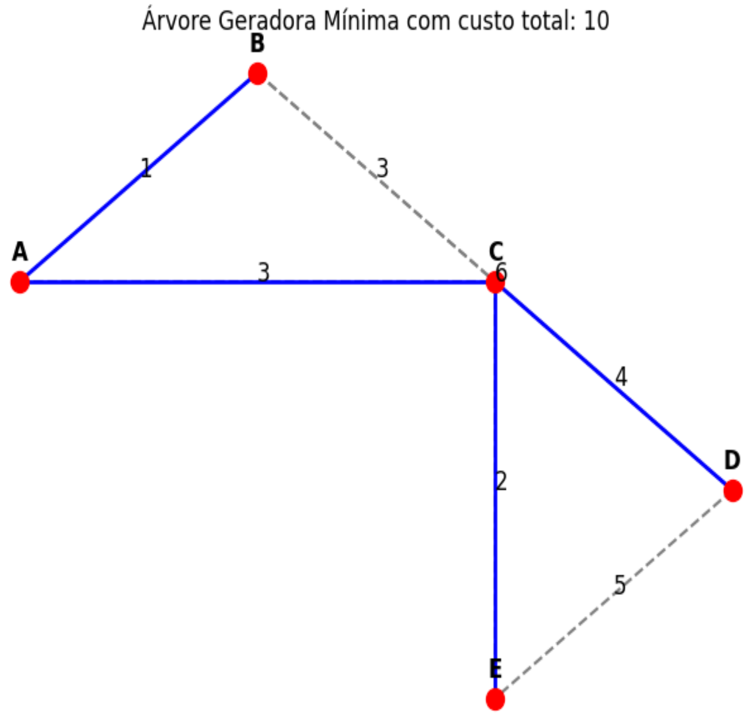
Implementação Kruskal (sem NetworkX)

```
import matplotlib.pyplot as plt

# função de união e busca (union-find)
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, u):  #encontra representante elemento u, procura quem é a
raiz de quem
        if self.parent[u] != u:
            self.parent[u] = self.find(self.parent[u])
        return self.parent[u]

    def union(self, u, v): #definição de cada união
        root_u = self.find(u)
        root_v = self.find(v) #vertice que unirá com u
        if root_u != root_v:
            # união rank
            if self.rank[root_u] > self.rank[root_v]:
                self.parent[root_v] = root_u
            elif self.rank[root_u] < self.rank[root_v]:
                self.parent[root_u] = root_v
            else:
                self.parent[root_v] = root_u
                self.rank[root_u] += 1

# função que realiza o kruskal
def kruskal(n, edges):
    # ordena pelo peso
    edges.sort(key=lambda x: x[2])
    uf = UnionFind(n)
    mst = []
    total_cost = 0

    for u, v, weight in edges:
        # se u e v n estão no mesmo conj, n forma ciclo
        if uf.find(u) != uf.find(v):
            uf.union(u, v)
            mst.append((u, v, weight))
            total_cost += weight

    return mst, total_cost

# definição grafo (arestas e vertices) com seus pesos
vertices = {'A': 0, 'B': 1, 'C': 2, 'D': 3, 'E': 4}
edges = [
```

```
    (vertices['A'], vertices['B'], 1),
    (vertices['A'], vertices['C'], 3),
    (vertices['B'], vertices['C'], 3),
    (vertices['B'], vertices['D'], 6),
    (vertices['C'], vertices['D'], 4),
    (vertices['C'], vertices['E'], 2),
    (vertices['D'], vertices['E'], 5)
]

mst, total_cost = kruskal(len(vertices), edges)

positions = {
    'A': (0, 1),
    'B': (1, 2),
    'C': (2, 1),
    'D': (3, 0),
    'E': (2, -1)
}

# Desenho do grafo inicial e da MST
def plot_graph(edges, mst, positions):
    plt.figure(figsize=(8, 6))

    # Desenhando todas as arestas do grafo original
    for u, v, weight in edges:
        u_pos = positions[list(vertices.keys())[u]]
        v_pos = positions[list(vertices.keys())[v]]
        plt.plot([u_pos[0], v_pos[0]], [u_pos[1], v_pos[1]], 'gray',
linestyle='--')
        plt.text((u_pos[0] + v_pos[0]) / 2, (u_pos[1] + v_pos[1]) / 2,
str(weight), color='black', fontsize=12)

    # Desenhando as arestas da MST em uma cor diferente
    for u, v, weight in mst:
        u_pos = positions[list(vertices.keys())[u]]
        v_pos = positions[list(vertices.keys())[v]]
        plt.plot([u_pos[0], v_pos[0]], [u_pos[1], v_pos[1]], 'b', linewidth=2)

    # Desenhando os vértices
    for vertex, pos in positions.items():
        plt.plot(pos[0], pos[1], 'ro', markersize=10)
        plt.text(pos[0], pos[1] + 0.1, vertex, ha='center', fontsize=12,
fontweight='bold')

    plt.title(f"Árvore Geradora Mínima com custo total: {total_cost}")
    plt.axis('off')
    plt.show()

plot_graph(edges, mst, positions)
```

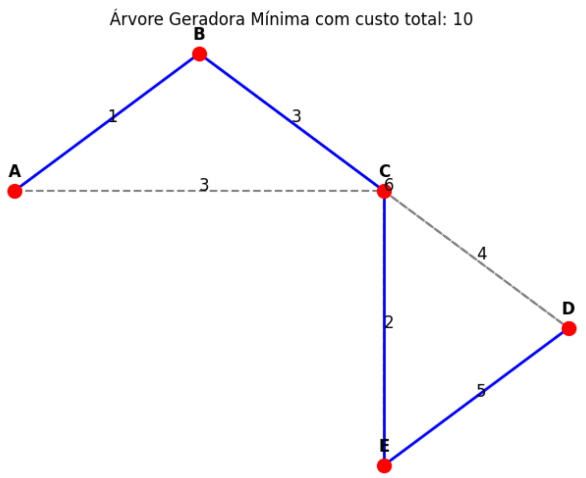


