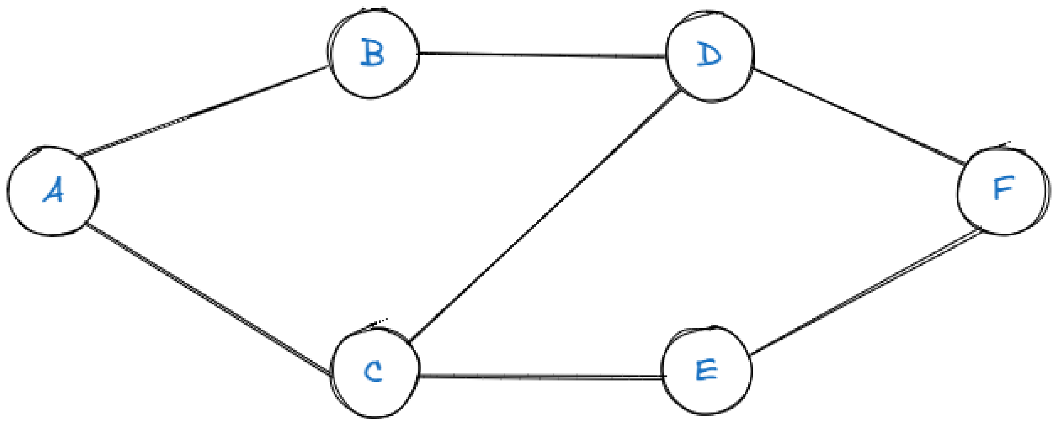
GRAFOS

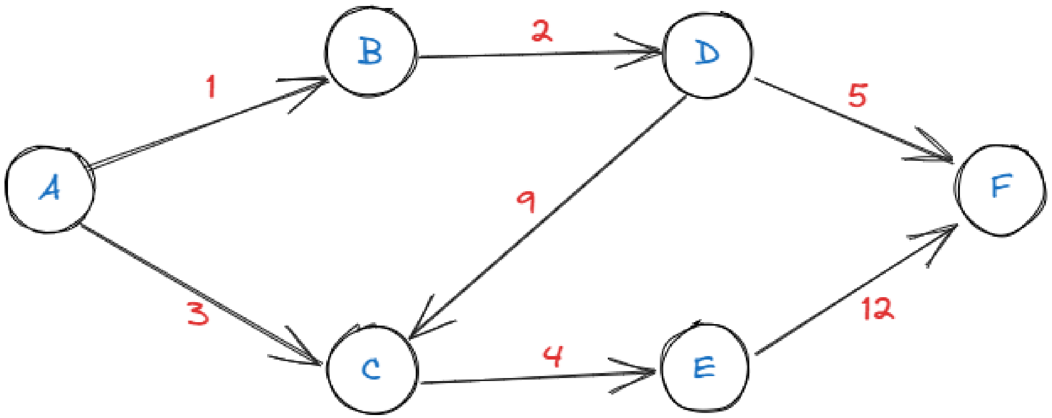
Grafo de Exemplo 1



Representação como um Dicionário

```
graph = {
    "A": ["B", "C"],
    "B": ["A", "D"],
    "C": ["A", "D", "E"],
    "D": ["B", "C", "F"],
    "E": ["C", "F"],
    "F": ["D", "E"]
}
```

Grafo de Exemplo 2



Representação como um Dicionário

```
graph = {
    "A": {"B": 1, "C": 3},
    "B": {"D": 2},
    "C": {"E": 4},
    "D": {"C": 9, "F": 5, },
    "E": {"F": 12},
    "F": {}
}
```

Algoritmo BFS

```
BFS(G,s)
for u:V[G]–{s} do
        cor[u] ← BRANCO
        d[u] ← ∞
        π[u] ← NIL
cor[s] ← CINZA
d[s] ← 0
π[s] ← NIL
Q ← {s}
while Q ≠ Ø do
      u ← dequeue[Q]
      for v:Adj[u] do
       if cor[v] = BRANCO then
            cor[v] ← CINZA
            d[v] ← d[u] + 1
            π[v] ← u
            enqueue(Q, v)
      cor[u] ← PRETO
```

G = (V, E) com lista de adjacências

cor[u]: cor do vértice u

π[u] predecessor de u na árvore de largura

(π[u] = NIL quando não houver)

d[u] distância entre o vértice inicial e u.

