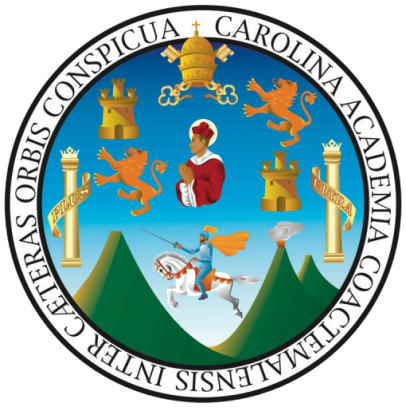
Universidad San Carlos de Guatemala
Facultad de Ingeniería
Ciencias y Sistemas
Inteligencia Artificial 1



MANUAL TÉCNICO
Proyecto 3

Pablo Medrano Velásquez	201222552
David Eduardo Garcia Garcia	201020136
Kevin Martin Samayoa Urizar	200915348

Índice

MazeBot

Una aplicación web tridimensional que simula un entorno de laberinto y un robot autónomo capaz de resolverlo utilizando algoritmos de búsqueda. El sistema permite la carga dinámica de laberintos mediante archivos JSON, facilitando la experimentación con diferentes configuraciones.

MazeBot está diseñado para fomentar la comprensión de conceptos de inteligencia artificial aplicados a entornos virtuales, integrando visualización 3D, simulación de comportamientos inteligentes y estrategias de resolución de problemas. La aplicación se ha desarrollado utilizando HTML, CSS, JS y Three.js y es accesible en cualquier navegador web moderno.

Arquitectura del Sistema

```
IA1_1S2025_P3_G19/
├── src/                  # Código fuente principal
│   ├── algorithms/       # Implementaciones de algoritmos
│   ├── css/              # Estilos de la aplicación
│   ├── js/               # Lógica principal
│   └── models/           # Modelos 3D
├── assets/               # Recursos estáticos
├── samples/              # Laberintos de ejemplo
└── manuales/             # Documentación técnica y de usuario
```

Componentes Principales

Frontend

- index.html: Página principal de la aplicación
- test_robot.html: Página de pruebas del robot
- testdos.html y testtres.html: Páginas de pruebas adicionales

Backend (JavaScript)

- main.js: Punto de entrada principal
- mazeLoader.js: Gestión de carga de laberintos
- renderer.js: Renderizado 3D
- robot.js: Lógica del robot
- config.js: Configuraciones globales

Algoritmos

- bfs.js: Búsqueda en Anchura
- dfs.js: Búsqueda en Profundidad
- astar.js: Búsqueda A*

Funcionalidades Principales

Visualización 3D

- Renderizado de laberintos en 3D usando Three.js
- Controles de cámara interactivos
- Visualización en tiempo real del movimiento del robot

Algoritmos de Búsqueda

1. Búsqueda en Anchura (BFS)
    - Explora el laberinto nivel por nivel
    - Garantiza el camino más corto
    - Complejidad: O(V + E)
2. Búsqueda en Profundidad (DFS)
    - Explora lo más lejos posible en cada rama
    - Eficiente en memoria
    - Complejidad: O(V + E)
3. Búsqueda A*
    - Utiliza heurísticas para optimizar la búsqueda
    - Combina BFS con búsqueda informada
    - Complejidad: O(E log V)

Gestión de Laberintos

Validación de Laberintos

- Carga de laberintos mediante archivos JSON
- Validación de datos
- Formato de archivo estandarizado

Interfaz de Usuario

Controles

- Cámara:
    - Rotación: Clic izquierdo + arrastrar
    - Desplazamiento: Clic derecho + arrastrar
    - Zoom: Rueda del ratón
    - Reinicio: Tecla 'R'

Elementos de la UI

- Selector de algoritmos
- Cargador de archivos
- Panel de estadísticas
- Controles de simulación

Formato de Datos

Estructura JSON del Laberinto

```
{
   "ancho": 10,    // Ancho del laberinto
   "alto": 10,     // Alto del laberinto
   "inicio": [0, 0], // Posición inicial [x, y]
   "fin": [9, 9],   // Posición final [x, y]
   "paredes": [     // Posiciones de las paredes
      [1, 0],
      [1, 1],
      [1, 2]
      // ... más posiciones de paredes
   ]
}
```

Manejo de Colisiones

El sistema de colisiones está implementado principalmente en la clase Robot y se maneja de la siguiente manera:

1. Detección de Colisiones:
    a. Se mantiene un conjunto (Set) de paredes del laberinto para verificación rápida
    b. Cada pared se almacena como una cadena en formato "x,y"
    c. La verificación se realiza en el método isValidMove

