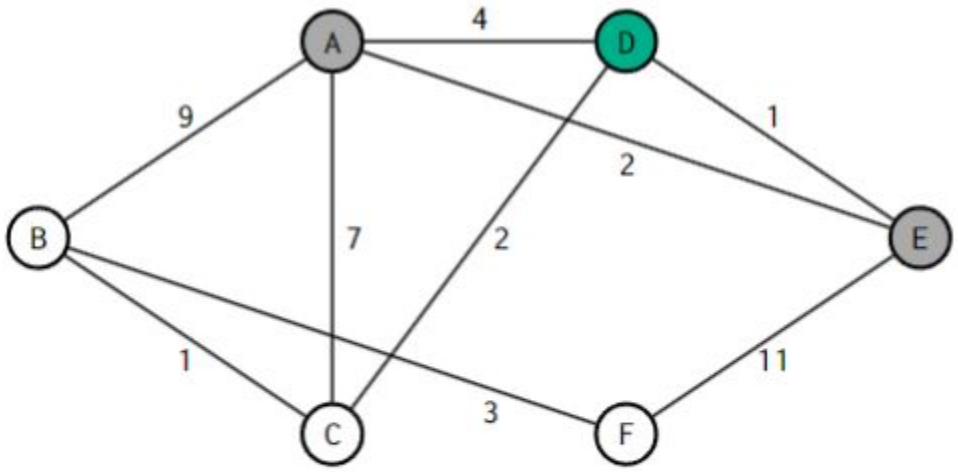
Algoritmo de Dijkstra



	A	B	C	D	E	F
dist(u, A)	0	9	7	3	2	**13**

$U = \{ \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{F} \}$

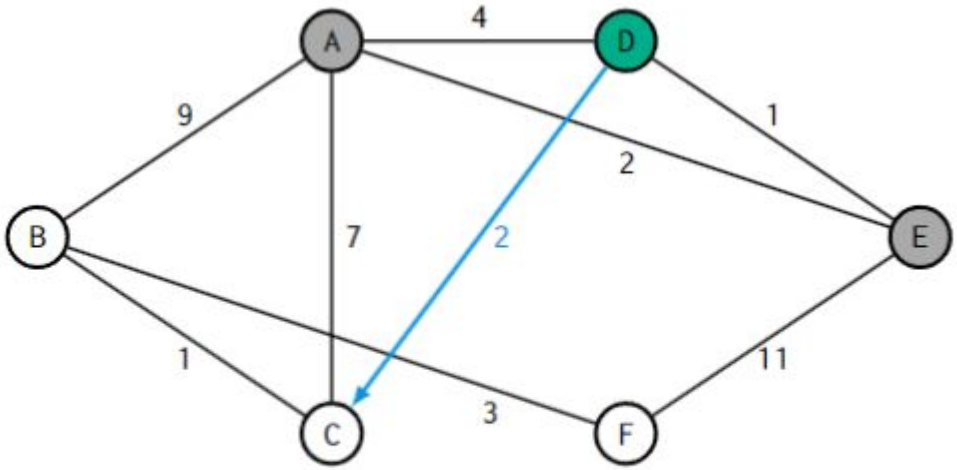
Algoritmo de Dijkstra



	A	B	C	D	E	F
$\text{dist}(u, \text{A})$	0	9	7	3	2	13

$U = \{ \text{B}, \text{C}, \text{F} \}$

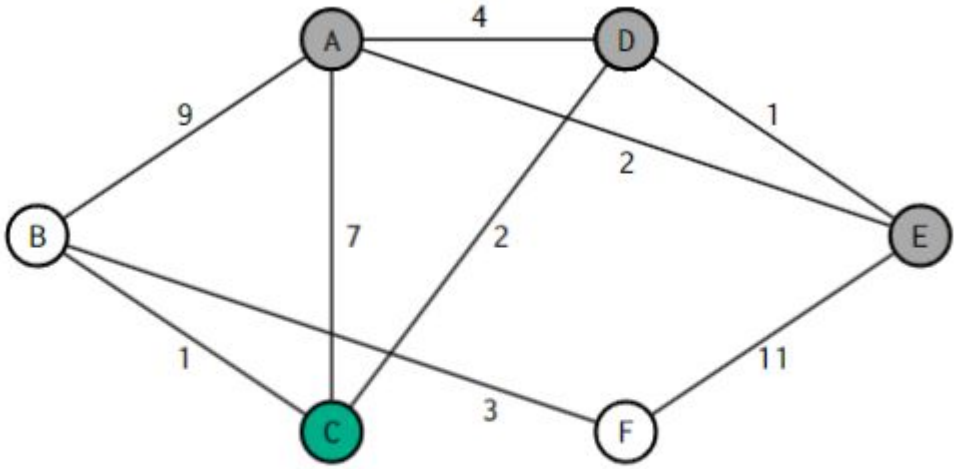
Algoritmo de Dijkstra



	A	B	C	D	E	F
$\text{dist}(u, \text{A})$	0	9	**5**	3	2	13

$U = \{ \text{B}, \text{C}, \text{F} \}$

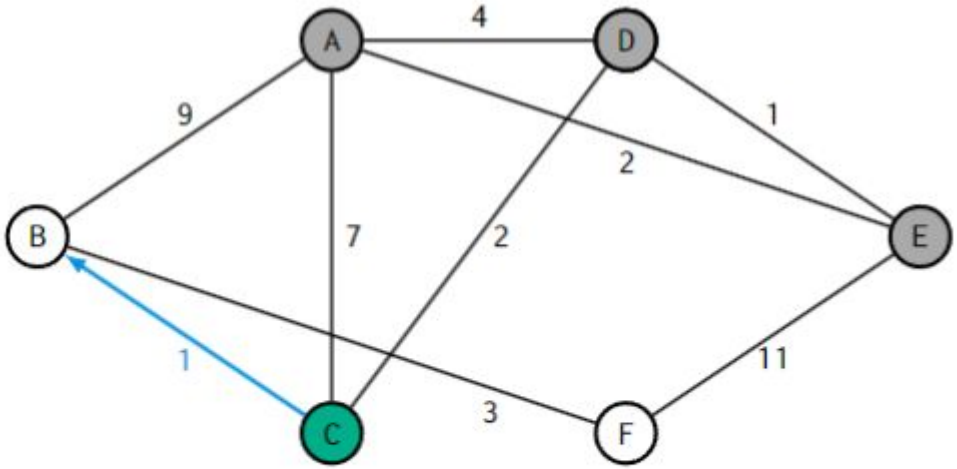
Algoritmo de Dijkstra



	A	B	C	D	E	F
dist(u, A)	0	9	5	3	2	13

$U = \{ \mathsf{B}, \mathsf{F} \}$