Implementação Prim (sem NetworkX)

```
import matplotlib.pyplot as plt
import heapq

def prim(n, edges):
    # lista adjacente para grafo
    graph = {i: [] for i in range(n)}
    for u, v, weight in edges:
        graph[u].append((weight, v))  # (peso, vertice)
        graph[v].append((weight, u))  # grafo não direcionado

    # inicia min_heap
    min_heap = [(0, 0)]  # (peso, vertice inicial)
    visited = set()
    mst = []
    total_cost = 0

    while min_heap:
        weight, u = heapq.heappop(min_heap)

        # se vertice visitado, ignorado
        if u in visited:
            continue

        # marca vertice visitado
        visited.add(u)
        total_cost += weight
```

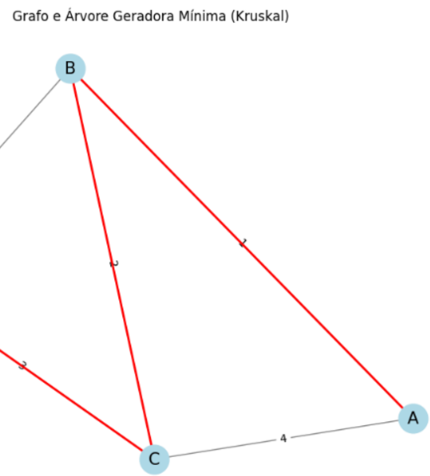
```
        # se n é aresta inicial adicona
        if weight != 0:
            mst.append((prev_vertex, u, weight))

        # adiciona aresta vertice atual
        for next_weight, v in graph[u]:
            if v not in visited:
                heapq.heappush(min_heap, (next_weight, v))
                prev_vertex = u  # armazena vertice anterior

    return mst, total_cost

# grado com arestas e peso
edges = [
    (0, 1, 1),  # A -- B
    (0, 2, 3),  # A -- C
    (1, 2, 3),  # B -- C
    (1, 3, 6),  # B -- D
    (2, 3, 4),  # C -- D
    (2, 4, 2),  # C -- E
    (3, 4, 5)   # D -- E
]

# chama prim
mst, total_cost = prim(5, edges)

# coodenadas
positions = {
    0: (0, 1),  # A
    1: (1, 2),  # B
    2: (2, 1),  # C
    3: (3, 0),  # D
    4: (2, -1)  # E
}

# nome vetices
vertex_names = {0: 'A', 1: 'B', 2: 'C', 3: 'D', 4: 'E'}

def plot_graph(edges, mst, positions, vertex_names):
    plt.figure(figsize=(8, 6))

    # desenhando todas as arestas do grafo original
    for u, v, weight in edges:
        if u in positions and v in positions:  # verifica se u e v
estão no dicionário positions
            u_pos = positions[u]
            v_pos = positions[v]
```

```
            plt.plot([u_pos[0], v_pos[0]], [u_pos[1], v_pos[1]],
'gray', linestyle='--')
            plt.text((u_pos[0] + v_pos[0]) / 2, (u_pos[1] +
v_pos[1]) / 2, str(weight), color='black', fontsize=12)

    # dando cor
    for u, v, weight in mst:
        if u in positions and v in positions:
            u_pos = positions[u]
            v_pos = positions[v]
            plt.plot([u_pos[0], v_pos[0]], [u_pos[1], v_pos[1]],
'b', linewidth=2)

    for vertex, pos in positions.items():
        plt.plot(pos[0], pos[1], 'ro', markersize=10)
        plt.text(pos[0], pos[1] + 0.1, vertex_names[vertex],
ha='center', fontsize=12, fontweight='bold')

    plt.title(f"Árvore Geradora Mínima com custo total:
{total_cost}")
    plt.axis('off')
    plt.show()

# chamando função p/ plotar grafo e a MST
plot_graph(edges, mst, positions, vertex_names)
```



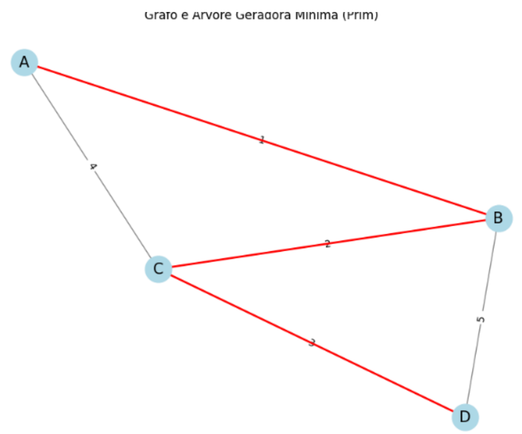
Implementação de Kruskal com a biblioteca NetworkX

```
import networkx as nx
import matplotlib.pyplot as plt

# grafo ponderado
G = nx.Graph()
G.add_weighted_edges_from([
    ('A', 'B', 1),
    ('A', 'C', 4),
    ('B', 'C', 2),
