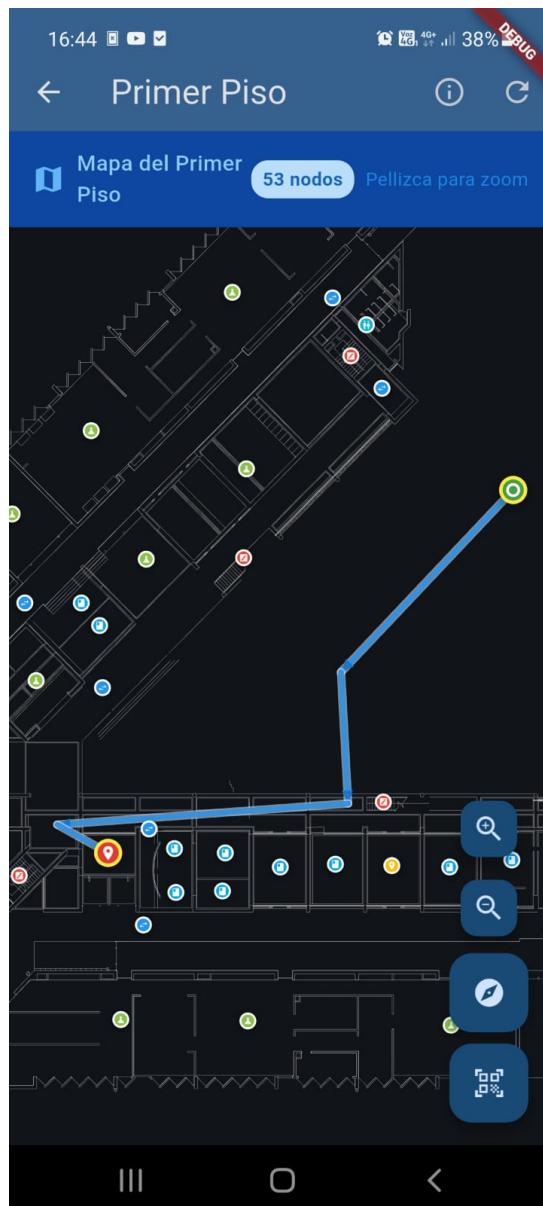




Taller de Integración
Universidad de Magallanes
*Aplicación de navegación
interna para la Facultad de
Ingeniería*



Estudiante: Diego Vidal
Programa: Ingeniería Civil en Computación e Informática
Departamento: Departamento de Ingeniería en Computación
Profesor guía: Dra. Patricia Maldonado
Fecha: 08/01/2026

Índice

1. Introducción	3
1.1. Antecedentes	3
1.2. Período en que se realizó	3
1.3. Objetivo General	3
1.4. Objetivos Específicos	3
2. Desarrollo, hallazgos y resultados	4
2.1. Metodología utilizada	4
2.2. Presentación de los Hallazgos y Resultados	4
2.2.1. Digitalización de planos arquitectónicos de la facultad	4
2.2.2. Investigación sobre la creación de aplicaciones móviles	6
2.2.3. Preparación del entorno de desarrollo y repositorio de Github	6
2.2.4. Creación de la interfaz gráfica	6
2.2.5. Integración de los mapas a la aplicación	12
2.2.6. Modelado del grafo de salas y conexiones	12
2.2.7. Implementación del algoritmo A* para cálculo de rutas	19
2.2.8. Creación y generación automatizada de códigos QR	23
2.2.9. Sistema de lectura e interpretación de códigos QR	25
2.2.10. Solución al problema de compatibilidad de formatos QR	28
2.2.11. Sistema de navegación y cálculo de rutas	30
2.2.12. Pantalla de selección de destino	33
2.2.13. Integración del escáner QR con la navegación	34
2.2.14. Pruebas de la aplicación	34
3. Conclusiones y Recomendaciones	35
3.1. Conclusiones	35
3.2. Recomendaciones	35
4. Referencias	36

Resumen

El presente proyecto desarrolla una aplicación de navegación interna cuyo objetivo es optimizar la orientación y el desplazamiento dentro de la Facultad de Ingeniería de la Universidad de Magallanes. Esta solución surge de la necesidad de mejorar la accesibilidad y la eficiencia al momento de encontrar salas, oficinas y laboratorios, especialmente para estudiantes nuevos y personas visitantes. Para su desarrollo, se implementó un sistema de mapeo digital basado en planos arquitectónicos del edificio, integrando tecnologías de posicionamiento soportadas por códigos QR y un algoritmo de búsqueda A para el cálculo de rutas óptimas dentro de cada piso. El sistema permite a los usuarios navegar de forma eficiente dentro de un piso seleccionado, visualizando rutas claras sobre mapas interactivos.*

1. Introducción

1.1. Antecedentes

Los alumnos al ingresar por primera vez a la Universidad pueden experimentar desorientación a la hora de ir a alguna sala o laboratorio, ya que no saben cómo llegar a esos lugares en específico. Para resolver esta problemática, se busca crear una aplicación móvil para poder navegar dentro de la facultad.

1.2. Período en que se realizó

El proyecto fue desarrollado en el segundo semestre del año 2025 y parte del mes de enero de 2026 en las dependencias de la Facultad de Ingeniería de la Universidad de Magallanes. Las pruebas y evaluaciones se realizaron usando planos digitales y simulaciones de desplazamiento en sus principales pasillos y áreas de uso común.

1.3. Objetivo General

Desarrollar una aplicación móvil de navegación interna que permita a los usuarios ubicarse y desplazarse eficientemente dentro de cada piso de la Facultad de Ingeniería, utilizando algoritmos de búsqueda de rutas óptimas y mapas digitales interactivos.

1.4. Objetivos Específicos

- Analizar la estructura espacial y las necesidades de orientación dentro de la Facultad de Ingeniería.
- Diseñar y modelar grafos de navegación para cada piso del edificio, incluyendo salas, pasillos, laboratorios y oficinas.
- Implementar el algoritmo A* para calcular rutas óptimas entre puntos de interés dentro del mismo piso.
- Integrar tecnologías de posicionamiento mediante códigos QR que permitan identificar la ubicación del usuario.
- Desarrollar una interfaz gráfica intuitiva con visualización de mapas SVG interactivos y control de zoom.

2. Desarrollo, hallazgos y resultados

2.1. Metodología utilizada

Para el desarrollo del proyecto se optó por una metodología incremental, dividiendo el trabajo en etapas que permitieron validar cada componente (mapas, algoritmo, interfaz) de forma independiente. Las etapas fueron las siguientes:

1. Digitalización de planos arquitectónicos de la facultad.
2. Investigación sobre la creación de aplicaciones móviles.
3. Preparación del entorno de desarrollo y repositorio de Github.
4. Creación de la interfaz gráfica de la aplicación.
5. Integración de los mapas a la aplicación.
6. Modelado del grafo de salas y conexiones.
7. Implementación del algoritmo A* para cálculo de rutas.
8. Creación y generación automatizada de códigos QR.
9. Sistema de lectura e interpretación de códigos QR.
10. Solución al problema de compatibilidad de formatos QR.
11. Sistema de navegación y cálculo de rutas.
12. Pantalla de selección de destino.
13. Integración del escáner QR con la navegación.

2.2. Presentación de los Hallazgos y Resultados

2.2.1. Digitalización de planos arquitectónicos de la facultad

Los mapas arquitectónicos de la facultad ya estaban digitalizados en formato DWG (formato utilizado por AutoCAD). A partir de estos archivos, se realizó una conversión a DXF (Drawing Exchange Format) para permitir su edición en software libre.

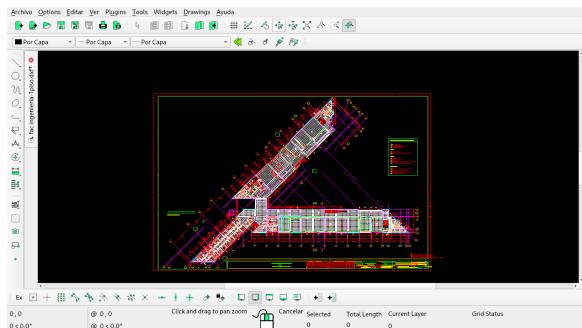
En LibreCAD se eliminaron elementos no relevantes para un mapa de navegación (mobiliario y detalles decorativos), conservando paredes y pasillos principales; adicionalmente, se corrigieron errores presentes en los planos originales, ya que faltaban varias salas, laboratorios y oficinas.

Una vez corregidos y simplificados, los planos se exportaron a formato SVG (Scalable Vector Graphics) para su integración en la aplicación móvil. No obstante, la exportación directa a SVG desde LibreCAD no mantenía las proporciones adecuadas, lo que hacía que, al visualizar el mapa en un visor de imágenes o navegador web, se presentara demasiado pequeño o ni siquiera pudiera visualizarse.

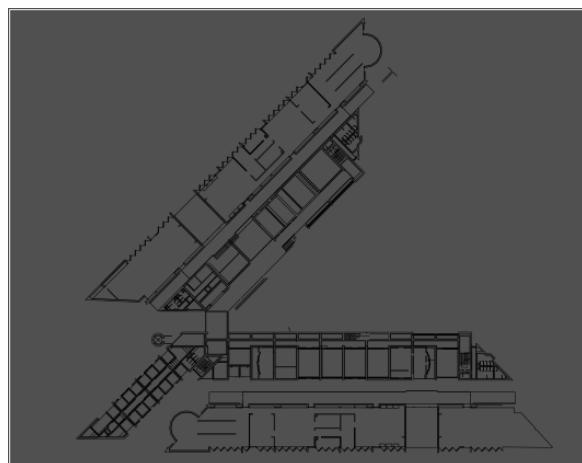
Para corregirlo, se empleó Inkscape (editor de gráficos vectoriales), ajustando las dimensiones del mapa para garantizar una visualización correcta en la aplicación móvil.



Figura 1: Logos de LibreCAD (arriba) e Inkscape (abajo)



Antes: Plano original



Después: Plano simplificado

Figura 2: Comparación del proceso de digitalización: plano original con elementos decorativos (izq.) vs. plano simplificado para navegación (der.)

2.2.2. Investigación sobre la creación de aplicaciones móviles

Con el fin de elegir la mejor forma de desarrollar la aplicación móvil, se consideraron las plataformas móviles más utilizadas por el estudiantado (Android e iOS). En consecuencia, se investigaron las diferentes tecnologías y frameworks existentes para desarrollo móvil. Las opciones principales fueron las siguientes:

- Desarrollo nativo: Crear aplicaciones independientes para cada plataforma (Java/Kotlin para Android y Swift/Objective-C para iOS).
- Desarrollo multiplataforma: Utilizar frameworks como React Native, Flutter o Xamarin para mantener una sola base de código que funcione en ambas plataformas.

Después de analizar las opciones, se optó por utilizar Flutter, dado que permite crear aplicaciones nativas de alto rendimiento para ambas plataformas desde una única base de código. Además, cuenta con una comunidad amplia y activa, junto con una extensa oferta de paquetes y complementos.