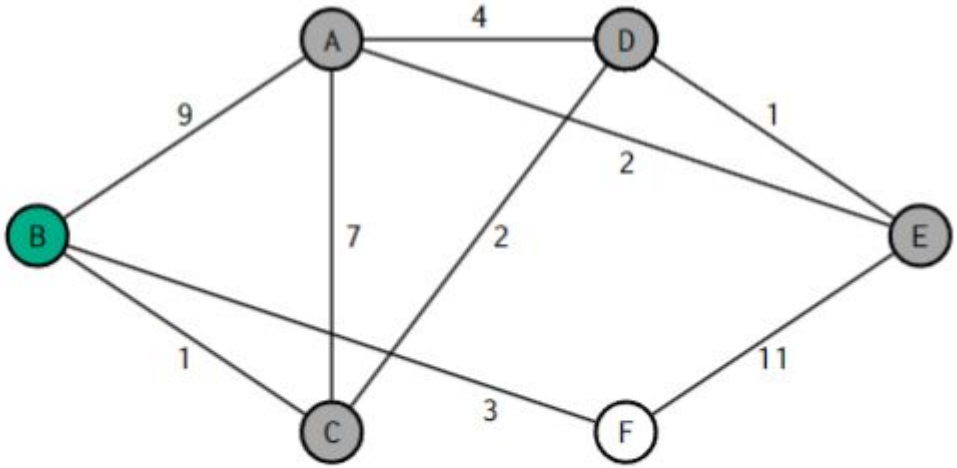
Algoritmo de Dijkstra



	A	B	C	D	E	F
dist(u, A)	0	**6**	5	3	2	13

$U = \{ \mathrm{B}, \mathrm{F} \}$

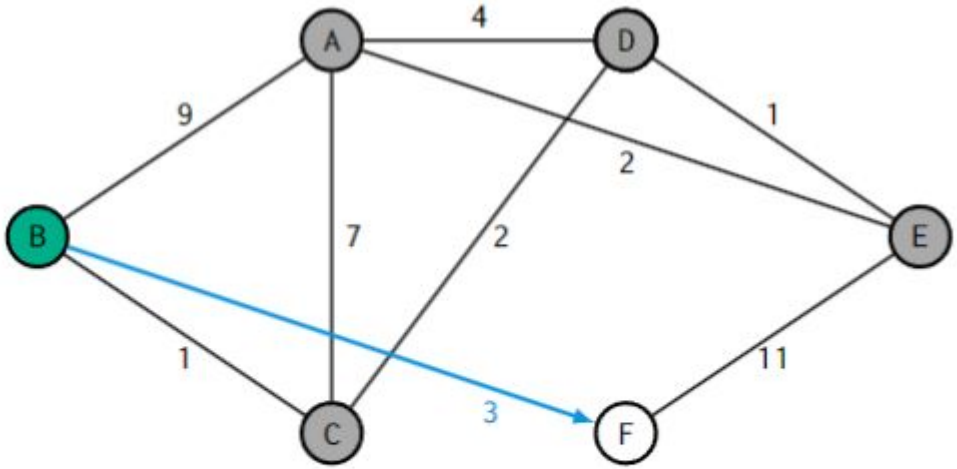
Algoritmo de Dijkstra



	A	B	C	D	E	F
$\text{dist}(u, \text{A})$	0	6	5	3	2	13

$U = \{ \text{F} \}$

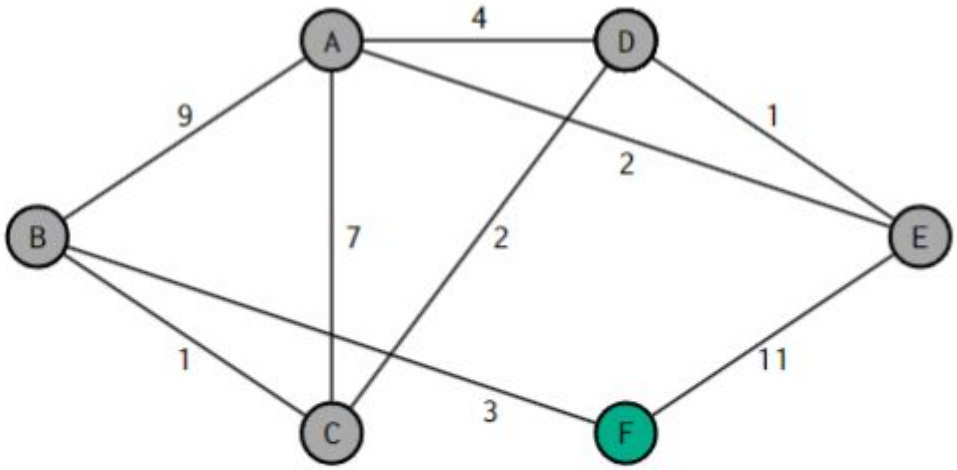
Algoritmo de Dijkstra



	A	B	C	D	E	F
$\text{dist}(u, \text{A})$	0	6	5	3	2	**9**

$U = \{ \text{F} \}$

Algoritmo de Dijkstra



	A	B	C	D	E	F	
$\text{dist}(u, \text{A})$	0	6	5	3	2	9	$U = \emptyset$

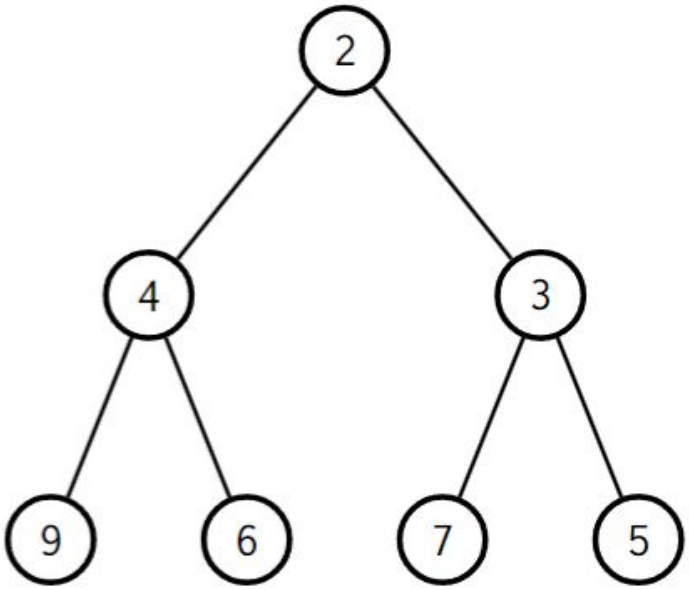
Algoritmo de Dijkstra - Identificação do Vértice Mais Próximo

