UNIVERSIDAD CATÓLICA SANTO TORIBIO DE MOGROVEJO

FACULTAD DE INGENIERÍA

ESCUELA DE SISTEMAS Y COMPUTACIÓN

SEMESTRE ACADÉMICO 2024-II

ALUMNOS:

Perez Davila, Junior

Paucar Mejia, Fabiana

Cortez Villaseca, Franco

DOCENTE: LUJAN SEGURA, EDWAR GLORIMER

TEMA: Trabajo Grupal – Juego de Mario Bros con A*

CHICLAYO, LAMBAYEQUE, PERÚ - 2024

ÍNDICE

1. Descripción del Juego

- **Nombre del juego:**
  Mario Bros - Algoritmo A* para Búsqueda de Rutas

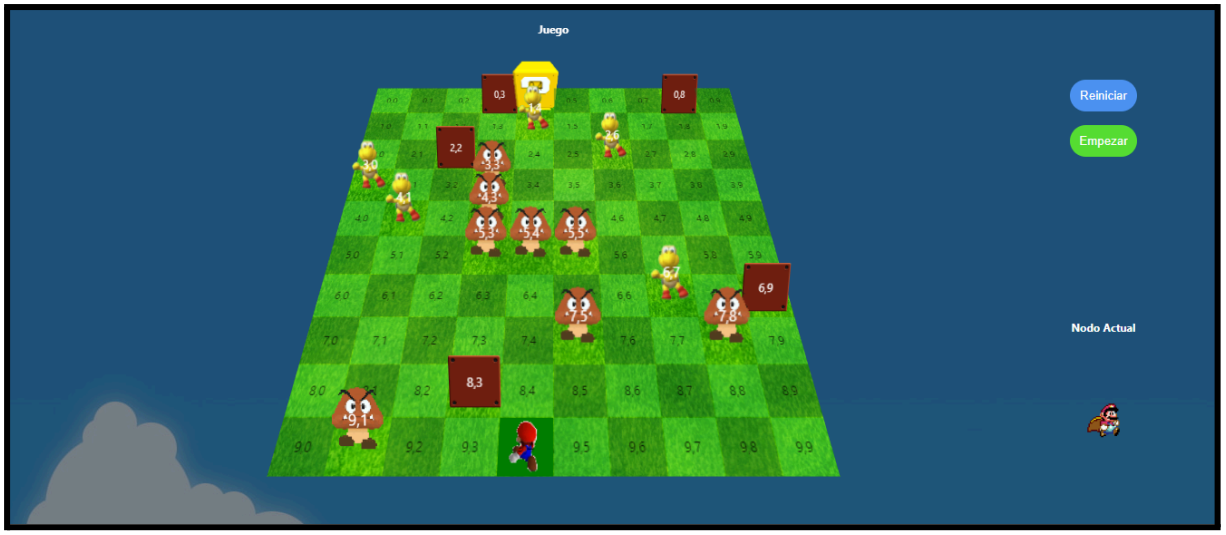
- **Implementación de código y algoritmo**
  Implementado en HTML, CSS y JavaScript se basa en el uso del algoritmo A* (A estrella) para que el personaje de Mario Bros encuentre el camino óptimo entre obstáculos y llegue a su meta. Demuestra cómo el algoritmo toma decisiones para guiar al personaje a través de su entorno.

- **Contenido del Juego**
  ➔ **HTML (index.html):** Contiene la estructura principal del juego, incluyendo la disposición del área de juego y los elementos visuales.
  ➔ **CSS (style.css):** Define el estilo y la presentación del juego, utilizando imágenes y gráficos que representan el mundo de Mario Bros. También se encarga de la disposición de los obstáculos y el diseño del tablero de juego.
  ➔ **JavaScript (scriptAestrellaMario.js):** Implementa el algoritmo A* para calcular el camino más corto desde la posición inicial de Mario hasta el objetivo, evitando obstáculos colocados en el tablero. El script se encarga de la lógica del juego y la interacción entre los elementos visuales.

