

# UNIVERSIDAD AUTÓNOMA “TOMÁS FRÍAS”

CARRERA DE “INGENIERÍA DE SISTEMAS”

## INVESTIGACIÓN OPERATIVA 2



### LABORATORIO N°1 APLICACIÓN DE ALGORITMOS DE GRAFOS A CIUDADES DE BOLIVIA

NOMBRE: HERSON JOSE CANAZA DELGADO

Potosí – Bolivia  
2024

## Algoritmo de Dijkstra

El **algoritmo de Dijkstra** es un método que se utiliza para encontrar la **ruta más corta** entre un nodo inicial y todos los demás nodos en un grafo ponderado (un grafo donde las aristas tienen un peso o costo). Es ampliamente utilizado en problemas de **optimización de rutas**, como el cálculo de las distancias más cortas entre puntos en una red de carreteras, sistemas de transporte, o redes informáticas.

### Características principales:

- Funciona en **grafos dirigidos o no dirigidos** que no tienen pesos negativos en las aristas.
- Busca la ruta más corta desde un **nodo fuente** a todos los demás nodos.
- Utiliza una estructura de datos (como una cola de prioridad) para seleccionar el nodo no visitado con la **distancia mínima**.
- Es un algoritmo **greedy** (codicioso), ya que selecciona el nodo más cercano en cada paso.

```

src > graph > paths > TS Dijkstra > Dijkstra
1 import Vertex from '../Vertex';
2 import Edge from '../Edge';
3 import getCanvas from '../graph-ui/canvas/canvas';
4 import { delay } from '../graph-ui/utils';
5
6 export const Dijkstra = class {
7   private vertices: number;
8   private adjMatrix: number[][];
9
10  constructor(vertices: number) {
11    this.vertices = vertices;
12    this.adjMatrix = Array.from({ length: vertices }, () =>
13      Array(vertices).fill(Infinity)
14    );
15  }
16
17  addEdge(src: number, dest: number, weight: number) {
18    this.adjMatrix[src][dest] = weight;
19    this.adjMatrix[dest][src] = weight; // Si el grafo es dirigido, elimina esta linea.
20  }
21
22  dijkstra(src: number) {
23    const dist = Array(this.vertices).fill(Infinity);
24    const sptSet = Array(this.vertices).fill(false);
25    dist[src] = 0;
26
27    for (let count = 0; count < this.vertices - 1; count++) {
28      const u = this.minDistance(dist, sptSet);
29      sptSet[u] = true;
30
31      for (let v = 0; v < this.vertices; v++) {
32        if (
33          !sptSet[v] &&
34          this.adjMatrix[u][v] !== Infinity &&
35          dist[v] > dist[u] + this.adjMatrix[u][v] {
36          dist[v] = dist[u] + this.adjMatrix[u][v];
37        }
38      }
39    }
40    this.printSolution(dist);
41  }
42
43  private minDistance(dist: number[], sptSet: boolean[]): number {
44    let min = Infinity;
45    let minIndex = -1;
46
47    for (let v = 0; v < this.vertices; v++) {
48      if (!sptSet[v] && dist[v] < min) {
49        min = dist[v];
50        minIndex = v;
51      }
52    }
53    return minIndex;
54  }
55
56  private printSolution(dist: number[]) {
57    console.log('Vertex \t Distance from Source');
58    for (let i = 0; i < this.vertices; i++) {
59      console.log(`${i} \t ${dist[i]}`);
60    }
61  }
62
63  };
64
65  };

```

```

src > graph > paths > TS Dijkstra > Dijkstra
6 export const Dijkstra = class {
22   dijkstra(src: number) {
35     dist[u] !== Infinity &&
36     dist[u] + this.adjMatrix[u][v] < dist[v]
37   ) {
38     dist[v] = dist[u] + this.adjMatrix[u][v];
39   }
40 }
41 }
42 }
43 this.printSolution(dist);
44 }
45
46 private minDistance(dist: number[], sptSet: boolean[]): number {
47   let min = Infinity;
48   let minIndex = -1;
49
50   for (let v = 0; v < this.vertices; v++) {
51     if (!sptSet[v] && dist[v] < min) {
52       min = dist[v];
53       minIndex = v;
54     }
55   }
56   return minIndex;
57 }
58
59 private printSolution(dist: number[]) {
60   console.log('Vertex \t Distance from Source');
61   for (let i = 0; i < this.vertices; i++) {
62     console.log(`${i} \t ${dist[i]}`);
63   }
64 }
65
66 };