- Uma informação que não foi explorada é, dado um vértice, encontrar qual é o vizinho mais próximo desse vértice
- Para resolver esse problema, pode-se utilizar uma min-heap, que irá manter na estrutura um conjunto de vizinhos dos vértices visitados, extraindo sempre aquele de menor distância no momento da descoberta

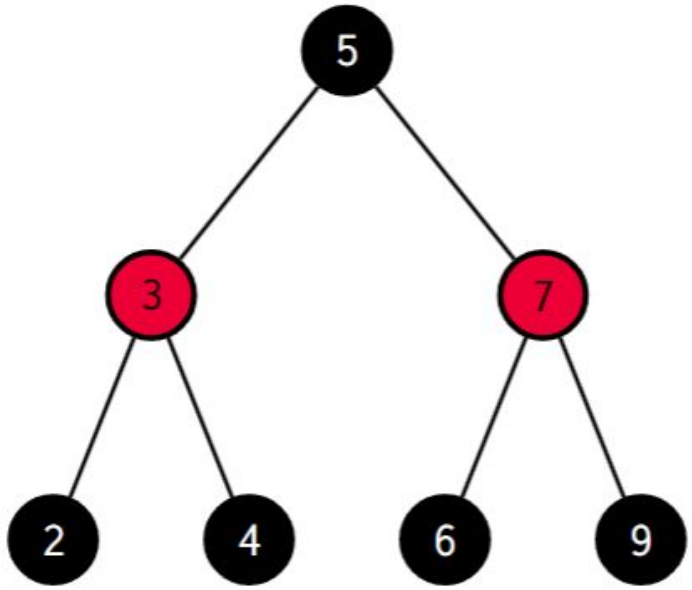
Algoritmo de Dijkstra - Identificação do Vértice Mais Próximo

- Também é possível utilizar uma árvore auto balanceada
  - Uma árvore red-black ou outra estrutura
  - Se utilizar um set, ter o cuidado de dar manutenção na inserção de um elemento que já se encontra na estrutura, atualizando, se necessário, a menor distância

Algoritmo de Dijkstra - Identificação do Vértice Mais Próximo



min heap
(priority_queue)



red-black tree
(set)

```
void dijkstra(int origem)
{
    memset(dist, INF, sizeof(dist));
    dist[origem] = 0;

    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
    pq.push({0, origem});

    while (!pq.empty()) {
        auto [d, u] = pq.top(); pq.pop();
        if (d > dist[u]) {
            continue;
        }
        for (auto [v, w]: G[u]) {
            if (dist[v] > dist[u] + w) {
                dist[v] = dist[u] + w;
                pq.push({ dist[v], v });
            }
        }
    }
}
```

Algoritmo de Dijkstra

- Um detalhe importante é que o algoritmo de dijkstra computa as distâncias mínimas, mas não quais são os caminhos mínimos
- É possível determinar um caminho mínimo, mantendo uma estrutura que contém a identificação de qual vértice levou a ter aquela distância (um vetor de predecessores ou o pai daquele vértice)

```cpp
void djikstra(int origem)
{
    memset(dist, INF, sizeof(dist));
    dist[origem] = 0;
    predecessor[origem] = origem;

    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
    pq.push({0, origem});

    while (!pq.empty()) {
        auto [d, u] = pq.top(); pq.pop();
        if (d > dist[u]) {
            continue;
        }
        for (auto [v, w]: G[u]) {
            if (dist[v] > dist[u] + w) {
                dist[v] = dist[u] + w;
                predecessor[v] = u;
                pq.push({ dist[v], v });
            }
        }
    }
}
```

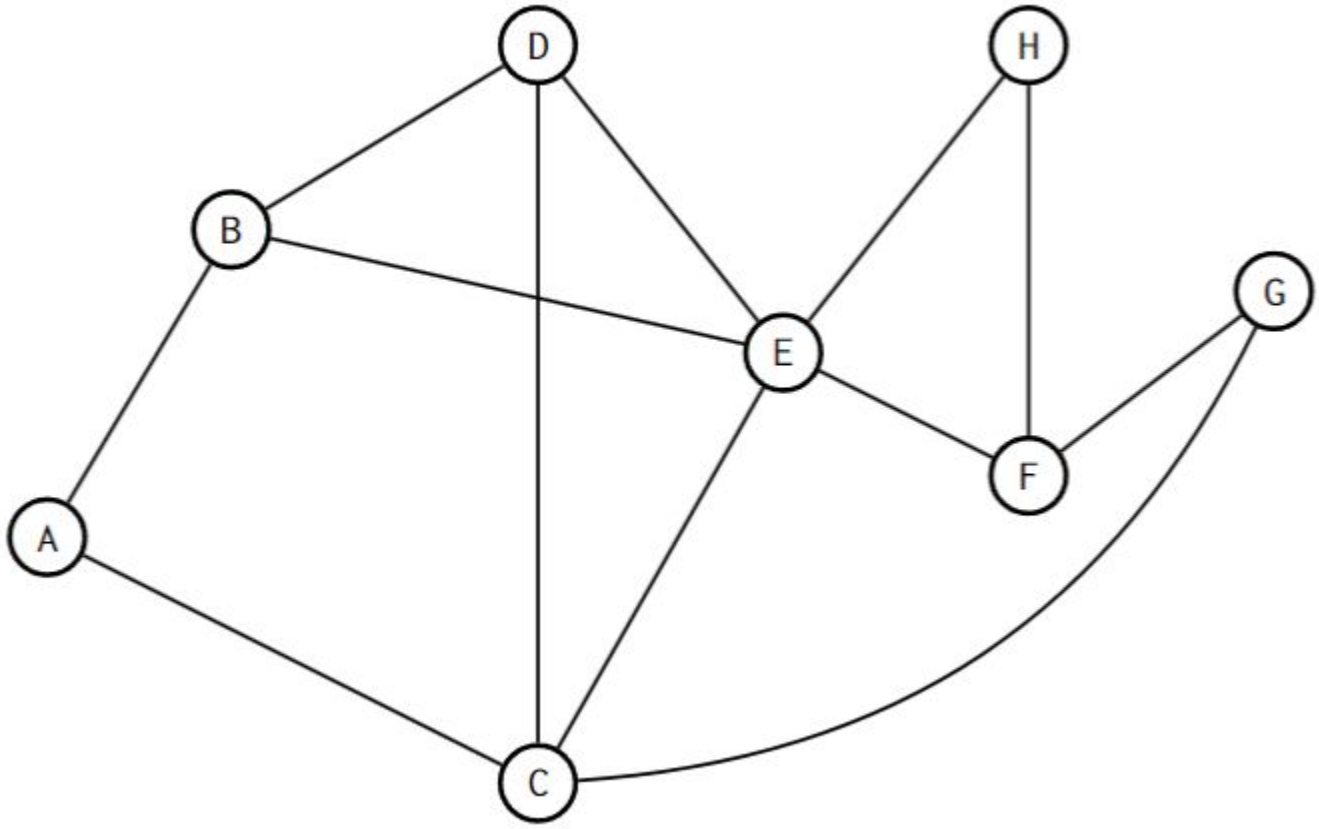
```cpp
vector<pair<int, int>> caminho_minimo(int origem, int destino)
{
    vector<pair<int, int>> caminho;
    int v = destino;

    while (v != origem) {
        caminho.push_back({predecessor[v], v});
        v = predecessor[v];
    }

    reverse(caminho.begin(), caminho.end());

    return caminho;
}
```

