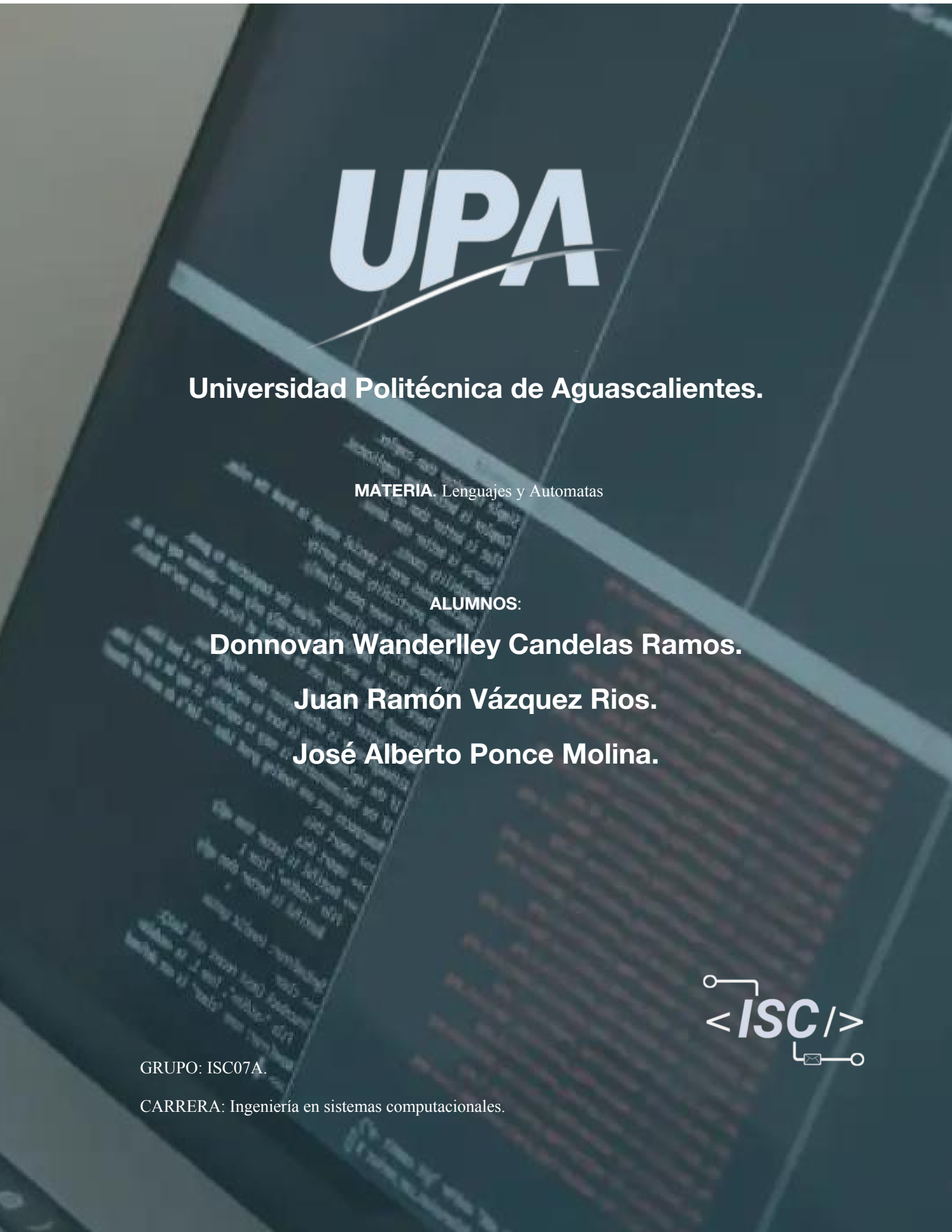


Universidad Politécnica de Aguascalientes.

MATERIA. Lenguajes y Automatas

ALUMNOS:

Donnovan Wanderlley Candelas Ramos.

Juan Ramón Vázquez Rios.

José Alberto Ponce Molina.

GRUPO: ISC07A.

CARRERA: Ingeniería en sistemas computacionales.

