Universidad Tecnológica de la Mixteca
Instituto de Computación

Ingeniería en Computación

Inteligencia Artificial

Semestre 2021-B

Laberinto

Alumna:

Beatriz Bedolla Martínez

Profesor:

Mtro. Juan Juárez Fuentes

21 de Junio de 2021

Objetivo

Generar un laberinto con la ayuda de algoritmos de búsqueda, donde un agente tome un punto inicial y un punto final, e iniciar su recorrido usando estos metodos de busqueda de IA

Introducción

Una Búsqueda en profundidad (en inglés DFS o Depth First Search) es un algoritmo de búsqueda no informada utilizado para recorrer todos los nodos de un grafo o árbol (teoría de grafos) de manera ordenada, pero no uniforme. Su funcionamiento consiste en ir expandiendo todos y cada uno de los nodos que va localizando, de forma recurrente, en un camino concreto. Cuando ya no quedan más nodos que visitar en dicho camino, regresa (Backtracking), de modo que repite el mismo proceso con cada uno de los hermanos del nodo ya procesado.

Búsqueda en anchura (en inglés BFS - Breadth First Search) es un algoritmo de búsqueda no informada utilizado para recorrer o buscar elementos en un grafo (usado frecuentemente sobre árboles). Intuitivamente, se comienza en la raíz (eligiendo algún nodo como elemento raíz en el caso de un grafo) y se exploran todos los vecinos de este nodo. A continuación para cada uno de los vecinos se exploran sus respectivos vecinos adyacentes, y así hasta que se recorra todo el árbol.
Formalmente, BFS es un algoritmo de búsqueda sin información, que expande y examina todos los nodos de un árbol sistemáticamente para buscar una solución. El algoritmo no usa ninguna estrategia heurística.

Ambas técnicas constituyen métodos sistemáticos para visitar todos los vértices y arcos del grafo, exactamente una vez y en un orden específico predeterminado, por lo cual podríamos decir que estos algoritmos simplemente nos permiten hacer recorridos controlados dentro del grafo con algún propósito.

Siendo la búsqueda una de las operaciones más sencillas y elementales en cualquier estructura de datos, se han estandarizado el uso de estos algoritmos para ello, por lo que se conocen como algoritmos de búsqueda. Sin embargo, es importante resaltar que pueden utilizarse para muchísimas otras operaciones con grafos que no necesariamente incluyan la búsqueda de algún elemento dentro del grafo.

Una de esas muchas operaciones será lo que se describe a continuación en el cual se implementará un laberinto, generando el grafo con el algoritmo de búsqueda DFS y la solución de este con ambos algoritmos.

Desarrollo

Como primera instancia se procedió a encontrar un objetivo, y los requerimientos que el proyecto debe tener. En la siguiente imagen se muestra el laberinto objetivo.