2.2.3. Preparación del entorno de desarrollo y repositorio de Github

Se prepararon los equipos para desarrollar la aplicación móvil (con sistemas operativos Windows 10 y Debian 13), instalando Flutter SDK, un editor de código (Visual Studio Code) y los emuladores de Android e iOS para realizar pruebas durante el desarrollo. Una vez listo el entorno, se creó un repositorio en GitHub para alojar el código fuente del proyecto y permitir la colaboración entre los integrantes del equipo.

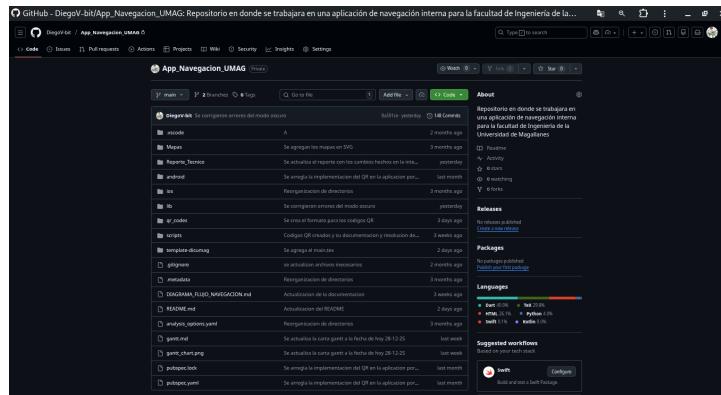


Figura 4: Repositorio de GitHub del proyecto con la estructura de archivos y commits

2.2.4. Creación de la interfaz gráfica

La interfaz gráfica de la aplicación se desarrolló usando Flutter como framework principal, aprovechando su capacidad de crear interfaces nativas y fluidas con un único código base. El diseño de la app se enfocó en la simplicidad y usabilidad, considerando que los usuarios principales serían estudiantes nuevos y visitantes que necesitan orientarse rápidamente dentro de la facultad.



Figura 3: Logos de Flutter (arriba) y Dart (abajo)

Estructura de la aplicación La aplicación se estructuró en dos pantallas principales que permiten una navegación intuitiva:

Pantalla de inicio (PantallaInicio). Esta pantalla funciona como punto de entrada de la aplicación y presenta las siguientes características:



Figura 5: Pantalla de inicio de la aplicación mostrando las cuatro opciones de pisos con iconos distintivos

- **Diseño visual atractivo:** Se implementó un gradiente de color que va desde un azul claro en la parte superior hasta blanco en la parte inferior, creando una interfaz agradable visualmente y manteniendo la identidad institucional mediante el uso del color azul.
- **Encabezado informativo:** En la parte superior se muestra un ícono de ubicación de gran tamaño junto con el título “Navegación Interna” y un subtítulo descriptivo, lo que permite al usuario identificar inmediatamente el propósito de la aplicación.
- **Sistema de tarjetas por piso:** La funcionalidad principal se presenta mediante cuatro tarjetas interactivas, una por cada piso de la facultad. Cada tarjeta incluye:
 - Un ícono distintivo que representa el tipo de espacios del piso (ciencia para laboratorios, escuela para aulas, libro para salas de estudio, y edificio para administración)
 - Un título claro con el nombre del piso
 - Una descripción breve de los espacios que contiene
 - Un indicador visual de navegación (flecha)
 - Códigos de color diferenciados (verde, naranja, morado y rojo) para facilitar la identificación rápida

Esta organización permite que el usuario seleccione rápidamente el piso que desea explorar con solo tocar la tarjeta correspondiente.

Pantalla de mapa (PantallaMapa). Una vez seleccionado un piso, el usuario es dirigido a la pantalla de mapa, que constituye el componente central de la aplicación. Esta pantalla implementa las siguientes funcionalidades:

- **Visualización de mapas SVG:** Se integró el paquete `flutter_svg` para cargar y mostrar los mapas arquitectónicos en formato SVG. La aplicación determina dinámicamente qué archivo SVG cargar según el piso seleccionado, utilizando las rutas:

- Primer piso: `Mapas/Primer_piso_fac_ing_simple.svg`
- Segundo piso: `Mapas/Segundo_piso_fac_ing_simple.svg`
- Tercer piso: `Mapas/Tercer_piso_fac_ing_simple.svg`
- Cuarto piso: `Mapas/Cuarto_piso_fac_ing_simple.svg`

- **Navegación interactiva:** Se implementó el widget `InteractiveViewer` de Flutter, que proporciona:

- Desplazamiento táctil para explorar diferentes áreas del mapa
- Zoom mediante gestos de pellizco en pantallas táctiles
- Límites de escala configurables (mínimo 1.0x, máximo 4.0x) para mantener la visualización en rangos útiles

- **Controles de zoom:** Se añadieron tres botones flotantes en la parte inferior derecha que permiten:

- Acercar el mapa (zoom in) aumentando la escala en un 20 %
- Alejar el mapa (zoom out) reduciendo la escala en un 20 %
- Centrar el mapa en el patio de ingeniería (nodo P1_Patio_de_ingenieria con coordenadas 567, 478), representado con un ícono de brújula (`Icons.explore`) similar a Google Maps

El botón de centrado fue diseñado para proporcionar un punto de referencia consistente a los usuarios, permitiendo reorientarse rápidamente al centro de la facultad independientemente del nivel de zoom o desplazamiento actual.

- **Gestión de estados de carga:** La aplicación implementa un sistema robusto para manejar diferentes estados:

- Estado de carga: Muestra un indicador circular de progreso con el texto “Cargando mapa...”
- Estado de error: En caso de que el archivo SVG no se encuentre o no pueda cargarse, se muestra un mensaje de error detallado con el ícono de advertencia, el nombre del archivo que se intentó cargar y la descripción del error específico

- **Barra informativa:** Se incluyó una barra azul clara debajo del encabezado que muestra el nombre del piso actual y proporciona una indicación visual de que se puede hacer zoom con gestos de pellizco.

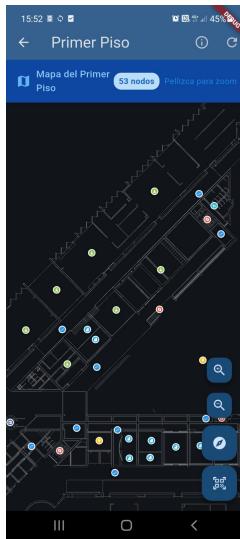


Figura 6: Vista del mapa SVG del primer piso con controles de zoom y centrado en la esquina inferior derecha

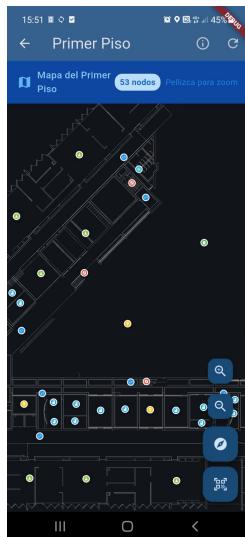


Figura 7: *
Sin zoom

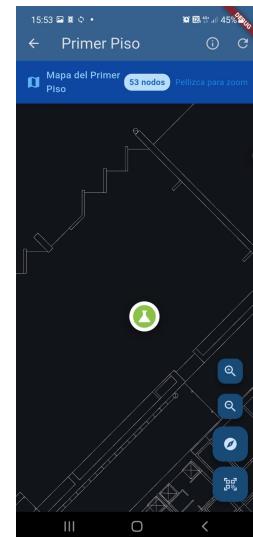


Figura 8: *
Zoom al máximo

Figura 9: Comparación de niveles de zoom: vista general del piso completo (izq.) vs. vista detallada con zoom máximo aplicado (der.)

Implementación técnica Widget principal y navegación. La aplicación utiliza `MaterialApp` como widget raíz, configurando el tema principal con Material Design 3 (`useMaterial3: true`) para aprovechar los componentes de diseño más modernos de Flutter. El sistema de navegación se basa en `Navigator.push()`, que permite transiciones fluidas entre la pantalla de inicio y las pantallas de mapas.

Gestión de transformaciones. Para controlar el zoom y desplazamiento del mapa, se implementó un `TransformationController` que mantiene una matriz de transformación 4x4 (`Matrix4`). Esta matriz permite aplicar operaciones de escala y traslación al mapa de forma eficiente, proporcionando una experiencia de usuario fluida incluso con mapas de gran tamaño.

Las operaciones de zoom se realizan clonando la matriz actual, aplicando la escala deseada y actualizando el controlador, lo que garantiza que las transformaciones se realicen de forma acumulativa y coherente.

Carga asíncrona de recursos. La aplicación implementa un patrón de carga asíncrona mediante `FutureBuilder`, que permite mostrar indicadores de progreso mientras se cargan los archivos SVG. Este enfoque mejora significativamente la experiencia del usuario, especialmente en dispositivos con menor rendimiento o cuando los archivos de mapa son de gran tamaño.

Consideraciones de diseño Durante el desarrollo de la interfaz se tomaron varias decisiones importantes:

1. **Material Design:** Se adoptó el sistema de diseño Material de Google para garantizar consistencia visual y aprovechar componentes probados en cuanto a usabilidad.
2. **Retroalimentación visual:** Todos los elementos interactivos proporcionan retroalimentación visual mediante efectos de elevación (`elevation`) en las tarjetas y efectos de onda (`InkWell`) al tocar.
3. **Accesibilidad:** Se implementaron tooltips en todos los botones de acción para ayudar a los usuarios a comprender la funcionalidad de cada control.
4. **Manejo de errores:** Se diseñó una pantalla de error informativa que no solo indica que algo salió mal, sino que proporciona detalles técnicos útiles para diagnosticar el problema.
5. **Soporte para temas claro y oscuro:** La aplicación incluye un sistema completo de temas que permite al usuario elegir entre modo claro, modo oscuro o seguir la configuración del sistema. Esta funcionalidad mejora la experiencia de usuario al permitir personalización según preferencias individuales y condiciones de iluminación.

Sistema de temas y personalización La aplicación implementa un sistema robusto de gestión de temas utilizando `ThemeMode` de Flutter, que permite tres modos de operación:

- **Modo Claro:** Interfaz con colores claros optimizada para ambientes bien iluminados
- **Modo Oscuro:** Interfaz con colores oscuros que reduce la fatiga visual en ambientes con poca luz y ahorra batería en pantallas OLED
- **Modo Automático:** Sigue la configuración del sistema operativo del dispositivo

El tema se gestiona a nivel de aplicación mediante un estado `ThemeMode` que se propaga a través de la jerarquía de widgets. La aplicación define dos temas completos basados en Material Design 3:

```

1  MaterialApp(
2   themeMode: _themeMode,
3
4   // Tema claro
5   theme: ThemeData(
6     colorScheme: ColorScheme.fromSeed(
7       seedColor: Colors.blue,
8       brightness: Brightness.light,
9     ),
10    useMaterial3: true,
11  ),
12
13  // Tema oscuro
14  darkTheme: ThemeData(
15    colorScheme: ColorScheme.fromSeed(
16      seedColor: Colors.blue,
17      brightness: Brightness.dark,
18    ),
19    useMaterial3: true,
20  ),
21)

```

Listing 1: Configuración de temas

Pantalla de Configuración. Para facilitar el cambio de tema, se implementó una pantalla de configuración (**PantallaAjustes**) accesible desde el botón de configuración en la barra de aplicación de la pantalla de inicio. Esta pantalla incluye:

- Selector de modo de tema con tres opciones mediante controles de radio
- Iconos representativos para cada modo (automático, claro, oscuro)
- Descripción explicativa de cada opción
- Aplicación inmediata del cambio de tema sin necesidad de reiniciar la aplicación
- Sección informativa mostrando la versión de la aplicación y datos institucionales

El diseño adaptable de la interfaz garantiza que todos los colores de fondo, texto, tarjetas y controles se ajusten automáticamente al tema seleccionado, manteniendo contraste adecuado y legibilidad en ambos modos.

