Documentación Técnica del Proyecto: `mapa-upa`

Este documento proporciona una guía detallada sobre la arquitectura, estructuras de datos, flujo de la aplicación y componentes del proyecto `mapa-upa`.

1. Visión General

`mapa-upa` es una aplicación web interactiva desarrollada con **Next.js** que permite visualizar un mapa del campus (UPA) y calcular rutas óptimas entre puntos de interés utilizando el algoritmo *A (A-Star)**.

Tecnologías Clave

- **Frontend**: Next.js 16 (React 19), TypeScript.
- **Mapas**: Leaflet (vía `react-leaflet`), OpenStreetMap.
- **Estado**: Zustand (gestión de estado global).
- **Estilos**: Tailwind CSS, Shadcn/UI.
- **GeoDatos**: `@tmcw/togeojson` (KML → GeoJSON).

---

2. Estructuras de Datos

Los datos originales provienen de un archivo GeoJSON (derivado de un KML). Para que la aplicación pueda utilizarlos, se transforman en tiempo de ejecución:

1. **Entrada (GeoJSON)**: Datos crudos con geometrías complejas.
2. **Proceso**:
    - Se extraen los puntos (`Point`) y se convierten en objetos `Vertex`.
    - Se extraen las líneas (`LineString`) y se convierten en objetos `Edge`.
    - Se calculan las distancias reales (pesos) y se vinculan los vértices con las aristas.

3. **Resultado**: Un conjunto de datos limpio y tipado (`Vertex[]` y `Edge[]`) que es *entendible* y procesable directamente por el algoritmo A* y los componentes de React.

2.1 Vértice (`Vertex`)

Representa un nodo o punto de interés en el grafo del mapa.

```
// types/vertex.ts
type Vertex = {
  id: number;          // Identificador único (índice en arra
y)
  name: string;        // Identificador interno (ej: "V1")
  label: string;       // Nombre legible (ej: "Edificio A")
  coordinates: {       // Ubicación geográfica
    lat: number;
    lon: number;
  };
};
```

2.2 Arista (`Edge`)

Representa la conexión entre dos vértices, con un peso asociado (distancia).

```
// types/edge.ts
type Edge = {
  id: number;          // Identificador único
  name: string;        // Nombre (ej: "E1")
  weight: number;      // Distancia en metros (costo para A*)
  coordinates: { lat: number; lon: number }[]; // Trayectoria
visual de la línea
  from: Vertex;        // Nodo origen
  to: Vertex;          // Nodo destino
};
```

3. Análisis detallado: Algoritmo A* (`scripts/aStar.ts`)

El archivo `scripts/aStar.ts` contiene la lógica core para encontrar la ruta más corta. A diferencia de Dijkstra, A* utiliza una función heurística para guiar la búsqueda hacia el objetivo, haciéndolo más eficiente para mapas espaciales.

3.1 Definición de Tipos Internos

Para el funcionamiento del algoritmo, se definen estructuras auxiliares:

- `CameFromEntry`: Rastrea cómo llegamos a un nodo. Fundamental para reconstruir el camino al final.

```
type CameFromEntry = {
  prev: number;   // ID del nodo anterior
  edgeId: number; // ID de la arista usada para llegar aqu
í
}
```

- `AStarResult`: Lo que retorna la función principal.

```
type AStarResult = {
  path: PathResult | null;       // El camino encontrado
(si existe).
  visitedOrder: number[];        // Lista ordenada de nodo
s visitados (para la animación).
  cameFrom: Map<number, CameFromEntry>; // Mapa completo d
e exploración.
}
```

3.2 Lógica del Algoritmo

La función `aStar` sigue estos pasos precisos:

1. **Inicialización**:
    - `openSet`: Un `Set` que contiene los nodos por evaluar. Inicialmente solo tiene `startId`.

- `gScore`: Un `Map` que guarda el costo real más barato conocido desde el inicio hasta el nodo actual. `gScore[start] = 0`.
- `fScore`: Un `Map` que guarda `gScore + heurística`. Estima el costo total pasando por este nodo. `fScore[start] = h(start)`.
- `visitedOrder`: Array para registrar el orden exacto de exploración.

