UNIVERSIDAD AUTÓNOMA “TOMÁS FRÍAS”

CARRERA DE “INGENIERÍA DE SISTEMAS”

INVESTIGACIÓN OPERATIVA II

LABORATORIO

APLICACIÓN DE ALGORITMOS DE GRAFOS A CIUDADES DE BOLIVIA

NOMBRE: UNIV. LUIS DANIEL ACUÑA OYOLA

DOCENTE: ING DITMAR CASTRO ANGULO

POTOSÍ – BOLIVIA

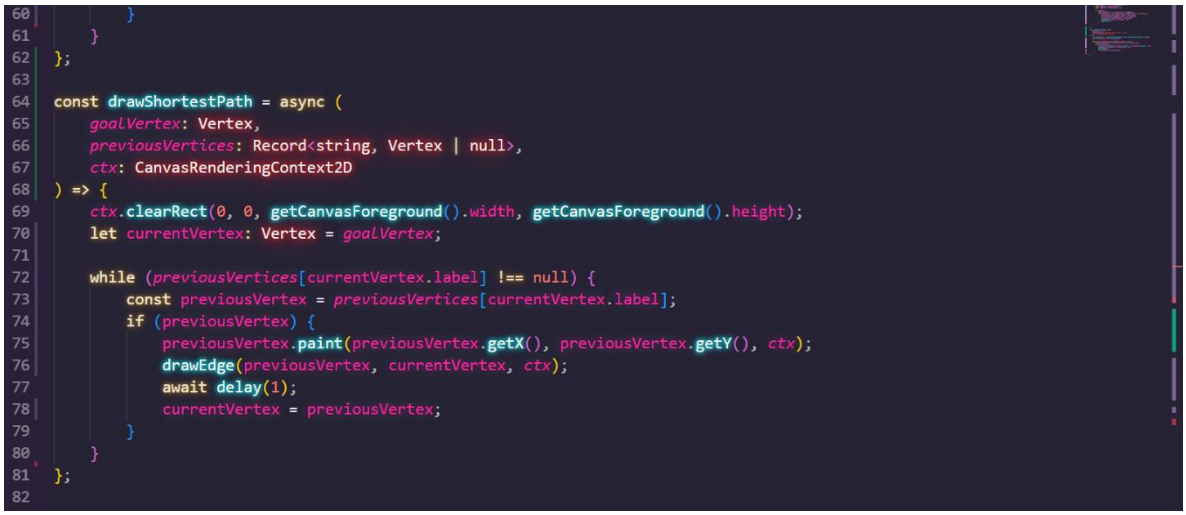
2024

LABORATORIO

ALGORITMO DE DIJKSTRA:

- **IMPLEMENTAR EL ALGORITMO DE DIJKSTRA EN LOS GRAFOS PROPORCIONADOS**

```ts
import Vertex from '../Vertex';
import { getCanvasForeground } from '../../graph-ui/canvas/canvas';
import { delay, drawEdge } from '../../graph-ui/utils';
import MinHeap from './MinHeap';

export const Dijkstra = async (initialVertex: Vertex, goalVertex: Vertex) => {
    if (initialVertex === goalVertex) {
        alert("El vértice inicial y el de destino son los mismos");
        return;
    }

    const shortestDistances: Record<string, number> = { [initialVertex.label]: 0 };
    const previousVertices: Record<string, Vertex | null> = {};
    const priorityQueue = new MinHeap();
    priorityQueue.insert(initialVertex, 0);

    const ctx = getCanvasForeground().getContext('2d');
    if (!ctx) {
        throw new Error('No se pudo obtener el contexto 2D');
    }

    while (!priorityQueue.isEmpty()) {
        const currentNode = priorityQueue.extractMin();
        if (!currentNode) break;

        const { vertex: currentVertex, distance: currentDistance } = currentNode;
        currentVertex.paint(currentVertex.getX(), currentVertex.getY(), ctx);
        await delay(1);

        await exploreNeighbors(currentVertex, currentDistance, shortestDistances, previousVertices, priorityQueue,

        if (currentVertex === goalVertex) break;
    }

    await drawShortestPath(goalVertex, previousVertices, ctx);
    return { shortestDistances, previousVertices };
};

const exploreNeighbors = async (
    currentVertex: Vertex,
    currentDistance: number,
    shortestDistances: Record<string, number>,
    previousVertices: Record<string, Vertex | null>,
    priorityQueue: MinHeap,
    ctx: CanvasRenderingContext2D
) => {
    for (const edge of currentVertex.getNeighbors()) {
        const neighbor = edge.destination;
        const edgeWeight = edge.weight || 0;

        if (neighbor) {
            const newDistance = currentDistance + edgeWeight;
            if (newDistance < (shortestDistances[neighbor.label] || Infinity)) {
                shortestDistances[neighbor.label] = newDistance;
                previousVertices[neighbor.label] = currentVertex;
                priorityQueue.insert(neighbor, newDistance);
                drawEdge(currentVertex, neighbor, ctx);
                await delay(1);
            }
```



```
        }
    }
};

const drawShortestPath = async (
    goalVertex: Vertex,
    previousVertices: Record<string, Vertex | null>,
    ctx: CanvasRenderingContext2D
) => {
    ctx.clearRect(0, 0, getCanvasForeground().width, getCanvasForeground().height);
    let currentVertex: Vertex = goalVertex;

    while (previousVertices[currentVertex.label] !== null) {
        const previousVertex = previousVertices[currentVertex.label];
        if (previousVertex) {
            previousVertex.paint(previousVertex.getX(), previousVertex.getY(), ctx);
            drawEdge(previousVertex, currentVertex, ctx);
            await delay(1);
            currentVertex = previousVertex;
        }
    }
};
```