fila Q (FIFO) com vértices de cor cinza

BFS - Simples

```python
def bfs(graph, start):
    visited = set()
    queue = []
    visited.add(start)
    queue.append(start)

    while queue:
        vertex = queue.pop()
        print(vertex)

        for adj in graph[vertex]:
            if adj not in visited:
                visited.add(adj)
                queue.append(adj)
```

BFS - Slide

```python
def bfs(graph, start = None):
    colors = {}
    distance = {}
    parent = {}
    queue = []

    if start is None:
        start = list(graph.keys())[0]

    for vertex in graph:
        colors[vertex] = "WHITE"
        distance[vertex] = float('inf')
        parent[vertex] = None

    colors[start] = "GRAY"
    distance[start] = 0
    queue.append(start)

    while queue:
        vertex = queue.pop(0)
        for adj in graph[vertex]:
            if colors[adj] == "WHITE":
                colors[adj] = "GRAY"
                distance[adj] = distance[vertex] + 1
                parent[adj] = vertex
                queue.append(adj)
        colors[vertex] = "BLACK"
```

CUSTO: $O(V + E)$

Algoritmo DFS

```
DFS(G)
for u:V[G]  do
   cor[u] ← BRANCO
   p[u] ← NIL
tempo ← 0
for  u:V[G] do
   if cor[u] = BRANCO then
            VisitaDFS(u)
```

```
VisitaDFS(u)
cor[u] ← CINZA
d[u] ← tempo ← tempo + 1
for v:Adj[u] do
   if cor[v] = BRANCO then
               p[v] ← u
               VisitaDFS(v)
cor[u] ← PRETO
[u] ← tempo ← tempo + 1
```

CUSTO: $O(V + E)$

DFS - Simples

```python
def dfs(graph, start, visited = None):
    if visited is None:
        visited = set()

    visited.add(start)
    print(start)

    for adj in graph[start]:
        if adj not in visited:
            dfs(graph, adj, visited)
```

DFS - Slide

```python
def dfs_visit(vertex, graph, colors, parents, discovery_time, finish_time):
        colors[vertex] = 'GRAY'
        discovery_time[0] += 1
        finish_time[vertex] = None

        for adj in graph[vertex]:
            if colors[adj] == 'WHITE':
                parents[adj] = vertex
                dfs_visit(adj, graph, colors, parents, discovery_time, finish_time)

        colors[vertex] = 'BLACK'
        discovery_time[0] += 1
        finish_time[vertex] = discovery_time[0]
```

```python
def dfs(graph, start = None):
    vertices = list(graph.keys())
    colors = {}
    parent = {}
    discovery_time = [0]
    finish_time = {}

    for vertex in vertices:
        colors[vertex] = 'WHITE'
        parent[vertex] = None

    if start is None:
        for vertex in vertices:
            if colors[vertex] == 'WHITE':
                dfs_visit(vertex, graph, colors, parent, discovery_time, finish_time)
    else:
        dfs_visit(start, graph, colors, parent, discovery_time, finish_time)

```

Topological-Sort (G)

Execute DFS(G) e calcule o tempo final $f[v]$ para todo $v \in V$

Quando a visita para um vértice for completada, proceda sua inserção

na cabeça de uma **lista encadeada**

retorne a **lista encadeada** de vértices

Ordenação Topológica

```python
def dfs_order(graph, start, visited):
    order = []
    visited.add(start)

    for adj in graph[start]:
        if adj not in visited:
            order += dfs_order(graph, adj, visited)

    return order + [start]

def topological_sorting(graph, start = None):
    visited = set()
    order = []

    if start:
        order += dfs_order(graph, start, visited)

    for vertex in graph.keys():
        if vertex not in visited:
            order += dfs_order(graph, vertex, visited)

    return order[::-1]
```

CUSTO: $O(V + E)$

Grafo Trasnposto

```python
def transpose(graph):
    transpose = {node: [] for node in graph}

    for node in graph:
        for adj in graph[node]:
            transpose[adj].append(node)

    return transpose
```

CUSTO: $O(V + E)$

Algoritmo SCC

calcula f[u] para cada vértice

SCC(G)
 DFS(G)
 calcula G^T
 DFS(G^T), onde o laço principal vai em ordem decrescente de f[u] do 1o. DFS
 return cada árvore floresta resultante do 2o. DFS é um SCC

CUSTO: $O(V + E)$

Componentes Fortemente Conectado (SCC)