Esta implementación inicial de la interfaz gráfica estableció las bases para el desarrollo posterior de funcionalidades más avanzadas, como el sistema de navegación basado en grafos y el cálculo de rutas óptimas entre diferentes puntos de la facultad.

2.2.5. Integración de los mapas a la aplicación

Con la interfaz gráfica ya creada, se integraron los mapas al proyecto para permitir su visualización dentro de la aplicación. Una vez incorporados, los mapas se visualizan correctamente y presentan una interacción adecuada con el usuario.

Cabe destacar que el mapa del primer piso y el de los laboratorios, al ser recursos independientes, podrían dificultar el mantenimiento si se trabajaran por separado. Por ello, se consideró conveniente fusionar ambos mapas para reducir complejidad y evitar futuras complicaciones en la aplicación.

2.2.6. Modelado del grafo de salas y conexiones

La aplicación hace uso de un modelo de grafo para representar la topología de cada piso del edificio (dicho de otra forma, se representa mediante el grafo las salas de clase, oficinas y laboratorios de toda la facultad de ingeniería). Este modelo permite calcular las mejores rutas entre distintos puntos de interés mediante algoritmos de búsqueda en grafos.

Las distancias que están asociadas a las conexiones del grafo representan costos relativos de desplazamiento, los cuales están asociados a la geometría del plano del edificio. Estas distancias no corresponden necesariamente a metros reales, sino que permiten modelar el esfuerzo de desplazamiento entre nodos dentro del entorno. Esta aproximación es suficiente para la navegación interna, ya que el objetivo es determinar rutas óptimas dentro del edificio y no realizar mediciones físicas exactas.

Estructura del modelo de datos **Clase Nodo.** Representa un punto de interés en el mapa (Sala, laboratorio, oficina, etc.).

```

1 class Nodo {
2     final String id; // Ejemplo: "P1_A101"
3     final double x; // Coordenada X en el mapa SVG
4     final double y; // Coordenada Y en el mapa SVG
5
6     Nodo({required this.id, required this.x, required this.y});
7
8     factory Nodo.fromJson(Map<String, dynamic> json) {
9         return Nodo(id: json['id'], x: json['x'], y: json['y']);
10    }
11
12    Map<String, dynamic> toJson() => {'id': id, 'x': x, 'y': y};
13 }
```

Listing 2: Clase Nodo

Clase Conexion. Representa una arista entre dos nodos, con la información de distancia.

```

1 class Conexion {
2     final String origen;      // ID del nodo origen
3     final String destino;    // ID del nodo destino
4     final double distancia;   // Distancia euclíadiana
5
6     const Conexion({required this.origen,
7                     required this.destino,
8                     required this.distancia});
9
10    factory Conexion.fromJson(Map<String, dynamic> json) {
11        return Conexion(
12            origen: json['origen'] as String,
13            destino: json['destino'] as String,
14            distancia: (json['distancia'] as num).toDouble(),
15        );
16    }
17 }
```

Listing 3: Clase Conexion

Clase Grafo. Contenedor principal que agrupa los nodos y conexiones de un piso.

```

1 class Grafo {
2     final List<Nodo> nodos;
3     final List<Conexion> conexiones;
4
5     const Grafo({required this.nodos, required this.conexiones});
6
7     factory Grafo.fromJson(Map<String, dynamic> json) {
8         return Grafo(
9             nodos: (json['nodos'] as List<dynamic>)
10                .map((e) => Nodo.fromJson(e as Map<String, dynamic>))
11                .toList(),
12             conexiones: (json['conexiones'] as List<dynamic>)
13                .map((e) => Conexion.fromJson(e as Map<String, dynamic>))
14                .toList(),
15         );
16     }
17 }

```

Listing 4: Clase Grafo

Sistema de tipificación de nodos La aplicación implementa un sistema de clasificación de nodos mediante un `enum` que permite categorizar los diferentes tipos de ubicaciones en el mapa. Se definió el `enum` `TipoNodo` con once valores diferentes:

- **entrada:** Puntos de acceso al edificio
- **pasillo:** Zonas de tránsito general
- **interseccion:** Puntos donde se cruzan múltiples pasillos
- **esquina:** Puntos de cambio de dirección
- **puerta:** Accesos a salas o espacios cerrados
- **escalera:** Conexiones verticales entre pisos
- **ascensor:** Elevadores para accesibilidad
- **bano:** Servicios higiénicos
- **laboratorio:** Espacios de prácticas y experimentación
- **salaClases:** Aulas y salas de docencia
- **puntoInteres:** Ubicaciones destacadas o de referencia

Cada tipo de nodo tiene asociado propiedades visuales específicas mediante una extensión del `enum` que proporciona:

```

1 extension TipoNodoExtension on TipoNodo {
2     String get nombre {
3         switch (this) {
4             case TipoNodo.entrada: return 'Entrada';
5             case TipoNodo.pasillo: return 'Pasillo';
6             case TipoNodo.interseccion: return 'Interseccion';
7             case TipoNodo.escalera: return 'Escalera';
8             case TipoNodo.ascensor: return 'Ascensor';
9             case TipoNodo.bano: return 'Bano';
10            case TipoNodo.laboratorio: return 'Laboratorio';
11            case TipoNodo.salaClases: return 'Sala de Clases';
12            // ... otros casos
13        }
14    }
15 }

```

```

16 IconData get icono {
17     switch (this) {
18         case TipoNodo.entrada: return Icons.door_front;
19         case TipoNodo.escalera: return Icons.stairs;
20         case TipoNodo.ascensor: return Icons.elevator;
21         case TipoNodo.bano: return Icons.wc;
22         case TipoNodo.laboratorio: return Icons.science;
23         case TipoNodo.salaClases: return Icons.school;
24         // ... otros casos
25     }
26 }
27
28 Color get color {
29     switch (this) {
30         case TipoNodo.entrada: return Colors.green;
31         case TipoNodo.escalera: return Colors.orange;
32         case TipoNodo.ascensor: return Colors.purple;
33         case TipoNodo.bano: return Colors.blue;
34         case TipoNodo.laboratorio: return Colors.teal;
35         case TipoNodo.salaClases: return Colors.indigo;
36         // ... otros casos
37     }
38 }
39

```

Listing 5: Extensión de TipoNodo para propiedades visuales

La aplicación puede inferir el tipo de nodo basándose en el identificador mediante análisis de subcadenas. Por ejemplo, si el ID contiene “entrada”, se clasifica como `TipoNodo.entrada`; si contiene “lab”, se clasifica como `TipoNodo.laboratorio`. Este sistema de inferencia automática facilita la asignación de tipos durante la creación de los datos del grafo y asegura consistencia en la visualización.

Sistema de coordenadas Normalización SVG. El sistema utilizado para las coordenadas fue normalizar las coordenadas del archivo SVG a 1200×800 píxeles, para todos los pisos, independientemente del tamaño real del archivo SVG. Esto permite:

- Consistencia entre los distintos pisos
- Simplificación del cálculo de las distancias
- Escalado dinámico a cualquier resolución de pantalla

Transformación de coordenadas. La aplicación implementa un sistema de transformación bidireccional. Para la transformación de SVG a pantalla, se calcula la escala manteniendo el aspect ratio (`BoxFit.contain`), se determinan las dimensiones escaladas y los offsets para centrado, y finalmente se aplica la transformación a las coordenadas. Para la transformación inversa de pantalla a SVG, se aplica la operación inversa restando los offsets y dividiendo por la escala.

Transformación de coordenadas: detalles de implementación La aplicación utiliza un sistema de coordenadas normalizado que permite independencia entre las dimensiones del SVG y la pantalla del dispositivo.

Coordenadas SVG → Pantalla. Dado que el contenedor del mapa tiene exactamente las dimensiones del SVG original (1200×800 píxeles), las coordenadas se mapean directamente sin necesidad de escala adicional:

```

1 Offset _calcularPosicionEscalada(double x, double y) {
2     // Mapeo 1:1 ya que SizedBox coincide con dimensiones SVG
3     return Offset(x, y);
4 }

```

Listing 6: Transformación SVG a pantalla

El `InteractiveViewer` se encarga de aplicar zoom y desplazamiento sobre este contenedor de tamaño fijo, manteniendo la proporción de los elementos.

Coordenadas Pantalla → SVG. Para capturar toques del usuario y convertirlos a coordenadas SVG:

```

1 Offset _calcularCoordenadasSVG(Offset screenPosition) {
2     // Como el contenedor es 1:1 con SVG,
3     // no hay conversion de escala necesaria
4     return screenPosition;
5 }
```

Listing 7: Transformación pantalla a SVG

Gestión del zoom. El sistema de zoom respeta límites definidos:

```

1 void zoom(double scaleFactor) {
2     final controller = _transformationController;
3     final currentScale = controller.value.getMaxScaleOnAxis();
4
5     // Evitar division por cero
6     final safeCurrent = currentScale <= 0 ? 1.0 : currentScale;
7
8     // Aplicar limites (1.0 a 4.0)
9     final targetScale = (safeCurrent * scaleFactor)
10    .clamp(_minScale, _maxScale);
11    final relativeFactor = targetScale / safeCurrent;
12
13    // Centrar zoom en el punto medio de la pantalla
14    final center = controller.toScene(
15        Offset(
16            MediaQuery.of(context).size.width / 2,
17            MediaQuery.of(context).size.height / 2,
18        ),
19    );
20
21    // Aplicar transformacion
22    final matrix = controller.value.clone();
23    matrix
24        ..translate(center.dx, center.dy)
25        ..scale(relativeFactor)
26        ..translate(-center.dx, -center.dy);
27
28    controller.value = matrix;
29 }
```

Listing 8: Control de zoom con límites

Este enfoque garantiza que el zoom siempre se aplique respecto al centro visible de la pantalla, proporcionando una experiencia de usuario más natural.

Centrado en punto de referencia. La aplicación implementa un sistema de centrado que enfoca el mapa en el patio de ingeniería al iniciar y al presionar el botón de brújula:

```

1 void _centrarEnPatio() {
2     // Coordenadas del patio de ingenieria en el piso 1
3     const double patioX = 567.0;
4     const double patioY = 478.0;
5
6     // Obtener el tamano de la pantalla
7     final screenWidth = MediaQuery.of(context).size.width;
8     final screenHeight = MediaQuery.of(context).size.height;
9
10    // Calcular el desplazamiento necesario para centrar el patio
11    // Centramos el punto del patio en el centro de la pantalla
12    final dx = (screenWidth / 2) - patioX;
13    final dy = (screenHeight / 2) - patioY;
```

```

14 // Crear matriz de transformacion con escala 1.0 y traduccion al patio
15 final matrix = Matrix4.identity()
16   ..translate(dx, dy);
17
18 _transformationController.value = matrix;
19 }
20
21 void _resetZoom() {
22   // Centra la vista en el patio de ingenieria (coordenadas: 567, 478)
23   _centrarEnPatio();
24 }
25

```

Listing 9: Centrado en el patio de ingeniería

Esta funcionalidad se ejecuta automáticamente al abrir el mapa mediante `WidgetsBinding.instance.addPostFrameCallback` en el método `initState()`, asegurando que el usuario siempre comience con el patio de ingeniería como punto de referencia central.