    ('B', 'D', 5),
    ('C', 'D', 3)
])

# econtra árvore geradora mínima usando kruskal com networkX
mst_kruskal = nx.minimum_spanning_tree(G, algorithm='kruskal')

# exibindo árvore (aresta, peso)
positions = nx.spring_layout(G)  #layout de mola para organizar o
grafo

# plotando grafo completo
plt.figure(figsize=(8, 6))
nx.draw(G, pos=positions, with_labels=True, node_color='lightblue',
edge_color='gray', node_size=700, font_size=15)
nx.draw_networkx_edge_labels(G, pos=positions, edge_labels={(u, v):
f"{data['weight']}" for u, v, data in G.edges(data=True)})

# plotando MST com cor
nx.draw_networkx_edges(mst_kruskal, pos=positions,
edge_color='red', width=2)

plt.title("Grafo e Árvore Geradora Mínima (Kruskal)")
plt.show()
```



Implementação de Prim usando NetworkX

```
import networkx as nx

# grafo ponderado
G = nx.Graph()
G.add_weighted_edges_from([
    ('A', 'B', 1),
    ('A', 'C', 4),
    ('B', 'C', 2),
    ('B', 'D', 5),
    ('C', 'D', 3)
])

# encontra árvore min usando prim com networkx
mst_prim = nx.minimum_spanning_tree(G, algorithm='prim')

positions = nx.spring_layout(G)

#plotando o grafo completo
plt.figure(figsize=(8, 6))
nx.draw(G, pos=positions, with_labels=True, node_color='lightblue',
edge_color='gray', node_size=700, font_size=15)
nx.draw_networkx_edge_labels(G, pos=positions, edge_labels={(u, v):
f"{data['weight']}" for u, v, data in G.edges(data=True)})

#plotando a MST com arestas destacadas em vermelho
nx.draw_networkx_edges(mst_prim, pos=positions, edge_color='red',
width=2)

plt.title("Grafo e Árvore Geradora Mínima (Prim)")
plt.show()
```



Impletação do Kruskal sem NetworkX, em função do tempo de execução

```
import heapq
import time
import random

def prim(n, edges):
    # lista adjacente grafo
    graph = {i: [] for i in range(n)}
    for u, v, weight in edges:
        graph[u].append((weight, v))  # (peso, vertice)
        graph[v].append((weight, u))  # grafo não direcionado

    # min heap
    min_heap = [(0, 0)]  # (peso, vertice inicial)
    visited = set()
    mst = []
    total_cost = 0

    while min_heap:
        weight, u = heapq.heappop(min_heap)

        # vertice visitado, ignorado
        if u in visited:
            continue

        # vertice visistado
        visited.add(u)
        total_cost += weight

        # se não é aresta inicial adiciona
        if weight != 0:
            mst.append((prev_vertex, u, weight))

        # adc aresta vertice atual
        for next_weight, v in graph[u]:
            if v not in visited:
                heapq.heappush(min_heap, (next_weight, v))
                prev_vertex = u  # armazena vertice anterior para a
aresta

    return mst, total_cost

# medir tempo prim
def measure_time_prim(n, edges):
    start_time = time.time()
    prim(n, edges)
    end_time = time.time()
    return end_time - start_time
```

```python
# grafos com número vértices arestas
def generate_graph(num_vertices, num_edges):
    edges = []
    for _ in range(num_edges):
        u = random.randint(0, num_vertices - 1)