- **EVALÚA EL TIPO DE SUPERFICIE Y SENTIDO DE LA VÍA EN EL CÁLCULO DEL ALGORITMO**

  Cada tipo de carretera puede tener un peso distinto que refleja factores como su capacidad, la velocidad máxima permitida o las condiciones del tráfico. Por ejemplo, una carretera principal podría tener un peso menor, lo que indica que es más rápida y directa en comparación con una vía residencial, que generalmente es más lenta. Al incorporar el tipo de carretera, el algoritmo de Dijkstra puede priorizar rutas que, aunque sean más largas en distancia, resulten más rápidas en términos de tiempo de viaje. Además, la superficie de la carretera influye en la velocidad de desplazamiento: las superficies más lisas, como el asfalto, permiten velocidades más altas en contraste con caminos de tierra, que tienden a reducir la velocidad del trayecto.

  Las vías de un solo sentido también afectan directamente el grafo, ya que no permiten transitar en sentido contrario. Esto puede obligar a tomar rutas más largas si el algoritmo no considera la dirección de las vías, lo que podría llevar a subestimar el tiempo real de viaje. Implementar una penalización para las vías de un solo sentido puede influir en la elección de caminos por parte del algoritmo.

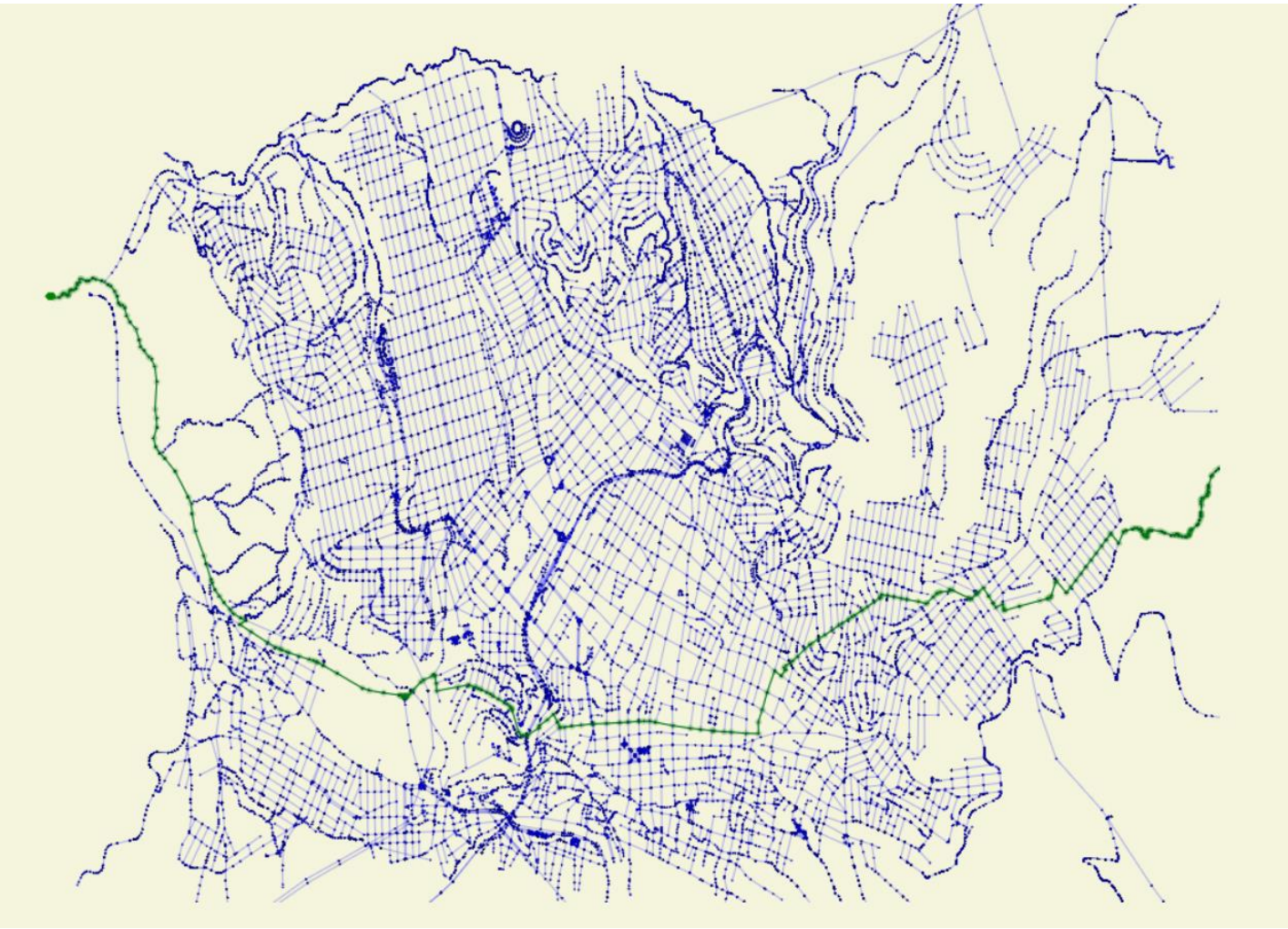
  Al integrar estos atributos, el algoritmo de Dijkstra no solo calculará la ruta más corta en términos de distancia, sino también en función del tiempo o el costo, proporcionando una solución más realista y útil.

- **MIDE Y REPORTA EL TIEMPO DE EJECUCIÓN DEL ALGORITMO**

  Se implementó el algoritmo de Dijkstra para identificar las rutas más cortas en la ciudad de Potosí, utilizando un extremo de la ciudad ('114.42353777773678_229.00422222237103') y otro extremo de la ciudad (‘1057.628800000064_306.54866666696034’) como puntos de referencia. La ejecución del algoritmo tomó 2:22,79 minutos, durante los cuales se

exploró toda la red vial, evaluando diferentes caminos y conexiones hasta alcanzar el destino final.

Este proceso no solo se enfocó en encontrar la ruta más corta en términos de distancia, sino que también consideró factores como el tipo de superficie de las calles y la dirección de las vías, lo que contribuyó a optimizar aún más el resultado. Una vez identificado el destino, se trazó visualmente la ruta más corta, lo que proporcionó una representación clara del recorrido entre el punto de partida y el destino final.



ALGORITMO PRIM:

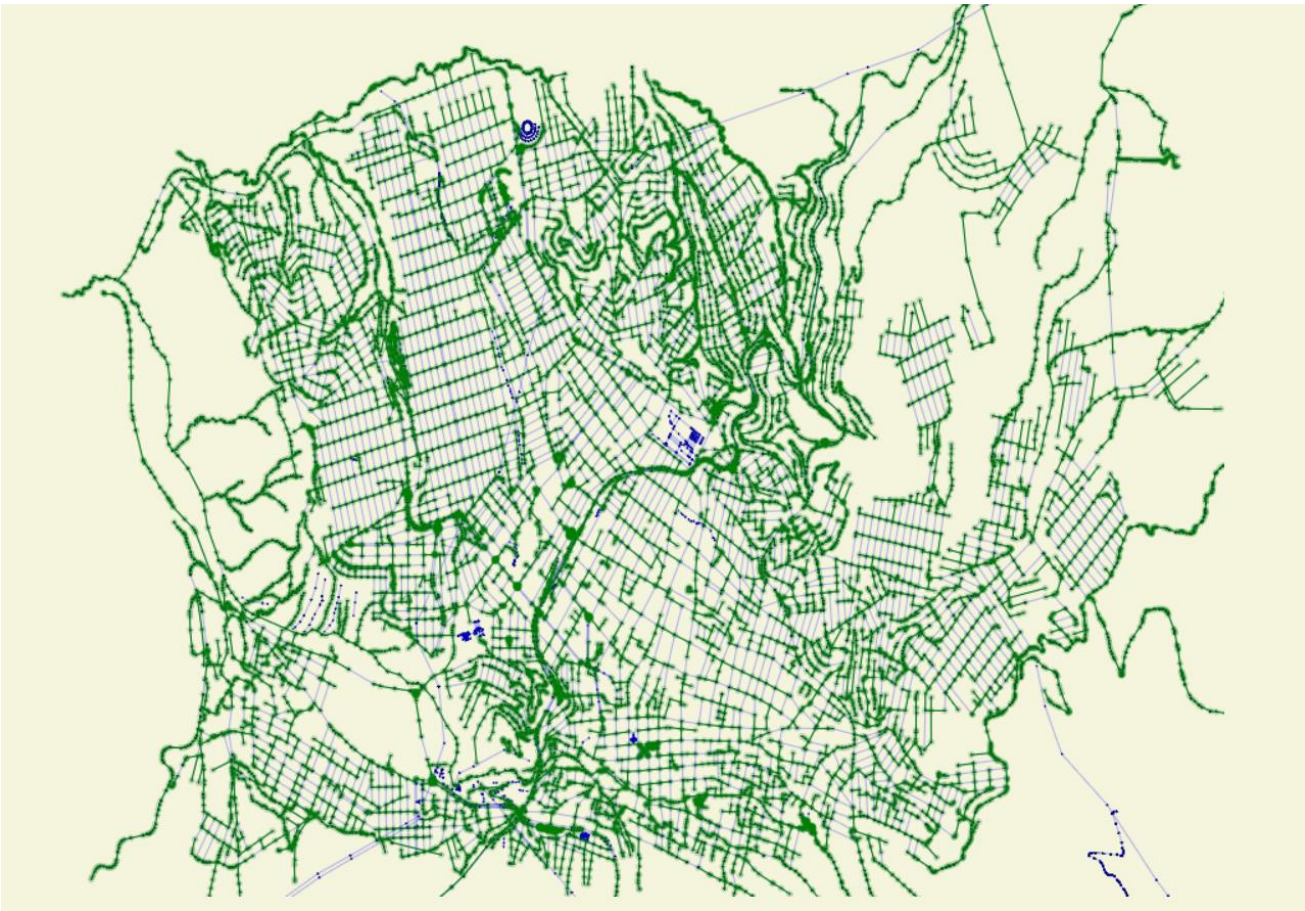
- **IMPLEMENTAR EL ALGORITMO DE PRIM EN EL GRAFO DE CIUDADES Y MUESTRA EL ARBOL DE EXPANSIÓN MINIMA RESULTANTE**

```typescript
import Vertex from '../Vertex';
import Edge from '../Edge';
import MinHeap from './MinHeap';
import { getCanvasForeground } from '../../graph-ui/canvas/canvas';
import { delay, drawEdge } from '../../graph-ui/utils';

export const Prim = async (startVertex: Vertex): Promise<Edge[]> => {
    const mst: Edge[] = [];
    const visitedVertices: Set<string> = new Set();
    const edgeQueue = new MinHeap();
    const ctx = getCanvasForeground().getContext('2d');

    if (!ctx) {
        throw new Error('No se pudo obtener el contexto 2D');
    }

    addNeighborEdgesToQueue(startVertex, edgeQueue);

    while (!edgeQueue.isEmpty()) {
        const smallestEdgeData = edgeQueue.extractMin();
        if (!smallestEdgeData) continue;

        const { edge: currentEdge } = smallestEdgeData;
        const { source: sourceVertex, destination: destinationVertex } = currentEdge;

        if (!destinationVertex || visitedVertices.has(destinationVertex.label)) {
            continue;
        }

        visitedVertices.add(destinationVertex.label);
        await paintAndConnectVertices(ctx, sourceVertex, destinationVertex, currentEdge);
        addNeighborEdgesToQueue(destinationVertex, edgeQueue, visitedVertices);
    }

    return mst;
;

onst addNeighborEdgesToQueue = (vertex: Vertex, edgeQueue: MinHeap, visitedVertices: Set<string> = new Set()):
oid => {
    for (const edge of vertex.getNeighbors()) {
        if (!edge.destination || visitedVertices.has(edge.destination.label)) {
            continue;
        }
        edgeQueue.insert(edge, edge.weight);
    }
;

onst paintAndConnectVertices = async (ctx: CanvasRenderingContext2D, sourceVertex: Vertex | null,
    destinationVertex: Vertex, edge: Edge): Promise<void> => {
    if (sourceVertex) {
        destinationVertex.paint(destinationVertex.getX(), destinationVertex.getY(), ctx);
        await delay(1);
        drawEdge(sourceVertex, destinationVertex, ctx);
    }
;
```