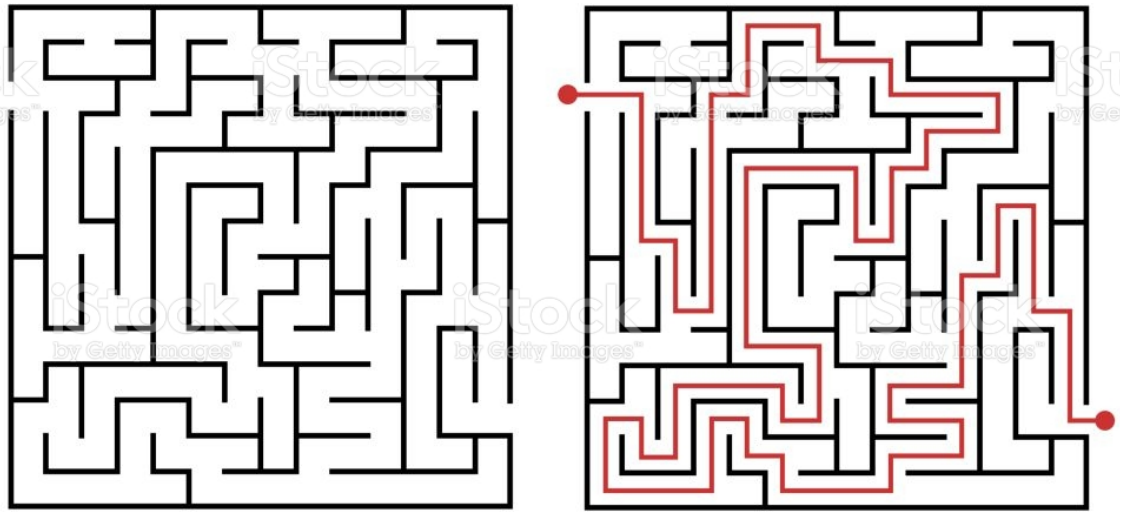


Imagen 1. Laberinto.

Página principal

En el archivo **index.html** se encuentran el diseño de la página principal.

Estructura

En el archivo **index.js** se definió el canvas, donde se especifica la estructura del laberinto, el anchura y altura del canvas, el código se muestra en la imagen 2.

```
import EventHandle from './event_handle.js';

document.addEventListener("DOMContentLoaded", () => {
  const canvas = document.getElementById("canvas");
  const ctx = canvas.getContext('2d');

  canvas.width = 500;
  canvas.height = 500;

  ctx.fillStyle = 'black';
  ctx.fillRect(0, 0, canvas.width, canvas.height);

  EventHandle(ctx, canvas);
});
```

Imagen 2. Estructura del laberinto.

De este archivo se llama al archivo **event_handle.js**, en este archivo se definen todas las órdenes que va obtener la página principal y los diferentes archivos a los que llamará para cumplir estas instrucciones. Aquí se hacen las llamadas para el generador del laberinto, y la solución con ambos algoritmos.

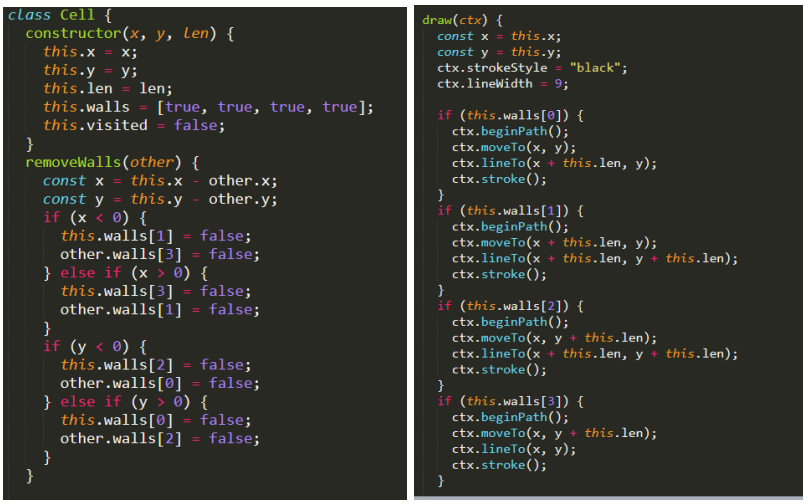
```
import GenerateDFS from './dfs.js';
import SolveDFS from './dfs_solve.js';
import Maze from './maze.js';
import SolveBFS from './bfs_solve.js'

const eventHandle = (ctx, canvas) => {
  let bastion = new Maze(canvas);
  let gen;
  let solve;

  function eventHelper() {
    ctx.clearRect(0, 0, canvas.width, canvas.height);
    $("button").prop("disabled", true);
    if (bastion) {
      bastion = new Maze(canvas);
    }
    bastion.solver = null;
    bastion.solved = false;
    bastion.solving = false;
    bastion.generating = true;
  }

  function fastAlgoHelper(bastionRunner) {
    let txt = "$("#instant-toggle-text").text()";
      if (txt === "ON") {
        bastionRunner.fast = true;
      }
  }

  function resetMaze(maze) {
    maze.cells.forEach( row => {
      row.forEach( cell => {
        cell.frontier = false;
        cell.explored = false;
        cell.path = false;
        cell.backtrack = false;
      });
    });
  }
```

```
function resetMaze(maze) {
  maze.cells.forEach( row => {
    row.forEach( cell => {
      cell.frontier = false;
      cell.explored = false;
      cell.path = false;
      cell.backtrack = false;
    });
  });
}

function solveHelper(maze, id) {
  $("button").prop("disabled", true);
  $(id).addClass("button-on");
  maze.solver = solve;
  maze.solving = true;
}

$("#dfs-gen").click(() => {
  eventHelper();
  gen = new GenerateDFS(bastion);
  bastion.generator = gen;
  fastAlgoHelper(bastion);
  $("#dfs-gen").addClass("button-on");
  bastion.begin();
});

$("#dfs-solve").click(() => {
  if (bastion.generator) {
    if (bastion.solved) {
      resetMaze(bastion);
    }
    solve = new SolveDFS(bastion);
    solveHelper(bastion, "#dfs-solve");
  }
});
```