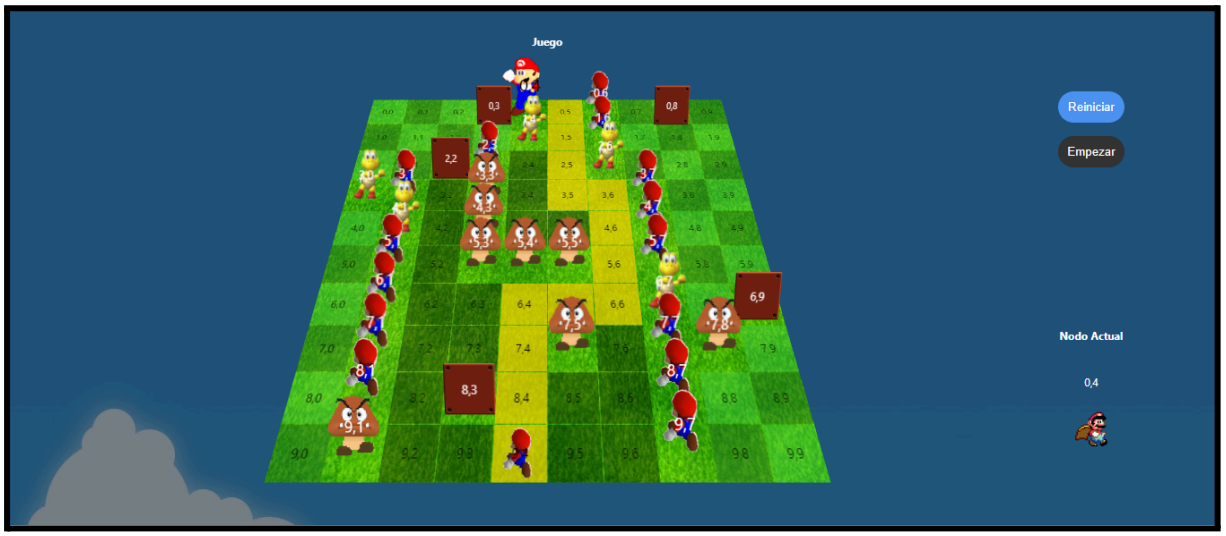
- **Recursos:** Contiene imágenes, audios, etc. Necesarios para representar a Mario y otros elementos visuales en el juego como los obstáculos.

- **Objetivo del Juego**
  El objetivo del juego es que Mario logre llegar a la meta evitando los obstáculos distribuidos en el entorno. Utilizando el algoritmo A*, el sistema calcula el camino más eficiente y lo representa visualmente mientras Mario avanza por el escenario.

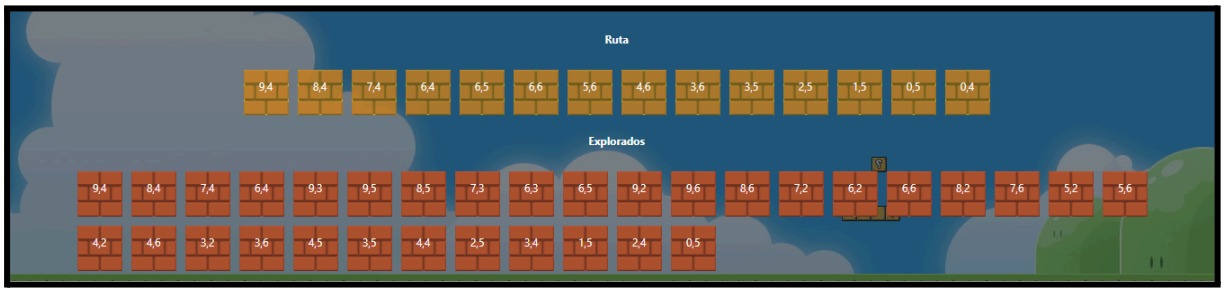
- **Funcionalidades Clave**
  - Implementación del algoritmo A* para la búsqueda de rutas.
  - Interfaz gráfica sencilla con representación visual de obstáculos y caminos.
  - Interfaz para la representación de la frontera de los nodos, además de la lista de nodos actual, explorados, señalización del nodo objetivo y representación de la ruta.
  - Uso de gráficos clásicos, música y GIFs del personaje de Mario Bros para una experiencia amigable.