        v = random.randint(0, num_vertices - 1)
        weight = random.randint(1, 10)
        if u != v:
            edges.append((u, v, weight))
    return edges

test_cases = [
    (10, 14),        # 10 vértices e 14 arestas
    (100, 140),      # 100 vértices e 140 arestas
    (1000, 1400),    # 1000 vértices e 1400 arestas
    (10000, 14000)   # 10000 vértices e 14000 arestas
]

# impressão dos tempos
for vertices, edges in test_cases:
    edges_list = generate_graph(vertices, edges)
    time_taken = measure_time_prim(vertices, edges_list)
    print(f"Tempo para grafo com {vertices} vértices e {edges}
arestas: {time_taken:.6f} segundos")
```

```
Tempo para grafo com 10 vértices e 14 arestas: 0.000025 segundos
Tempo para grafo com 100 vértices e 140 arestas: 0.000384 segundos
Tempo para grafo com 1000 vértices e 1400 arestas: 0.004969 segundos
Tempo para grafo com 10000 vértices e 14000 arestas: 0.042117 segundos
```

Analise em função do tempo para Kurskal com Network

```python
import networkx as nx
import time
import random

# gerar grafo ponderado aleatorio
def generate_weighted_graph(num_nodes, num_edges):
    G = nx.Graph()
    edges = set()
    while len(edges) < num_edges:
        u = random.randint(0, num_nodes - 1)
        v = random.randint(0, num_nodes - 1)
        if u != v and (u, v) not in edges and (v, u) not in edges:  # evita loops e duplicatas
            weight = random.uniform(1, 10)  # peso aleatório entre 1 e 10
            edges.add((u, v, weight))
    G.add_weighted_edges_from(edges)
    return G

# medir o tempo kruskal
def measure_kruskal_time(num_nodes, num_edges):
    G = generate_weighted_graph(num_nodes, num_edges)
    start_time = time.time()
    mst_kruskal = nx.minimum_spanning_tree(G, algorithm='kruskal')
    end_time = time.time()
    return end_time - start_time

scenarios = [
    (10, 14),
    (100, 140),
    (1000, 1400),
    (10000, 14000)
]

#exibe os tempos de execução
print("Tempo de execução do algoritmo de Kruskal em diferentes cenários:")
for num_nodes, num_edges in scenarios:
    exec_time = measure_kruskal_time(num_nodes, num_edges)
    print(f"Grafo com {num_nodes} vértices e {num_edges} arestas: {exec_time:.4f} segundos")
```

```
Tempo de execução do algoritmo de Kruskal em diferentes cenários:
Grafo com 10 vértices e 14 arestas: 0.0006 segundos
Grafo com 100 vértices e 140 arestas: 0.0013 segundos
Grafo com 1000 vértices e 1400 arestas: 0.0107 segundos
Grafo com 10000 vértices e 14000 arestas: 0.2366 segundos
```

Analise em função do tempo para Prim com Network

```
import networkx as nx
import time
import random

# grafos ponderados aleatorio
def generate_weighted_graph(num_nodes, num_edges):
    G = nx.Graph()
    edges = set()
    while len(edges) < num_edges:
        u = random.randint(0, num_nodes - 1)
        v = random.randint(0, num_nodes - 1)
        if u != v and (u, v) not in edges and (v, u) not in edges:  # evita loops e duplicatas
            weight = random.uniform(1, 10)  # peso aleatório entre 1 e 10