Imagen 3. Archivo principal.

Generador

En el archivo cell.js declare funciones en las cuales se toma una celda, cada celda tiene con cuatro paredes, a partir de esa celda aleatoria se selecciona una celda vecina que aún no se ha visitado, el algoritmo quitará la pared entre las 2 celdas y marca la celda como visitada y así sucesivamente. Cuando llega a un callejón sin salida, retrocede por el camino hasta llegar a una celda con un vecino no visitado, continuando la generación del camino visitando esta nueva celda no visitada, hasta que llegue a la primera celda.



```
class Cell {
  constructor(x, y, len) {
    this.x = x;
    this.y = y;
    this.len = len;
    this.walls = [true, true, true, true];
    this.visited = false;
  }
  removeWalls(other) {
    const x = this.x - other.x;
    const y = this.y - other.y;
    if (x < 0) {
      this.walls[1] = false;
      other.walls[3] = false;
    } else if (x > 0) {
      this.walls[3] = false;
      other.walls[1] = false;
    }
    if (y < 0) {
      this.walls[2] = false;
      other.walls[0] = false;
    } else if (y > 0) {
      this.walls[0] = false;
      other.walls[2] = false;
    }
  }
```

```
draw(ctx) {
  const x = this.x;
  const y = this.y;
  ctx.strokeStyle = "black";
  ctx.lineWidth = 9;

  if (this.walls[0]) {
    ctx.beginPath();
    ctx.moveTo(x, y);
    ctx.lineTo(x + this.len, y);
    ctx.stroke();
  }
  if (this.walls[1]) {
    ctx.beginPath();
    ctx.moveTo(x + this.len, y);
    ctx.lineTo(x + this.len, y + this.len);
    ctx.stroke();
  }
  if (this.walls[2]) {
    ctx.beginPath();
    ctx.moveTo(x, y + this.len);
    ctx.lineTo(x + this.len, y + this.len);
    ctx.stroke();
  }
  if (this.walls[3]) {
    ctx.beginPath();
    ctx.moveTo(x, y + this.len);
    ctx.lineTo(x, y);
    ctx.stroke();
  }
```

Imagen 4. Dibuja celdas.

