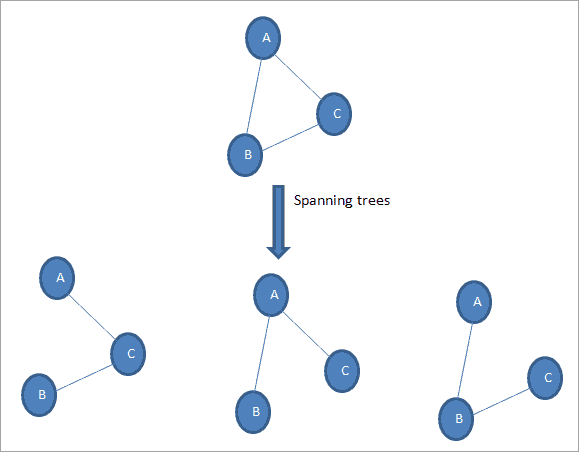
**MINIMUN SPANNING TREE (MST)**

Definição:

Árvore Geradora: é uma árvore não-enraizada cujo conjunto de vértices coincide com o conjunto de vértices do grafo e cujas arestas são (algumas das) arestas do grafo.

Exemplo:



Árvore Geradora Mínima: é uma árvore geradora que tem o menor peso possível, sendo o peso definido como a soma dos pesos das arestas da árvore. (Em outras palavras, uma árvore geradora é qualquer subgrafo que conecta todos os vértices do grafo, enquanto uma árvore geradora mínima é uma árvore geradora que conecta todos os vértices do grafo com o menor custo possível. Para que uma árvore geradora exista, o grafo deve ser conexo. Isso significa que deve haver um caminho entre quaisquer dois vértices do grafo.)

Exemplo:

6 9

0 1 3

0 2 8

0 3 4

0 5 10

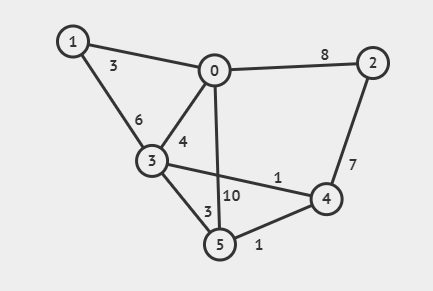
1 3 6

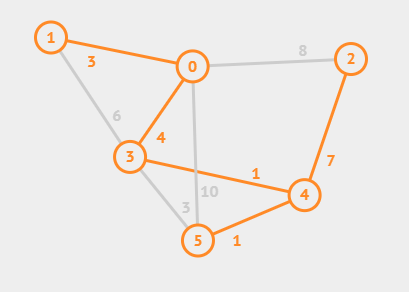
2 4 7

3 4 1

3 5 3

4 5 1





Algoritmos para encontrar a Árvore Mínima Geradora

A árvore geradora mínima pode ser encontrada usando algoritmos gulosos, que constroem a árvore incrementalmente, adicionando uma aresta de cada vez. Os dois algoritmos gulosos mais comuns para encontrar a árvore geradora mínima são o algoritmo de Prim e o algoritmo de Kruskal.

O algoritmo de Prim começa com um vértice arbitrário e, em seguida, adiciona as arestas de menor peso que conectam esse vértice a outros vértices que ainda não estão na árvore. O algoritmo de Kruskal, por outro lado, começa com um conjunto vazio de arestas e, em seguida, adiciona as arestas de menor peso, uma a uma, até que todos os vértices estejam conectados.

IMPLEMENTAÇÃO EM JAVA DOS ALGORITMOS DE PRIM E KRUSKAL

A Classe Aresta:

1. class Edge { // Class to represent an edge in the graph

2.     int src, dest, weight; // Source vertex, destination vertex, and weight of the edge

3.

4.     /\*\*

5.      \* Edge constructor

6.      \* @param src Source vertex

7.      \* @param dest Destination vertex

8.      \* @param weight Weight of the edge

9.      \*/

10.     public Edge(int src, int dest, int weight) { // Edge constructor

11.         this.src = src; // Initialize the source, destination, and weight of the edge

12.         this.dest = dest; // Initialize the source, destination, and weight of the edge

13.         this.weight = weight; // Initialize the source, destination, and weight of the edge

14.     } // End of Edge constructor

15. } // End of Edge class

16.