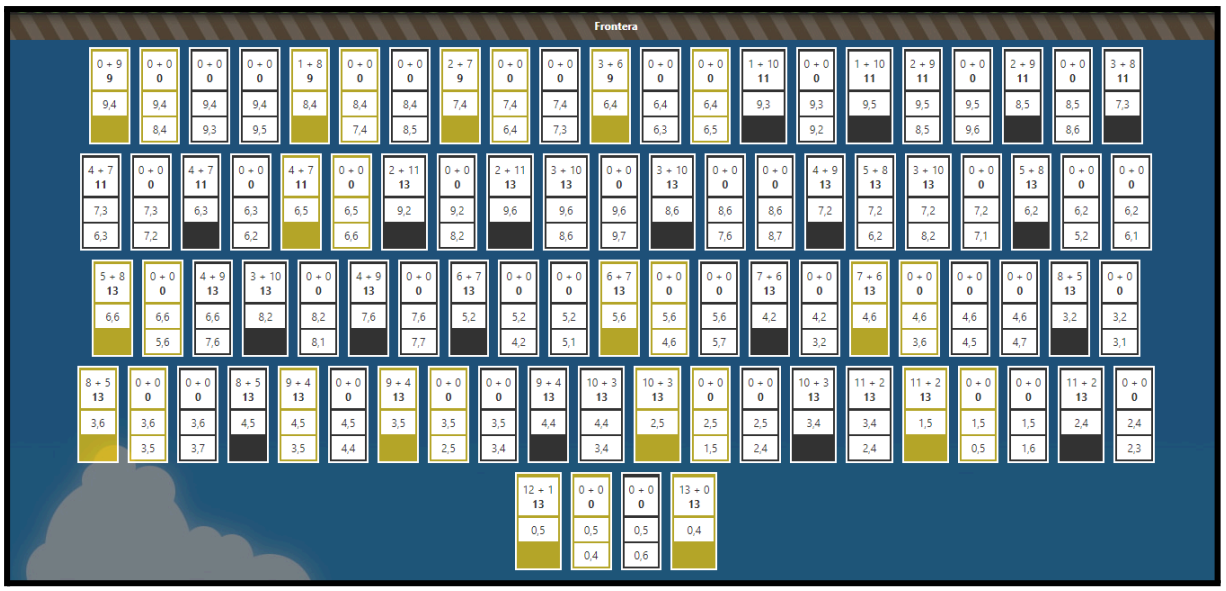
Juego al inicio antes de ejecutar algoritmo



Juego habiendo sido recorrido con el algoritmo A, señalando la ruta final*



Ruta final encontrada con los nodos que usó para llegar al objetivo



La frontera de los nodos explorados, donde se señala de aquellos relevantes para encontrar la ruta,

2. Algoritmo de Búsqueda A*

El algoritmo A* es una técnica de búsqueda de caminos que combina las características de la búsqueda de costo uniforme y la búsqueda heurística, utilizando una función de costo que incluye dos componentes: g(n), que es el costo acumulado desde el punto de inicio hasta el nodo actual, y h(n), que es una estimación heurística del costo restante desde el nodo actual hasta el destino. El objetivo de A* es encontrar el camino más corto minimizando el valor total de estas dos funciones.

Funcionamiento del Algoritmo A*

A* comienza explorando los nodos que están más cercanos al punto de inicio y, simultáneamente, selecciona los nodos que parecen estar más cerca del destino basándose en la heurística (generalmente la distancia de Manhattan en un tablero cuadriculado como el de tu juego). A* es eficiente porque evita explorar caminos que son claramente subóptimos, concentrándose en las rutas más prometedoras.

La función f(n)f(n)f(n) le da al algoritmo A* la capacidad de encontrar el camino más corto de forma eficiente, porque:

- **g(n):** Evalúa el costo real dado por el problema.
- **h(n):** Representa la heurística, predice cuán lejos está la meta, ayudando a priorizar los nodos más prometedores.

En el Juego

1. **Búsqueda del Camino Más Corto:**

   Por ejemplo, si Mario está ubicado en la esquina inferior izquierda del tablero y la meta está en la esquina superior derecha, el algoritmo A* calcula todos los posibles caminos, pero prioriza aquellos que dirigen a Mario hacia la meta de forma más directa, evitando retroceder innecesariamente.

2. **Obstáculos en el Camino:**

   Por ejemplo, si hay un muro de bloques que separa a Mario de la meta, A* intentará encontrar la ruta más cercana alrededor del muro. Si existen dos rutas, una larga y otra corta, A* seleccionará la ruta más corta aunque implique rodear el obstáculo de manera más compleja.

