UNIVERSIDAD AUTÓNOMA “TOMÁS FRÍAS”

CARRERA DE “INGENIERÍA DE SISTEMAS”

INVESTIGACIÓN OPERATIVA II (SIS-544)

LABORATORIO:
APLICACIÓN DE ALGORITMOS DE GRAFOS A CIUDADES DE BOLIVIA

NOMBRE: UNIV. ALEX ADRIÁN MÉNDEZ MOREIRA

Potosí – Bolivia

2024

Algoritmo de Dijkstra (Optimización de rutas):

- **Implementa el algoritmo de Dijkstra en los grafos proporcionados .**

```
export const Dijkstra = async (source: Vertex, destination: Vertex) => {
    if (source === destination) {
        alert("El origen y el destino son el mismo");
        return;
    }

    const distances: Record<string, number> = { [source.label]: 0 };
    const previous: Record<string, Vertex | null> = {};
    const minHeap = new MinHeap();
    minHeap.insert(source, 0);

    const ctx = getCanvasForeground().getContext('2d');
    if (!ctx) {
        throw new Error('Fallo al obtener el contexto 2D');
    }

    while (!minHeap.isEmpty()) {
        const current = minHeap.extractMin();
        if (!current) break;

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

        await processNeighbors(currentVertex, currentDistance, distances,
previous, minHeap, ctx);

        if (currentVertex === destination) break;
    }

    await drawShortestPath(destination, previous, ctx);
    return { distances, previous };
};

const processNeighbors = async (
    currentVertex: Vertex,
    currentDistance: number,
    distances: Record<string, number>,
    previous: Record<string, Vertex | null>,
    minHeap: MinHeap,
    ctx: CanvasRenderingContext2D
) => {
    for (const edge of currentVertex.getNeighbors()) {