Índice

1. Descripción General

2. Propósito de la Aplicación

3. Pila Tecnológica

4. Arquitectura General de la Aplicación

5. Punto de Entrada y Bootstrap

6. Orquestador Central: Grafos

7. Estructuras de Datos

8. Componentes de la Interfaz de Usuario

9. Algoritmo de Búsqueda de Rutas

10. Flujo de Interacción del Usuario

11. Estructura del Proyecto

12. Patrón de Flujo de Datos

13. Características Clave del Diseño

    13.1. Unidireccionalidad del flujo de datos

    13.2. Patrón de diseño aplicado

14. Conclusión

1. Descripción General

Este documento describe de manera detallada la arquitectura, estructura y funcionamiento del sistema Nodos-UPA, una aplicación web desarrollada con React que representa el campus de la Universidad Politécnica de Aguascalientes como un grafo ponderado. El sistema permite calcular rutas óptimas entre distintas ubicaciones del campus utilizando el algoritmo de Dijkstra.

2. Propósito de la Aplicación

El propósito de Nodos-UPA es proporcionar una herramienta interactiva que permita a los usuarios:

- Visualizar el campus como una red de nodos interconectados
- Seleccionar un nodo de origen y un nodo destino
- Calcular la ruta más corta entre ambos puntos
- Mostrar la distancia total y la secuencia de nodos recorridos

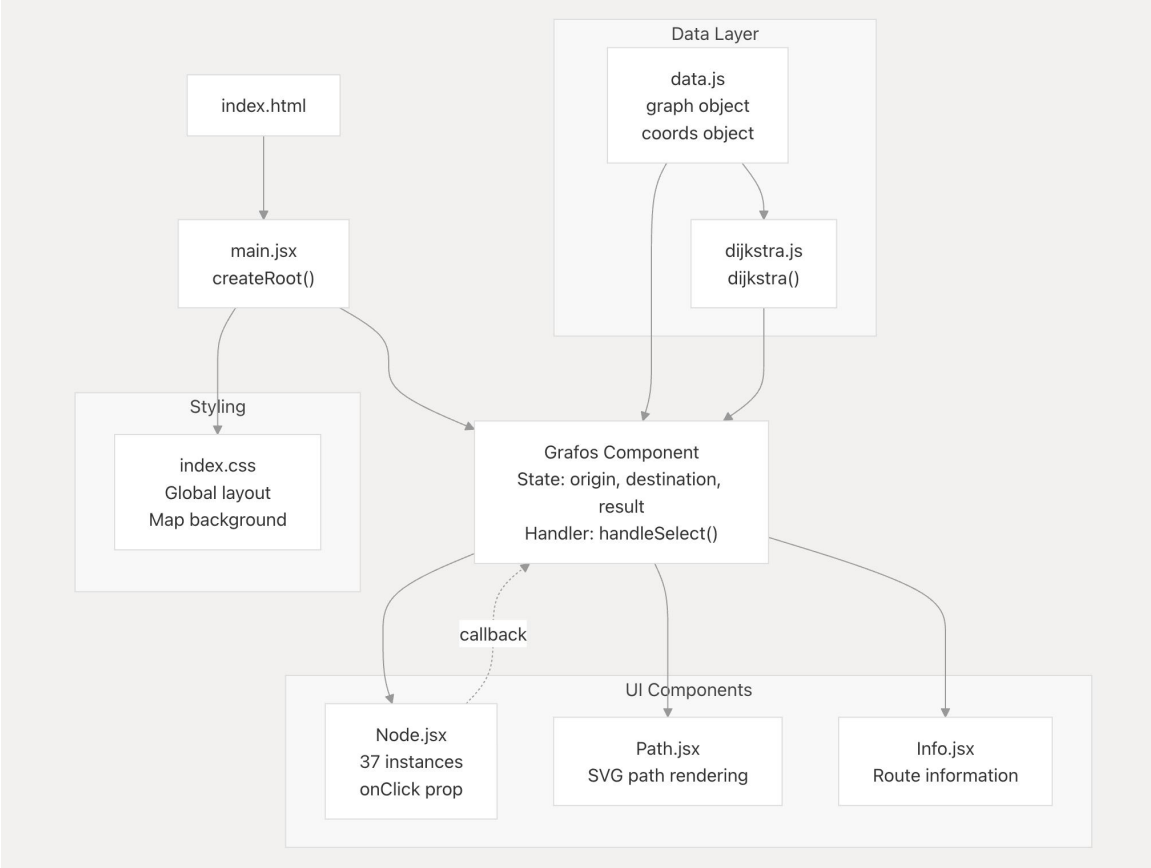
3. Pila Tecnológica

La aplicación fue desarrollada utilizando las siguientes tecnologías:

- React 18: interfaz basada en componentes
- Vite: servidor de desarrollo y empaquetado
- SWC: transpilador de JavaScript y JSX
- ESLint: validación de estándares de código
- CSS3: estilos y diseño visual

4. Arquitectura General

La aplicación sigue un modelo de orquestación centralizada, donde el componente principal Grafos controla el estado global, la lógica de negocio y la comunicación entre componentes.



5. Punto de Entrada

El archivo main.jsx actúa como punto de entrada de la aplicación. En él se importa el componente principal, se cargan los estilos globales y se renderiza la aplicación en el DOM utilizando createRoot, además de envolverla en StrictMode.

6. Orquestador Central

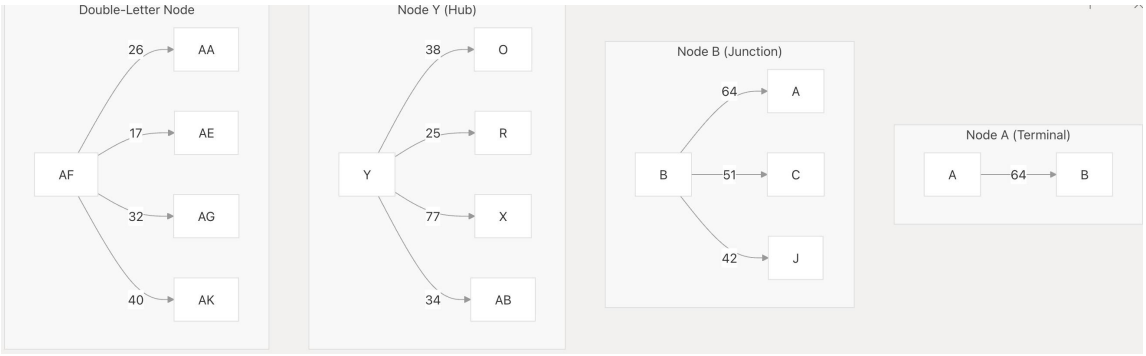
El componente Grafos.jsx administra:

- El nodo de origen
- El nodo destino
- El resultado del algoritmo

También implementa la lógica de selección basada en tres clics: origen, destino y reinicio.

Ejemplos de definiciones de nodos

El siguiente diagrama ilustra la estructura de varios nodos representativos:



7. Estructuras de Datos

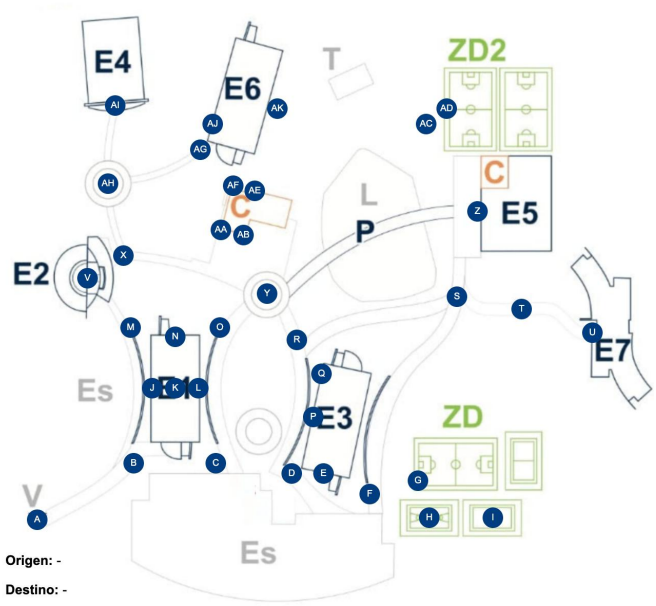
El sistema utiliza dos estructuras principales:

graph: lista de adyacencia ponderada que representa el grafo

coords: coordenadas cartesianas para la representación visual

Ambas estructuras comparten los mismos identificadores de nodos.

```
// ==================== COORDENADAS ====================
export  const coords = {
    A: { x: 21, y: 515 },
    B: { x: 115, y: 460 },
    C: { x: 195, y: 460 },
    D: { x: 269, y: 470 },
    E: { x: 300, y: 470 },
    F: { x: 345, y: 490 },
    G: { x: 392, y: 477 },
    H: { x: 403, y: 513 },
    I: { x: 465, y: 513 },
    J: { x: 133, y: 387 },
    K: { x: 155.5, y: 387 },
    L: { x: 178, y: 387 },
    M: { x: 112, y: 328 },
    N: { x: 155.5, y: 337 },
    O: { x: 199, y: 328 },
    P: { x: 290, y: 415 },
    Q: { x: 298, y: 373 },
    R: { x: 274, y: 340 },
    S: { x: 430, y: 298 },
    T: { x: 493, y: 310 },
    U: { x: 562, y: 333 },
    V: { x: 70, y: 280 },
    X: { x: 105, y: 258 },
    Y: { x: 245, y: 295 },
    Z: { x: 450, y: 215 },
    AA: { x: 200, y: 233 },
    AB: { x: 222, y: 238 },
    AC: { x: 400, y: 130 },
    AD: { x: 420, y: 115 },
    AE: { x: 233, y: 195 },
    AF: { x: 212, y: 190 },
    AG: { x: 180, y: 155 },
    AH: { x: 90, y: 188 },
    AI: { x: 97, y: 112 },
    AJ: { x: 192, y: 130 },
    AK: { x: 255, y: 115 },
};
```

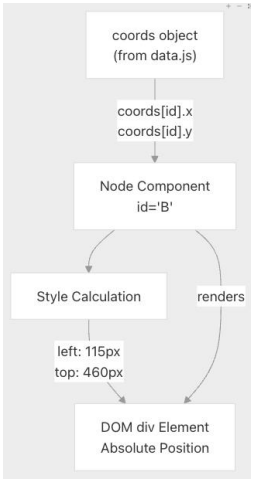



```
// Representación del campus como grafo ponderado

export const graph = {
    A: { B: 64 },
    B: { A: 64, C: 51, J: 42 },
    C: { B: 51, D: 36, L: 42 },
    D: { C: 36, E: 20, P: 43 },
    E: { D: 20, F: 14 },
    F: { E: 14, G: 45, H: 37, S: 96 },
    G: { F: 45 },
    H: { F: 37, I: 40 },
    I: { H: 40 },
    J: { B: 42, K: 10, M: 35 },
    K: { J: 10, L: 10, N: 36 },
    L: { C: 42, K: 10, O: 35, P: 48 },
    M: { J: 35, N: 14, V: 34 },
    N: { K: 36, M: 14, O: 14 },
    O: { L: 35, N: 14, R: 36, Y: 38 },
    P: { D: 43, L: 48, Q: 28 },
    Q: { P: 28, R: 15 },
    R: { O: 36, Q: 15, S: 53, Y: 25 },
    S: { F: 96, R: 53, T: 41, Z: 22 },
    T: { S: 41, U: 50 },
    U: { T: 50 },
    V: { M: 34, X: 11 },
    X: { V: 11, Y: 77, AH: 65 },
    Y: { O: 38, R: 25, X: 77, AB: 34 },
    Z: { S: 22, AC: 74 },
    AA: { AB: 15, AF: 26 },
    AB: { Y: 34, AA: 15, AE: 26 },
    AC: { Z: 74, AD: 15, AE: 70 },
    AD: { AC: 15 },
    AE: { AB: 26, AC: 70, AF: 17 },
    AF: { AA: 26, AE: 17, AG: 32, AK: 40 },
    AG: { AF: 32, AH: 35, AJ: 22 },
    AH: { X: 65, AG: 35, AI: 45 },
    AI: { AH: 45, AJ: 59 },
    AJ: { AG: 22, AI: 59 },
    AK: { AF: 40 },
};
```