3. **Áreas de Camino Libre:**

   Por ejemplo, si el área entre Mario y la meta es mayormente libre de obstáculos, A* calculará el camino más directo. Sin embargo, si aparecen obstáculos pequeños, como bloques dispersos, A* identificará la ruta menos costosa que rodee estos obstáculos en lugar de evitarlos completamente con un rodeo mayor.

4. **Distancia Estimada (Heurística):**

   Por ejemplo, si Mario está muy cerca de la meta pero existen varios bloques entre ambos, A* utilizará la distancia restante estimada para decidir si es más eficiente saltar algunos bloques o

rodearlos. Esta estimación permite al algoritmo mantener un equilibrio entre explorar nuevas rutas y avanzar hacia la meta.

5. **Nodos Explorados:**

   Por ejemplo, si Mario ya ha explorado un camino que no conduce eficientemente a la meta, A* marcará ese camino como menos probable para futuras exploraciones y buscará una ruta alternativa que tenga un costo menor, evitando que Mario recorra caminos que ya ha descartado.

3. Funcionamiento del Programa

El programa que se encarga del funcionamiento del juego empleando el algoritmo A*, es el script en JS, ya que se encarga de hacer el recorrido en el tablero y mostrar la información de los nodos que empleó para llegar al objetivo.

a. Definición de Variables y Constantes

Se define primero las constantes a utilizar durante todo el programa, como a cantidad de columnas y filas del tablero junto con el tamaño de cada celda, el tiempo de ejecución para que demore en marcar las celdas y poder apreciar el proceso; el nivel dificultad que hace referencia a la cantidad de obstáculos que aparecerán aleatoriamente, además de los posibles tipos de obstáculos a encontrar, definir el estado inicial y el estado final, y por último, el audio a utilizar en ciertos momentos del juego.

```
const cols = 10;
const rows = 10;
const sizeCell = 50;
const grid = [];
const timeExecute = 75;
const nivelDificultad = 20;
const enemies = ["muro", "hongo", "tortuga"];
let start, goal;
const sonido = document.querySelector("#sonido");
```

b. Creación de Tablero

Se establece una función para obtener un número aleatorio entero que se usará para colocar obstáculos que sirven como muros al momento de la creación del tablero. Luego, se obtiene el elemento grid desde el HTML, y se definen las dimensiones de la cuadrícula utilizando las variables cols (columnas) y rows (filas). Las dimensiones se ajustan dinámicamente con CSS Grid. Se inicia un ciclo anidado que recorre filas (rows) y columnas (cols). Para cada celda, se crea un elemento div, que representa una celda en la cuadrícula visual, y se almacena en un arreglo bidimensional grid[i][j].

Cuando se genera, cada celda se inicializa con sus coordenadas (x e y), valores de costo f, g y heurística h, y una propiedad booleana isWall que indica si es un obstáculo. También se guarda una referencia al elemento visual (cellElement).

El nodo de inicio se define en la posición inferior central de la cuadrícula, y se le asigna una clase start y un ID para el personaje (Mario). La meta (o el objetivo) se define en la posición superior central, con la clase goal. Se define un patrón de colores para las celdas de la cuadrícula, alternando entre dos clases cell_1 y cell_2 para crear un aspecto de tablero de ajedrez, lo que ayuda a la diferenciación visual entre las celdas. Las celdas que no son ni la de inicio ni la de meta pueden convertirse en obstáculos o muros aleatoriamente, según el nivel de dificultad. Si una celda es seleccionada como muro, también puede tener un enemigo asignado de forma aleatoria, utilizando la función getRandomInt.