Grafo direcionado

```python
def dfs_comp(graph, start, visited):
    component = [start]
    visited.add(start)

    for adj in graph[start]:
        if adj not in visited:
            component += dfs_comp(graph, adj, visited)

    return component

def scc_directed_graph(graph):
    topological_order = topological_sorting(graph)
    graphT = transpose(graph)
    components = []
    visited = set()

    for vertex in topological_order:
        if vertex not in visited:
            components.append(dfs_comp(graphT, vertex, visited))

    return components
```

Componentes Fortemente Conectado (SCC)

Grafo não direcionado

```python
def dfs_comp(graph, start, visited):
    component = [start]
    visited.add(start)

    for adj in graph[start]:
        if adj not in visited:
            component += dfs_comp(graph, adj, visited)

    return component

def scc_undirected_graph(graph):
    components = []
    visited = set()

    for vertex in graph.keys():
        if vertex not in visited:
            components.append(dfs_comp(graph, vertex, visited))

    return components
```

DIJKSTRA(G, s)

INICIAMENORCAMINHO(G,S) O(V)
$S = \emptyset$
$Q = V[G]$ O(V)
while $Q \neq \emptyset$
    $u =$ EXTRACTMIN(Q) ExtractMin tem custo lg V e é executado V vezes: O(V lg V)
    $S = S \cup \{u\}$
    for each $v \in \text{Adj}[u]$ Relaxa tem custo lg V e é
        RELAXA(u, v) executado E vezes: O(E lg V)

Custo total = O(V) + O(E lg V) + O(V lg V) = O(E lg V)

BELLMAN-FORD(G, s)

INICIAMENORCAMINHO(G, s) O(V)
for $i = 1$ **to** $n - 1$
    for each $(u, v) \in E[G]$
        RELAX(u, v) O(V*E)
for each $(u, v) \in E[G]$
    if $d[v] > d[u] + w(u, v)$ O(E)
        return FALSE
return TRUE

Custo total = O(V*E)

Dijkstra

```python
import heapq

def dijkstra(graph, start):
    distances = {node:float('inf') for node in graph}
    parent={node:None for node in graph}

    distances[start] = 0
    queue = [(0, start)]

    while queue:
        current_distance, current_node = heapq.heappop(queue)
        for next_node, weight in graph[current_node].items():
            distance_temp = current_distance + weight
            if distance_temp < distances[next_node]:
                distances[next_node] = distance_temp
                parent[next_node] = current_node
                heapq.heappush(queue, (distance_temp, next_node))

    return distances, parent
```

Bellman Ford

```python
def bellman_ford(graph, node):
    distances = {node:float('inf') for node in graph}
    nodes = graph.keys()
    distances[node] = 0

    for _ in range(len(nodes) - 1):
        for current_node in nodes:
            for next_node, weight in graph[current_node].items():
                distance_temp = distances[current_node] + weight
                if distance_temp < distances[next_node]:
                    distances[next_node] = distance_temp

    for current_node in nodes:
        for next_node, weight in graph[current_node].items():
            distance_temp = distances[current_node] + weight
            if distance_temp < distances[next_node]:
                return (False, distances)

    return (True, distances)
```

Se Q for implementado por uma Min-Heap, inicialização (linhas 1-5) é feita em O(V).
O laco while é executado V vezes.
O custo de Extract-Min é O(lg V).
A atribuição da linha 10 envolve, implicitamente, uma mudança de elementos na heap (colocar o menor no topo). Cada Decrease-Key tem custo O(lg V).

```
PRIM(G, r)
1  for each u ∈ V[G]
2      key[u] = ∞
3      π[u] = NIL
4  key[r] = 0
5  Q = V[G]
6  while Q ≠ ∅
7      u = EXTRACT-MIN(Q)
8      for each v ∈ Adj[u]
9          if v ∈ Q and w(u, v) < key[v]
10             key[v] = w(u, v)
11             π[v] = u
```

|V| chamadas EXTRACT-MIN = O(VlgV)

<= |E| chamadas DECREASE-KEY = O(ElgV)

Custo Total: O(V lg V + E lg V) = O(E lg V)
Em um grafo conexo |E| >= |V|

Inicialização: O(V)
Ordenação arestas: O(E log E)
Find-set e Union: O(E log V)
Custo total: O(E log E)

```
KRUSKAL(G, r)
A = ∅
for each v ∈ G[V]
    MAKE-SET(v)
ordene E em ordem crescente de pesos
for each (u, v) da lista ordenada
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
return A
```