Árvore Geradora Mínima

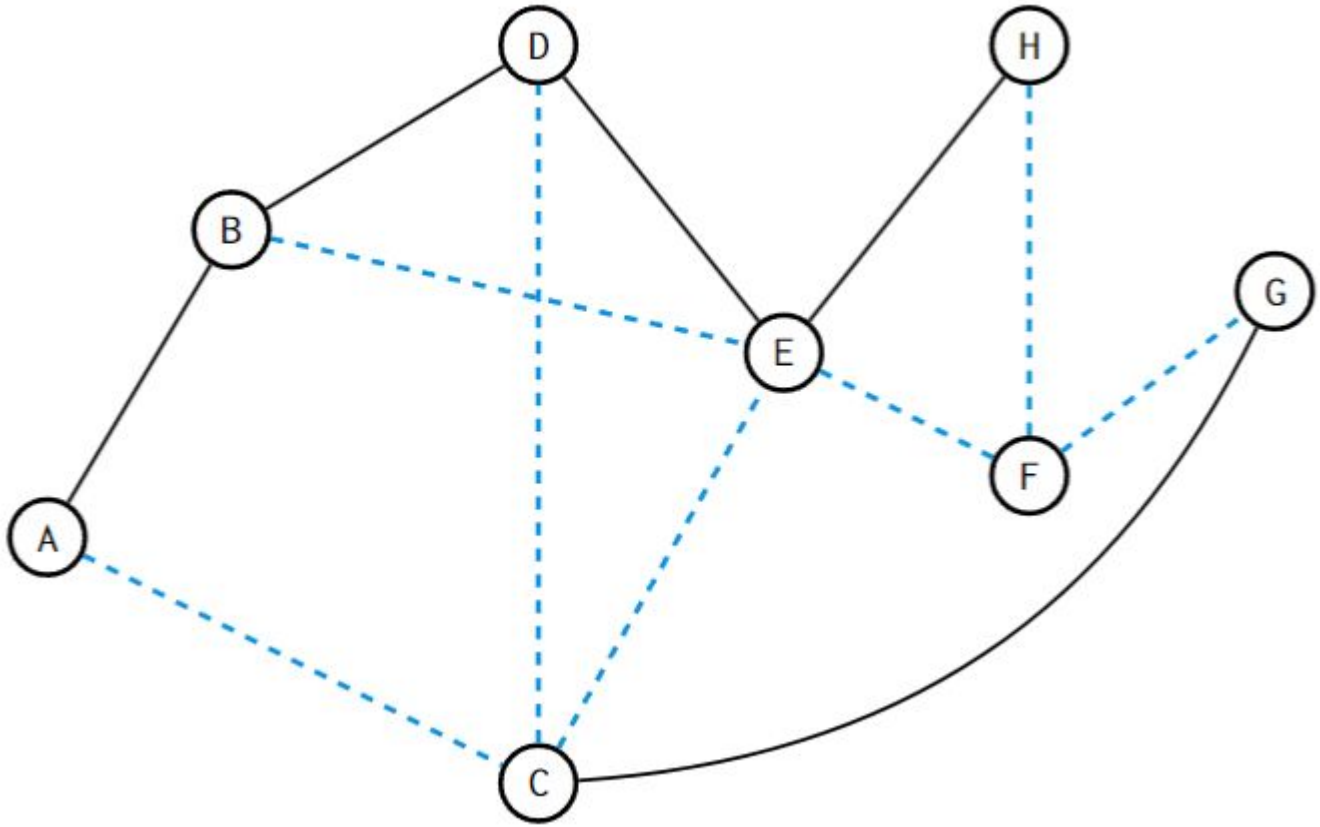
Árvores Geradoras

- Seja G(V, E) um grafo
- Uma árvore geradora de G é um subgrafo T (V, E') de G tal que T é uma árvore que contém todos os vértices de G