Esta imagen muestra el laberinto creado y la solución de este. La estructura del proyecto se centra en tener un objeto "Maze" que incluye un objeto "Maze Generator" y un objeto "Maze Solver", con controladores de eventos para asignar los diferentes algoritmos que se utilizarán y mostrarán en el lienzo. En la imagen 5 se muestra el código del algoritmo DFS que genera el laberinto, y en la imagen 6 se muestra el laberinto generado que se encuentran el archivo generator.js que importa del archivo cell.js.

```
import Generator from './generator.js';

class GenerateDFS extends Generator {
  constructor(maze) {
    super(maze);
    const y = Math.floor(Math.random() * maze.cells.length);
    const x = Math.floor(Math.random() * maze.cells[0].length);
    this.current = maze.cells[y][x];
    this.current.visited = true;
    this.stack = [];
    this.start = this.current;
  }

  adjacentCells(cell) {
    const maze = this.maze;
    const inBounds = () => {
        if (maze.x < 0 || maze.x > maze.w || maze.y < 0 || maze.y > maze.h) {
          return false;
        } else {
          return true;
        }
      };
    let neighbors = [];
    let x;
    let y;
    const add = [[-maze.len, 0],[maze.len, 0], [0, maze.len], [0, -maze.len]];

    for (let i=0; i < add.length; i++) {
      x = cell.x + add[i][0];
      y = cell.y + add[i][1];
      let neighborCell = this.findCell(x, y);
      if (inBounds() && neighborCell && !neighborCell.visited) {
        neighbors.push(neighborCell);
      }
    }
    if (neighbors.length > 0) {
      return neighbors[Math.floor(Math.random() * neighbors.length)];
    }
  }
```

```
  draw(ctx) {
    const maze = this.maze;
    if (maze.fast === true) {
      this.fastAlgo();
    } else {
      this.slowAlgo();
    }
    maze.cells.forEach( row => {
      row.forEach( cell => {
        cell.draw(ctx);
      });
    });
    if (this.current) {
      if (maze.generating) {
        this.current.highlight(ctx);
      }
    }
    ctx.strokeRect(0, 0, maze.w, maze.h);
  }
}
```

Imagen 5. Generador de laberinto.

La funcionalidad principal de DFS se basa en la pila para garantizar que se hayan explorado las "ramas" más lejanas del laberinto antes de volver a las intersecciones anteriores.

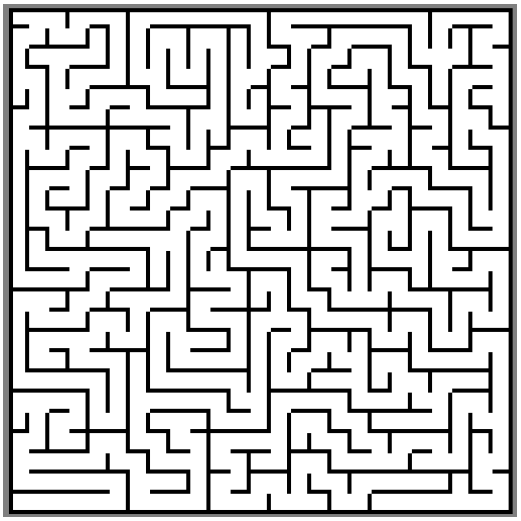


Imagen 6. Laberinto generado

Resuelve DFS

El algoritmo genera un laberinto aleatorio. Para resolver el laberinto se usó el algoritmo DFS y BFS. DFS atraviesa una ruta en particular hasta que no tiene más rutas para explorar. Si el buscador de caminos llega a una intersección con múltiples

caminos, seleccionará un camino al azar. En ese punto, retrocederá a una intersección anterior del laberinto y bajará por una ruta diferente.

```
import Solver from './sol.js';
class SolveDFS extends Solver {
  constructor(maze) {
    super(maze);

    this.start.parent = null;
    this.current = this.start;
    this.stack = [];
  }

  DFSsolver() {
    if (this.current !== this.finish) {
      const neighbors = this.adjacentCells(this.current);
      if (neighbors) {
        const next = neighbors[Math.floor(Math.random() * neighbors.length)];
        next.explored = true;
        this.stack.push(this.current);
        this.current = next;
      } else if (this.stack.length > 0) {
        this.current.backtrack = true;
        this.current = this.stack.pop();
      }
    }
  }

  path() {
    if (this.stack.length > 0) {
      this.stack.pop().path = true;
    } else {
      this.maze.solving = false;
      this.maze.solved = true;
    }
  }
}
```

Imagen 7. Resuelve DFS.