2. Validación de Movimientos:

```
isValidMove(fromPos, toPos) {
    // Verifica límites del laberinto
    if (toX < 0 || toX ⩾ this.maze.ancho || toY < 0 || toY ⩾ this.maze.alto) {
        return false;
    }

    // Verifica colisión con paredes
    if (this.walls.has(`${toX},${toY}`)) {
        return false;
    }

    // Verifica que el movimiento sea ortogonal y de un solo paso
    const dx = Math.abs(toX - fromX);
    const dy = Math.abs(toY - fromY);
    if (dx + dy ⩵ 1) {
        return false;
    }

    return true;
}
```

Renderizado del Robot

El robot se renderiza de dos maneras posibles:

1. Modelo 3D:
    - Se carga un modelo GLB (formato 3D)
    - Tiene animaciones para diferentes estados:
    - idle: Estado inactivo
    - walk: Caminando
    - run: Corriendo

    - tpose: Pose T (para debugging)

2. Cubo Simple (modo alternativo):
    - Se usa cuando useCube = true
    - Dimensiones: 0.6 x 0.5 x 0.6 (configurable en CONFIG)
    - Color: Púrpura (0x9B59B6)
3. Animaciones:
    - Velocidad de movimiento: 0.4 segundos por movimiento
    - Velocidad de rotación: π radianes por segundo
    - Duración de transición: 0.5 segundos

Renderizado del Escenario

El escenario se renderiza con las siguientes características:

```
- Luz ambiental: 0xffffff (intensidad 1.0)
  - Luz direccional principal: 0xffffff (intensidad 1.5)
  - Luz de relleno: 0x7ec0ee (intensidad 0.8)
  - Luz frontal: 0xffffff (intensidad 0.5)
  - Luz de rebote: Hemisferio (0xffffff a 0x444444)
```

Elementos del Laberinto

Piso:

- Color: Gris (0x808080)
- Material: MeshStandardMaterial con rugosidad 0.8

Paredes:

- Altura: 1.5 unidades
- Grosor: 0.1 unidades
- Color: Azul (0x4A90E2)
- Material: MeshPhongMaterial con brillo 30

Marcadores:

- Inicio: Verde (0x2ECC71)
- Fin: Rojo (0xE74C3C)
- Con efectos de brillo (glow)

Efectos Visuales:

- Niebla: Color blanco (0xFFFFFF), densidad 0.001
- Sombras suaves (PCFSoftShadowMap)
- Antialiasing activado

Colores por Algoritmo

Los colores se utilizan para visualizar el proceso de exploración y el camino final:

1. Exploración:
    - BFS: Azul (0x4A90E2)
    - DFS: Verde (0x2ECC71)
    - A*: Amarillo (0xF1C40F)*
2. Retroceso (backtracking):
    - Color: Rojo claro (0xFF6B6B)
    - Opacidad: 0.5
3. Camino Final:
    - Color: Amarillo (0xF1C40F)
    - Efecto de brillo (glow)
    - Opacidad: 0.7
    - Incluye números de paso para seguimiento
4. Marcadores de Exploración:
    - Tamaño: 80% del tamaño de la celda
    - Altura: 0.02 unidades
    - Opacidad: 0.5
    - Efecto de brillo adicional

El sistema utiliza una combinación de materiales y efectos para crear una visualización clara y atractiva del proceso de resolución del laberinto, con una clara distinción entre los diferentes algoritmos y estados del proceso de búsqueda.

Requisitos Técnicos

Requisitos del Sistema

- Navegador web moderno con soporte WebGL
- Python 3.x o servidor web estático
- Conexión a internet (para cargar Three.js)

Dependencias

- Three.js para renderizado 3D
- JavaScript ES6+
- HTML5 y CSS3

Solución de Problemas

Problemas Comunes

1. Pantalla en Blanco
    - Verificar soporte WebGL
    - Revisar consola del navegador
    - Comprobar carga de archivos
2. Problemas de Rendimiento
    - Reducir tamaño del laberinto
    - Ajustar configuraciones
    - Cerrar pestañas innecesarias

Mantenimiento y Desarrollo

Agregar Nuevos Algoritmos

1. Crear archivo en src/algorithms/
2. Implementar clase del algoritmo
3. Registrar en main.js
4. Agregar opción en UI

Modificar la UI

1. Editar src/css/styles.css
2. Actualizar src/js/config.js

3. Modificar HTML según necesidad

Licencia y Contribución

- Licencia MIT
- Contribuciones mediante pull requests
- Issues para reportar problemas

Algoritmos de Búsqueda

Cada algoritmo tiene sus propias ventajas y casos de uso:

BFS: Mejor para encontrar el camino más corto
DFS: Mejor para explorar espacios grandes con memoria limitada
A*: Mejor para encontrar el camino óptimo de manera eficiente

Búsqueda por Anchura o Breadth-First Search (BFS):

BFS es un algoritmo fundamental para la búsqueda en grafos y laberintos porque garantiza encontrar el camino más corto (óptimo) en laberintos no ponderados. Explora todos los nodos a una misma “profundidad” antes de avanzar, lo que lo hace ideal para comparar soluciones óptimas y analizar el rendimiento en laberintos simples.