Al final de la función, se carga y reproduce una pista de música para el juego.

```
function getRandomInt(max) {
    return Math.floor(Math.random() * max);
}

function createGrid() {
    const gridElement = document.getElementById('grid');
    gridElement.style.gridTemplateColumns = `repeat(${cols},${sizeCell}px)`;
    gridElement.style.gridTemplateRows = `repeat(${rows},${sizeCell}px)`;

    for (let i = 0; i < rows; i++) {
        grid[i] = [];
        for (let j = 0; j < cols; j++) {
            const cellElement = document.createElement('div');
            cellElement.classList.add('cell');
            gridElement.appendChild(cellElement);

            const cell = {
                x: i,
                y: j,
                f: 0,
                g: 0, // Distancia acumulada desde el inicio
                h: 0, // Heurística (distancia Manhattan hasta el objetivo)
                isWall: false,
                parent: null,
                element: cellElement
            };
            grid[i][j] = cell;
            cellElement.innerHTML = `<span>${i},${j}</span>`;

            if (i === parseInt((rows - 1)) && j === parseInt((cols - 1)/2)) {
                start = cell;
                cellElement.classList.add('start');
                cellElement.setAttribute("id","mario_inicial")
            } else if (i === 0 && j === parseInt((cols - 1)/2)) {
                goal = cell;
                cellElement.classList.add('goal');
            }

            if ((i % 2 == 0 && j % 2 != 0 ) || (j % 2 == 0 && i % 2 != 0 )) {
                cellElement.classList.add("cell_1");
            } else if ((j % 2 != 0 && i % 2 != 0 ) || (j % 2 == 0 && i % 2 ==
0 )) {
                cellElement.classList.add("cell_2");
            }

            if (Math.random() < nivelDificultad / 100 && cell !== start &&
cell !== goal) {
                cell.isWall = true;
                cellElement.classList.add('wall');
```

```
                var indiceAleatorio = getRandomInt(enemies.length);
                cellElement.classList.add(enemies[indiceAleatorio]);
            }
        }
    }
    sonido.innerHTML = '<source src="aud/yt1s.com - Super Mario Bros
Soundtrack.mp3" type="audio/mpeg">';
    sonido.load();
    sonido.play();
}
```

c. Información de recorrido

Esta parte del código se enfoca en la visualización y actualización de elementos relacionados con la búsqueda en el tablero, principalmente cómo se gestionan y muestran los nodos explorados, la frontera, el nodo actual y la reconstrucción de la ruta final.

Se definen las variables globales de ruta y frontera seleccionan los elementos HTML donde se mostrarán visualmente la ruta y la frontera de nodos, utilizando los identificadores #Ruta y #Frontera. Servirán como contenedores para la información que se irá actualizando durante la ejecución del algoritmo.

La función de la Frontera agrega el nodo padre y sus vecinos a la visualización de la frontera.

- **Nodo padre**: Se crea una columna con información sobre el nodo actual (parent). Si el nodo es el de inicio, se muestran guiones en lugar de valores numéricos.
- **Vecinos**: Para cada vecino del nodo actual, se crea otra columna con la información de costos (g, h, f), el nodo padre y las coordenadas del vecino.

Cada vecino se muestra en una nueva columna, lo que permite seguir el árbol de búsqueda mientras el algoritmo explora diferentes nodos.

La función de nodo actual actualiza la visualización del nodo actual en el HTML. Muestra las coordenadas del nodo en el que el algoritmo se encuentra en un momento dado. Mientras que la función de Nodos Explorados actualiza el listado de nodos que ya han sido explorados y cerrados y no serán revisitados por el algoritmo.

La función de Reconstrucción de camino, ademas de resaltar de amarillo los nodo que son parte de la ruta final, es la encargada de reconstruir la ruta final desde el nodo objetivo hasta el nodo de inicio en el campo de Ruta, donde añade todos los todos que formaron parte de esta. Además, después de reconstruir el camino, se resaltan las columnas correspondientes a los nodos que forman parte de la ruta final en la tabla de la frontera. Si se logra mostrar la ruta, es porque Mario llegó al final y se reproduce la música de fondo y una animación de baile, simbolizando que ha logrado llegar al objetivo y el algoritmo cumplió su trabajo.