Estructura de archivos JSON Cada piso de la facultad tiene su archivo JSON correspondiente con una estructura que incluye un array de nodos (con propiedades id, x, y, tipo y nombre) y un array de conexiones (con origen, destino y distancia).

Los identificadores de nodos siguen el patrón `P{piso}-{tipo}{numero}`, por ejemplo:

- `P1_Entrada` para la entrada de la facultad
- `P2_Sala_23` para la sala 23 del segundo piso
- `P3_Departamento_matematica_fisica` para el departamento de matemática y física del tercer piso

Los archivos se ubican en la ruta `lib/data/` con los nombres `grafo_piso1.json`, `grafo_piso2.json`, `grafo_piso3.json` y `grafo_piso4.json`.

Carga dinámica del grafo La aplicación implementa una utilidad de carga (`grafo_loader.dart`) que permite cargar asíncronamente los archivos JSON desde los assets del proyecto. La función determina dinámicamente qué archivo cargar según el número de piso seleccionado, lee el contenido del archivo mediante `rootBundle.loadString()`, decodifica el JSON y crea las instancias de las clases del modelo de datos.

La inicialización del mapa se realiza de forma asíncrona, cargando primero los nodos y esperando al primer frame para tener el `RenderBox` disponible antes de marcar el componente como inicializado.

Visualización de nodos Los nodos se visualizan en el mapa como marcadores circulares sobre el mapa SVG. Cada marcador se posiciona utilizando las coordenadas transformadas del sistema SVG al sistema de la pantalla, con un tamaño de 12×12 píxeles.

Los marcadores incluyen:

- Color de fondo según el tipo de nodo (inferido o explícito)
- Forma circular con borde blanco de 1.5 píxeles
- Sombra para mejorar la visibilidad
- Icono representativo del tipo de nodo (puerta, pasillo, escalera, etc.)
- Interactividad mediante `GestureDetector` para mostrar información al tocar

Herramientas de desarrollo La aplicación incluye un modo de desarrollo para facilitar la creación del grafo. Este modo permite:

- **Captura de coordenadas:** Tocar el mapa en modo debug registra las coordenadas SVG del punto seleccionado
- **Creación de nodos:** Un diálogo permite especificar el tipo y el ID del nodo, generando automáticamente el JSON correspondiente

- **Generación automática de IDs:** Los identificadores se generan siguiendo el patrón establecido, incluyendo el número de piso y un timestamp
- **Creación de conexiones:** Interfaz para seleccionar nodo origen y destino, calculando automáticamente la distancia euclídea
- **Visualización de conexiones:** Las conexiones se muestran como líneas con flechas direccionales y etiquetas de distancia
- **Exportación de datos:** Funcionalidad para copiar al portapapeles el JSON generado de nodos y conexiones
- **Estadísticas:** Visualización de la cantidad de nodos por tipo y validación de la integridad del grafo
- **Migración de tipos:** Herramienta para agregar automáticamente tipos a nodos existentes mediante inferencia
- **Recarga dinámica:** Capacidad de recargar los nodos desde el archivo JSON sin reiniciar la aplicación

El modo debug utiliza marcadores visuales de color naranja para diferenciar los puntos capturados de los nodos permanentes del grafo, y permite activarse o desactivarse mediante un botón en la interfaz.

Herramientas avanzadas de desarrollo Además de las herramientas básicas mencionadas, el modo debug incluye funcionalidades avanzadas para facilitar el desarrollo y mantenimiento del sistema:

Visualización de conexiones en tiempo real. El sistema permite visualizar las conexiones entre nodos directamente sobre el mapa mediante `CustomPainter`:

- Líneas que conectan nodos relacionados
- Flechas direccionales para conexiones unidireccionales
- Etiquetas con la distancia entre nodos
- Colores diferenciados para conexiones activas y debug

Estadísticas del grafo. Un diálogo de estadísticas muestra:

- Cantidad de nodos por tipo (entradas, pasillos, escaleras, etc.)
- Porcentaje de nodos con tipo explícito vs. inferido
- Total de conexiones en el grafo
- Nodos huérfanos (sin conexiones)

Herramienta de migración de tipos. Permite agregar automáticamente el campo `tipo` a nodos existentes:

```

1 Future<void> _migrarNodosConTipo() async {
2   final raw = await rootBundle.loadString(rutaGrafoJson);
3   final data = json.decode(raw) as Map<String, dynamic>;
4
5   final nodosActualizados = (data['nodos'] as List).map((nodo) {
6     final nodoMap = nodo as Map<String, dynamic>;
7
8     // Si ya tiene tipo, mantenerlo
9     if (nodoMap.containsKey('tipo')) return nodoMap;
10
11    // Inferir tipo basado en ID
12    final tipoInferido = _obtenerTipoNodoPorId(
13      nodoMap['id'] as String
14    );
15
16    return {
17      ...nodoMap,

```

```

18     'tipo': tipoInferido?.name ?? 'puntoInteres',
19   };
20 }).toList();
21
22 final jsonActualizado = JsonEncoder.withIndent('  ')
23   .convert({'nodos': nodosActualizados, ...data});
24
25 Clipboard.setData(ClipboardData(text: jsonActualizado));
26 }

```

Listing 10: Migración automática de tipos

Generador de códigos QR. Desde el diálogo de información de cualquier nodo, se puede generar su código QR:

- Formato JSON compatible con el scanner de la aplicación
- Vista previa del contenido del QR
- Opción para copiar al portapapeles
- Útil para pruebas sin necesidad de imprimir códigos físicos

Recarga dinámica de nodos. Permite actualizar el grafo sin reiniciar la aplicación:

- Lee el archivo JSON actualizado desde los assets
- Reemplaza los nodos en memoria
- Mantiene el estado de selección de origen/destino
- Actualiza la visualización automáticamente

Prueba de rutas en diálogo de coordenadas. Al crear un nuevo nodo, se puede:

- Seleccionar nodos origen y destino de los existentes
- Calcular una ruta de prueba con A*
- Ver la distancia total y los nodos intermedios
- Validar que el nuevo nodo esté correctamente conectado

Diagnóstico del sistema. Un diálogo de diagnóstico verifica:

- Existencia de archivos SVG de todos los pisos
- Validez de archivos JSON del grafo
- Tamaño y estructura de cada archivo
- Cálculo de ruta demo entre nodos conocidos

Activación y desactivación del modo debug El modo debug se controla mediante:

- Constante global `kDebugMode` al inicio de `main.dart`
- Botón flotante en la interfaz para activar/desactivar dinámicamente
- Estado persistente durante la sesión de la aplicación
- Limpieza automática de datos debug al desactivar

Cuando se desactiva el modo debug, se limpian automáticamente:

- Coordenadas capturadas
- Conexiones temporales
- Marcadores visuales de depuración

2.2.7. Implementación del algoritmo A* para cálculo de rutas

Para calcular las mejores rutas entre los distintos puntos de interés dentro de la facultad, se implementó un algoritmo de búsqueda en grafos. El algoritmo elegido fue **A*** (**A estrella**), un método de búsqueda heurística que encuentra la ruta más corta entre dos puntos dentro de un grafo.

Fundamentos del algoritmo A* A* es un algoritmo de búsqueda informada que combina dos valores fundamentales:

- $g(n)$: el costo real acumulado desde el nodo inicial hasta el nodo n
- $h(n)$: una heurística que estima el costo faltante desde el nodo n hacia el objetivo

De este modo, evalúa cada nodo mediante la función:

$$f(n) = g(n) + h(n) \quad (1)$$

Esta combinación permite que A* sea eficiente y encuentre rutas óptimas siempre que la heurística sea admisible (no sobreestima el costo real). La heurística utilizada en la aplicación es la **distancia euclíadiana** basada en las coordenadas (x, y) de los nodos:

$$h(n_1, n_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \quad (2)$$

El algoritmo A* utiliza una heurística basada en la distancia euclíadiana entre nodos, calculada a partir de sus coordenadas en el plano. Esta heurística permite guiar la búsqueda de manera eficiente, sin necesidad de conocer distancias físicas exactas. Su uso es adecuado en este contexto, ya que el edificio se modela como un entorno discreto representado mediante un grafo.

Implementación en Dart La implementación se realizó en el archivo `a_estrella.dart`, integrándose con la clase `Grafo` que modela los nodos (con sus coordenadas) y la adyacencia entre ellos.

Estructura de la clase AStar. Se creó la clase `AStar`, que funciona en dos modos para mantener flexibilidad:

- Como **instancia**, recibiendo un objeto `Grafo` y proporcionando un método `calcular()`
- Como **API estática**, mediante `AStar.calcularRuta()`, manteniendo compatibilidad con versiones previas

```

1 class AStar {
2   final Grafo grafo;
3   AStar(this.grafo);
4
5   List<String> calcular({required String origen,
6                           required String destino}) {
7     return AStar.calcularRuta(
8       grafo: grafo,
9       origen: origen,
10      destino: destino
11    );
12  }
13}
```

Listing 11: Estructura de la clase AStar

Mapa de adyacencias. El algoritmo genera primero un mapa de adyacencias a partir del grafo, donde cada nodo queda asociado a sus vecinos con los costos de distancia correspondientes:

```

1 final mapa = grafo.generarMapaAdyacencia();
2 final nodos = grafo.nodos;
3 final nodosMap = {for (var n in nodos) n.id: n};
```

Tablas internas del algoritmo. Se inicializan tres estructuras de datos esenciales:

- **gScore**: almacena el costo acumulado desde el origen
- **fScore**: almacena el costo estimado total ($f = g + h$)
- **prev**: registra los predecesores para reconstruir la ruta

Todos los valores comienzan como infinito excepto el nodo inicial:

```

1  for (var n in mapa.keys) {
2      gScore[n] = double.infinity;
3      fScore[n] = double.infinity;
4      prev[n] = null;
5  }
6  gScore[origen] = 0;
7  fScore[origen] = _heuristica(nodosMap[origen]!,
8                               nodosMap[destino]!);

```

Proceso principal del algoritmo. El algoritmo itera sobre un conjunto de nodos abiertos siguiendo estos pasos:

1. Selecciona el nodo con menor **fScore** del conjunto abierto
2. Si es el destino, reconstruye la ruta y finaliza
3. Si no, revisa cada vecino del nodo actual:
 - Calcula un costo tentativo: $g_{tentativo} = g_{actual} + distancia(actual, vecino)$
 - Si es menor al costo conocido, actualiza **gScore**, **fScore** y **prev**
 - Agrega el vecino al conjunto abierto

```

1  final abiertos = <String>{origen};
2
3  while (abiertos.isNotEmpty) {
4      final actual = abiertos.reduce(
5          (a, b) => fScore[a]! < fScore[b]! ? a : b
6      );
7
8      if (actual == destino) {
9          return _reconstruir(prev, destino);
10     }
11
12     abiertos.remove(actual);
13
14     for (var vecino in mapa[actual]!.keys) {
15         final tentativeG = gScore[actual]! + mapa[actual]![vecino]!;
16
17         if (tentativeG < gScore[vecino]!) {
18             prev[vecino] = actual;
19             gScore[vecino] = tentativeG;
20             fScore[vecino] = tentativeG +
21                 _heuristica(nodosMap[vecino]!, nodosMap[destino]!);
22             abiertos.add(vecino);
23         }
24     }
25 }
26 return [] ; // No se encontró ruta

