Estructura del Grafo:

Representaremos el grafo usando un mapa de listas de adyacencia, donde cada nodo tendrá una lista de nodos vecinos y los pesos de las aristas correspondientes.

Clase Arista:

Crearemos una clase Edge para representar cada conexión (arista) entre dos nodos y su peso.

Algoritmo de Dijkstra:

Usaremos una cola de prioridad (PriorityQueue) para seleccionar siempre el nodo con la distancia mínima.
Mantendremos un arreglo de distancias para almacenar la distancia más corta conocida desde el nodo de origen a cada nodo.

Clase Arista:

Representa una arista en el grafo. Cada Edge tiene un nodo (el nodo de destino) y un weight (peso de la arista).

Representación del Grafo:

Usamos un Map<Integer, List<Edge>> para almacenar las aristas. Cada nodo tiene una lista de aristas que representan sus vecinos y los pesos.

Método addArista:

Este método agrega una arista al grafo. Si el grafo fuera no dirigido, también agregaremos la arista en sentido contrario.

Método removeNodo:

Este método remueve un nodo al grafo y todas las aristas adyacentes al nodo a eliminar.

Método mostarVecinos:

Este método Itera sobre el grafo base y crea una lista con todos los nodos adyacentes a cada nodo iterado. Luego devuelve por pantalla el resultado mostrando el nodo vecino y su peso.

Método dijkstra:

Inicializamos las distancias a todos los nodos como Integer.MAX_VALUE (infinito).
Establecemos la distancia del nodo de inicio a 0.
Usamos una cola de prioridad (**PriorityQueue**) para procesar los nodos en orden de la distancia mínima acumulada.
Para cada nodo, recorremos sus vecinos. Si encontramos una distancia más corta a un vecino, actualizamos la distancia y la agregamos de nuevo a la cola de prioridad.

Impresión de Resultados:

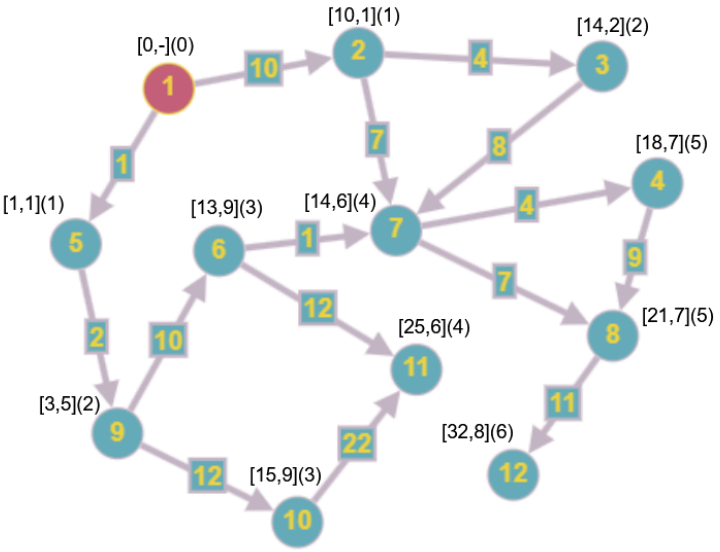
Después de ejecutar Dijkstra, imprimimos la distancia mínima desde el nodo de origen a cada nodo del grafo.

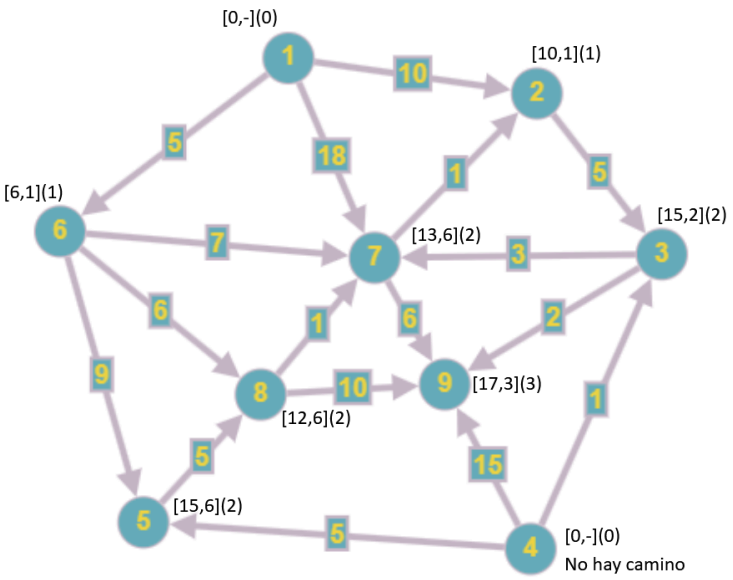
Explicación del Flujo:

Comenzamos en el nodo de origen (en este caso, el nodo 1).
Para cada nodo procesado, actualizamos las distancias mínimas de sus vecinos.
Usamos una cola de prioridad para siempre seleccionar el siguiente nodo con la distancia acumulada más corta.
Una vez que todos los nodos han sido procesados, tenemos la distancia mínima desde el nodo de origen a todos los demás nodos.