Árvores Geradoras



Árvores Geradoras



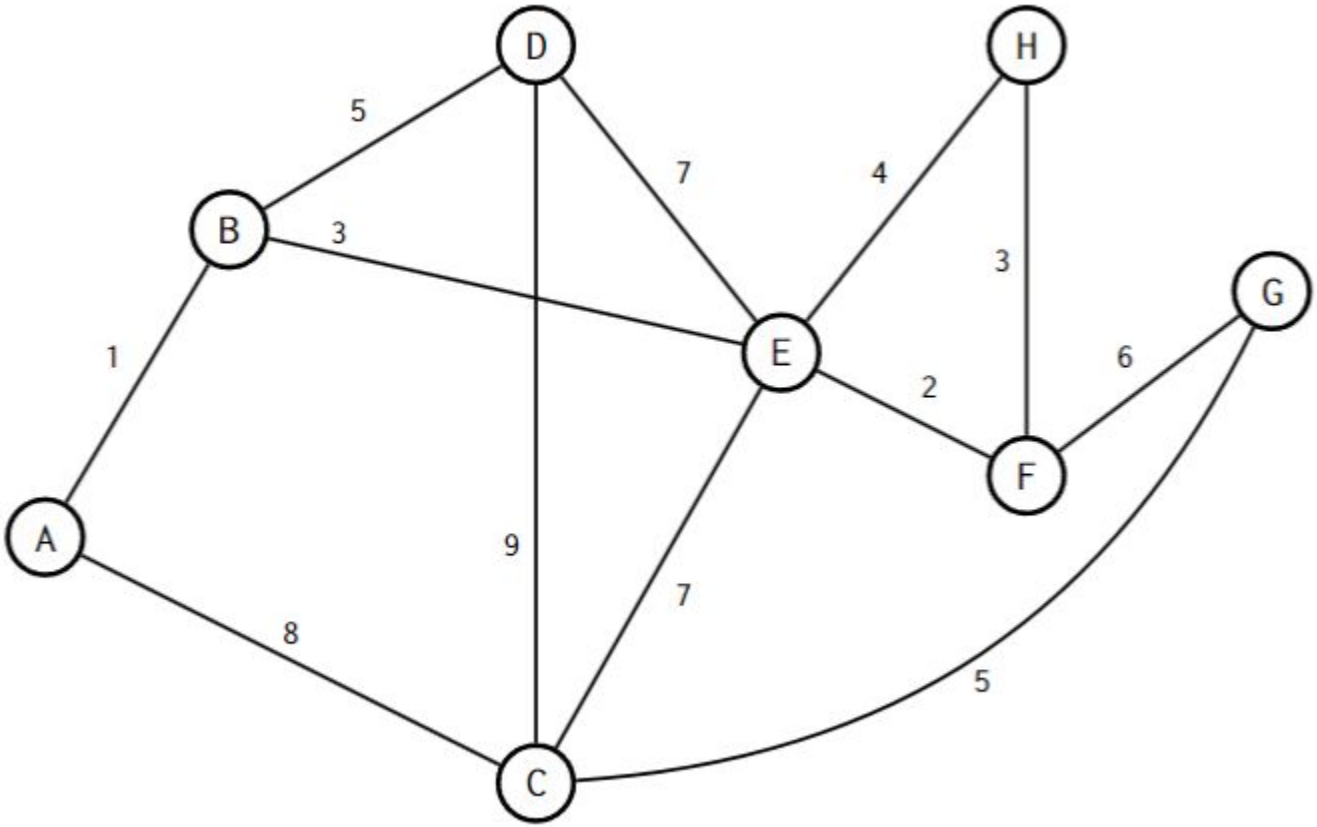
Árvore Geradora Mínima (MST)

- Seja G(V, E) um grafo ponderado (não direcionado)
- Uma árvore geradora T (V, E') de G é uma árvore geradora mínima (Minimum Spanning Tree) de G se a soma

$$c(T) = \sum_{e \in E'} w(e)$$

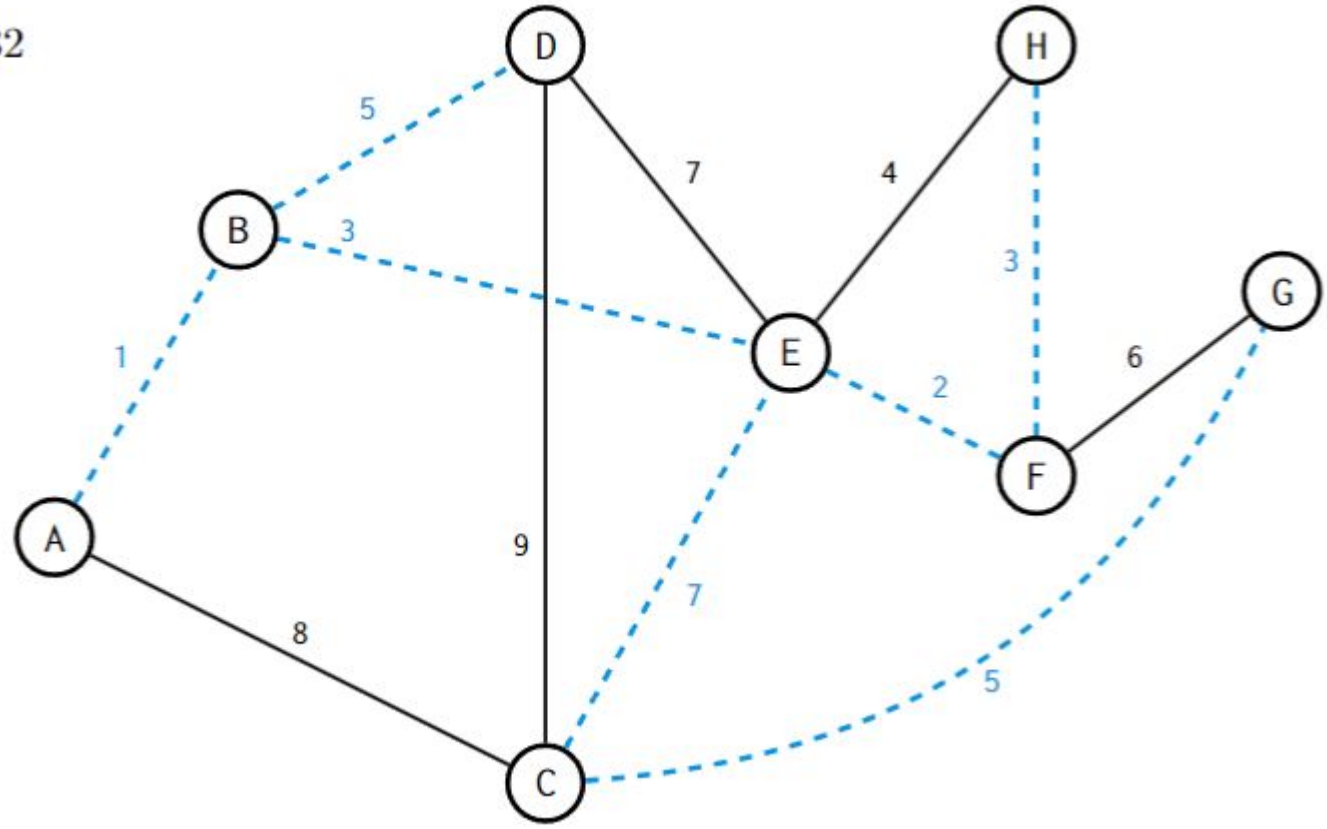
é mínima

Árvore Geradora Mínima (MST)