2. **Bucle Principal (`while (openSet.size > 0)`)**:
   - **Selección del Mejor Nodo**: Se itera sobre `openSet` para encontrar el nodo con el menor `fScore`. Este es el nodo "más prometedor".
   - **Condición de Parada**: Si el nodo actual es el `goalId`, ¡hemos llegado! Se llama a `reconstructPath` y se retorna el resultado.
   - **Exploración**: Se mueve el nodo actual de `openSet` a procesado (se elimina del set). Se añade a `visitedOrder`.
3. **Expansión de Vecinos**:
   - Para cada vecino del nodo actual:
     - Se calcula un `tentativeG` (gScore actual + peso de la arista).
     - Si este camino es mejor que el que ya conocíamos para ese vecino (o si no lo conocíamos):
       - Actualizamos `cameFrom` (apuntando al nodo actual).
       - Actualizamos `gScore` y `fScore` del vecino.
       - Añadimos el vecino al `openSet` si no estaba ya.
4. **Reconstrucción del Camino (`reconstructPath`)**:
   - Esta función auxiliar comienza desde el `goalId`.
   - Retrocede paso a paso usando el mapa `cameFrom` hasta llegar al `startId`.
   - Devuelve los arrays de nodos y aristas en el orden correcto (desde el inicio hasta el final).

3.3 Heurística

La función `aStar` recibe un callback `getHeuristic`. En `Map.tsx`, esta función se define como la **distancia Haversine** (distancia en línea recta sobre la esfera

terrestre) entre el nodo evaluado y el objetivo. Esto asegura que el algoritmo priorice explorar nodos que geográficamente se acercan al destino.

---

4. Arquitectura y Flujo de la Aplicación

El flujo de datos principal sigue estos pasos:

4.1 Carga y Procesamiento de Datos

1. **Inicio (`layout.tsx`)**: Llama a la acción `loadKml()` del store global.
2. **Obtención (`useMapStore.ts`)**: Se descarga el archivo `map.kml` desde `public/`.
3. **Conversión**:
   - `KML` → `GeoJSON` (usando `@tmcw/togeojson`).
   - `GeoJSON` → `Graph Data` (usando `scripts/transformGeoJson.ts`).
   - Este último paso calcula pesos usando la fórmula de **Haversine** y conecta vértices con aristas.

4.2 Interacción del Usuario

1. **Selección (`DataPanel.tsx`)**: El usuario elige un "Origen" y "Destino" mediante menús desplegables.
2. **Actualización de Estado**: Se actualizan `startId` y `goalId` en el `useMapStore`.

4.3 Ejecución del Algoritmo A*

1. **Detección (`Map.tsx`)**: El componente detecta cambios en el origen/destino.
2. **Cálculo (`scripts/aStar.ts`)**:
   - Ejecuta `aStar(startId, goalId, graph, heuristic)`.
   - **Heurística**: Distancia euclidiana/Haversine directa al destino.
3. **Resultado**: Devuelve el camino óptimo (`path`) y la secuencia de exploración (`visitedOrder`).

4.4 Visualización

1. **Renderizado (`Map.tsx`)**:

- Dibuja vértices como `CircleMarker`.
- Dibuja aristas como `Polyline`.

2. **Animación**: Un temporizador (`setInterval`) recorre `visitedOrder`, actualizando el estado visual de las aristas para mostrar cómo el algoritmo explora el grafo antes de encontrar la ruta final.

---

5. Detalles de los Componentes

`components/Map.tsx`

Es el componente principal de visualización.

- **Responsabilidad**: Renderizar el mapa, manejar clics en nodos y orquestar la animación del algoritmo.
- **Lógica Clave**:
  - Construye el grafo en memoria (`useMemo`) a partir de `edges` y `vertices`.
  - Ejecuta `aStar` cada vez que cambian los puntos de inicio/fin.
  - Mantiene un estado `visitedIndex` para la animación paso a paso.

`store/useMapStore.ts`

Gestión de estado global con **Zustand**.

- Almacena la "Base de Datos" en memoria del mapa (`vertices`, `edges`).
- Controla variables de UI como el nivel de zoom y los puntos seleccionados.
- Contiene la lógica asíncrona (`loadKml`) para inicializar la app.

`scripts/transformGeoJson.ts`

Utilidad de transformación de datos backend-to-frontend.

- Procesa el GeoJSON plano.
- Identifica qué puntos son vértices y qué líneas son aristas.
- Calcula distancias geográficas reales para los pesos de las aristas.
- Realiza el "matching" geométrico para saber qué vértices conecta cada arista.