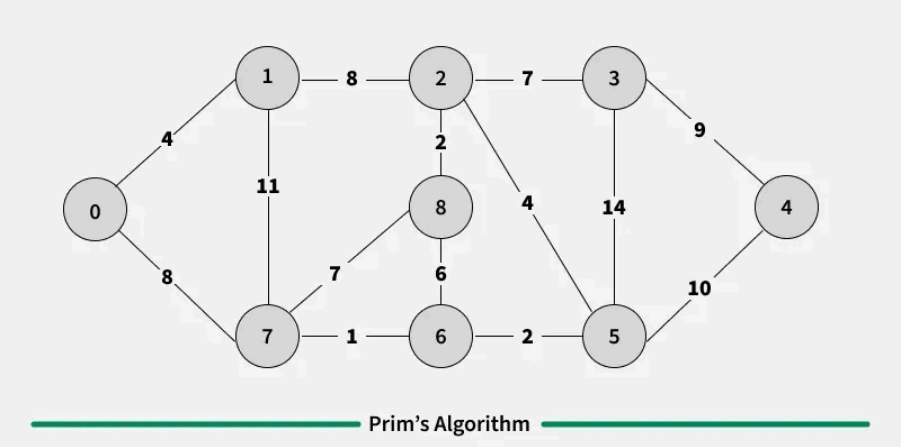
**Algoritmos y trayectorias especiales**

Esto se refiere al conjunto de técnicas y procedimientos computacionales diseñados para planificar rutas o movimientos óptimos en espacios físicos o virtuales, especialmente en sistemas como la robótica, videojuegos, inteligencia artificial y navegación autónoma. Estos algoritmos determinan trayectorias específicas que deben seguirse evitando obstáculos, minimizando tiempo o recursos, y adaptándose a condiciones cambiantes del entorno.

Entre estos podremos encontrar:

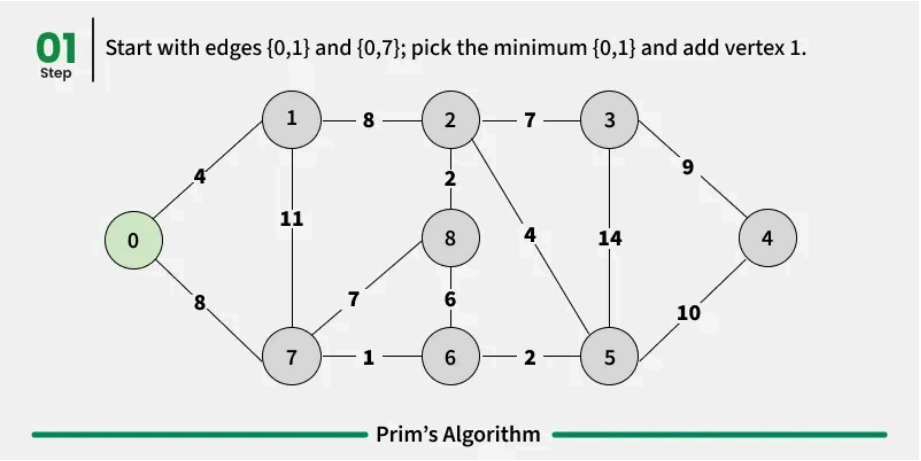
1. **Algoritmo de Prim:** Este algoritmo se utiliza para obtener un árbol de expansión mínima en un grafo no dirigido y conectado, garantizando que el costo total sea el más bajo posible. Su funcionamiento se basa en ir incorporando vértices al árbol uno a uno, eligiendo siempre la arista de menor peso que conecte un nodo ya incluido con otro que aún no lo está.

   El proceso inicia desde un solo nodo y, a partir de ahí, va explorando los nodos vecinos, evaluando todas las conexiones disponibles para seguir construyendo el árbol con eficiencia.