        const neighbor = edge.destination;
        const weight = edge.weight || 0;

        if (neighbor) {
            const newDistance = currentDistance + weight * calculateWeight(edge);
```

```
            if (newDistance < (distances[neighbor.label] || Infinity)) {
                distances[neighbor.label] = newDistance;
                previous[neighbor.label] = currentVertex;
                minHeap.insert(neighbor, newDistance);
                drawEdge(currentVertex, neighbor, ctx);
                await delay(1);
            }
        }
    }
};

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

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

- **Evalúa el impacto del tipo de superficie y sentido de la vía en el cálculo del algoritmo.**

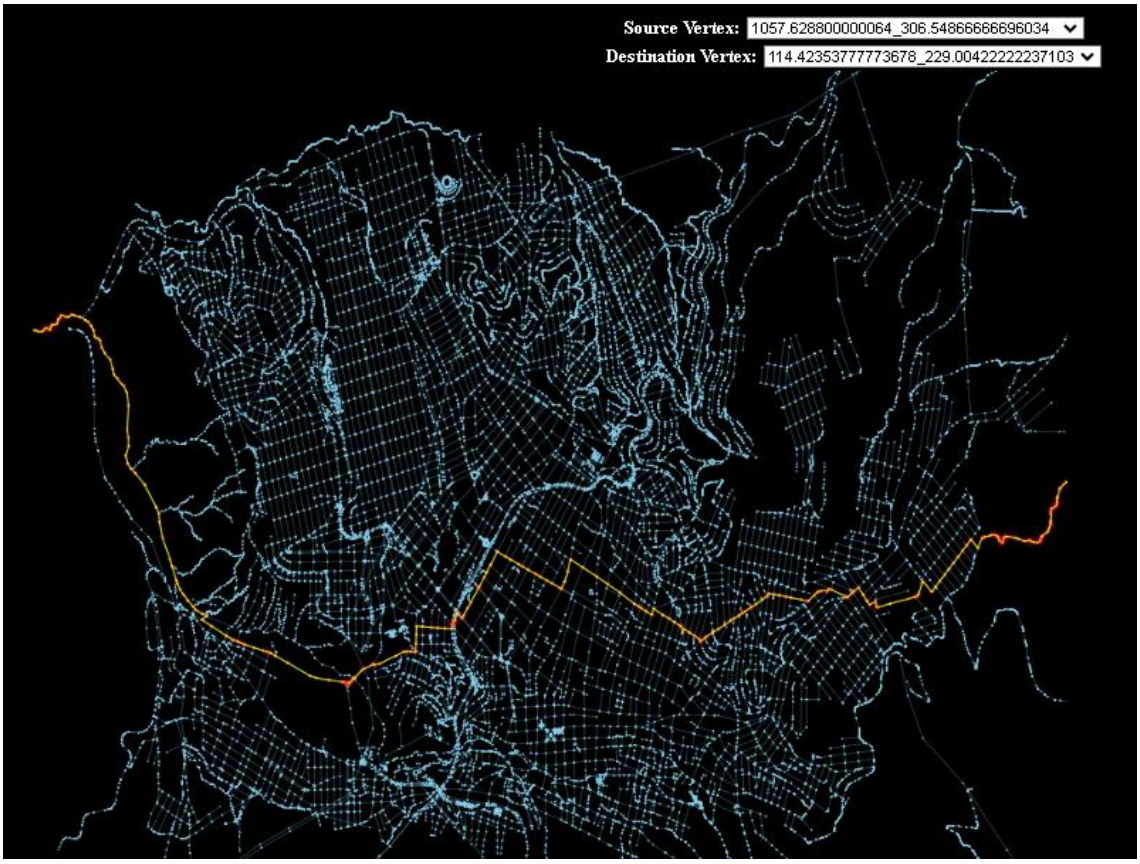
  Cada tipo de carretera puede tener un peso diferente que refleja su capacidad, la velocidad máxima permitida o las condiciones de tráfico. Por ejemplo, una carretera primaria (primary) podría tener un peso más bajo, lo que indica que es más rápida y directa en comparación con una residencial (residential), que podría ser más lenta. Al considerar el tipo de carretera, el algoritmo de Dijkstra puede favorecer rutas que, aunque sean más largas en distancia, son más rápidas en tiempo de viaje. Así mismo la superficie de la carretera puede afectar la velocidad de desplazamiento. Las superficies más suaves, como el asfalto, permitirán velocidades más altas en comparación con caminos de tierra, que podrían ralentizar el trayecto.