            edges.add((u, v, weight))
    G.add_weighted_edges_from(edges)
    return G

#medir o tempo prim
def measure_prim_time(num_nodes, num_edges):
    G = generate_weighted_graph(num_nodes, num_edges)
    start_time = time.time()
    mst_prim = nx.minimum_spanning_tree(G, algorithm='prim')
    end_time = time.time()
    return end_time - start_time

scenarios = [
    (10, 14),
    (100, 140),
    (1000, 1400),
    (10000, 14000)
]

# tempos de execução
print("Tempo de execução do algoritmo de Prim em diferentes cenários:")
for num_nodes, num_edges in scenarios:
    exec_time = measure_prim_time(num_nodes, num_edges)
    print(f"Grafo com {num_nodes} vértices e {num_edges} arestas: {exec_time:.4f} segundos")
```

```
Tempo de execução do algoritmo de Prim em diferentes cenários:
Grafo com 10 vértices e 14 arestas: 0.0003 segundos
Grafo com 100 vértices e 140 arestas: 0.0021 segundos
Grafo com 1000 vértices e 1400 arestas: 0.0077 segundos
Grafo com 10000 vértices e 14000 arestas: 0.1076 segundos
```

Analise geral em função do tempo entre os quatro diferentes tipos de implementação.

```
import random
import time
import heapq
import networkx as nx

# gerar grado ponderado
def generate_weighted_graph(num_nodes, num_edges,
use_networkx=False):
    if use_networkx:
        G = nx.Graph()
        edges = set()
        while len(edges) < num_edges:
            u = random.randint(0, num_nodes - 1)
            v = random.randint(0, num_nodes - 1)
            if u != v and (u, v) not in edges and (v, u) not in
edges:
                weight = random.uniform(1, 10)
                edges.add((u, v, weight))
                G.add_edge(u, v, weight=weight)
        return G
    else:
        G = {}
        edges = set()
        while len(edges) < num_edges:
            u = random.randint(0, num_nodes - 1)
            v = random.randint(0, num_nodes - 1)
            if u != v and (u, v) not in edges and (v, u) not in
edges:
                weight = random.uniform(1, 10)
                edges.add((u, v, weight))
                if u not in G:
                    G[u] = []
                if v not in G:
                    G[v] = []
                G[u].append((v, weight))
                G[v].append((u, weight))
        return G

# kruskal sem networkX
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, u):
```

```
        if self.parent[u] != u:
            self.parent[u] = self.find(self.parent[u])
        return self.parent[u]

    def union(self, u, v):
        root_u = self.find(u)
        root_v = self.find(v)
        if root_u != root_v:
            if self.rank[root_u] > self.rank[root_v]:
                self.parent[root_v] = root_u
            elif self.rank[root_u] < self.rank[root_v]:
                self.parent[root_u] = root_v
            else:
                self.parent[root_v] = root_u
                self.rank[root_u] += 1
            return True
        return False

def kruskal_no_nx(graph, n):
    edges = []
    for u in graph:
        for v, weight in graph[u]:
            if u < v:  # evita arestas duplicadas
                edges.append((weight, u, v))
    edges.sort()  # ordena por peso
    uf = UnionFind(n)
    mst = []
    for weight, u, v in edges:
        if uf.union(u, v):
            mst.append((u, v, weight))
    return mst

# Prim sem networkX
def prim_no_nx(graph, n):
    mst = []
    visited = [False] * n
    min_heap = [(0, 0)]  # começa do nó 0 com peso 0
    while min_heap:
        weight, u = heapq.heappop(min_heap)
        if not visited[u]:
            visited[u] = True
            if weight > 0:
                mst.append((u, weight))
            for v, edge_weight in graph[u]:
                if not visited[v]:
                    heapq.heappush(min_heap, (edge_weight, v))
    return mst

# medir o tempo de execução de cada algoritmo
```

```python
def measure_time(algorithm, graph, n, m):
    start_time = time.time()
    result = algorithm(graph, n)
    end_time = time.time()
    return end_time - start_time

scenarios = [
    (10, 14),
    (100, 140),
    (1000, 1400),
    (10000, 14000)
]

# comparação de todos os algoritmos
for num_nodes, num_edges in scenarios:
    # Gerar grafo
    G_nx = generate_weighted_graph(num_nodes, num_edges,
use_networkx=True)
    G_no_nx = generate_weighted_graph(num_nodes, num_edges,
use_networkx=False)

    # medindo os tempos de execução
    print(f"\nAnalisando Grafo com {num_nodes} vértices e
{num_edges} arestas:")

    # kruskal com networkX
    start_time = time.time()
    mst_kruskal_nx = nx.minimum_spanning_tree(G_nx,
algorithm='kruskal')
    kruskal_nx_time = time.time() - start_time
    print(f"Kruskal com NetworkX: {kruskal_nx_time:.4f} segundos")

    # prim com networkX
    start_time = time.time()
    mst_prim_nx = nx.minimum_spanning_tree(G_nx, algorithm='prim')
    prim_nx_time = time.time() - start_time
    print(f"Prim com NetworkX: {prim_nx_time:.4f} segundos")

    # kruskal sem networkX
    kruskal_no_nx_time = measure_time(kruskal_no_nx, G_no_nx,
num_nodes, num_edges)
    print(f"Kruskal sem NetworkX: {kruskal_no_nx_time:.4f}
segundos")

    # prim sem networkX
    prim_no_nx_time = measure_time(prim_no_nx, G_no_nx, num_nodes,
num_edges)
    print(f"Prim sem NetworkX: {prim_no_nx_time:.4f} segundos")
```

```
Analisando Grafo com 10 vértices e 14 arestas:
Kruskal com NetworkX: 0.0004 segundos
Prim com NetworkX: 0.0002 segundos
Kruskal sem NetworkX: 0.0001 segundos
Prim sem NetworkX: 0.0000 segundos

Analisando Grafo com 100 vértices e 140 arestas:
Kruskal com NetworkX: 0.0011 segundos
Prim com NetworkX: 0.0007 segundos
Kruskal sem NetworkX: 0.0003 segundos
Prim sem NetworkX: 0.0002 segundos

Analisando Grafo com 1000 vértices e 1400 arestas:
Kruskal com NetworkX: 0.0120 segundos
Prim com NetworkX: 0.0085 segundos
Kruskal sem NetworkX: 0.0030 segundos
Prim sem NetworkX: 0.0019 segundos

Analisando Grafo com 10000 vértices e 14000 arestas:
Kruskal com NetworkX: 0.2572 segundos
Prim com NetworkX: 0.1092 segundos
Kruskal sem NetworkX: 0.0452 segundos
Prim sem NetworkX: 0.0305 segundos
```