A Classe Grafo:

1. class Graph { // Class to represent a graph

2.     int V, E; // Number of vertices and edges

3.     ArrayList<Edge> edges; // Array list of edges

4.

5.     /\*\*

6.      \* Graph constructor

7.      \* @param V Number of vertices

8.      \* @param E Number of edges

9.      \*/

10.     public Graph(int V, int E) { // Graph constructor

11.         this.V = V; // Initialize the number of vertices

12.         this.E = E; // Initialize the number of edges

13.         edges = new ArrayList<>(E); // Initialize the array list of edges

14.     } // End of Graph constructor

15.

16.     // Function to add an edge to the graph

17.     /\*\*

18.      \* Function to add an edge to the graph

19.      \* @param src Source vertex

20.      \* @param dest Destination vertex

21.      \* @param weight Weight of the edge

22.      \*/

23.     void addEdge(int src, int dest, int weight) { // Add an edge to the graph

24.         edges.add(new Edge(src, dest, weight)); // Add an edge to the graph

25.     } // End of addEdge method

26. }

A Classe Principal:

1. public class MinimumSpanningTree { // Main class

2.     /\*\*

3.      \* Main method

4.      \* @param args Command line arguments

5.      \*/

6.     public static void main(String[] args) { // Main method

7.         int V = 5; // Number of vertices

8.         int E = 5; // Number of edges

9.

10.         Graph graph = new Graph(V, E); // Create a graph object with V vertices and E edges

11.

12.         // Add edges with weights to the graph

13.         graph.addEdge(0, 1, 3);

14.         graph.addEdge(1, 4, 4);

15.         graph.addEdge(2, 4, 1);

16.         graph.addEdge(3, 2, 2);

17.         graph.addEdge(0, 3, 3);

18.

19.         // Kruskal's Minimum Spanning Tree

20.         graph.kruskalMST(); // Print the result of Kruskal's algorithm

21.

22.         // Prim's Minimum Spanning Tree

23.         graph.primMST(); // Print the result of Prim's algorithm

24.     } // End of Main method

25. } // End of Main class

26.

O Algoritmo de Kruskal:

1. void kruskalMST() { // Kruskal's algorithm

2.         Collections.sort(edges, Comparator.comparingInt(e -> e.weight)); // Sort the edges by weight in ascending order

3.

4.         int[] parent = new int[V]; // Create an array of subsets

5.         for (int i = 0; i < V; i++) // Create V subsets with single elements

6.             parent[i] = i; // Create V subsets with single elements

7.

8.         ArrayList<Edge> result = new ArrayList<>(); // Store the result

9.

10.         int i = 0, e = 0; // Index of the next edge and the number of edges

11.         while (e < V - 1 && i < E) { // Loop until we find V-1 edges or we traverse all the edges

12.             Edge nextEdge = edges.get(i++); // Get the next edge

13.             int x = find(parent, nextEdge.src); // Find the set of the source vertex

14.             int y = find(parent, nextEdge.dest); // Find the set of the destination vertex

15.

16.             if (x != y) { // If the source and destination vertices are not in the same set

17.                 result.add(nextEdge); // Add the edge to the result

18.                 union(parent, x, y); // Union the two sets

19.                 e++; // Increment the number of edges

20.             } // End of if statement

21.         } // End of while loop

22.

23.         // Print the result

24.         System.out.println("Kruskal's Minimum Spanning Tree:"); // Print the result

25.         for (Edge edge : result) { // Loop through the result

26.             System.out.println(edge.src + " - " + edge.dest + " : " + edge.weight); // Print the result in the format of "src - dest : weight"

27.         } // End of loop

28.     } // End of kruskalMST method

29.

30.     // Helper function to find the set of an element in Kruskal's algorithm

31.     /\*\*

32.      \* Find method to find the set of an element in Kruskal's algorithm

33.      \* @param parent Array of parents

34.      \* @param i Index of the element

35.      \* @return

36.      \*/

37.     int find(int[] parent, int i) { // Find method

38.         if (parent[i] != i) // If i is not the parent of itself

39.             parent[i] = find(parent, parent[i]); // Recursively find the parent of i

40.         return parent[i]; // Return the parent of i

41.     } // End of find method

42.