  Las vías de un solo sentido impactan directamente en el grafo, ya que no se puede transitar en la dirección opuesta. Esto puede generar rutas más largas si el algoritmo no tiene en cuenta la dirección, lo que podría llevar a subestimar el tiempo de viaje real. Si se implementa una penalización para las vías de un solo sentido, esto puede influir en la selección de caminos por parte del algoritmo.

Al incorporar estos atributos, el algoritmo de Dijkstra no solo calculará el camino más corto en términos de distancia, sino también en términos de tiempo o costo, proporcionando una solución más útil y realista.

- **Mide y reporta el tiempo de ejecución del algoritmo.**

  Se utilizó el algoritmo de Dijkstra para encontrar las rutas más cortas en la ciudad de Potosí, tomando como puntos de referencia los extremos de la ciudad. La ejecución del algoritmo duró 2.30 minutos, durante los cuales se exploró toda la red vial, revisando diferentes caminos y conexiones hasta alcanzar el destino final.

  Este proceso no solo buscó la ruta más corta en términos de distancia, sino que también tuvo en cuenta aspectos como el tipo de superficie de las calles y el sentido de las vías, lo que ayudó a optimizar aún más el resultado. Una vez que se localizó el destino, se trazó visualmente la ruta más corta, proporcionando una representación clara del recorrido entre el punto de inicio y el de destino.

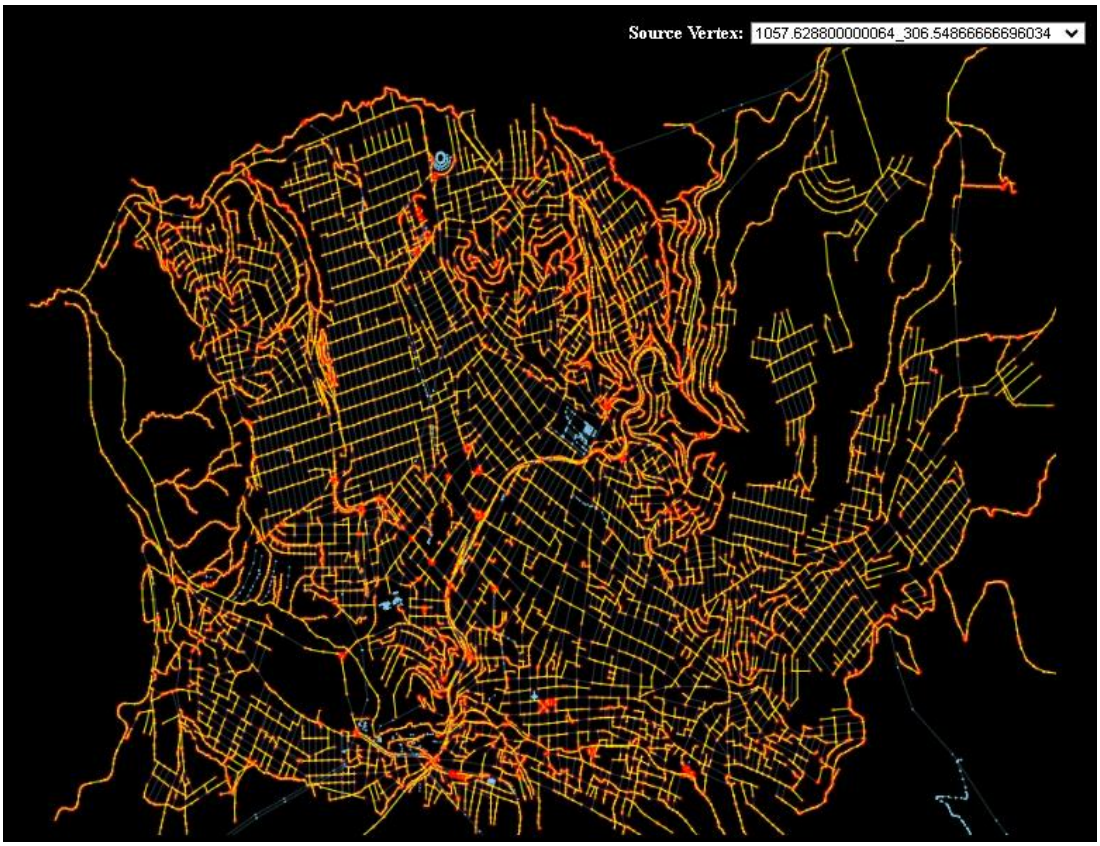


Algoritmo de Prim (Optimización de conexiones):

- **Implementa el algoritmo de Prim en el grafo de ciudades y muestra el Árbol de Expansión Mínima resultante.**