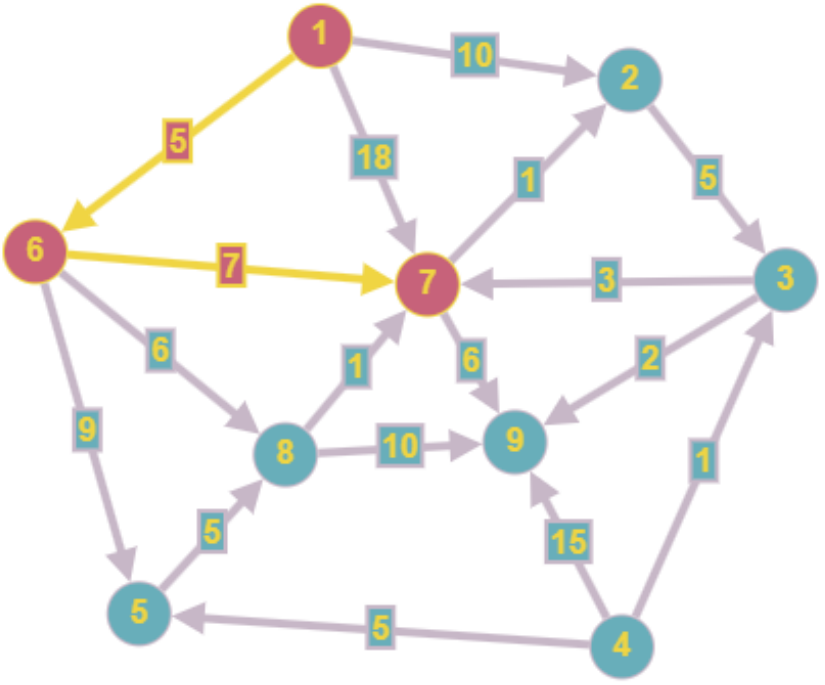
Este enfoque es eficiente para grafos con pesos positivos, y la complejidad es aproximadamente $O((V+E) * \log V)$, donde E es el número de aristas y V es el número de nodos.

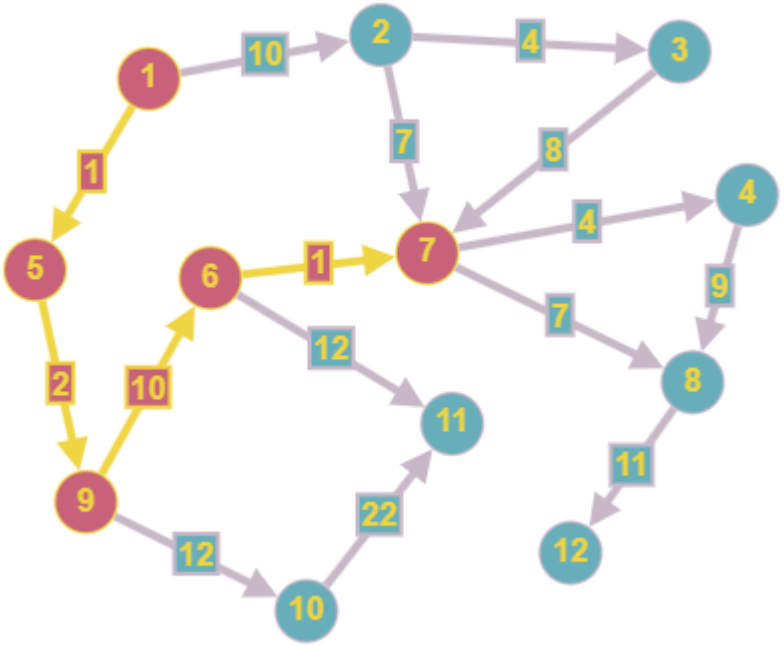
Cálculo del camino más corto:





Enunciado B(camino mas corto desde el vértice 1 hasta el 7):