Todas estas funciones utilizan setTimeout para crear un pequeño retraso entre la actualización de cada nodo, lo que permite una animación gradual del recorrido en la cuadrícula para apreciar el proceso detalladamente.

```
const ruta = document.querySelector('#Ruta');
const arbol = document.querySelector('#Frontera');

function addToFrontera(parent, neighbors) {
    const fronteraContainer = document.getElementById('Frontera');

    // Crear la columna para el nodo padre, si es el inicio se muestra con "-"
    const parentColumn = document.createElement('div');
    parentColumn.classList.add('table-column');

    const parentCostCell = document.createElement('div');
    parentCostCell.classList.add('table-cell');
    parentCostCell.innerHTML = parent ? `<span> ${parent.g} +
${parent.h}</span><b>${parent.f}</b>` : "g: -, h: -, f: -";

    const parentNameCell = document.createElement('div');
    parentNameCell.classList.add('table-cell');
    parentNameCell.innerHTML = parent ? `${parent.x},${parent.y}` : "-";

    parentColumn.appendChild(parentCostCell);
    parentColumn.appendChild(parentNameCell);
    fronteraContainer.appendChild(parentColumn);

    neighbors.forEach(neighbor => {
        const neighborColumn = document.createElement('div');
        neighborColumn.classList.add('table-column');

        const costCell = document.createElement('div');
        costCell.classList.add('table-cell');
        costCell.innerHTML = `
            <span> ${neighbor.g} + ${neighbor.h} </span><b>${neighbor.f}</b>
        `;

        const parentNodeCell = document.createElement('div');
        parentNodeCell.classList.add('table-cell');
        parentNodeCell.innerText = `${parent.x},${parent.y}`;

        const neighborCell = document.createElement('div');
        neighborCell.classList.add('table-cell');
        neighborCell.innerText = `${neighbor.x},${neighbor.y}`;

        neighborColumn.appendChild(costCell);
        neighborColumn.appendChild(parentNodeCell);
        neighborColumn.appendChild(neighborCell);

        fronteraContainer.appendChild(neighborColumn);
    });
}
```

```
function updateNodoActual(current) {
    const nodoElement = document.getElementById('Nodo');
    nodoElement.innerHTML = `<p>${current.x},${current.y}</p>`;
}

function updateExplorados(closedSet) {
    const exploradosElement = document.getElementById('Explorados');
    exploradosElement.innerHTML = '';

    closedSet.forEach(node => {
        const nodeElement = document.createElement('p');
        nodeElement.innerText = `${node.x},${node.y}`;
        exploradosElement.appendChild(nodeElement);
    });
}

function reconstructPath(current) {
    let delay = 0;
    const fronteraContainer = document.getElementById('Frontera').children;

    sonido.innerHTML = '<source src="gangnam style.mp3" type="audio/mpeg">';
    sonido.load();
    sonido.play();

    const caminoCoords = [];

    ruta.innerHTML = '';
    while (current) {
        caminoCoords.push(`${current.x},${current.y}`);
        const node = current;

        setTimeout(() => {
            if (node && node.element) {
                node.element.classList.add('path');
                ruta.innerHTML += `<p>${node.x},${node.y}</p>`;
            }
        }, delay * 20);

        current = current.parent;
        delay++;
    }

    setTimeout(() => {
        for (let column of fronteraContainer) {
            const coords =
column.querySelector('.table-cell:last-child').innerText;
            if (caminoCoords.includes(coords)) {
                column.classList.add('final-path');
            }
        }
    }, delay * 20);
```

```
}
```

d. Implementación de Algoritmo A*

En esta parte se implementa el algoritmo de búsqueda A* de forma asíncrona para encontrar el camino más corto entre un nodo de inicio (start) y un objetivo (goal) en una cuadrícula (grid). Incluye funciones de soporte para obtener los vecinos de un nodo, calcular la heurística, y animación del proceso paso a paso.

La función para obtener nodos vecinos, devuelve los vecinos del nodo que está explorando, siempre y cuando no se salgan de los límites de la cuadrícula. Los vecinos se obtienen basándose en las coordenadas (x, y) del nodo actual.

Se usa también una función para calcular la distancia heurística entre un nodo y el objetivo utilizando la distancia Manhattan, donde simplemente suma las diferencias absolutas entre las coordenadas X e Y del nodo actual y el objetivo, proporcionando una estimación del costo restante.