Resuelve BFS

BFS visita todas las celdas adyacentes de la celda actual antes de hacer que las celdas adyacentes subsiguientes sean cada una la siguiente celda actual en repetir el proceso.

```
import Solver from './sol.js';

class SolveBFS extends Solver {
  constructor(maze) {
    super(maze);

    this.current = this.start;
    this.queue = [];
  }

  algorithm() {
    if (this.current !== this.finish) {
      const neighbors = this.adjacentCells(this.current);
      if (neighbors) {
        neighbors.forEach(neighbor => {
          neighbor.parent = this.current;
          this.queue.push(neighbor);
        });
      }
      let next = this.queue.shift();
      next.explored = true;
      this.current = next;
    }
  }
}

export default SolveBFS;
```

Imagen 8. Resolver BFS.

A continuación se muestra el juego en ejecución y le dan clic en “Generar”, automáticamente generará el laberinto como se muestra en la imagen 9.

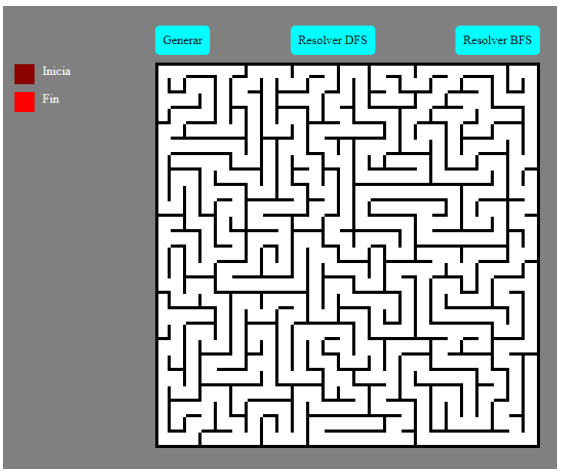


Imagen 9.Juego Laberinto.

Después de generar el laberinto se indicara con qué algoritmo se desea resolver si es con DFS dar clic en “Resolver DFS”, el color azul cielo indica cuando el algoritmo pasa por las diferentes celdas y azul marino cuando estas ya fueron visitadas y regresa a la inicial.

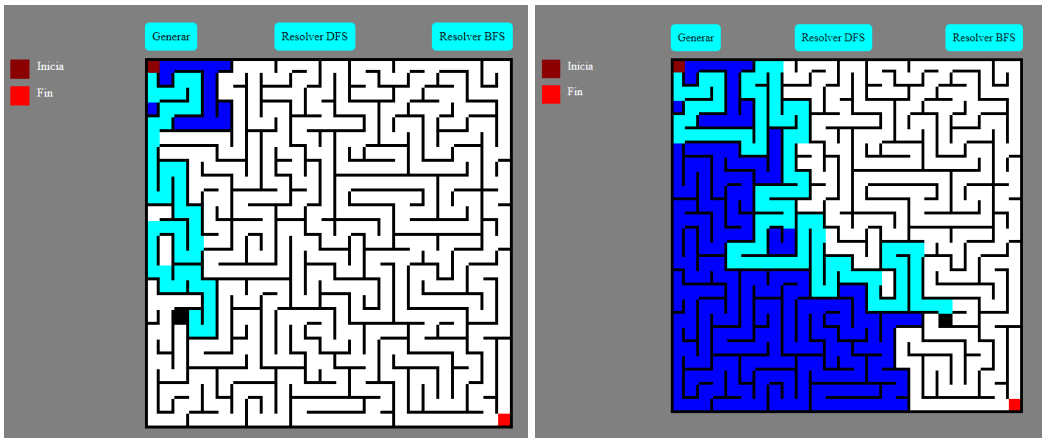


Imagen 10. Laberinto DFS.