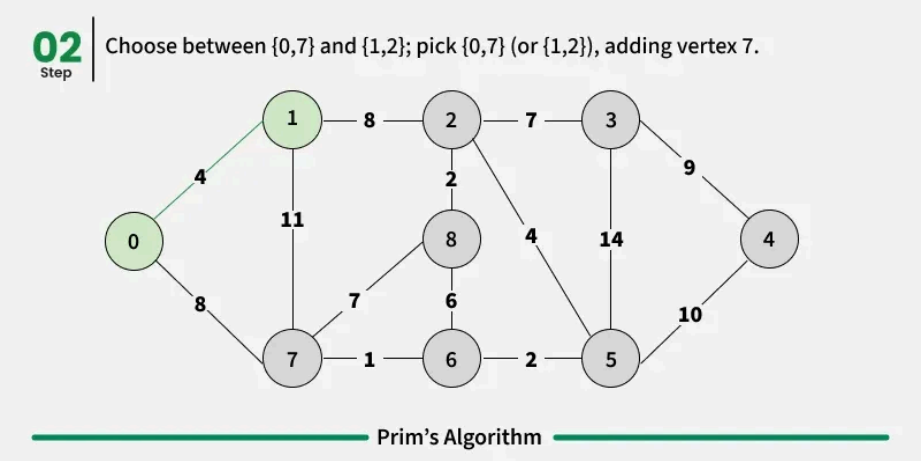


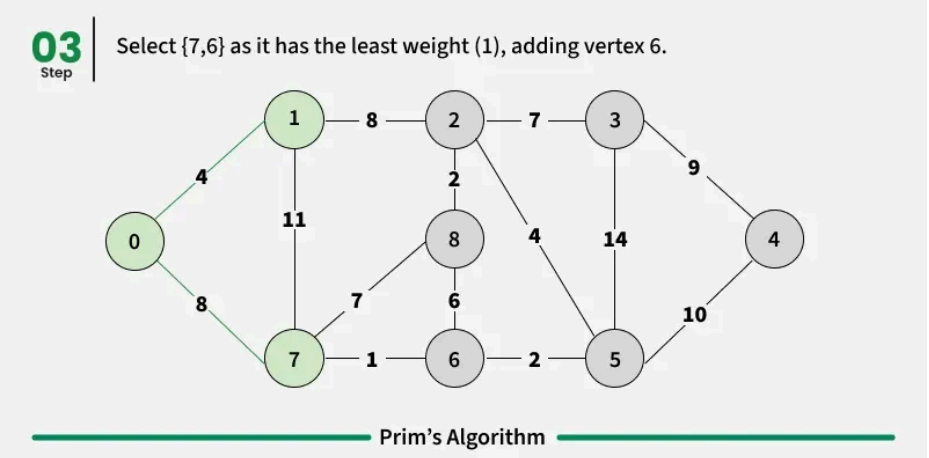

**¿Cómo funciona?**

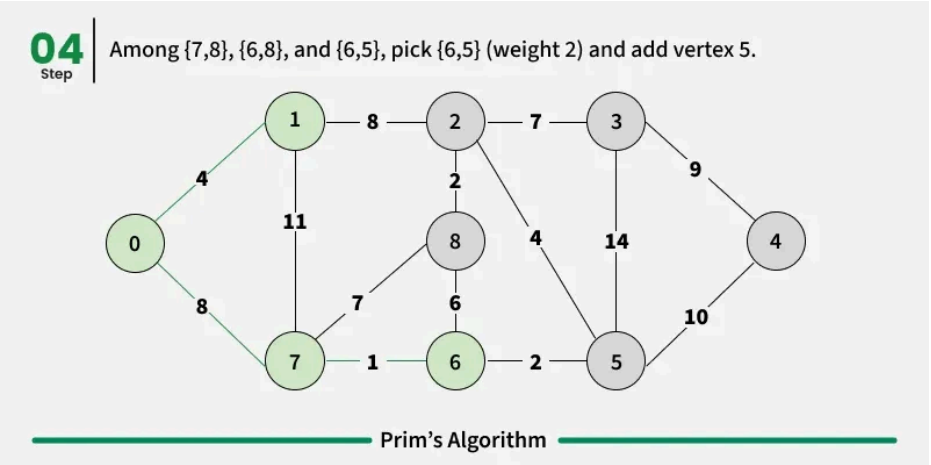
***Paso 1:*** Determine un vértice arbitrario como vértice inicial del MST. En el diagrama a continuación, elija 0.