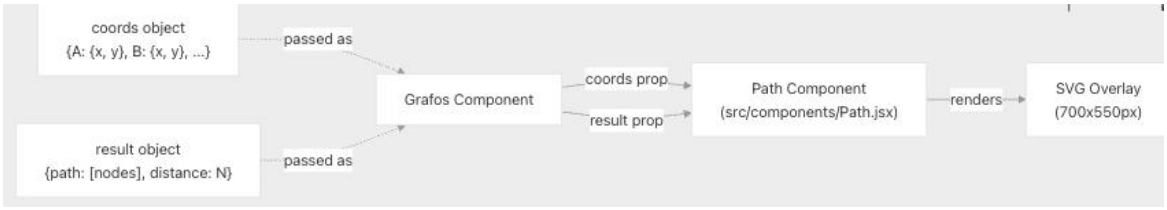
8. Componentes UI

La interfaz está compuesta por tres componentes principales:

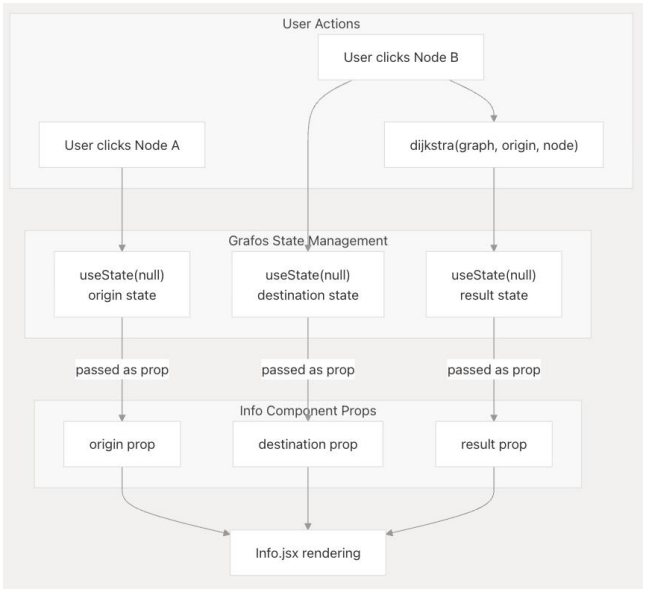
Node: representa nodos interactivos, no tiene estado interno solo es puramente presencial.



Path: dibuja la ruta calculada, no contiene un estado interno solo es presencial.



Info: muestra información de la ruta y si existe un resultado te mostrara la ruta.



Cada componente cumple una única responsabilidad.

9. Algoritmo

La aplicación utiliza el algoritmo de Dijkstra para calcular la ruta más corta entre dos nodos del campus, considerando únicamente pesos positivos. Este algoritmo permite obtener tanto la secuencia óptima de nodos como la distancia total recorrida entre un punto de origen y un punto de destino.

El algoritmo inicializa la distancia de todos los nodos como infinita y establece sus predecesores como nulos. Posteriormente, asigna una distancia inicial de cero al nodo de inicio. En cada iteración se selecciona el nodo no visitado con la menor distancia acumulada y se evalúan sus nodos vecinos, actualizando las distancias y predecesores cuando se encuentra un camino más corto.

El proceso continúa hasta que se alcanza el nodo destino o no existen más nodos disponibles para evaluar. Una vez finalizado, la ruta óptima se reconstruye recorriendo los nodos desde el destino hacia el origen mediante los predecesores calculados. Como resultado, el algoritmo devuelve la ruta seguida y la distancia total del recorrido.