Al encontrar el camino de punto inicial al final, el camino es de color gris como se muestra en la imagen 11.

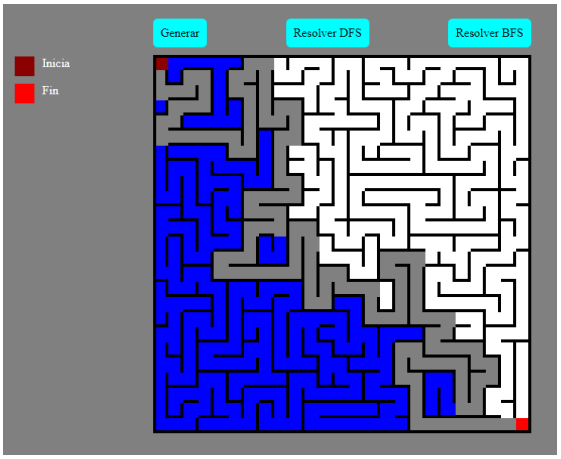


Imagen 11. Laberinto resuelto con DFS.

En la siguiente imagen se muestra el mismo laberinto pero resuelto con el algoritmo BFS al dar clic en “Resolver BFS”. Si se desea generar otro laberinto aleatorio se debe volver a dar clic en “Generar”. De igual manera el color azul cielo marca el camino recorrido y el camino gris es la solución del laberinto.

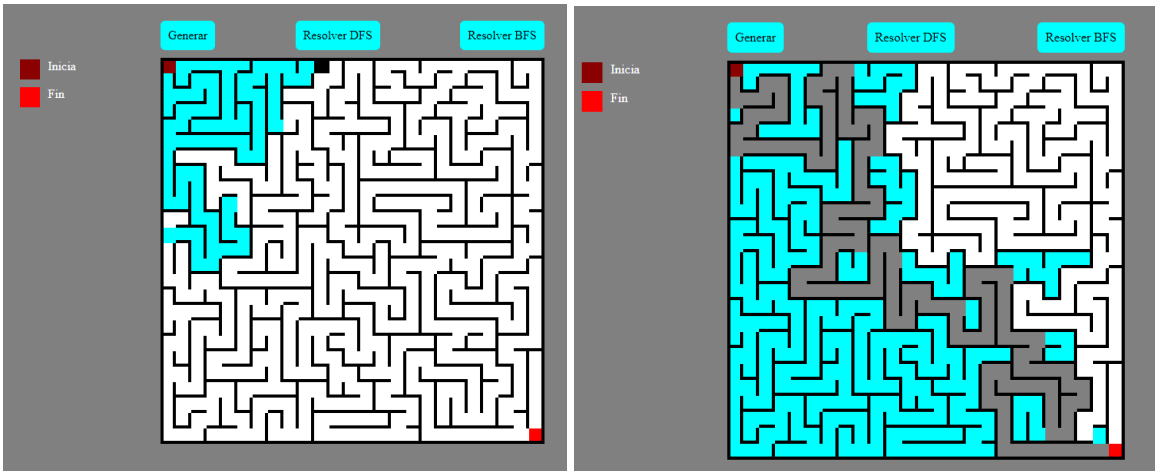


Imagen 12. Laberinto resuelto con BFS.

Conclusión

Como hemos visto los algoritmos son útiles para solucionar problemas comunes y con la ayuda de estos solucionarlo de manera rápida como lo es en el juego del laberinto. Al realizar este proyecto se cumplio el objetivo presentado, además de que se reanudó el aprendizaje acerca de los diferentes algoritmos de búsqueda de la IA, esto ayudó a la comprensión e implementación de la codificación, obteniendo como resultado un buen juego de laberinto.

Referencias

https://es.wikipedia.org/wiki/Búsqueda_en_anchura
https://es.wikipedia.org/wiki/Búsqueda_en_profundidad