Árvore Geradora Mínima (MST)

$c(T) = 32$



Árvore Geradora Mínima (MST)

- Propriedades:
  - A MST é única apenas se todos os pesos forem distintos
  - A árvore geradora máxima pode ser obtida invertendo os sinais dos pesos de todas das arestas

Árvore Geradora Mínima (MST)

- MSTs são utilizadas em diversas áreas que envolvem, principalmente, tráfego:
  - Construção de redes (telefônicas ou de outros tipos) ou malhas elétricas
  - Segmentação de imagens
  - Sistemas de Informações Geográficos
  - etc

Árvore Geradora Mínima (MST)

- Existem dois algoritmos mais conhecidos para a MST:
  1. Algoritmo de Kruskal
     - Algoritmo guloso que une vértices enquanto florestas
     - Utiliza uma estrutura chamara Disjoint Set Union (DSU)
  2. Algoritmo de Prim (detalhado a seguir)
     - Algoritmo guloso que une vértices não visitados
     - Utiliza heaps para identificar vértices próximos

Algoritmo de Prim

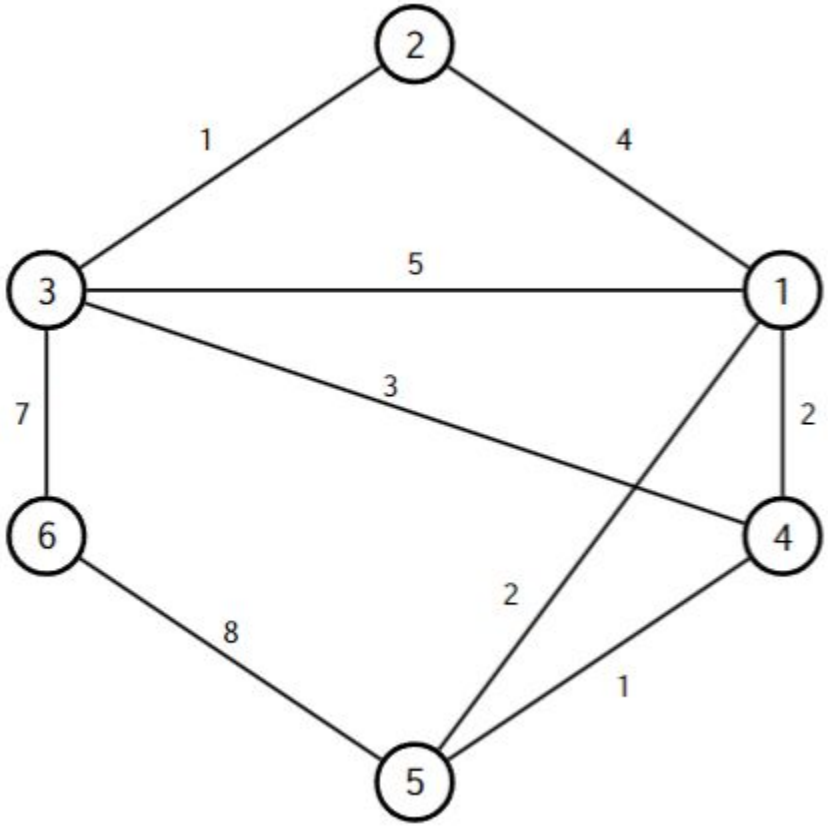
- Descoberto por diferentes pessoas em épocas próximas
    - Ficou mais famoso por Robert Clay Prim (1957)
- Encontra uma MST usando uma abordagem gulosa
- Um vértice u é escolhido para iniciar um componente conectado C
- Enquanto C ≠ V , é identificado o vértice u ≠ C mais próximo de C
- Então u é inserido em C e essa aresta faz parte de uma MST
- Complexidade: O(E log V)

Algoritmo de Prim - Pseudocódigo

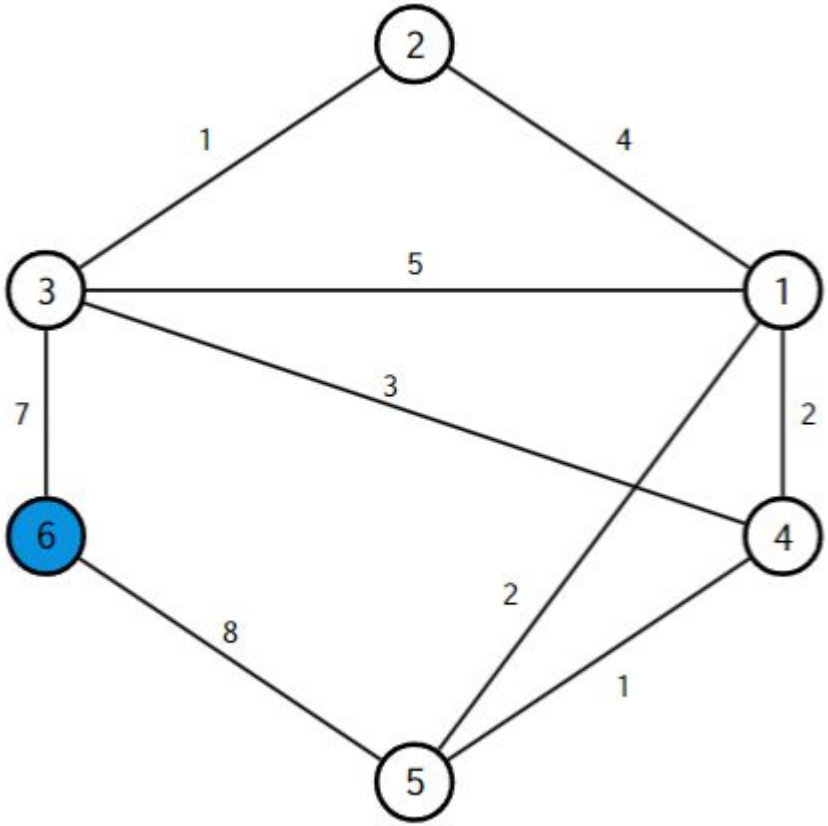
- Entrada: um grafo ponderado G(V, E)
- Saída: uma MST de G