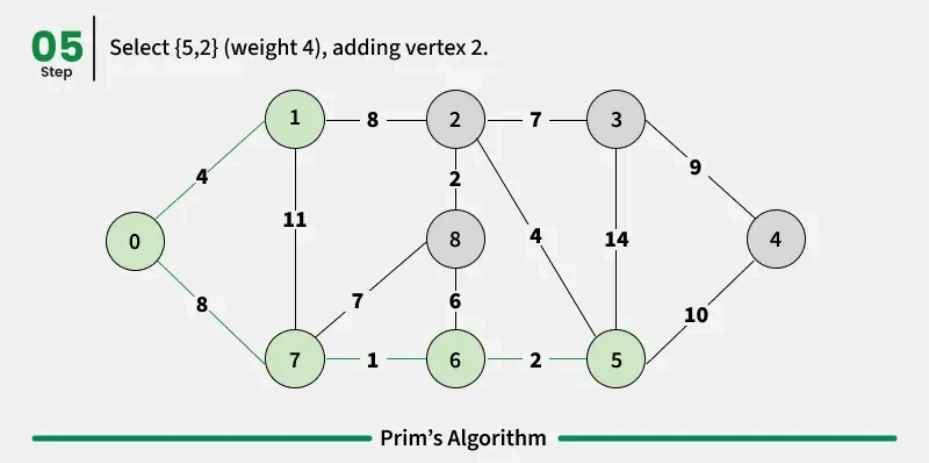



***Paso 2:*** Siga los pasos 3 a 5 hasta que haya vértices no incluidos en el MST (conocidos como vértices marginales).

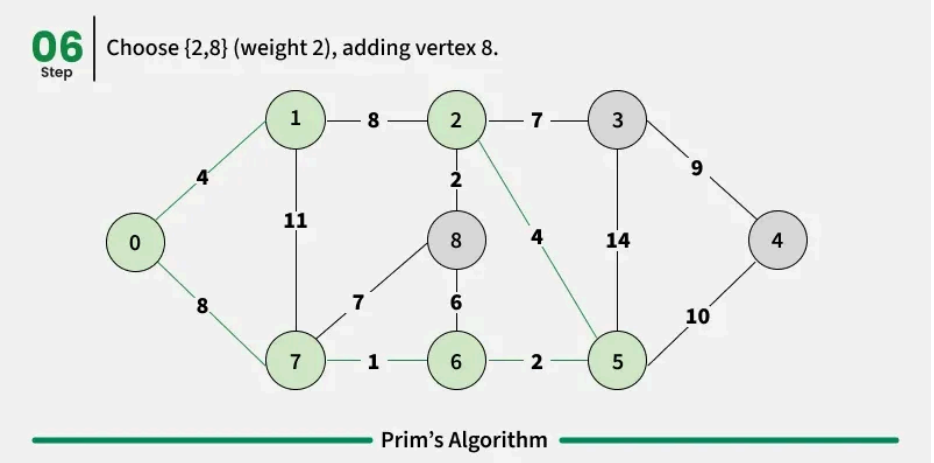


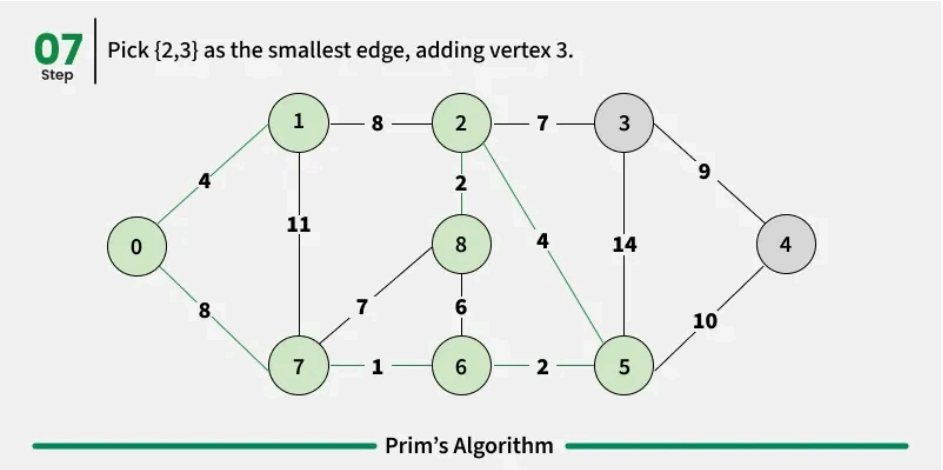