```

Listing 12: Bucle principal de A*

Reconstrucción de la ruta. Cuando se alcanza el nodo destino, se reconstruye el camino recorriendo el mapa de predecesores desde el objetivo hacia el origen:

```

1 static List<String> _reconstruir(
2     Map<String, String?> prev,
3     String destino) {
4     final ruta = <String>[];
5     var actual = destino;
6
7     while (prev[actual] != null) {
8         ruta.insert(0, actual);
9         actual = prev[actual]!;
10    }
11
12    ruta.insert(0, actual); // Agregar origen
13    return ruta;
14}

```

Listing 13: Reconstrucción de la ruta

Integración con la interfaz gráfica El algoritmo se integró con la interfaz desarrollada en Flutter mediante la pantalla de selección de destino (`pantalla_seleccion_destino.dart`). El proceso de uso es el siguiente:

1. El usuario escanea un código QR que identifica su ubicación actual (nodo origen)
2. Se abre una pantalla que muestra el origen y permite seleccionar el destino mediante un menú desplegable
3. Al presionar “Calcular Ruta”, se crea una instancia de AStar:

```

1 final ruta = AStar.calcularRuta(
2     grafo: widget.grafo,
3     origen: widget.nodoOrigenId,
4     destino: _nodoDestinoSeleccionado!,
5 );

```

4. La ruta resultante es una lista ordenada de IDs de nodos
5. La distancia total se calcula sumando las distancias euclidianas entre nodos consecutivos
6. La aplicación visualiza el recorrido paso a paso en una lista numerada
7. Al confirmar, la ruta se dibuja sobre el mapa SVG mediante un `CustomPainter` (`RutaPainter`)

Visualización de rutas en el mapa La clase `RutaPainter` implementa un pintor personalizado que dibuja la ruta calculada sobre el mapa:

- Convierte los IDs de nodos a coordenadas de pantalla
- Dibuja líneas conectando los nodos secuencialmente
- Utiliza colores distintivos (azul para el camino, verde para origen, rojo para destino)
- Agrega efectos visuales como sombras y gradientes para mejorar la legibilidad
- Dibuja círculos en cada nodo de la ruta para destacar los puntos de paso

Este proceso permite navegación precisa entre salas, laboratorios, oficinas y pasillos, proporcionando al usuario una guía visual clara para desplazarse dentro de la facultad.

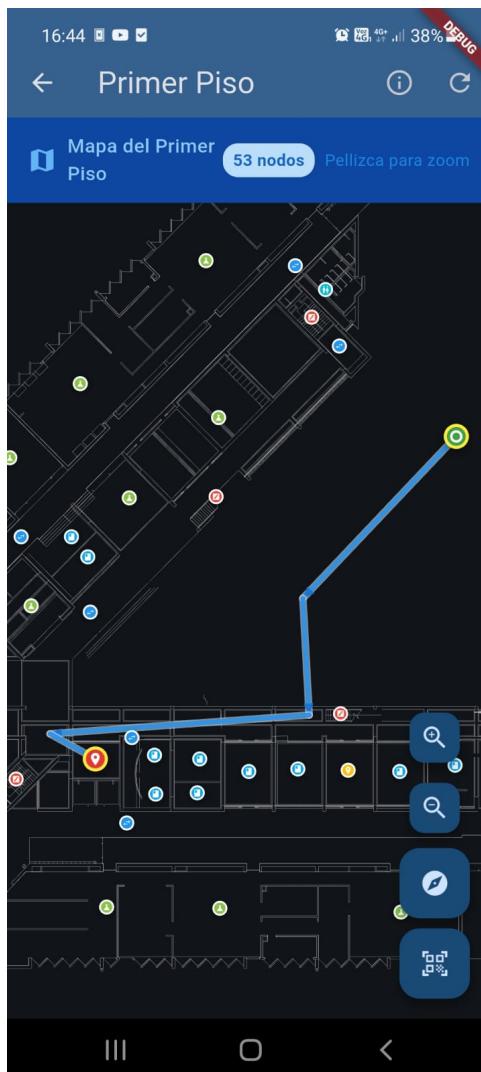


Figura 10: Ruta óptima calculada con el algoritmo A* desde el origen (verde) hasta el destino (rojo), mostrando el camino con líneas azules y nodos intermedios

Ventajas de A* sobre alternativas La elección de A* sobre el algoritmo de Dijkstra se justifica por varias razones:

- **Eficiencia:** A* explora menos nodos al usar la heurística para guiar la búsqueda hacia el objetivo
- **Optimalidad:** Garantiza encontrar la ruta más corta cuando la heurística es admisible
- **Rendimiento:** En grafos espaciales como los mapas de edificios, la distancia euclíadiana es una excelente heurística que reduce significativamente el espacio de búsqueda
- **Escalabilidad:** El algoritmo funciona eficientemente incluso con grafos grandes (los 4 pisos suman más de 100 nodos)

Integración con el sistema de navegación por QR El sistema de códigos QR permite a los usuarios escanear marcadores físicos ubicados en puntos estratégicos de la facultad. Cada código QR contiene información en formato JSON que identifica un nodo específico del grafo:

```

1 {
2   "type": "nodo",
3   "id": "P1_Entrada_1",
4   "piso": 1,
5   "x": 100,
6   "y": 200

```

7 }

Listing 14: Formato de códigos QR

La aplicación valida y parsea estos códigos mediante la clase `QRUtils` en `codigo_qr.dart`, que soporta múltiples formatos (JSON moderno y formatos legacy) para mantener compatibilidad. Una vez identificado el nodo origen, A* calcula la ruta óptima al destino seleccionado por el usuario.

Conclusión sobre la implementación de A* La integración del algoritmo A* con el modelo de grafos y la interfaz gráfica permite una experiencia de navegación interna eficiente y clara. El algoritmo determina rutas óptimas usando distancias reales y una heurística adecuada para espacios 2D, mientras Flutter proporciona la visualización interactiva del plano de la facultad.

Este diseño modular facilita extender la aplicación a nuevos pisos o edificios simplemente añadiendo los mapas SVG correspondientes y sus archivos JSON de grafos, sin necesidad de modificar la lógica del algoritmo de búsqueda.

2.2.8. Creación y generación automatizada de códigos QR

Para la implementación del sistema de navegación interior, se desarrolló un programa en Python que automatiza la generación de códigos QR para cada nodo definido en los grafos de navegación. Este sistema permite generar de forma rápida y consistente todos los códigos QR necesarios para los 4 pisos de la Facultad de Ingeniería, facilitando el despliegue físico del sistema.

Arquitectura del sistema de generación El sistema de generación de códigos QR está compuesto por tres scripts principales ubicados en el directorio `scripts/`:

1. **generar_qrs.py**: Script principal que genera códigos QR para todos los pisos del edificio de forma automatizada. Incluye generación masiva de QRs, lectura automática de archivos JSON de grafos, creación de estructura de carpetas organizada, estadísticas detalladas de generación y generación automática de documentación README.
2. **generar_qr_piso.py**: Script auxiliar de utilidad para regenerar códigos QR de un piso específico, útil durante el desarrollo y mantenimiento. Uso: `python generar_qr_piso.py [número_piso]`
3. **verificar_formato_qr.py**: Script de pruebas que valida que los QRs generados sean compatibles con el formato esperado por la aplicación Flutter.

Dependencias del sistema El sistema utiliza las siguientes bibliotecas de Python, definidas en `requirements.txt`:

`qrcode[pil]==8.2`: Biblioteca principal para la generación de códigos QR. Proporciona soporte para diferentes versiones de QR (1-40), múltiples niveles de corrección de errores (L, M, Q, H), personalización de tamaño y borde, y exportación a múltiples formatos de imagen.

`Pillow>=11.0.0`: Biblioteca de procesamiento de imágenes (PIL - Python Imaging Library). Se utiliza para renderización de códigos QR en formato PNG, manipulación de colores y contraste, configuración de la calidad de salida y soporte para diferentes formatos de imagen.

Configuración de generación La configuración principal del sistema establece los parámetros óptimos para la generación de códigos QR:

```

1 QR_CONFIG = {
2     'version': 1,
3     'error_correction': qrcode.constants.ERROR_CORRECT_H,
4     'box_size': 10,
5     'border': 4,
6 }
```

Listing 15: Configuración de generación de QR

Los parámetros significan:

- **version**: 1 - Define el tamaño del QR. Versión 1 es el QR más pequeño posible (21×21 módulos), que se ajusta automáticamente si el contenido es mayor.

- **error_correction:** ERROR_CORRECT_H - Nivel de corrección de errores al 30 %. Este nivel permite que el QR funcione incluso si está parcialmente dañado, sucio o deteriorado, ideal para instalaciones físicas en ambientes educativos.
- **box_size:** 10 - Tamaño en píxeles de cada módulo del QR. Con 10 píxeles, un QR versión 1 genera una imagen de aproximadamente 290×290 píxeles.
- **border:** 4 - Tamaño del borde en módulos. El estándar QR requiere mínimo 4 módulos de borde para garantizar el escaneo correcto.

Funciones principales del sistema `leer_grafo_json(ruta_json)`: Lee y valida archivos JSON de grafos de navegación. Implementa validaciones de existencia del archivo, formato JSON válido, presencia de la clave 'nodos' y manejo robusto de errores con mensajes descriptivos.

`extraer_numero_piso(nodo_id)`: Extrae el número de piso del identificador del nodo siguiendo la convención de nomenclatura del proyecto. El formato esperado es P{numero}_{descripcion} (ejemplo: P1_Entrada_1).

`crear_datos_qr(nodo, piso_default)`: Genera el contenido JSON que será codificado en cada código QR. El formato de salida incluye los campos `type`, `id`, `piso`, `x` e `y`.

`generar_qr_imagen(datos_qr, ruta_salida)`: Genera la imagen PNG del código QR con la configuración especificada. El proceso interno crea un objeto QRCode, agrega los datos JSON, optimiza el tamaño, genera la imagen en blanco y negro y guarda en formato PNG comprimido.

`generar_qrs_desde_grafo(ruta_json, carpeta_salida, numero_piso)`: Función principal que coordina la generación masiva de QRs para un piso completo. Lee el archivo JSON del grafo, extrae la lista de nodos, crea la carpeta de salida y genera cada imagen QR reportando el progreso.

`generar_qrs_todos_los_pisos(directorio_base)`: Función de orquestación que genera códigos QR para todos los pisos del edificio. Procesa los 4 pisos secuencialmente y muestra estadísticas completas del proceso.