- **MIDE Y REPORTA EL TIEMPO DE EJECUCIÓN DEL ALGORITMO**

  Se aplicó el algoritmo de Prim para determinar el árbol de expansión mínima (MST) en la ciudad de Potosí, comenzando desde un extremo de la ciudad (‘114.42353777773678_229.00422222237103’). Este proceso tuvo una duración aproximada de 1:07,52 minutos, durante los cuales se analizó toda la red vial de la ciudad, garantizando que todos los puntos del grafo fueran visitados.

  Durante esta ejecución, el algoritmo evaluó las conexiones entre las distintas áreas, buscando la forma más eficiente de unir todos los puntos con el menor costo posible.



ANÁLISIS DE BFS Y DFS:

- **REALIZA UN ANÁLISIS DE SU COMPORTAMIENTO EN LOS GRAFOS DE LAS CIUDADES BOLIVIANAS**

  Por otro lado, el algoritmo de Búsqueda en Profundidad (DFS) se distingue por su método de exploración, que se adentra en cada rama del grafo lo más profundamente posible antes de retroceder. Comienza en un vértice de origen y sigue un camino hasta llegar a un punto muerto, momento en el cual regresa y busca otros caminos por explorar.

Al implementar DFS en el grafo de la ciudad de Potosí, se notó que el recorrido fue altamente eficiente en términos de la cantidad de nodos visitados. Este algoritmo se completó en solo 09,89 segundos, lo que representa una velocidad considerablemente mayor en comparación con el algoritmo BFS. Esto pone de manifiesto la capacidad de DFS para manejar grandes conjuntos de datos de manera más ágil, especialmente en entornos urbanos donde la complejidad de las conexiones puede ser significativa.

Por otro lado, el algoritmo BFS, o Búsqueda en Anchura, opera explorando todos los vértices en un nivel de profundidad antes de avanzar al siguiente. Comienza en un vértice de origen y visita todos los nodos vecinos, luego se mueve a los vecinos de esos nodos, continuando este proceso de forma sistemática.

Al aplicar BFS al grafo de la ciudad de Potosí, se observó que el recorrido completo tomó 2:11,89 minutos. Este tiempo puede parecer extenso, especialmente al considerar la necesidad de recorrer una ciudad con numerosas intersecciones y caminos. La lentitud del algoritmo se debe a su enfoque: espera a que todos los nodos en el mismo nivel sean visitados antes de continuar, lo que puede resultar en un tiempo de procesamiento prolongado en redes de transporte más complejas.

- **COMPARA SUS RESULTADOS CON LOS OBTENIDOS UTILIZANDO PRIM Y DIJKSTRA EN TERMINOS DE ESTRUCTURA DE GRAFOS, Y REPORTA OBSERVACIONES SOBRE EFICIENCIA Y APLICABILIDAD EN DIFERENTES CONTEXTOS**

  En cuanto a eficiencia temporal, el algoritmo de búsqueda en profundidad (DFS) se destaca como el más ágil, seguido por el algoritmo de Prim, luego el algoritmo de Dijkstra, y finalmente, el algoritmo de búsqueda en anchura (BFS). Este orden de velocidad revela cómo cada uno de estos algoritmos maneja la exploración y evaluación de nodos en un grafo.

  Al considerar la optimización de rutas y conexiones, el algoritmo de Dijkstra sobresale al identificar las rutas más cortas, teniendo en cuenta tanto la distancia como el tiempo requerido para recorrerlas. En contraste, el algoritmo de Prim se centra en minimizar los costos de conexión entre los nodos. Aunque BFS y DFS son herramientas eficaces para explorar y recorrer grafos, su enfoque es diferente.

  En términos de complejidad de implementación, incorporar atributos como el tipo de superficie de las carreteras y la dirección de circulación en el algoritmo de Dijkstra puede complicar un poco su funcionamiento. Sin embargo, esta adición mejora significativamente la precisión en la identificación de las rutas óptimas.