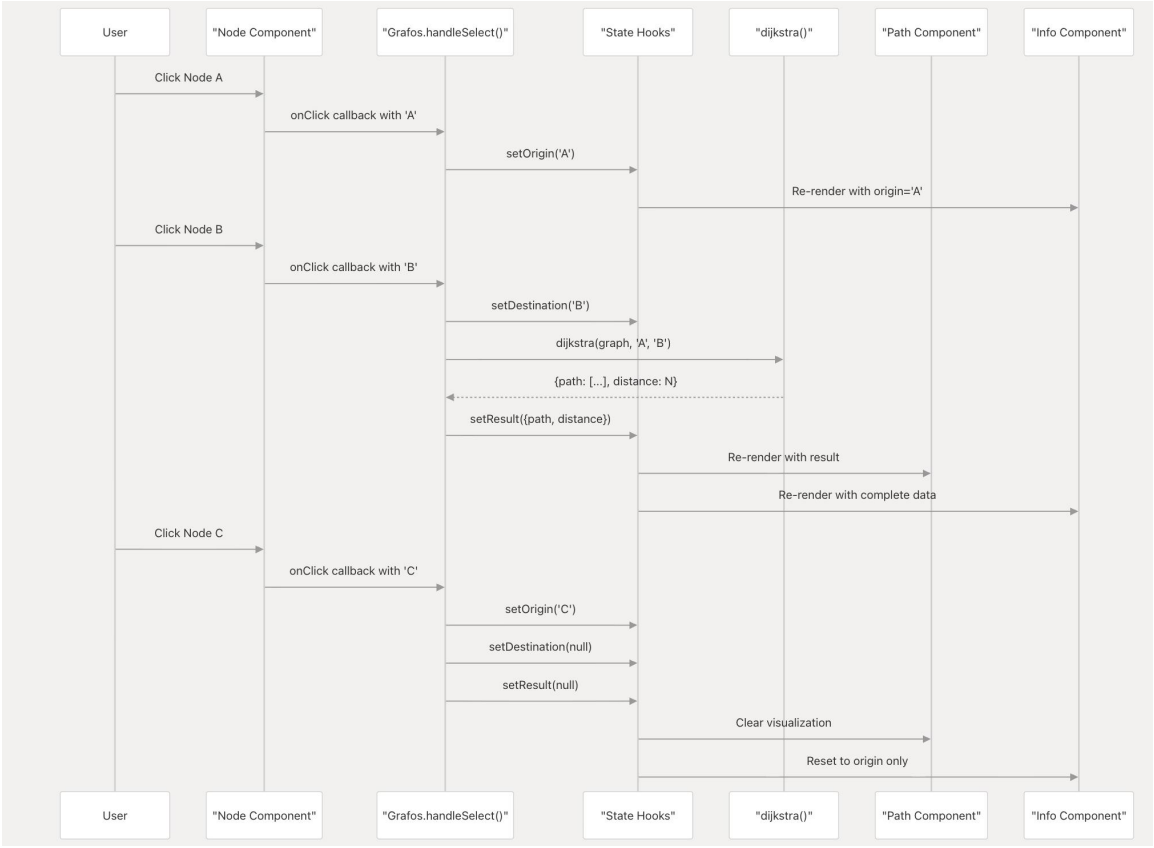
En la aplicación, el algoritmo recibe como entradas el grafo ponderado del campus y los identificadores de los nodos de origen y destino. Su salida es un objeto que contiene la ruta óptima como una lista de nodos y la distancia total calculada. Este resultado es utilizado por los componentes de la interfaz para representar visualmente el camino y mostrar la información correspondiente al usuario.

El algoritmo asume que el grafo es conexo y que todos los pesos son no negativos. En caso de que no exista un camino válido entre los nodos seleccionados, la distancia permanecerá como infinita.

```
// ================== DIJKSTRA ==================

export function dijkstra(graph, start, end) {
    const distances = {};
    const previous = {};
    const visited = {};

    for (let node in graph) {
        distances[node] = Infinity;
        previous[node] = null;
    }
    distances[start] = 0;

    while (true) {
        let closestNode = null;
        let shortestDistance = Infinity;

        for (let node in distances) {
        if (!visited[node] && distances[node] < shortestDistance) {
            shortestDistance = distances[node];
            closestNode = node;
        }
        }

        if (closestNode === null) break;
        if (closestNode === end) break;

        visited[closestNode] = true;

        for (let neighbor in graph[closestNode]) {
        const newDistance = distances[closestNode] + graph[closestNode][neighbor];
        if (newDistance < distances[neighbor]) {
            distances[neighbor] = newDistance;
            previous[neighbor] = closestNode;
        }
        }
    }

    const path = [];
    let current = end;
    while (current) {
        path.unshift(current);
        current = previous[current];
    }

    return { path, distance: distances[end] };
}
```