1. Escolha um vértice u ∈ V e faça C = {u}, M = ∅
2. Enquanto C ≠ V :
   - Escolha o vértice v ≠ C mais próximo de C
   - Inclua v em C e a aresta que une v a C em M
3. Retorne M

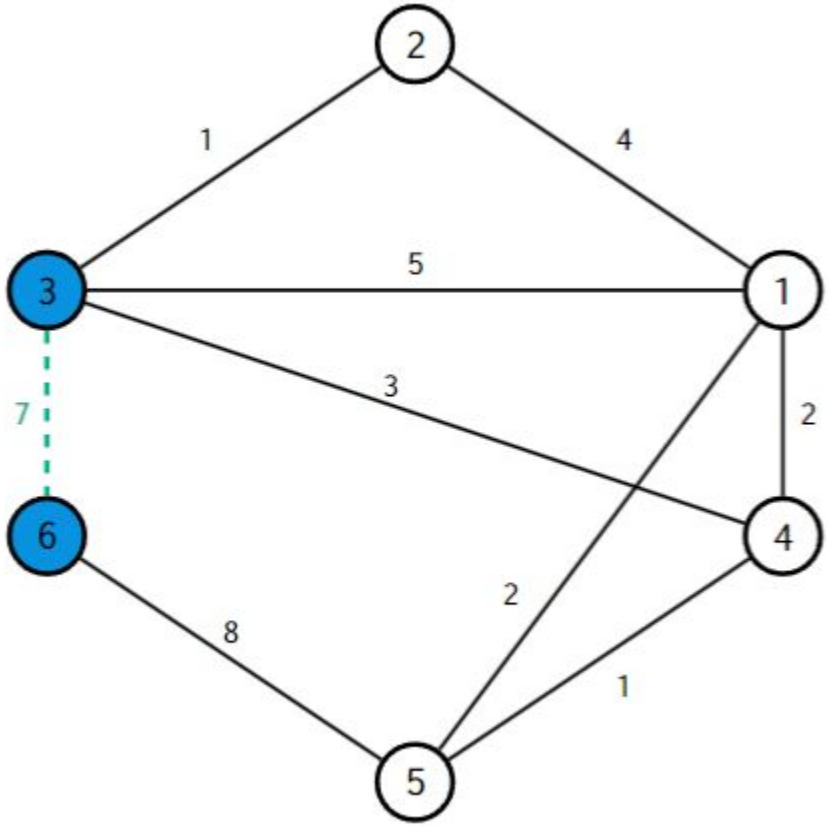
Algoritmo de Prim



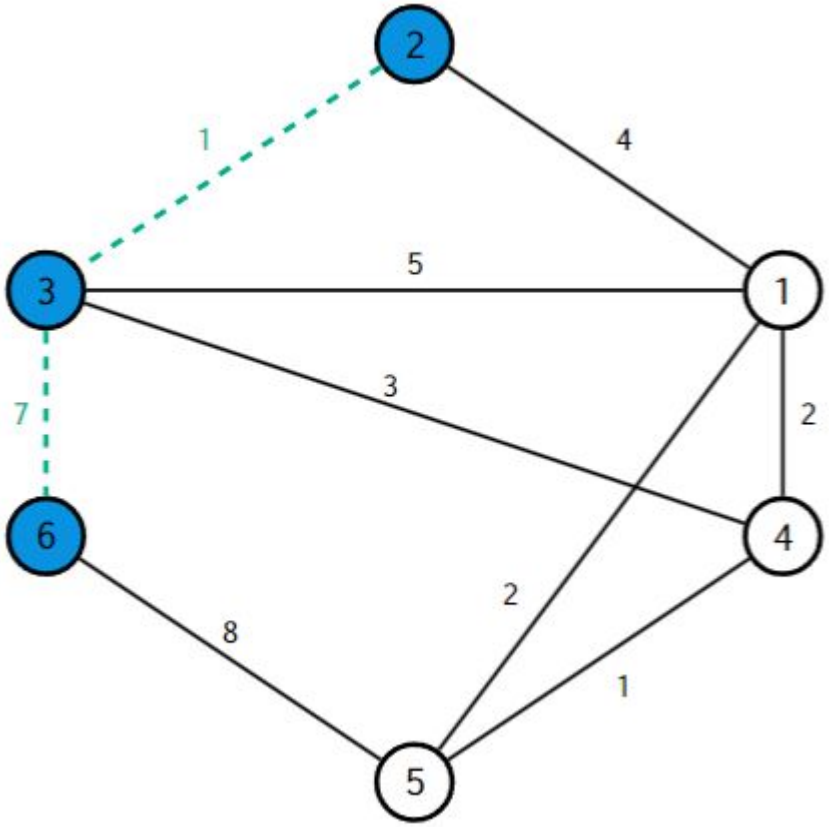
Algoritmo de Prim



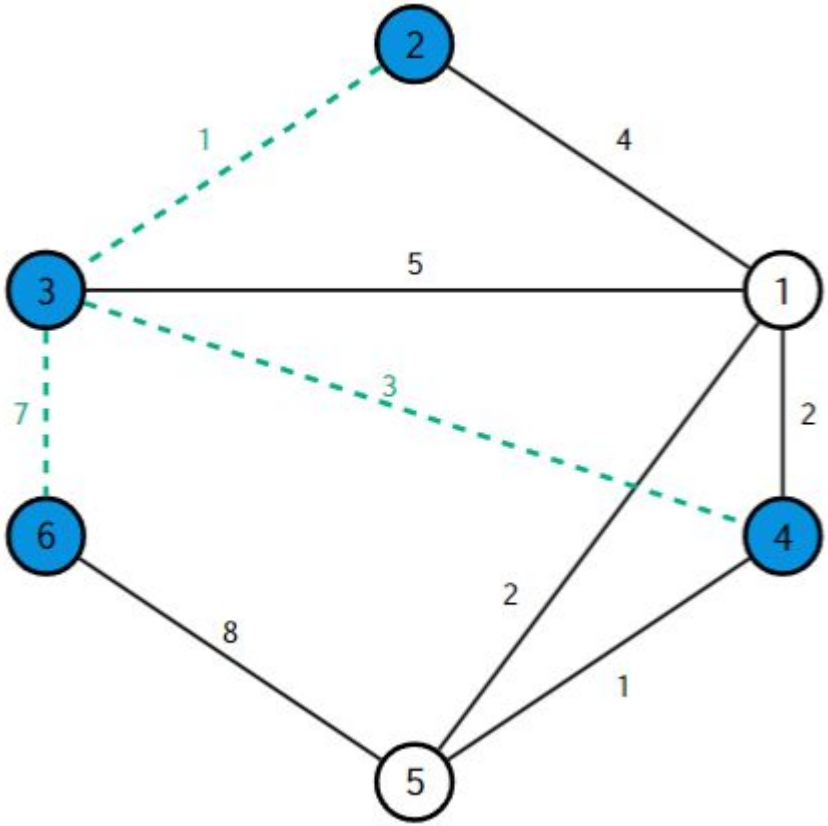
Algoritmo de Prim



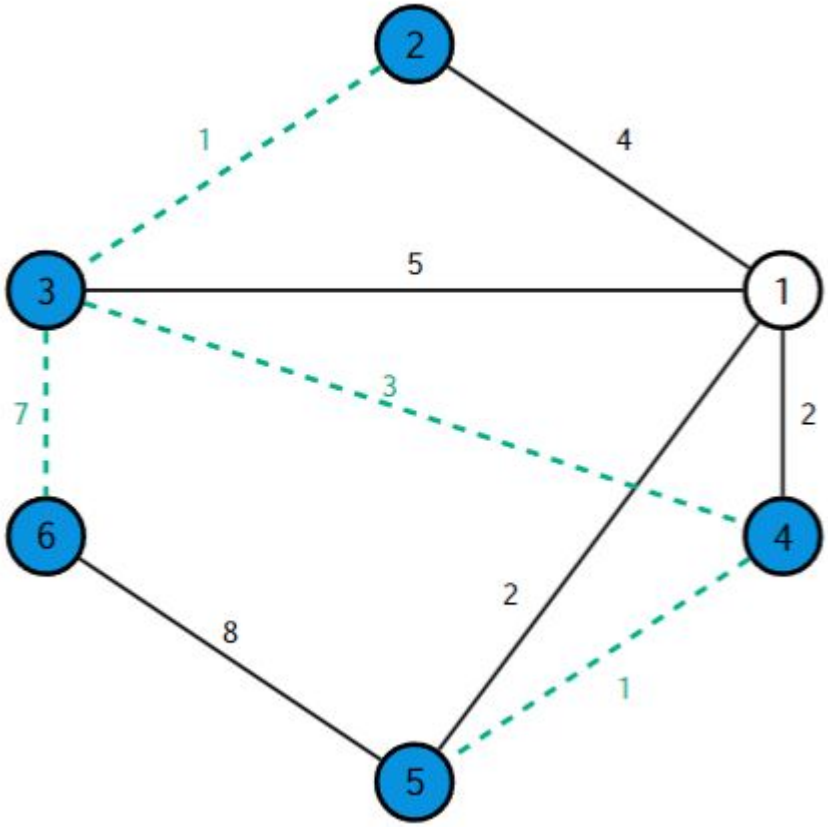
Algoritmo de Prim



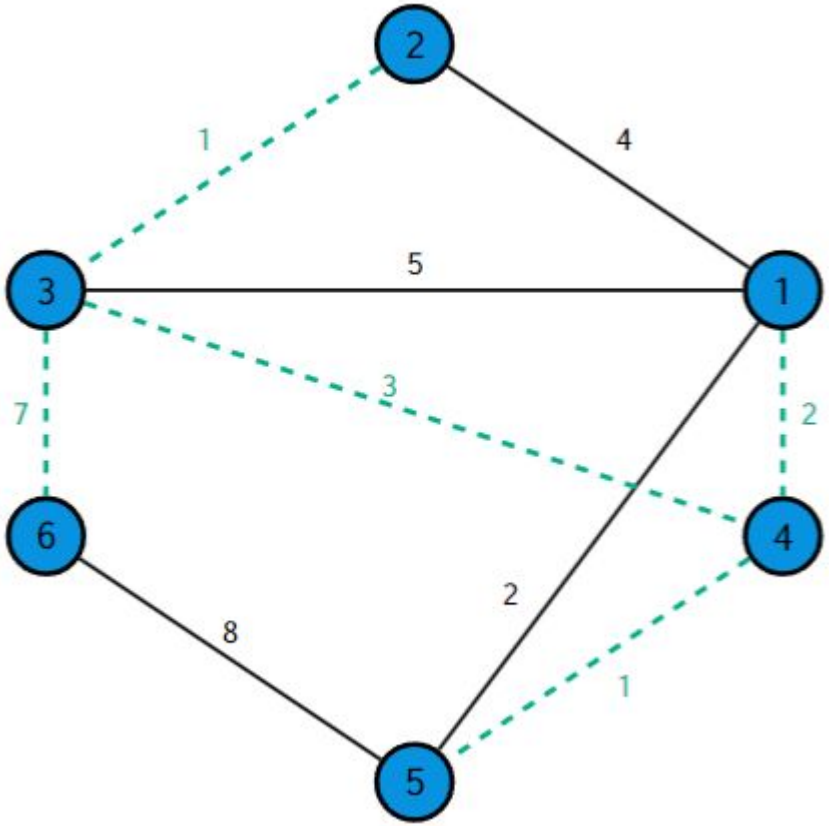
Algoritmo de Prim



Algoritmo de Prim



Algoritmo de Prim



Algoritmo de Prim - Identificação do Vértice Mais Próximo

- De maneira análoga ao algoritmo de dijkstra, o algoritmo de prim utiliza uma fila de prioridades, para encontrar o vértice mais próximo
  - O primeiro elemento inserido na árvore é o vértice de início, apontando para ele próprio com peso zero

```cpp
typedef tuple<int, int, int> tiii;

vector<pair<int, int>> prim(int origem) {
    priority_queue<tiii, vector<tiii>, greater<tiii>> pq;
    vector<pair<int, int>> mst; // Receberá as arestas da MST
    vector<bool> visitado(N + 1, false);

    pq.push({0, origem, origem}); // {peso, origem, destino}
    while (!pq.empty()) {
        auto [p, u, v] = pq.top(); pq.pop();
        if (visitado[v]) {
            continue;
        }
        visitado[v] = true;
        mst.push_back({u, v});
        for (auto [r, w]: G[v]) {
            if (!visitado[r]) {
                pq.push({w, v, r}); // Lembrar de inserir primeiro o peso
            }
        }
    }
    return mst;
}
```

Conclusão