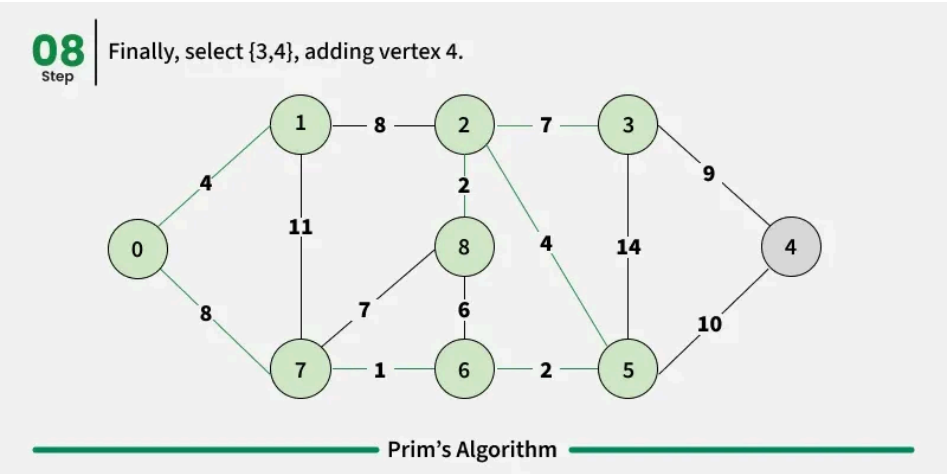

***Paso 3:*** Encuentre las aristas que conectan cualquier vértice del árbol con los vértices marginales.



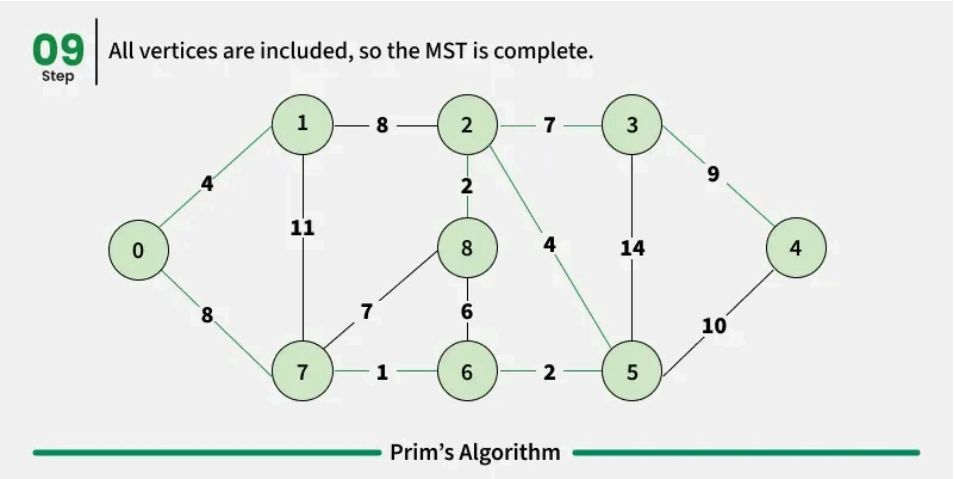
***Paso 4:*** Encuentre el mínimo entre estas aristas.

