LABORATORIO-INFORME

APLICACIÓN DE ALGORITMOS DE GRAFOS A CIUDADES DE BOLIVIA

Nombre: Univ. Jacob Santos Ayaviri Condori

Docente: Ing. Ditmar Castro Angulo

ALGORTIMO DE DIJKSTRA

- **Implementación del algoritmo de Dijkstra en los grafos**

```
export const Dijkstra = async (source: Vertex, destination: Vertex) => {
    if (source === destination) {
        alert("El punto de partida y el de llegada son los mismos");
        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('no se pudo 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;
            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;
        }
    }
};
```

- **Análisis del impacto del tipo de superficie y la dirección de la vía en el cálculo del algoritmo**

Cada tipo de carretera puede estar asociado a un peso específico, que representa factores como su capacidad, la velocidad máxima permitida o las condiciones de tráfico. Por ejemplo, una carretera principal (primary) podría asignarse un peso menor, indicando que es más rápida y eficiente en comparación con una carretera residencial (residential), que probablemente sea más lenta. Al incluir el tipo de carretera, el algoritmo de Dijkstra puede priorizar rutas que, aunque sean más largas en términos de distancia, resulten más rápidas en tiempo de viaje. Asimismo, la superficie de la carretera puede influir en la velocidad de desplazamiento. Superficies más suaves, como el asfalto, permiten velocidades más elevadas en comparación con caminos de tierra, que pueden ralentizar el trayecto.

Las vías de un solo sentido también afectan directamente la representación del grafo, ya que impiden la circulación en sentido contrario. Esto podría generar rutas más extensas si el algoritmo no toma en cuenta la dirección, lo que podría resultar en una subestimación del tiempo de viaje real. Al introducir una penalización para las vías unidireccionales, se puede influir en la elección de caminos que realiza el algoritmo.

Incorporar estos atributos en el cálculo permite que el algoritmo de Dijkstra no solo determine el trayecto más corto en términos de distancia, sino también en términos de tiempo o costo, proporcionando así una solución más práctica y ajustada a la realidad.

- **Medición y reporte del tiempo de ejecución del algoritmo**

Se empleó el algoritmo de Dijkstra para encontrar las rutas más cortas en la ciudad de Potosí, tomando como referencia los puntos más distantes de la ciudad. La ejecución del algoritmo duró 2.15 minutos, durante los cuales se exploró la totalidad de la red vial, analizando diversas rutas y conexiones hasta llegar al destino final.

Este procedimiento no solo identificó la ruta más corta en términos de distancia, sino que también tuvo en cuenta el tipo de superficie de las calles y la dirección de las vías, optimizando aún más el resultado. Tras localizar el destino, se dibujó visualmente la ruta óptima, ofreciendo una representación clara del trayecto entre el punto de inicio y el destino.

ALGORITMO PRIM

```
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('no se pudo obtener conexion 2D con el canvas');
    }
    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);
    }
};
```

- **Medición y reporte del tiempo de ejecución del algoritmo**
  Se utilizó el algoritmo de Prim para calcular el árbol de expansión mínima (MST) en la ciudad de Potosí, comenzando desde uno de los extremos de la ciudad. La ejecución del proceso tomó **1.05 minutos**, durante los cuales se analizó toda la red vial, asegurando que se visitaran todos los nodos del grafo.
  Durante este tiempo, el algoritmo evaluó las conexiones entre las distintas áreas de la ciudad, buscando la manera más eficiente de conectar todos los puntos con el menor costo posible. Este enfoque permitió optimizar la red de caminos, minimizando el recorrido total sin dejar ninguna área sin cubrir

Análisis de BFS y DFS

Comportamiento de los Algoritmos en Ciudades Bolivianas

El algoritmo de Búsqueda en Anchura (BFS) explora todos los nodos de un nivel antes de pasar al siguiente. Al aplicarlo al grafo de la ciudad de Potosí, se observó que el recorrido completo tardó **2.25 minutos**. Este tiempo es considerable, especialmente en una ciudad con muchas intersecciones. La lentitud de BFS se debe a que espera a visitar

todos los nodos en un mismo nivel, lo que puede resultar en tiempos prolongados en redes complejas.

En contraste, la Búsqueda en Profundidad (DFS) sigue un enfoque diferente, profundizando en cada rama del grafo hasta llegar a un punto muerto, luego retrocede y busca otros caminos. Al implementar DFS en Potosí, el recorrido se completó en **19 segundos**, demostrando ser mucho más ágil que BFS. Esto resalta la eficacia de DFS en ambientes urbanos con conexiones complejas.

Comparación de Resultados

En términos de eficiencia, DFS es el más rápido, seguido de Prim, Dijkstra y, por último, BFS. Dijkstra se destaca por identificar las rutas más cortas, considerando tanto la distancia como el tiempo de recorrido, mientras que Prim minimiza los costos de conexión. Aunque BFS y DFS son útiles para explorar grafos, su enfoque no está diseñado para optimizar rutas.

Incorporar atributos como la superficie de las carreteras y el sentido de circulación en Dijkstra puede complicar su implementación, pero mejora la precisión en la búsqueda de rutas óptimas.