Especificaciones técnicas del QR Los códigos generados tienen las siguientes características: versión QR 1 con auto-ajuste (tamaño mínimo posible, óptimo para datos pequeños), corrección de errores nivel H de 30 % (máxima durabilidad en ambientes de alto tráfico), tamaño de imagen aproximado de 290×290 píxeles (suficiente para impresión a 5×5 cm con 300 DPI), formato PNG (compresión sin pérdida, ideal para impresión), colores blanco y negro (máximo contraste para lectura confiable), y tamaño de datos de 80-120 bytes (JSON compacto con información esencial).

El nivel de corrección de errores H permite recuperar hasta el 30 % de la información del código aunque esté dañada. Esto es crítico porque los stickers pueden sufrir desgaste físico (rayarse o deteriorarse), acumular suciedad (polvo o manchas), enfrentar condiciones de luz adversas (reflejo o sombras durante el escaneo) y tener impresión imperfecta (variaciones en la calidad).

Estructura de salida Después de ejecutar el script, se genera la siguiente estructura:

```
qr_codes/
|-- README.md          # Documentacion automatica
|-- piso1/
|   |-- QR_P1_Entrada_1.png
|   |-- QR_P1_Pasillo_Norte.png
|   +-- ... (50 archivos)
|-- piso2/
|   +-- ... (24 archivos)
|-- piso3/
|   +-- ... (22 archivos)
+- piso4/
    +-- ... (12 archivos)
```

Total: 108 códigos QR organizados por piso (50 en piso 1, 24 en piso 2, 22 en piso 3 y 12 en piso 4).

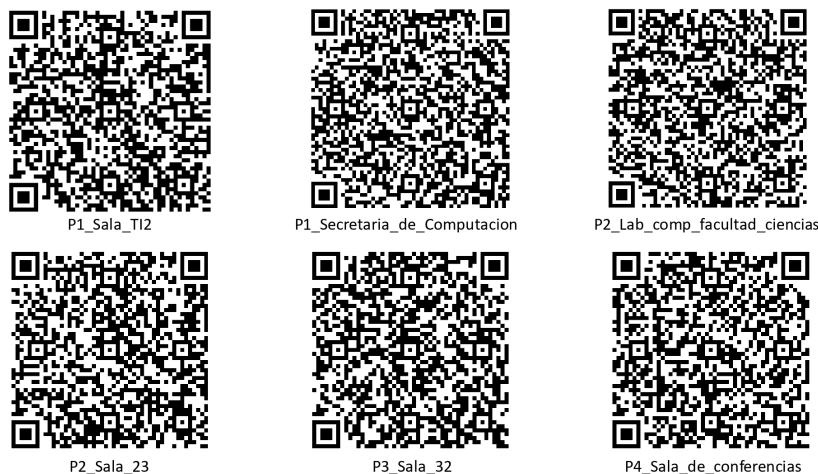


Figura 11: Ejemplos de códigos QR generados automáticamente para diferentes puntos de interés en los cuatro pisos

Consideraciones de implementación física Para la instalación de los códigos QR, se recomienda un tamaño de 5×5 cm (mínimo funcional 3×3 cm), resolución de 300 DPI o superior, material de stickers vinilo plastificado resistente al agua, e impresora láser preferiblemente para mejor contraste.

La instalación debe realizarse a una altura estándar de 1.5 metros desde el suelo, en superficies planas, visibles y accesibles, con orientación perpendicular a la línea de visión y evitando zonas con reflejo directo de iluminación.

El mantenimiento incluye limpieza con paño húmedo cada 3 meses, inspección mensual para verificar legibilidad y reemplazo cuando el daño supere el 30 %.

2.2.9. Sistema de lectura e interpretación de códigos QR

El sistema de códigos QR implementado en la aplicación permite a los usuarios identificar rápidamente ubicaciones, nodos del grafo, rutas completas e incluso coordenadas específicas dentro de los mapas SVG. Este módulo mejora la experiencia de navegación y agiliza el acceso a información relevante dentro de la facultad.

Objetivos de la implementación La integración de códigos QR cumple tres objetivos principales: identificación rápida de ubicaciones sin necesidad de buscar manualmente en el mapa, facilitar la navegación automática hacia salas, oficinas, laboratorios o puntos de interés, y compatibilidad con diversos formatos, permitiendo flexibilidad tanto operativa como en la fase de depuración.

Arquitectura del sistema QR El sistema se compone de tres módulos esenciales:

QRScannerScreen (pantalla.lectora_qr.dart): Pantalla encargada de activar la cámara, detectar códigos QR en tiempo real, validar y enviar los datos escaneados, y proveer controles como flash, pausa y vista previa.

QRUtils (codigo_qr.dart): Este módulo centraliza la lógica completa de interpretación y generación de códigos QR. Soporta múltiples tipos de códigos: formato **nodo**: para selección directa de nodos (ejemplo: `nodo:P1_Entrada_1`), formato **ruta**: para cálculo de ruta con A* (ejemplo: `ruta:P1_A|P1_B`), formato **piso**: para navegación entre pisos (ejemplo: `piso:1|nodo:P1_Entrada_1`), formato **coord**: para posicionamiento en SVG (ejemplo: `coord:900,350`), formato **ubicacion**: para alias amigables (ejemplo: `ubicacion:Ascensor`), y formato JSON generado por Python (ejemplo: `{"type": "nodo", "id": "P1_Entrada_1"}`).

Las funciones clave del módulo incluyen `parseQRCode()`, `esQRValido()`, `procesarQRConGrafo()`, conversiones de alias a ID y generación de QR dinámicos.

QRNavigation (navegacion_qr.dart): Define la reacción de la app ante cada tipo de QR procesado, incluyendo mostrar nodo, calcular rutas con A*, mostrar coordenadas y navegar automáticamente entre pantallas.

Flujo de funcionamiento El proceso completo de escaneo y procesamiento sigue estos pasos:

1. **Apertura del lector QR:** El usuario abre el lector y la app crea una instancia de `QRScannerScreen` con el piso y grafo actual.
2. **Lectura del QR:** El contenido escaneado se envía a `parseQRCode()`.
3. **Validación de formato:** Si el QR es soportado, se procede; de lo contrario, se avisa al usuario.
4. **Interpretación del contenido:** Según el tipo de QR, se extraen IDs de nodos, pisos, alias, coordenadas o rutas.
5. **Procesamiento:** La función `procesarQRConGrafo()` implementa lógica adicional, como buscar nodos en el grafo, calcular rutas usando A*, determinar distancias y convertir coordenadas.
6. **Acción final en la interfaz:** Usando `QRNavigation`, la app regresa al mapa, selecciona nodos, muestra rutas, centra el mapa e inicia navegación paso a paso.

Ejemplos de uso **Escanear nodo:** `nodo:P1_Entrada_1` selecciona automáticamente la entrada del piso 1.

Escanear ruta: `ruta:P1_Entrada_1|P1_Lab_Tesla` calcula la ruta con A*, muestra los pasos e inicia la navegación.

Escanear coordenadas: `coord:1020,540` centra el mapa en la posición indicada.

Escanear alias: `ubicacion:Ascensor` se interpreta como `P1_Ascensor`.

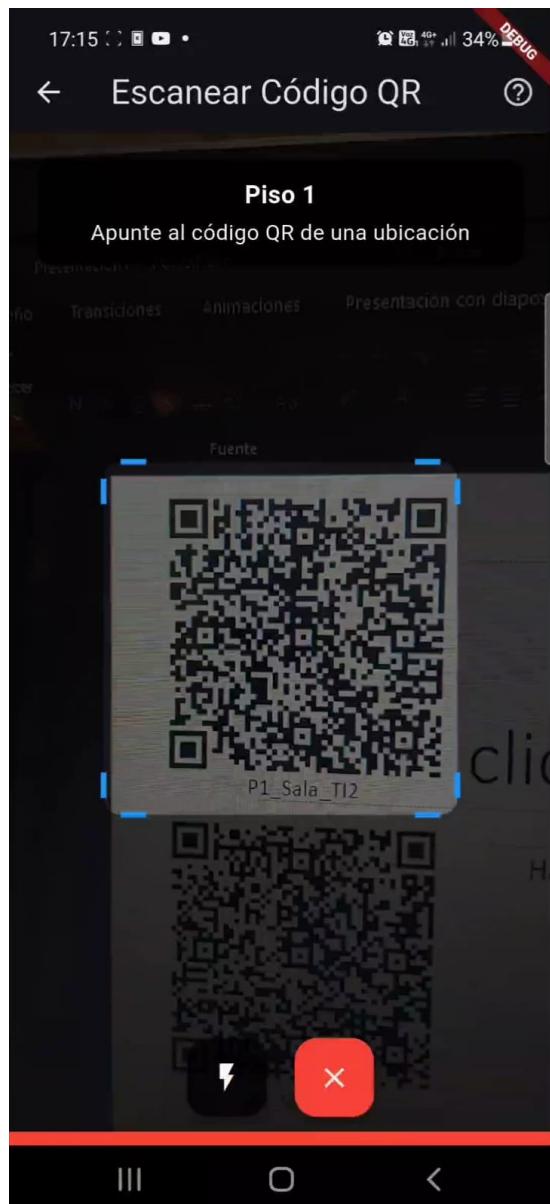


Figura 12: Pantalla de lectura de códigos QR con vista previa de cámara activa

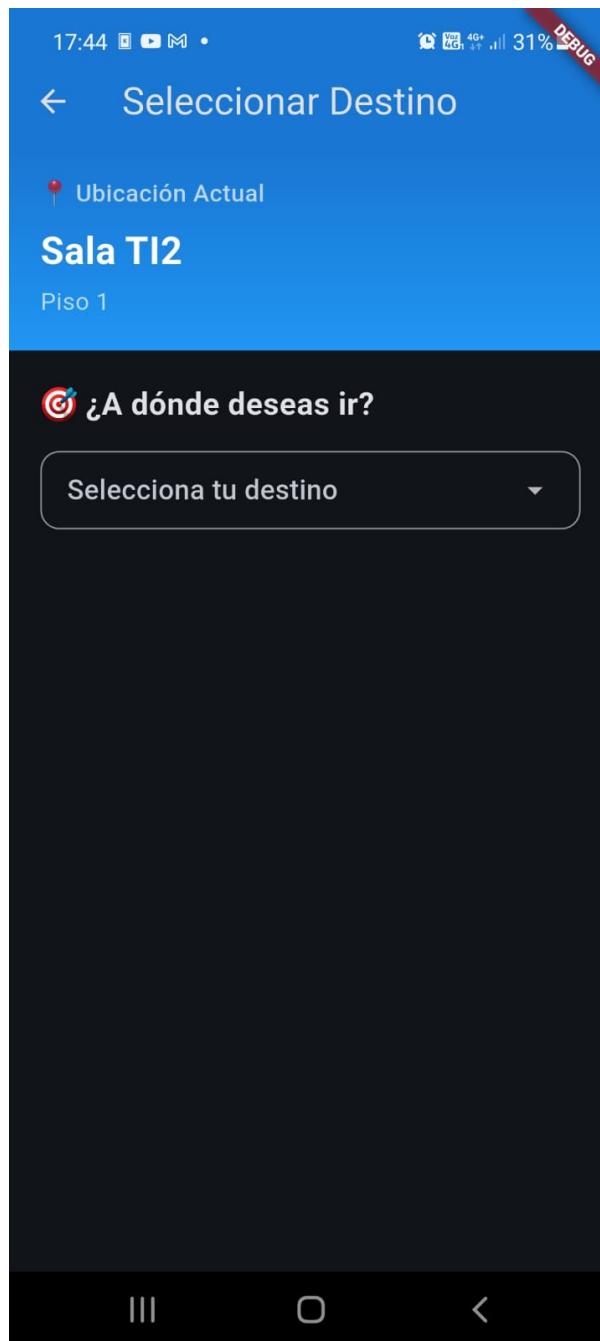


Figura 13: Mensaje de confirmación de lectura exitosa de código QR mostrando el ID del nodo identificado

Generación de códigos QR desde la aplicación En modo debug, desde el mapa se puede tocar un nodo y seleccionar "Generar QR", lo que copia al portapapeles un QR válido para ese nodo. Esta funcionalidad facilita el testing y la creación de nuevos códigos durante el desarrollo.

