// A\* para localização de pontos

// Defina a função do algoritmo A\*

function aStarAlgorithm(start, end, graph) {

// Defina a função heurística

function heuristic(a, b) {

return Math.abs(a.x - b.x) + Math.abs(a.y - b.y);

}

// Defina os conjuntos abertos e fechados

let openSet = [start];

let closedSet = [];

// Faça um loop até que o conjunto aberto esteja vazio

while (openSet.length > 0) {

// Encontre o nó com a pontuação f mais baixa

let current = openSet[0];

for (let i = 1; i < openSet.length; i++) {

if (openSet[i].f < current.f) {

current = openSet[i];

}

}

// Se o nó atual for o nó final, retorne o caminho

if (current === end) {

let path = [];

let temp = current;

while (temp.previous) {

path.push(temp);

temp = temp.previous;

}

return path.reverse();

}

// Remova o nó atual do conjunto aberto e adicione-o ao conjunto fechado

openSet.splice(openSet.indexOf(current), 1);

closedSet.push(current);

// Faça um loop pelos vizinhos do nó atual

let neighbors = graph.neighbors(current);

for (let i = 0; i < neighbors.length; i++) {

let neighbor = neighbors[i];

// Se o vizinho já estiver no conjunto fechado, pule-o

if (closedSet.includes(neighbor)) {

continue;

}

// Calcule a pontuação g provisória

let tentativeGScore = current.g + heuristic(neighbor, current);

// Se o vizinho não estiver no conjunto aberto, adicione-o

if (!openSet.includes(neighbor)) {

openSet.push(neighbor);

} else if (tentativeGScore >= neighbor.g) {

continue;

}

// Este caminho é o melhor até agora. Grave isto!

neighbor.previous = current;

neighbor.g = tentativeGScore;

neighbor.f = neighbor.g + heuristic(neighbor, end);

}

}

// Nenhum caminho foi encontrado

return null;

}

// Defina o objeto gráfico

let graph = {

nodes: [],

edges: [],

addNode: function(node) {

this.nodes.push(node);

},

addEdge: function(node1, node2) {

this.edges.push([node1, node2]);

this.edges.push([node2, node1]);

},

neighbors: function(node) {

let neighbors = [];

for (let i = 0; i < this.edges.length; i++) {

let edge = this.edges[i];

if (edge[0] === node) {

neighbors.push(edge[1]);

}

}

return neighbors;

}

};

// Defina o objeto Nó

function Node(x, y) {

this.x = x;

this.y = y;

this.g = 0;

this.f = 0;

this.previous = null;

}

// Defina os nós iniciais e finais

let start = new Node(0, 0);

let end = new Node(5, 5);

// Adicione os nós ao gráfico

graph.addNode(new Node(1, 1));

graph.addNode(new Node(1, 2));

graph.addNode(new Node(2, 2));

graph.addNode(new Node(3, 2));

graph.addNode(new Node(3, 3));

graph.addNode(new Node(4, 3));

graph.addNode(new Node(4, 4));

graph.addNode(new Node(5, 4));

// Adicione as arestas ao gráfico

graph.addEdge(graph.nodes[0], graph.nodes[1]);

graph.addEdge(graph.nodes[1], graph.nodes[2]);

graph.addEdge(graph.nodes[2], graph.nodes[3]);

graph.addEdge(graph.nodes[3], graph.nodes[4]);

graph.addEdge(graph.nodes[4], graph.nodes[5]);

graph.addEdge(graph.nodes[5], graph.nodes[6]);

graph.addEdge(graph.nodes[6], graph.nodes[7]);

graph.addEdge(graph.nodes[3], graph.nodes[7]);

// Encontre o caminho mais curto

let path = aStarAlgorithm(start, end, graph);

// Imprima o caminho

console.log(path);

========================

// Defina uma grade com obstáculos

const grid = [

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

];

// Define the start and end points

const start = [0, 0];

const end = [9, 9];

// Defina a função heurística

function heuristic(a, b) {

return Math.abs(a[0] - b[0]) + Math.abs(a[1] - b[1]);

}

// Defina o algoritmo A\*

function aStar(grid, start, end) {

const open = [start];

const closed = [];

const path = [];

const directions = [

[0, 1],

[0, -1],

[1, 0],

[-1, 0],

[1, 1],

[1, -1],

[-1, 1],

[-1, -1]