Posteriormente, está el algoritmo A*, el cual utiliza una función delay para mostrar el proceso más lentamente y apreciar el recorrido que hace en busca del camino más corto desde start a goal en el tablero. Dentro de esta se utilizan un openSet (para listar de nodos que aún están por explorar) y un closedSet (para listar de nodos ya explorados). Inicializa los valores de g, h, y f del nodo inicial y va explorando sus nodos hijos (o vecinos), convirtiéndolos en nodos padre y después expandiendo sus hijos, sin tomar los nodo padre ya tomados anteriormente. La manera en la que escoge qué nodo hijo expandir es gracias al valor f, que es la suma del costo y la heurística y solo expande quien posea el valor f menor de todos los hijos. Al mismo tiempo que esto pasa, se van mostrando los nodos explorados, el nodo actual y la frontera con los valores tomados. Cuando por fin llega al objetivo, se ejecutan las funciones de Ruta, detalladas anteriormente; caso contrario, si no se ha encontrado el objetivo, se muestra un mensaje en el campo de Ruta y se reproduce una animación acompañada de un sonido que indica el fracaso.

```
function getNeighbors(node) {
    const neighbors = [];
    const { x, y } = node;

    if (x > 0) neighbors.push(grid[x - 1][y]); // Arriba
    if (x < rows - 1) neighbors.push(grid[x + 1][y]); // Abajo
    if (y > 0) neighbors.push(grid[x][y - 1]); // Izquierda
    if (y < cols - 1) neighbors.push(grid[x][y + 1]); // Derecha

    return neighbors;
}

function heuristic(node, goal) {
    return Math.abs(node.x - goal.x) + Math.abs(node.y - goal.y); // Distancia
Manhattan
}
```

```
function delay(ms) {
    return new Promise(resolve => setTimeout(resolve, ms));
}

async function aStarSearch(start, goal) {
    let openSet = [start];
    let closedSet = [];

    start.g = 0;
    start.h = heuristic(start, goal);
    start.f = start.g + start.h;

    while (openSet.length > 0) {
        let current = openSet.reduce((prev, node) => node.f < prev.f ? node :
prev);

        updateNodoActual(current);
        addToFrontera(current, getNeighbors(current).filter(neighbor =>
!closedSet.includes(neighbor) && !neighbor.isWall));
        await delay(timeExecute);

        if (current === goal) {
            reconstructPath(current);
            return;
        }

        openSet = openSet.filter(node => node !== current);
        closedSet.push(current);
        current.element.classList.remove('open');
        current.element.classList.add('closed');
        updateExplorados(closedSet);

        let neighbors = getNeighbors(current).filter(neighbor =>
!closedSet.includes(neighbor) && !neighbor.isWall);
        for (const neighbor of neighbors) {
            let tentative_g = current.g + 1;

            if (!openSet.includes(neighbor)) {
                openSet.push(neighbor);
                neighbor.element.classList.add('open');
                await delay(timeExecute);
            } else if (tentative_g >= neighbor.g) {
                continue;
            }

            neighbor.g = tentative_g;
            neighbor.h = heuristic(neighbor, goal);
            neighbor.f = neighbor.g + neighbor.h;
            neighbor.parent = current;
            await delay(timeExecute);
```

```
        }
    }

    ruta.innerHTML = '<span>No se encontró el objetivo</span>';
    sonido.innerHTML = '<source src="loose.mp3" type="audio/mpeg">';
    sonido.load();
    sonido.play();
    document.getElementById("mario_inicial").classList.add('sad_mario');
}
```

e. Preparación de Juego

Aquí se inician las principales funciones apenas se entra al archivo html, generando el tablero del juego de Mario y preparando el funcionamiento del Algoritmo A* en el botón de “Iniciar” par que comience el recorrido, así como el botón “Reiniciar” para recargar la página cuando el usuario guste.

```
createGrid();

const btnEmpezar = document.querySelector('#iniciar');
btnEmpezar.addEventListener('click', () => {
    aStarSearch(start, goal);
    btnEmpezar.disabled = true;
});

document.querySelector('#reiniciar').addEventListener('click', () => {
    location.reload();
});
```

4. Conclusión

- A* Es adaptable a distintas situaciones, y al utilizarlo en un juego, exploramos cómo se puede modificar para distintos tipos de entornos y niveles, desde mapas más simples hasta escenarios más complejos y dinámicos, logrando observar y analizar cómo actuaría el algoritmo en distintas situaciones.
- El algoritmo A* Es muy eficiente pues toma g(n) y h(n) lo que garantiza una respuesta cercana a la óptima o la óptima en un tiempo razonable a diferencia de otros algoritmos ya sean informados o no informados.
- De la mano con la eficiencia mencionada anteriormente, puede darse un caso en el que hayan demasiados obstáculos lo cual genera que el algoritmo demore un poco más de tiempo en dar una respuesta lo cual nos invita a mejorar aún más el algoritmo o usar otro tipo de heurística o incluso una metaheurística.
- El algoritmo A* brinda una solución óptima en un tiempo razonable gracias al uso de heurística, además que al implementar un registro de los nodos explorados y por explorar, lo cual optimiza la toma de decisiones evitando caminos ya recorridos,