Ventajas del sistema QR El sistema proporciona navegación más rápida dentro del edificio, integración completa con el grafo y A*, soporte para mapas complejos con coordenadas, señalización física dentro de la facultad y simplificación del debugging del sistema.

2.2.10. Solución al problema de compatibilidad de formatos QR

Durante el desarrollo inicial del sistema, se identificó un problema crítico: los códigos QR generados por el script de Python mostraban el error "formato QR no soportado."^{al} ser escaneados por la aplicación Flutter.

Causa raíz del problema El script Python generaba códigos con formato JSON completo: {"type": "nodo", "id": "P1_EEntrada_1", "piso": 1, "x": 100, "z": 200}, mientras que la aplicación esperaba formatos de texto simple como nodo:P1_EEntrada_1, piso:1|nodo:P1_EEntrada_1, etc.

Solución implementada Se actualizó el archivo `lib/utils/codigo_qr.dart` para soportar ambos formatos, manteniendo compatibilidad retroactiva con los códigos existentes.

Los cambios incluyeron:

1. **Soporte JSON añadido:** Se implementó detección y parsing de formato JSON al inicio del proceso de validación. El sistema verifica si `qrData` comienza con '{' y termina con '}', decodifica el JSON con `json.decode()`, valida la presencia del campo `type`, extrae `id`, `piso`, `x` e `y` según el tipo, y usa fallback a formatos legacy si el JSON falla.
2. **Validación actualizada:** La función `esQRValido()` ahora valida primero si es JSON y mantiene compatibilidad con formatos legacy.
3. **Importación de librería:** Se añadió `import 'dart:convert'`; para el manejo de JSON.

Formatos soportados La aplicación ahora reconoce todos estos formatos:

JSON (generado por Python): {"type": "nodo", "id": "P1_EEntrada_1", "piso": 1, "x": 100, "z": 200}

Texto simple (formatos legacy): nodo:P1_EEntrada_1, piso:1|nodo:P1_EEntrada_1, ubicacion:Entrada Principal, coord:1004,460, ruta:P1_EEntrada_1|P1_Pasillo_Norte

ID directo: P1_EEntrada_1

Verificación y pruebas Se creó el script `scripts/verificar_formato_qr.py` que verifica la compatibilidad de todos los códigos generados. La verificación confirmó que los 108 QRs generados (50 en piso 1, 24 en piso 2, 22 en piso 3 y 12 en piso 4) son válidos y compatibles con la aplicación.

Script de verificación de formatos Para garantizar la compatibilidad entre los códigos QR generados y la aplicación, se emplea el script `scripts/verificar_formato_qr.py`, previamente descrito.

Funciones principales:

- Lee los archivos JSON de todos los pisos
- Genera datos QR de prueba para cada nodo
- Simula el proceso de decodificación de la aplicación
- Verifica compatibilidad con todos los formatos soportados
- Genera reporte de validación

```

1 def probar_qr_desde_grafo(ruta_json):
2     data = leer_grafo_json(ruta_json)
3     if not data:
4         return
5
6     nodos = data.get('nodos', [])
7     print(f"\n{'='*70}")
8     print(f"Probando {len(nodos)} nodos del archivo:")
9     print(f"  {ruta_json}")
10    print(f"{'='*70}\n")
11
12    validos = 0
13    invalidos = 0
14
15    for nodo in nodos:
16        qr_data = crear_datos_qr(nodo)
17
18        # Simular decodificación
19        es_valido = verificar_formato(qr_data)

```

```

20
21     if es_valido:
22         validos += 1
23     else:
24         invalidos += 1
25         print(f"  Warning  QR invalido: {nodo.get('id')}")
26
27     print(f"\nResumen:")
28     print(f"  QRs validos: {validos}")
29     print(f"  QRs invalidos: {invalidos}")

```

Listing 16: Verificación de formato QR

La verificación confirmó que los 108 códigos QR generados son compatibles con la aplicación, distribuidos en: 50 códigos en piso 1, 24 en piso 2, 22 en piso 3 y 12 en piso 4.

Compatibilidad retroactiva La solución mantiene:

- Funcionamiento de QRs antiguos (formatos legacy)
- Funcionamiento de QRs nuevos (formato JSON)
- Sin cambios en el resto del código
- Sin necesidad de regenerar QRs antiguos

La lógica de parsing JSON utiliza detección por delimitadores, parsing con manejo de excepciones, validación de campos requeridos, extracción segura de datos y fallback silencioso a otros formatos si el JSON falla. El formato JSON se verifica primero antes de intentar otros formatos, asegurando prioridad al formato generado por el sistema automatizado de Python.

2.2.11. Sistema de navegación y cálculo de rutas

La aplicación implementa un sistema de navegación dentro de cada piso del edificio, permitiendo calcular y visualizar rutas óptimas entre diferentes puntos de interés.

Restricción de navegación por piso El sistema actual está diseñado para navegar dentro de un mismo piso por simplicidad y eficiencia. Cuando el usuario intenta seleccionar un destino en un piso diferente al origen, la aplicación valida esta condición y muestra un mensaje de advertencia:

```

1 void _establecerDestino(String nodoId) async {
2     // Verificar que origen y destino esten en el mismo piso
3     final pisoOrigen = _extraerPisoDeNodoId(_origenSeleccionado);
4     final pisoDestino = _extraerPisoDeNodoId(nodoId);
5
6     if (pisoOrigen != pisoDestino) {
7         ScaffoldMessenger.of(context).showSnackBar(
8             const SnackBar(
9                 content: Text('El origen y destino deben estar en el mismo piso'),
10                backgroundColor: Colors.orange,
11            ),
12        );
13        return;
14    }
15
16    // Continuar con calculo de ruta usando A*
17    setState(() => _destinoSeleccionado = nodoId);
18    await _calcularYMostrarRuta(_origenSeleccionado!, nodoId);
19}

```

Listing 17: Validación de mismo piso

Esta decisión de diseño simplifica la experiencia del usuario y evita complejidades relacionadas con el cambio de pisos durante la navegación. Para navegar a otro piso, el usuario debe:

1. Regresar al menú principal
2. Seleccionar el piso destino deseado
3. Establecer un nuevo origen y destino dentro de ese piso

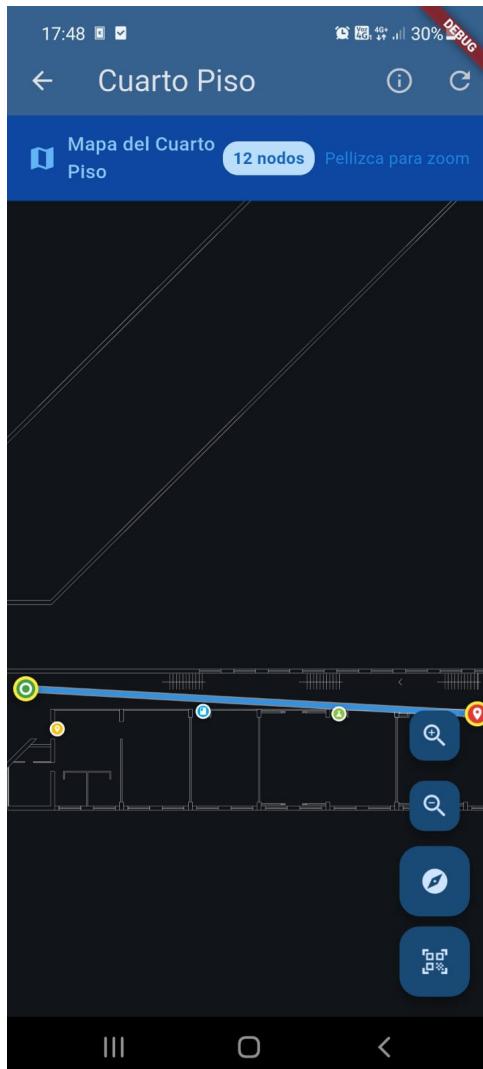


Figura 14: Selección de nodo origen (verde) y nodo destino (rojo) mediante interacción táctil en el mapa

Cálculo de rutas con A* Una vez validado que origen y destino están en el mismo piso, el sistema procede a calcular la ruta óptima:

```

1 Future<void> _calcularYMostrarRuta(String origen, String destino) async {
2   try {
3     final grafoJson = await rootBundle.loadString(rutaGrafoJson);
4     final grafoData = json.decode(grafoJson);
5     final grafo = Grafo.fromJson(grafoData);
6
7     // Calcular ruta usando A*
8     final ruta = AStar.calcularRuta(
9       grafo: grafo,
10      origen: origen,
11      destino: destino,
12    );
13
14   if (ruta.isEmpty) {

```

```

15     _mostrarError('No se encontro una ruta entre los puntos');
16     return;
17 }
18
19 // Actualizar estado con la ruta calculada
20 setState(() {
21     _rutaActiva.clear();
22     _rutaActiva.addAll(ruta);
23 });
24
25 // Calcular y mostrar distancia total
26 final distancia = _calcularDistanciaRuta(ruta);
27 _mostrarConfirmacion(
28     'Ruta calculada: ${distancia.toStringAsFixed(1)} metros'
29 );
30 } catch (e) {
31     _mostrarError('Error al calcular ruta: $e');
32 }
33 }
```

Listing 18: Cálculo de ruta en el mismo piso

Visualización de la ruta La ruta calculada se visualiza sobre el mapa mediante un sistema de líneas dibujadas con `CustomPainter`:

- **Líneas de ruta:** Conectan secuencialmente los nodos del camino calculado
- **Marcadores especiales:** El origen se destaca en verde y el destino en rojo
- **Nodos intermedios:** Se marcan en azul a lo largo del recorrido
- **Información de distancia:** Se muestra la distancia total en la interfaz

```

1 List<Widget> _buildLineasRuta() {
2     if (_rutaActiva.length < 2) return [];
3
4     final lineas = <Widget>[];
5
6     for (int i = 0; i < _rutaActiva.length - 1; i++) {
7         final nodoActual = _nodos.firstWhere(
8             (n) => n['id'] == _rutaActiva[i]
9         );
10        final nodoSiguiente = _nodos.firstWhere(
11            (n) => n['id'] == _rutaActiva[i + 1]
12        );
13
14        final inicio = _calcularPosicionEscalada(
15            nodoActual['x'] as double,
16            nodoActual['y'] as double,
17        );
18        final fin = _calcularPosicionEscalada(
19            nodoSiguiente['x'] as double,
20            nodoSiguiente['y'] as double,
21        );
22
23        lineas.add(
24            CustomPaint(
25                painter: RutaPainter(inicio: inicio, fin: fin),
26            ),
27        );
28    }
29
30    return lineas;
```