```
export const Prim = async (startVertex: Vertex): Promise<Edge[]> => {
    const mstEdges: Edge[] = [];
    const visited: Set<string> = new Set();
    const minHeap = new MinHeap();
    const ctx = getCanvasForeground().getContext('2d');
    if (!ctx) {
        throw new Error('Failed to get 2D context');
    }
    addEdgesToHeap(startVertex, minHeap);
    while (!minHeap.isEmpty()) {
        const currentEdgeData = minHeap.extractMin();
        if (!currentEdgeData) continue;
        const { edge: currentEdge } = currentEdgeData;
        const { source: currentVertex, destination: neighbor } = currentEdge;
        if (!neighbor || visited.has(neighbor.label)) {
            continue;
        }
        visited.add(neighbor.label);
        await paintAndDrawEdge(ctx, currentVertex, neighbor, currentEdge);
        addEdgesToHeap(neighbor, minHeap, visited);
    }
    return mstEdges;
};

const addEdgesToHeap = (vertex: Vertex, minHeap: MinHeap, visited: Set<string> =
new Set()): void => {
    for (const edge of vertex.getNeighbors()) {
        if (!edge.destination || visited.has(edge.destination.label)) {
            continue;
        }
        minHeap.insert(edge, edge.weight);
    }
};
const paintAndDrawEdge = async (ctx: CanvasRenderingContext2D, currentVertex:
Vertex | null, neighbor: Vertex, edge: Edge): Promise<void> => {
    if (currentVertex) {
        neighbor.paint(neighbor.getX(), neighbor.getY(), ctx);
        await delay(1);
        drawEdge(currentVertex, neighbor, ctx);
    }
};
```



- **Mide y reporta el tiempo de ejecución del algoritmo.**

  Se utilizó el algoritmo de Prim para encontrar el árbol de expansión mínima (MST) en la ciudad de Potosí, comenzando desde uno de sus extremos. Este proceso duró aproximadamente 1.10 minutos, durante los cuales se analizó toda la red vial de la ciudad, asegurando que se visitaran todos los puntos del grafo.

  A lo largo de esta ejecución, el algoritmo examinó las conexiones entre las diferentes áreas, buscando la manera 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.**

  El algoritmo **BFS, o Búsqueda en Anchura**, funciona de manera que explora todos los vértices en un nivel de profundidad antes de pasar al siguiente. Comienza en un vértice de origen, y visita todos los nodos vecinos. Luego, avanza a los vecinos de esos nodos, continuando este proceso de manera sistemática.
  Cuando aplicamos BFS al grafo de la ciudad de Potosí, notamos que el recorrido completo tardó 2.25 minutos, esta duración parecer ser un poco extensa, especialmente si

consideramos la necesidad de recorrer una ciudad con múltiples intersecciones y caminos. La lentitud del algoritmo se debe a su naturaleza: 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.

El algoritmo de **Búsqueda en Profundidad (DFS)** se caracteriza por su método de exploración, donde se adentra en cada rama del grafo lo más profundamente posible antes de regresar. Inicia en un vértice de origen y sigue un camino hasta que se encuentra con un punto muerto, momento en el cual retrocede y busca otros caminos por explorar.
Al implementar DFS en el grafo de la ciudad de Potosí, se observó que el recorrido fue altamente eficiente en términos de la cantidad de nodos visitados. Este algoritmo se completó en un tiempo de 19 segundos, lo que representa una velocidad considerablemente mayor en comparación con el algoritmo de Búsqueda en Amplitud (BFS). Esto resalta la capacidad del algoritmo para manejar grandes conjuntos de datos de manera más ágil, especialmente en entornos urbanos donde la complejidad de las conexiones puede ser significativa.

- **Compara sus resultados con los obtenidos utilizando Prim y Dijkstra en términos de estructura de grafos, y reporta observaciones sobre eficiencia y aplicabilidad en diferentes contextos.**

  En términos de eficiencia temporal, el algoritmo de búsqueda en profundidad (DFS) se presenta como el más rápido o ágil, seguido por el algoritmo de Prim, después viene el algoritmo de Dijkstra, y finalmente, el algoritmo de búsqueda en anchura (BFS). Este orden de velocidad nos muestra cómo cada uno de estos algoritmos gestiona la exploración y la evaluación de nodos en un grafo.

  Cuando hablamos de optimización de rutas y conexiones, el algoritmo de Dijkstra brilla al identificar las rutas más cortas, teniendo en cuenta tanto la distancia como el tiempo que se tarda en recorrerlas. Por otro lado, el algoritmo de Prim se enfoca en reducir los costos de conexión entre los nodos. En cambio, aunque BFS y DFS son herramientas efectivas para explorar o recorrer grafos.

  En lo que respecta a la complejidad de implementación, añadir atributos como el tipo de superficie de las carreteras y el sentido de circulación en el algoritmo de Dijkstra puede complicar un poco su funcionamiento, pero a la vez mejora significativamente la precisión al encontrar las rutas óptimas.