Características Principales:

- Explora el laberinto nivel por nivel
- Garantiza el camino más corto
- Usa una cola (FIFO - First In, First Out)

Implementación

```
class BFS {
   constructor(maze) {
       this.visited = new Set();    // Celdas visitadas
       this.queue = [];            // Cola para BFS
       this.parent = new Map();    // Mapa de padres para reconstruir el camino
   }
```

Proceso de Búsqueda:

Inicialización:

- Comienza en la posición inicial
- Marca la posición inicial como visitada
- Añade la posición inicial a la cola

Exploración

```
while (this.queue.length > 0) {
      const current = this.queue.shift();  // Obtiene el primer elemento
      // Explora los vecinos
      for (const neighbor of neighbors) {
          if (!this.visited.has(neighborStr)) {
              this.visited.add(neighborStr);
              this.parent.set(neighborStr, current);
              this.queue.push(neighbor);
          }
      }
  }
```

Visualización:

- Color: Verde (0x4CAF50)
- Muestra cada celda explorada en tiempo real
- Delay de 100ms entre exploraciones

Búsqueda Por Profundidad o Depth-First Search (DFS):

DFS es sencillo de implementar y útil para explorar completamente un laberinto. Aunque no garantiza el camino más corto, es eficiente en memoria y puede ser más rápido en encontrar una solución en laberintos con muchos caminos posibles. DFS permite visualizar cómo un robot puede “perderse” y retroceder, lo que es didáctico para entender la exploración exhaustiva.

Características Principales:

- Explora lo más lejos posible en cada rama
- Eficiente en memoria
- Puede no encontrar el camino más corto

Implementación

```
class DFS {
   constructor(maze) {
       this.visited = new Set();     // Celdas visitadas
       this.parent = new Map();     // Mapa de padres
   }
```

Proceso de Búsqueda:

Exploración Recursiva:

```
  async dfs(current, end, renderer) {
      this.visited.add(currentStr);
      // Explora cada vecino recursivamente
      for (const neighbor of neighbors) {
          if (!this.visited.has(neighborStr)) {
              this.parent.set(neighborStr, current);
              if (await this.dfs(neighbor, end, renderer)) {
                  return true;
              }
          }
      }
  }
```

Backtracking:

- Muestra el retroceso con color rojo (0xD32F2F)
- Mueve el robot de vuelta a la posición anterior
- Delay de 100ms entre movimientos

Búsqueda A estrella o A* (A-Star):

A* Es uno de los algoritmos de búsqueda más potentes y populares en inteligencia artificial. Utiliza heurísticas para guiar la búsqueda hacia la meta, combinando lo mejor de BFS (óptimo) y la eficiencia de una búsqueda informada. Es ideal para mostrar cómo la IA puede resolver problemas de manera más inteligente y rápida, especialmente en laberintos grandes o complejos.

Características Principales:

- Algoritmo de búsqueda informada
- Combina BFS con heurísticas
- Encuentra el camino óptimo

Implementación:

```
class AStar {
   constructor(maze) {
       this.visited = new Set();
       this.parent = new Map();
       this.gScore = new Map();    // Costo desde el inicio
       this.fScore = new Map();    // Costo total estimado
   }
```

Sistema de Puntuación:

```
// gScore: costo real desde el inicio
  this.gScore.set(startStr, 0);
  // fScore: gScore + heurística
  this.fScore.set(startStr, this.heuristic(start, end));
```

Heurística

```
  heuristic(pos, end) {
      // Distancia Manhattan
      return Math.abs(pos[0] - end[0]) + Math.abs(pos[1] - end[1]);
  }
```

Exploración

```
  while (openSet.length > 0) {
      const current = this.getLowestFScore(openSet, end);
```

```
    // Explora vecinos y actualiza puntuaciones
    for (const neighbor of neighbors) {
        const tentativeGScore = this.gScore.get(currentStr) + 1;
        if (tentativeGScore < this.gScore.get(neighborStr)) {
            // Actualiza puntuaciones y añade a openSet
        }
    }
}
```

Visualización:

- Color: Púrpura (0x9C27B0)
- Muestra exploración y consideración de vecinos
- Delay de 100ms para exploración, 50ms para vecinos

Características Comunes Entre Algoritmos

Validación de Vecinos:

```
getValidNeighbors(position) {
    const directions = [
        [0, 1],  // derecha
        [1, 0],  // abajo
        [0, -1], // izquierda
        [-1, 0]  // arriba
    ];
    // Filtra vecinos válidos
}
```

Reconstrucción del Camino:

```
reconstructPath(start, end) {
    const path = [end];
    let current = end.toString();
    while (current !== startStr) {
```

```
            const parent = this.parent.get(current);
            path.unshift(parent);
            current = parent.toString();
        }
        return path;
    }
```

Estadísticas:

- Cuenta de exploración
- Tiempo de ejecución
- Tamaño de la cola/conjunto abierto
- Número de celdas visitadas