31 }

Listing 19: Construcción de líneas de ruta

Limpieza de selección El usuario puede limpiar la selección actual y comenzar una nueva navegación en cualquier momento:

```

1 void _limpiarSeleccion() {
2     setState(() {
3         _origenSeleccionado = null;
4         _destinoSeleccionado = null;
5         _rutaActiva.clear();
6
7         // Limpiar conexiones debug si estan activas
8         if (_modoDebugActivo) {
9             _conexionesDebug.clear();
10        }
11    });
12
13 ScaffoldMessenger.of(context).showSnackBar(
14     const SnackBar(
15         content: Row(
16             children: [
17                 Icon(Icons.clear, color: Colors.white),
18                 SizedBox(width: 12),
19                 Text('Selección limpia'),
20             ],
21         ),
22         duration: Duration(seconds: 2),
23     ),
24 );
25 }
```

Listing 20: Limpieza de ruta activa

Ventajas del enfoque simplificado

- **Simplicidad:** La interfaz es más intuitiva al enfocarse en un piso a la vez
- **Rendimiento:** Reduce la carga computacional al trabajar con grafos individuales
- **Claridad visual:** El usuario siempre ve claramente el piso en el que está navegando
- **Mantenibilidad:** El código es más simple y fácil de mantener sin lógica multi-piso
- **Escalabilidad:** Permite agregar nuevos pisos sin complejidad adicional en el sistema de navegación

2.2.12. Pantalla de selección de destino

Para facilitar la selección de destinos, se implementó una pantalla dedicada (`pantalla_seleccion_destino.dart`) que permite al usuario buscar y elegir su destino de manera intuitiva.

Características principales **Lista de nodos disponibles.** La pantalla muestra todos los nodos del piso actual con las siguientes capacidades:

- Visualización de todos los destinos posibles dentro del mismo piso
- Agrupación visual por tipo de nodo (aulas, laboratorios, oficinas, etc.)
- Iconos distintivos según el tipo de ubicación
- Indicación clara del nodo origen seleccionado

Selección directa. El usuario puede:

- Tocar cualquier nodo en el mapa para establecerlo como origen o destino
- Ver información detallada de cada nodo al seleccionarlo
- Confirmar la selección mediante diálogos intuitivos
- Visualizar la ruta calculada inmediatamente después de seleccionar ambos puntos

Flujo de uso

1. El usuario escanea un código QR o toca un nodo en el mapa para establecer el origen
2. Selecciona un segundo nodo en el mismo piso como destino
3. La aplicación valida que ambos nodos estén en el mismo piso
4. Si la validación es exitosa, se calcula automáticamente la ruta con A*
5. La ruta se visualiza sobre el mapa con líneas y marcadores
6. El usuario puede limpiar la selección y comenzar una nueva ruta en cualquier momento

Limitación actual Actualmente, la aplicación solo permite navegar dentro del mismo piso. Si el usuario intenta seleccionar un destino en un piso diferente, se muestra un mensaje indicando que debe navegar manualmente al piso destino:

“El origen y destino deben estar en el mismo piso. Por favor, selecciona el piso destino desde el menú principal.”

Esta decisión de diseño simplifica la experiencia del usuario y mantiene la claridad en la navegación, evitando complejidades adicionales relacionadas con cambios de piso durante el recorrido.

2.2.13. Integración del escáner QR con la navegación

El sistema de códigos QR se integra perfectamente con el sistema de navegación, permitiendo establecer rápidamente el punto de origen:

1. El usuario escanea un código QR físico ubicado en un punto de la facultad
2. El sistema identifica el nodo correspondiente en el grafo del piso actual
3. Se establece automáticamente como punto de origen para la navegación
4. El usuario puede seleccionar un destino tocando otro nodo en el mapa
5. La aplicación calcula y muestra la ruta óptima usando A*

Este flujo permite una experiencia de usuario fluida, combinando la precisión del posicionamiento por QR con la flexibilidad de la selección visual de destinos.

2.2.14. Pruebas de la aplicación

Con lo desarrollado, se realizaron pruebas exhaustivas para validar la funcionalidad del sistema de navegación y códigos QR:

- El algoritmo A* fue probado en múltiples escenarios, confirmando rutas óptimas que tienen sentido con la topología real del edificio.
- Los mapas SVG fueron verificados para asegurar que los nodos y conexiones coinciden con los planos arquitectónicos.
- La integración del escáner QR fue testeada con todos los códigos generados, confirmando compatibilidad y correcta identificación de nodos.
- La interfaz de usuario fue evaluada para garantizar una experiencia intuitiva y fluida.
- Se realizaron pruebas de usabilidad con usuarios reales para obtener retroalimentación y mejorar la experiencia.

3. Conclusiones y Recomendaciones

3.1. Conclusiones

El desarrollo de la aplicación de navegación interna para la Facultad de Ingeniería de la Universidad de Magallanes cumplió exitosamente con los objetivos planteados, logrando implementar un sistema funcional que facilita la orientación de estudiantes y visitantes dentro del edificio.

Logros principales:

- Se digitalizaron y optimizaron los planos arquitectónicos de los 4 pisos del edificio, convirtiéndolos a formato SVG para su visualización interactiva.
- Se implementó el algoritmo A* para el cálculo de rutas óptimas, demostrando ser eficiente y preciso para la navegación en espacios interiores.
- Se desarrolló un sistema de modelado mediante grafos que representa fielmente la topología de cada piso, incluyendo más de 100 nodos distribuidos entre salas, laboratorios, oficinas y pasillos.
- Se integró exitosamente un sistema de códigos QR que permite a los usuarios identificar rápidamente su ubicación actual, generando automáticamente 108 códigos QR para todos los puntos de interés.
- Se creó una interfaz gráfica intuitiva utilizando Flutter, con soporte para zoom, desplazamiento y visualización clara de rutas sobre mapas vectoriales.

Decisiones de diseño:

Durante el desarrollo, se optó por simplificar el sistema de navegación limitándolo a rutas dentro del mismo piso. Esta decisión resultó beneficiosa por varias razones:

- **Claridad visual:** El usuario siempre tiene claro en qué piso se encuentra y hacia dónde se dirige.
- **Simplicidad de uso:** La interfaz es más intuitiva al no requerir gestión compleja de cambios de piso durante la navegación.
- **Mantenibilidad del código:** El código resultante es más simple y fácil de mantener, reduciendo la probabilidad de errores.
- **Rendimiento:** Al trabajar con grafos individuales por piso, la aplicación consume menos recursos computacionales.

Para navegar a otro piso, el usuario simplemente regresa al menú principal y selecciona el piso destino deseado, estableciendo una nueva ruta dentro de ese nivel.

3.2. Recomendaciones

Para futuras mejoras del sistema, se recomienda:

- **Implementar navegación multi-piso:** Desarrollar un gestor que calcule rutas completas atravesando múltiples pisos mediante escaleras y ascensores, proporcionando opciones alternativas cuando existan múltiples caminos.
- **Agregar navegación en tiempo real:** Integrar seguimiento continuo de la posición del usuario mediante Bluetooth beacons o Wi-Fi positioning para actualizar la ruta dinámicamente.
- **Incluir características de accesibilidad:** Identificar rutas accesibles para personas con movilidad reducida, priorizando ascensores sobre escaleras y evitando obstáculos.
- **Expandir la cobertura:** Extender el sistema a otros edificios del campus universitario, creando una red de navegación completa.
- **Añadir información contextual:** Incluir horarios de clases, disponibilidad de salas, y puntos de interés como cafeterías, baños y servicios estudiantiles.
- **Implementar modo offline:** Permitir el uso de la aplicación sin conexión a internet, descargando todos los mapas y datos necesarios de forma anticipada.

- **Desarrollar sistema de retroalimentación:** Permitir a los usuarios reportar problemas, sugerir mejoras o actualizar información sobre nodos y conexiones.
- **Agregar nuevos nodos a los mapas:** Incluir nuevos puntos de interés como puertas de emergencia, zonas de descanso y servicios adicionales.
- **Optimizar la interfaz de usuario:** Mejorar la experiencia visual y de interacción, incorporando animaciones suaves, transiciones y un diseño más atractivo.
- **Agregar otros edificios del campus:** Extender la aplicación para cubrir otros edificios de la universidad, creando una red de navegación completa.

El sistema actual proporciona una base sólida sobre la cual se pueden construir estas mejoras futuras, manteniendo la arquitectura modular y escalable que facilita la evolución del proyecto.

4. Referencias

Referencias

- [1] L. Torvalds y D. Diamond, *Just for Fun: The Story of an Accidental Revolutionary*. HarperBusiness, 2001.
- [2] A. S. Tanenbaum y H. Bos, *Modern Operating Systems*. 4. ed., Pearson, 2015.
- [3] Google LLC, “Flutter - Build apps for any screen,” Flutter Documentation. [En línea]. Disponible en: <https://flutter.dev/>. [Accedido: 08-ene-2026].
- [4] Google LLC, “Dart programming language,” Dart Documentation. [En línea]. Disponible en: <https://dart.dev/>. [Accedido: 08-ene-2026].
- [5] LibreCAD Community, “LibreCAD - Open Source 2D-CAD,” LibreCAD Official Website. [En línea]. Disponible en: <https://librecad.org/>. [Accedido: 08-ene-2026].
- [6] Inkscape Project, “Inkscape - Draw Freely,” Inkscape Official Website. [En línea]. Disponible en: <https://inkscape.org/>. [Accedido: 08-ene-2026].
- [7] Microsoft Corporation, “Visual Studio Code - Code Editing. Redefined,” VS Code Documentation. [En línea]. Disponible en: <https://code.visualstudio.com/>. [Accedido: 08-ene-2026].
- [8] GitHub, Inc., “GitHub: Where the world builds software,” GitHub Platform. [En línea]. Disponible en: <https://github.com/>. [Accedido: 08-ene-2026].
- [9] Python Software Foundation, “Python Programming Language,” Python Official Website. [En línea]. Disponible en: <https://www.python.org/>. [Accedido: 08-ene-2026].
- [10] L. Vanden, “qrcode - Python QR Code image generator,” PyPI. [En línea]. Disponible en: <https://pypi.org/project/qrcode/>. [Accedido: 08-ene-2026].
- [11] A. Clark et al., “Pillow - Python Imaging Library,” Pillow Documentation. [En línea]. Disponible en: <https://pillow.readthedocs.io/>. [Accedido: 08-ene-2026].
- [12] P. E. Hart, N. J. Nilsson, y B. Raphael, “A Formal Basis for the Heuristic Determination of Minimum Cost Paths,” *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100-107, 1968.
- [13] Google LLC, “Material Design - Design System,” Material Design Guidelines. [En línea]. Disponible en: <https://m3.material.io/>. [Accedido: 08-ene-2026].
- [14] W3C, “Scalable Vector Graphics (SVG) 1.1 Specification,” World Wide Web Consortium. [En línea]. Disponible en: <https://www.w3.org/TR/SVG11/>. [Accedido: 08-ene-2026].

Diego Vidal

Estudiantes de Ingeniería Civil en Computación e Informática

Universidad de Magallanes

Fecha de entrega