43.     // Helper function to perform union of two sets in Kruskal's algorithm

44.     /\*\*

45.      \* Union method to perform union of two sets in Kruskal's algorithm

46.      \* @param parent Array of parents

47.      \* @param x

48.      \* @param y

49.      \* @return

50.      \*/

51.     void union(int[] parent, int x, int y) { // Union method

52.         int xRoot = find(parent, x); // Find the set of x

53.         int yRoot = find(parent, y); // Find the set of y

54.         parent[xRoot] = yRoot; // Make yRoot as parent of xRoot

55.     } // End of union method

56.

O Algoritmo de Prim

1. // Prim's algorithm for Minimum Spanning Tree

2.     /\*\*

3.      \* Prim's algorithm for Minimum Spanning Tree

4.      \*/

5.     void primMST() { // Prim's algorithm

6.         PriorityQueue<Edge> minHeap = new PriorityQueue<>(Comparator.comparingInt(e -> e.weight)); // Create a min heap to store the edges

7.

8.         boolean[] inMST = new boolean[V]; // Store the vertices in the MST

9.         ArrayList<Edge> result = new ArrayList<>(); // Store the result

10.

11.         int startVertex = 0; // Start vertex is 0

12.         inMST[startVertex] = true; // Add the start vertex to the MST

13.

14.         for (Edge edge : edges) { // Loop through the edges

15.             if (edge.src == startVertex) { // If the source vertex is the start vertex

16.                 minHeap.add(edge); // Add the edge to the heap

17.             } // End of if statement

18.         } // End of loop

19.

20.         while (result.size() < V - 1) { // Loop until we find V-1 edges

21.             if (minHeap.isEmpty()) { // If the heap is empty, then the graph is not connected

22.                 System.out.println("The graph is not connected."); // Print the result

23.                 return; // Return

24.             } // End of if statement

25.

26.             Edge minEdge = minHeap.poll(); // Get the edge with the minimum weight

27.

28.             if (!inMST[minEdge.dest]) { // If the destination vertex is not in the MST

29.                 inMST[minEdge.dest] = true; // Add the destination vertex to the MST

30.                 result.add(minEdge); // Add the edge to the result

31.

32.                 for (Edge edge : edges) { // Loop through the edges

33.                     if (edge.src == minEdge.dest && !inMST[edge.dest]) { // If the source vertex is the destination vertex of the minimum edge and the destination vertex is not in the MST

34.                         minHeap.add(edge); // Add the edge to the heap

35.                     } // End of if statement

36.                 } // End of loop

37.             } // End of if statement

38.         } // End of while loop

39.

40.         // Print the result

41.         System.out.println("\nPrim's Minimum Spanning Tree:"); // Print the result

42.         for (Edge edge : result) { // Loop through the result

43.             System.out.println(edge.src + " - " + edge.dest + " : " + edge.weight); // Print the result in the format of "src - dest : weight"

44.         } // End of loop

45.     } // End of primMST method

46.

**COMPLEXIDADE E APLICAÇÕES**

A complexidade do algoritmo de Prim é O(V^2 + E), no pior caso, onde V é o número de vértices do grafo e E é o número de arestas. Essa complexidade é obtida porque o algoritmo precisa examinar todas as arestas do grafo para encontrar a aresta de menor peso que conecta um vértice da árvore a um vértice que ainda não está na árvore.

A complexidade do algoritmo de Kruskal é O(E log E), no pior caso. Essa complexidade é obtida porque o algoritmo precisa ordenar as arestas do grafo por peso, o que pode ser feito em tempo O(E log E). Em seguida, o algoritmo adiciona as arestas de menor peso, uma a uma, até que todos os vértices estejam conectados.

A facilidade de implementação do algoritmo de Prim ou do algoritmo de Kruskal depende da estrutura de dados utilizada para representar o grafo. Com uma matriz de adjacência, a implementação é relativamente simples, pois basta percorrer a matriz de adjacência para encontrar as arestas de menor peso. No entanto, a matriz de adjacência pode ocupar muita memória para grafos grandes.