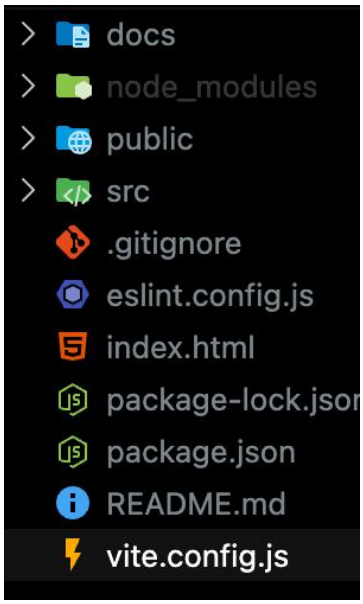
10. Flujo Usuario

1. El usuario selecciona un nodo de origen
2. Selecciona un nodo destino
3. Se ejecuta el algoritmo
4. Se dibuja la ruta
5. Un tercer clic reinicia el proceso



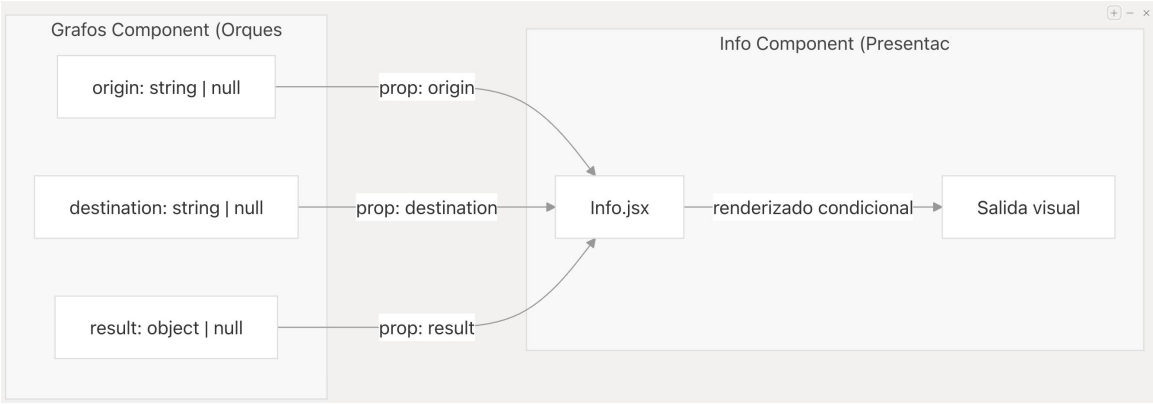
11. Estructura Proyecto

El proyecto sigue una estructura modular organizada por responsabilidades, separando componentes, algoritmos, estilos y recursos.



12. Flujo de Datos

La aplicación implementa un flujo de datos unidireccional, donde el estado reside únicamente en el componente Grafos y se transmite a los componentes hijos mediante props.



13. Características

- Gestión centralizada del estado
- Separación entre lógica y visualización
- Componentes especializados
- Ejecución del algoritmo bajo demanda

13.1. Unidireccionalidad del flujo de datos

El flujo de datos en la aplicación es unidireccional y va únicamente desde el componente Grafos hacia el componente Info. Grafos concentra el estado de la aplicación y envía la información

necesaria mediante props, mientras que Info se limita a mostrar esos datos sin manejar estado propio ni devolver eventos.

Los datos transmitidos incluyen el nodo de origen, el nodo destino y el resultado del algoritmo, que contiene la ruta y la distancia. El componente Info realiza un renderizado condicional, mostrando valores por defecto cuando no hay información disponible y presentando la ruta solo cuando existe un resultado válido.

Este enfoque corresponde a un componente presentacional, lo que mantiene una separación clara entre la lógica y la presentación.

13.2. Patrón de diseño aplicado

El componente Info sigue el patrón de componente presentacional puro (presentational component), ya que depende completamente de propiedades externas para su funcionamiento, carece de lógica interna compleja y no gestiona estado propio. Este enfoque favorece la claridad, reutilización y mantenimiento del código, reforzando la separación de responsabilidades dentro de la arquitectura del sistema.

14. Conclusión

Nodos-UPA es una aplicación sólida que integra teoría de grafos y desarrollo web moderno para resolver problemas reales de navegación dentro del campus universitario de forma visual e intuitiva.