];

while (open.length) {

let lowestIndex = 0;

for (let i = 0; i < open.length; i++) {

if (open[i].f < open[lowestIndex].f) {

lowestIndex = i;

}

}

const current = open[lowestIndex];

if (current[0] === end[0] && current[1] === end[1]) {

let temp = current;

path.push(temp);

while (temp.parent) {

path.push(temp.parent);

temp = temp.parent;

}

return path.reverse();

}

open.splice(lowestIndex, 1);

closed.push(current);

for (let i = 0; i < directions.length; i++) {

const direction = directions[i];

const neighbor = [current[0] + direction[0], current[1] + direction[1]];

if (

neighbor[0] < 0 ||

neighbor[0] >= grid.length ||

neighbor[1] < 0 ||

neighbor[1] >= grid[0].length

) {

continue;

}

if (grid[neighbor[0]][neighbor[1]] === 1) {

continue;

}

if (closed.find((node) => node[0] === neighbor[0] && node[1] === neighbor[1])) {

continue;

}

const g = current.g + 1;

const h = heuristic(neighbor, end);

const f = g + h;

const node = { position: neighbor, g, h, f, parent: current };

if (!open.find((element) => element.position[0] === neighbor[0] && element.position[1] === neighbor[1])) {

open.push(node);

}

}

}

return path;

}

// Chame o algoritmo A\*

const path = aStar(grid, start, end);

console.log(path);

=================================

1. Definir a grade com obstáculos: Este bloco define a grade que representa o mapa em que o algoritmo A\* será executado. Os obstáculos são representados por 1s e os espaços vazios por 0s.

2. Definir os pontos de início e fim: Este bloco define os pontos de início e fim no mapa. O algoritmo A\* encontrará o caminho mais curto entre esses dois pontos.

3. Definir a função heurística: Este bloco define a função heurística que o algoritmo A\* usará para estimar o custo restante de chegar ao ponto final a partir de um determinado ponto.

4. Definir o algoritmo A\*: Este bloco define o algoritmo A\* real. Ele usa a grade, os pontos de início e fim e a função heurística para encontrar o caminho mais curto entre os dois pontos.

5. Chamar o algoritmo A\*: Este bloco chama o algoritmo A\* e imprime o caminho mais curto encontrado.

1.

// Define a grid with obstacles

const grid = [

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

];

// Define the start and end points

const start = [0, 0];

const end = [9, 9];

2.

// Define the heuristic function

function heuristic(a, b) {

return Math.abs(a[0] - b[0]) + Math.abs(a[1] - b[1]);

}

3.

// Define the A\* algorithm

function aStar(grid, start, end) {

const open = [start];

const closed = [];

const path = [];

const directions = [

[0, 1],

[0, -1],

[1, 0],

[-1, 0],

[1, 1],

[1, -1],

[-1, 1],

[-1, -1]

];

while (open.length) {

let lowestIndex = 0;

for (let i = 0; i < open.length; i++) {

if (open[i].f < open[lowestIndex].f) {

lowestIndex = i;

}

}

const current = open[lowestIndex];

if (current[0] === end[0] && current[1] === end[1]) {

let temp = current;

path.push(temp);

while (temp.parent) {

path.push(temp.parent);

temp = temp.parent;

}

return path.reverse();

}

open.splice(lowestIndex, 1);

closed.push(current);

for (let i = 0; i < directions.length; i++) {

const direction = directions[i];

const neighbor = [current[0] + direction[0], current[1] + direction[1]];

if (

neighbor[0] < 0 ||

neighbor[0] >= grid.length ||

neighbor[1] < 0 ||

neighbor[1] >= grid[0].length

) {

continue;

}

if (grid[neighbor[0]][neighbor[1]] === 1) {

continue;

}

if (closed.find((node) => node[0] === neighbor[0] && node[1] === neighbor[1])) {

continue;

}

const g = current.g + 1;

const h = heuristic(neighbor, end);

const f = g + h;

const node = { position: neighbor, g, h, f, parent: current };

if (!open.find((element) => element.position[0] === neighbor[0] && element.position[1] === neighbor[1])) {

open.push(node);

}

}

}

return path;

}

4.

// Call the A\* algorithm

const path = aStar(grid, start, end);

console.log(path);

5.

// Call the A\* algorithm

const path = aStar(grid, start, end);

console.log(path);