Com uma lista de vértices, a implementação é um pouco mais complexa, pois é necessário manter uma estrutura de dados auxiliar para armazenar as arestas que ainda não foram adicionadas à árvore. No entanto, a lista de vértices é uma estrutura de dados mais eficiente em termos de memória para grafos grandes.

A árvore geradora mínima pode ser utilizada em diversas situações reais, como:

* Planejamento de redes de comunicação: para encontrar a rota mais barata para conectar todos os pontos de uma rede.
* Planejamento de redes elétricas: para encontrar a maneira mais eficiente de conectar todos os consumidores de energia.
* Planejamento de redes de distribuição: para encontrar a melhor maneira de distribuir mercadorias ou serviços para todos os clientes.
* Planejamento de redes de esgoto: para encontrar a maneira mais eficaz de coletar e transportar esgoto.

A árvore geradora mínima também pode ser utilizada em problemas de otimização, como:

* Problema do caixeiro viajante: para encontrar o caminho mais curto que visita todos os pontos em um mapa.
* Problema do corte de barras: para encontrar a maneira mais eficiente de cortar uma barra de material em pedaços de tamanhos específicos.
* Problema do carregamento de navios: para encontrar a maneira mais eficiente de carregar um navio com mercadorias.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**USO RESTRITO NOSSO**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Utilizem esse site para simular uma apresentação do funcionamento dos algoritmos:**

**https://www.softwaretestinghelp.com/minimum-spanning-tree-tutorial/#:~:text=Prim's%20algorithm%20is%20yet%20another,all%20the%20vertices%20are%20covered.**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**EXPLANATION OF MINIMUM SPANNING TREE ALGORITHM IN JAVA**

This Java code implements two algorithms to find the Minimum Spanning Tree (MST) of a weighted, undirected graph: Kruskal's algorithm and Prim's algorithm. Here's a breakdown of each part:

1. Classes and Data Structures:

* Edge class: Represents an edge in the graph with its source, destination, and weight.
* Graph class: Represents the entire graph with its number of vertices (V), edges (E), and a list of edges (edges).
* PriorityQueue: Used in Prim's algorithm to store and prioritize edges based on their weights.
* ArrayList: Used to store the final MST edges.

2. Kruskal's Algorithm:

1. Sort edges: All edges are sorted based on their weight in non-decreasing order.
2. Initialize sets for disjoint sets: An array parent is used to keep track of the sets in which each vertex belongs. Initially, each vertex forms its own set.
3. Iterate through edges: For each edge in the sorted list:
   * Find the sets of the source and destination vertices.
   * Check if adding the edge would create a cycle: If the source and destination belong to the same set, adding the edge would create a cycle, so skip it.
   * Otherwise, add the edge to the MST and merge the sets of the source and destination vertices.
4. Print MST: Once all edges have been processed, print the edges in the MST along with their weights.

3. Prim's Algorithm:

1. Initialize priority queue and set for visited nodes: A priority queue minHeap is used to store and prioritize edges based on their weights. A boolean array inMST is used to keep track of visited nodes.
2. Start with an arbitrary vertex: Choose any vertex as the starting point and mark it as visited.
3. Add edges connected to the visited node: Add all edges connected to the visited node to the minHeap.
4. Iterate until all nodes are visited: While there are still unvisited nodes:
   * Extract the minimum weight edge from the minHeap.
   * If the destination node of the extracted edge is not visited:
     + Mark the destination node as visited.
     + Add the extracted edge to the MST.
     + Add all edges connected to the newly visited node to the minHeap.
5. Print MST: Once all nodes have been visited, print the edges in the MST along with their weights.

4. Main Function:

1. Define graph parameters: V and E specify the number of vertices and edges, respectively.
2. Create graph object: Initialize an object of the Graph class.
3. Add edges: Use the addEdge method to add edges to the graph with their source, destination, and weight.
4. Call MST algorithms:
   * Kruskal's algorithm: Call the kruskalMST method to calculate and print the MST using Kruskal's algorithm.
   * Prim's algorithm: Call the primMST method to calculate and print the MST using Prim's algorithm.

Note:

* Both algorithms guarantee finding the minimum spanning tree.
* Kruskal's algorithm is generally faster for sparse graphs (few edges compared to vertices).
* Prim's algorithm is generally easier to implement.