```

## ALGORITMO DE PRIM

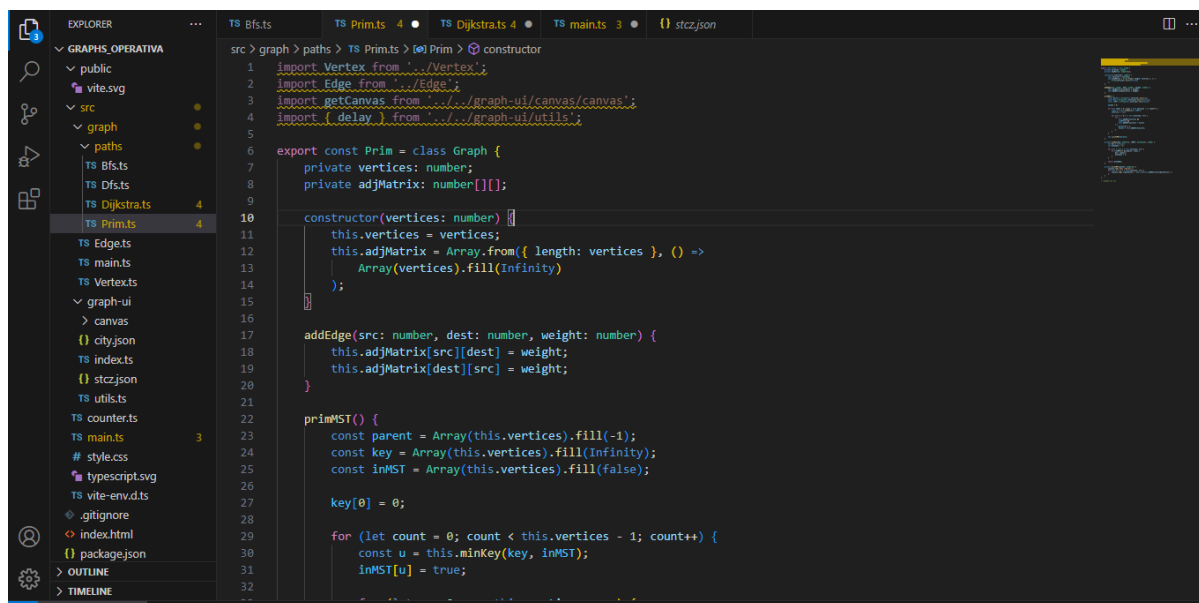
El **algoritmo de Prim** es uno de los métodos más conocidos para encontrar el **Árbol de Expansión Mínima (MST, por sus siglas en inglés)** en un **grafo no dirigido** y ponderado. Un árbol de expansión mínima es un subgrafo que conecta todos los vértices del grafo sin crear ciclos y con el menor costo total posible. Este algoritmo es de tipo **greedy** (codicioso), lo que significa que en cada paso selecciona la opción más ventajosa a corto plazo (es decir, la arista de menor peso) con la esperanza de que esta decisión conduzca a una solución global óptima.

### Características principales:

Funciona en grafos **no dirigidos** donde las aristas tienen pesos.

Encuentra un árbol de expansión mínima que conecta todos los nodos del grafo.

Es un algoritmo **greedy** (codicioso), seleccionando siempre la arista de menor peso que conecta un nodo al árbol de expansión.



```
src > graph > paths > TS Prim.ts > Prim > constructor
1  import Vertex from '../Vertex';
2  import Edge from '../Edge';
3  import getCanvas from '../graph-ui/canvas/canvas';
4  import { delay } from '../graph-ui/utils';
5
6  export const Prim = class Graph {
7    private vertices: number;
8    private adjMatrix: number[][];
9
10   constructor(vertices: number) {
11     this.vertices = vertices;
12     this.adjMatrix = Array.from({ length: vertices }, () =>
13       Array(vertices).fill(Infinity)
14     );
15   }
16
17   addEdge(src: number, dest: number, weight: number) {
18     this.adjMatrix[src][dest] = weight;
19     this.adjMatrix[dest][src] = weight;
20   }
21
22   primMST() {
23     const parent = Array(this.vertices).fill(-1);
24     const key = Array(this.vertices).fill(Infinity);
25     const inMST = Array(this.vertices).fill(false);
26
27     key[0] = 0;
28
29     for (let count = 0; count < this.vertices - 1; count++) {
30       const u = this.minKey(key, inMST);
31       inMST[u] = true;
32     }
33   }
34 }
```

```

src > graph > paths > TS Prim.ts > Prim > minKey
6  export const Prim = class Graph {
22  primMST() {
32
33      for (let v = 0; v < this.vertices; v++) {
34          if (
35              this.adjMatrix[u][v] &&
36              !inMST[v] &&
37              this.adjMatrix[u][v] < key[v]
38          ) {
39              parent[v] = u;
40              key[v] = this.adjMatrix[u][v];
41          }
42      }
43      this.printMST(parent);
44  }
45  private minKey(key: number[], inMST: boolean[]): number {
46      let min = Infinity;
47      let minIndex = -1;
48
49      for (let v = 0; v < this.vertices; v++) {
50          if (!inMST[v] && key[v] < min) {
51              min = key[v];
52              minIndex = v;
53          }
54      }
55      return minIndex;
56  }
57  private printMST(parent: number[]) {
58      console.log('Edge \tWeight');
59      for (let i = 1; i < this.vertices; i++) {
60          console.log(`${parent[i]} - ${i} \t${this.adjMatrix[i][parent[i]]}`);
61      }
62  }
63  }

```

## CONCLUSIONES

1. **Dijkstra** es el algoritmo más adecuado para encontrar rutas mínimas considerando pesos de las aristas (distancias, superficies de la vía).
2. Prim es útil para encontrar una estructura de conexión mínima entre ciudades.
3. BFS y DFS son algoritmos menos eficientes en el contexto de rutas optimizadas, pero útiles para explorar la conectividad.
4. Las características de las vías, como el tipo de superficie y el sentido, tienen un impacto significativo en la optimización de rutas, y su consideración en los algoritmos puede mejorar la precisión de los resultados.