Pseudocódigo:

```
Clase Arista
        Atributos:
        nodo: Entero
        peso: Entero

        Método Constructor(nodo: Entero, peso: Entero)
        this.nodo = nodo
        this.peso = peso

Fin Clase

Clase Grafo
        Atributos:
        grafo: Mapa<Entero, Lista<Arista>>

        Método Constructor(n: Entero)
        grafo = Nuevo Mapa
        Para i desde 1 hasta n
        grafo.put(i, Nueva Lista<Arista>())

        Método addArista(origen: Entero, destino: Entero, peso: Entero)
        grafo.get(origen).add(Nueva Arista(destino, peso))
        grafo.get(destino).add(Nueva Arista(origen, peso)) // Para grafo no
dirigido

        Método removeArista(origen: Entero, destino: Entero)
        grafo.get(origen).removeIf(arista -> arista.nodo == destino)
        grafo.get(destino).removeIf(arista -> arista.nodo == origen) // Para grafo
no dirigido

        Método removeNodo(nodo: Entero)
        Si no grafo.containsKey(nodo) Entonces
        Retornar // Si el nodo no existe, no hace nada
        Fin Si
        // Eliminar todas las aristas que apuntan al nodo desde otros nodos
        Para cada arista en Nueva Lista(grafo.get(nodo))
        nodoVecino = arista.nodo
        grafo.get(nodoVecino).removeIf(adj -> adj.nodo == nodo)
        Fin Para
        // Eliminar el nodo y sus aristas adyacentes
        grafo.remove(nodo)
```

```
        Fin Método

        Método getAdyacentes(nodo: Entero) Retorna Lista<Arista>
        Retornar grafo.getOrDefault(nodo, Nueva Lista<Arista>())

        Método mostrarVecinos()
        Imprimir "Vecinos de cada nodo:"
        Para cada nodo en grafo.keySet()
        adyacentes = grafo.get(nodo)
        Imprimir "Nodo " + nodo + ": "
        Si adyacentes.isEmpty() Entonces
        Imprimir "No tiene vecinos"
        Sino
        Para cada arista en adyacentes
                Imprimir " Nodo: " + arista.nodo
        Fin Para
        Imprimir ""
        Fin Si
        Fin Para
        Fin Método
Fin Clase

Clase AlgoritmoDijkstra
        Método main()
        n = 12 // Número de nodos en el grafo (1 a 12)
        grafo = Nuevo Grafo(n)

        // Agregar aristas al grafo
        grafo.addArista(1, 2, 10)
        grafo.addArista(1, 6, 5)
        grafo.addArista(1, 7, 18)
        // añadir demás aristas ...

        // Mostrar el grafo antes de eliminar el nodo
        Imprimir "Grafo antes de eliminar el nodo:"
        ejecutarDijkstra(grafo, 1, n)
        grafo.mostrarVecinos()

        // Eliminar un nodo y todas sus aristas adyacentes
        Entero remover = 7;
        grafo.removeNodo(remover)
        Imprimir "Grafo después de eliminar el nodo:"
        ejecutarDijkstra(grafo, 1, n)
        Imprimir ""
        grafo.mostrarVecinos()
        Fin Método
```

```
        Método ejecutarDijkstra(grafo: Grafo, origen: Entero, n: Entero)
        resultados = dijkstra(grafo, origen, n)

        // Imprimir las distancias y caminos desde el origen a cada nodo
        Imprimir "Distancias y caminos desde el nodo " + origen + ":"
        Para i desde 1 hasta n
        Si resultados.distancias[i] == MAXIMO_ENTERO Entonces
        Imprimir "Hasta el nodo " + i + " -> No hay camino"
        Sino
        Imprimir "Hasta el nodo " + i + " -> Distancia: " + resultados.distancias[i] +
                ", Camino: " + reconstruirCamino(resultados.predecesores, origen,i)
        Fin Si
        Fin Para
        Fin Método

        Clase ResultadosDijkstra
        Atributos:
        distancias: Arreglo de Enteros
        predecesores: Arreglo de Enteros

        Método Constructor(n: Entero)
        distancias = Nuevo Arreglo de Enteros[n + 1]
        predecesores = Nuevo Arreglo de Enteros[n + 1]
        Fin Clase

        Método dijkstra(grafo: Grafo, start: Entero, n: Entero) Retorna
ResultadosDijkstra
        resultados = Nuevo ResultadosDijkstra(n)
        Llenar resultados.distancias con MAXIMO_ENTERO
        Llenar resultados.predecesores con -1
        resultados.distancias[start] = 0

        colaPrioridad = Nueva ColaPrioridad()
        colaPrioridad.offer(Nueva Arista(start, 0))

        Mientras colaPrioridad no está vacía
        actual = colaPrioridad.poll()
        nodoActual = actual.nodo
        distanciaActual = actual.peso

        Si distanciaActual > resultados.distancias[nodoActual] Entonces
        Continuar
        Fin Si

        // Procesar cada vecino del nodo actual
        Para cada arista en grafo.getAdyacentes(nodoActual)
        vecino = arista.nodo
```

```
        peso = arista.peso
        nuevaDistancia = resultados.distancias[nodoActual] + peso

        // Si encontramos un camino más corto al vecino, lo actualizamos
        Si nuevaDistancia < resultados.distancias[vecino] Entonces
                resultados.distancias[vecino] = nuevaDistancia
                resultados.predecesores[vecino] = nodoActual
                colaPrioridad.offer(Nueva Arista(vecino, nuevaDistancia))
        Fin Si
        Fin Para
        Fin Mientras

        Retornar resultados
        Fin Método

        Método reconstruirCamino(predecesores: Arreglo de Enteros, origen:
Entero, destino: Entero) Retorna Cadena
        Si predecesores[destino] == -1 y origen != destino Entonces
        Retornar "No hay camino" // Si no hay predecesor y no es el nodo de
origen
        Fin Si

        camino = Nueva Lista
        Para nodo desde destino hasta -1
        camino.add(nodo)
        Fin Para
        Revertir camino
        Retornar camino.toString()
        Fin Método
Fin Clase
```