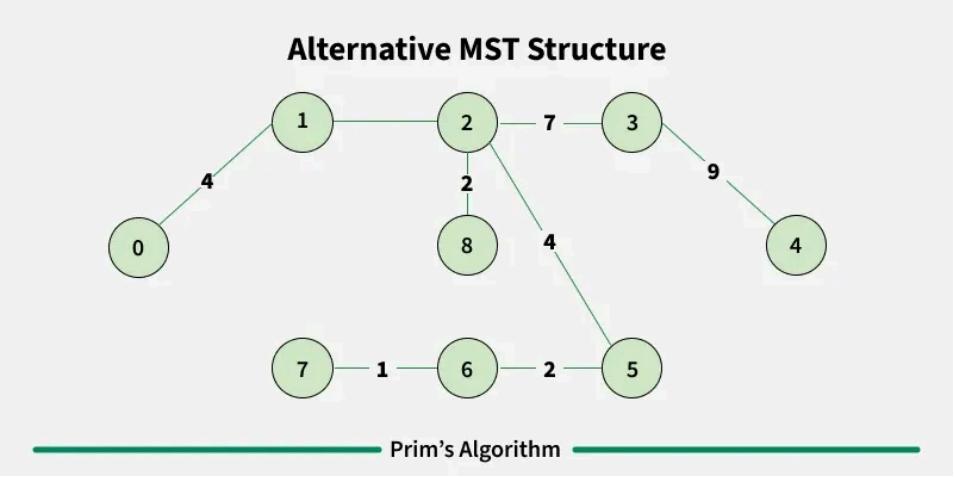



***Paso 5:*** Agregue la arista elegida al MST. Dado que solo consideramos las aristas que conectan los vértices marginales con el resto, nunca se produce un ciclo.



***Paso 6:*** Regrese al MST y salga.




Ejemplo de cómo se implementaría:

```
# A Python3 program for
# Prim's Minimum Spanning Tree (MST) algorithm.
# The program is for adjacency matrix
# representation of the graph

# Library for INT_MAX
import sys


class Graph():
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                      for row in range(vertices)]

    # A utility function to print
```

```
    # the constructed MST stored in parent[]
    def printMST(self, parent):
        print("Edge \tWeight")
        for i in range(1, self.V):
            print(parent[i], "-", i, "\t", self.graph[parent[i]][i])

    # A utility function to find the vertex with
    # minimum distance value, from the set of vertices
    # not yet included in shortest path tree
    def minKey(self, key, mstSet):

        # Initialize min value
        min = sys.maxsize

        for v in range(self.V):
            if key[v] < min and mstSet[v] == False:
                min = key[v]
                min_index = v

        return min_index

    # Function to construct and print MST for a graph
    # represented using adjacency matrix representation
    def primMST(self):

        # Key values used to pick minimum weight edge in cut
        key = [sys.maxsize] * self.V
        parent = [None] * self.V  # Array to store constructed MST
        # Make key 0 so that this vertex is picked as first vertex
        key[0] = 0
        mstSet = [False] * self.V

        parent[0] = -1  # First node is always the root of

        for cout in range(self.V):

            # Pick the minimum distance vertex from
            # the set of vertices not yet processed.
            # u is always equal to src in first iteration
            u = self.minKey(key, mstSet)

            # Put the minimum distance vertex in
            # the shortest path tree
            mstSet[u] = True

            # Update dist value of the adjacent vertices
```

```
            # of the picked vertex only if the current
            # distance is greater than new distance and
            # the vertex in not in the shortest path tree
            for v in range(self.V):

                # graph[u][v] is non zero only for adjacent vertices of
m
                # mstSet[v] is false for vertices not yet included in
MST
                # Update the key only if graph[u][v] is smaller than
key[v]
                if self.graph[u][v] > 0 and mstSet[v] == False \
                and key[v] > self.graph[u][v]:
                    key[v] = self.graph[u][v]
                    parent[v] = u

        self.printMST(parent)


# Driver's code
if __name__ == '__main__':
    g = Graph(5)
    g.graph = [[0, 2, 0, 6, 0],
               [2, 0, 3, 8, 5],
               [0, 3, 0, 0, 7],
               [6, 8, 0, 0, 9],
               [0, 5, 7, 9, 0]]

    g.primMST()
```