Actividad 3

Programación Dinámica: Laberinto

Diego Andrés Figueroa Peart
A01660987

Curso:

Análisis y diseño de algoritmos avanzados
(TC2038.652)

Profesor:

Cristhian Alejandro Ávila Sánchez

Fecha de entrega:

29 de noviembre de 2023

Actividad 3. Programación Dinámica: Laberinto

Planteamiento del problema

A lo largo de la programación, han existido diversos algoritmos para resolver laberintos; algunos de estos están diseñados para un autómata que está navegando el laberinto por primera vez y solamente puede ver a sus alrededores, mientras que otros están diseñados tomando en cuenta que se puede visualizar todo el laberinto al mismo tiempo.

Para mi resolución a esta problemática, estaré empleando el segundo de estos casos. Utilizando una variante del algoritmo de Dijkstra, este rellenará todas las celdas accesibles del laberinto desde la salida con su respectiva distancia hasta la celda final, y luego una función recorrerá este laberinto, navegando las distancias menores hasta llegar a la salida.

Datos de entrada:

- N = El tamaño del arreglo que representa a la señal.
- Laberinto = El arreglo bidimensional

Datos de salida:

- Una representación gráfica del camino más eficiente para llegar desde la celda inicial hasta la celda que representa la salida del laberinto, o, si este es imposible, el camino hasta la última celda navegable.

Funcionamiento:

1. Se define el tamaño del laberinto (N x N celdas).
2. Se genera un arreglo bidimensional de N*N, conteniendo 80% celdas libres, representadas por ceros, y 20% muros, representados por unos.
3. Se llama a la función floodFill.
4. Se genera un arreglo vacío llamado queue, al cual se le agregarán todas las celdas válidas que se visiten con sus distancias a la celda final.
5. Se agrega la celda inicial al queue con su distancia (0) y se inicia el bucle, que correrá mientras existan elementos en el queue.
6. Se toma el primer valor del queue y se visitan sus vecinos, si estos son válidos se agregan al queue asignándoles una distancia uno mayor a la de la celda origen.

7. Este proceso se repite hasta que no existan valores en el queue, lo cuál significa que todas las celdas accesibles desde la salida han sido exploradas.
8. Se navega el arreglo de distancia mayor a menor, hasta llegar a la salida.

Pseudocódigo:

```
fun floodFill(int x, int y, arr Laberinto)
{
	Queue = []
	n = 0
	Queue.append(x, y, n)
	While Queue:
		Pos = first(queue)
		N = PosN + 1
		If valid(PosX + 1, PosY):
			Queue.append(PosX + 1, PosY, N)
		If valid(PosX - 1, PosY):
			Queue.append(PosX - 1, PosY, N)
		If valid(PosX, PosY + 1):
			Queue.append(PosX, PosY + 1, N)
		If valid(PosX, PosY - 1):
			Queue.append(PosX, PosY - 1, N)
}
```

Complejidad:

Este algoritmo, en el mejor caso, tiene una complejidad computacional de O(1), dado el caso que haya una única celda en el laberinto o la celda inicial se encuentre rodeada de paredes. En el caso promedio y el peor caso, se visitarán todas las celdas que son alcanzables desde la celda inicial, y como el laberinto consiste de N * N celdas, la complejidad computacional será de $O(N^2)$.

Programa:

```
N = 50

laberinto = []
for i in range(N):
    row = []
    for j in range(N):
        if i == N - 1 and j == 0:
            row.append(1)
            continue
        if i == 0 and j == N - 1:
            row.append(1)
            continue
        k = random.random()
        if k < 0.20:
            row.append(0)
        else:
            row.append(1)
    laberinto.append(row)

def valid(x, y, arreglo):
    if x < 0 or x >= N or y < 0 or y >= N or arreglo[y][x] != -2:
        return False
    else:
        return True

def valid2(x, y, arreglo):
    if x < 0 or x >= N or y < 0 or y >= N or arreglo[y][x] == -1:
        return False
    else:
        return True

def floodFill(x, y, arreglo):
    queue = []
    n = 0
    queue.append([x, y, n])
    arreglo[y][x] = n

    while queue:
        pos = queue.pop(0)
        posX = pos[0]
        posY = pos[1]
        n = pos[2] + 1
```

```
        if valid(posX + 1, posY, arreglo):
            arreglo[posY][posX + 1] = n
            queue.append([posX + 1, posY, n])

        if valid(posX - 1, posY, arreglo):
            arreglo[posY][posX - 1] = n
            queue.append([posX - 1, posY, n])

        if valid(posX, posY + 1, arreglo):
            arreglo[posY + 1][posX] = n
            queue.append([posX, posY + 1, n])

        if valid(posX, posY - 1, arreglo):
            arreglo[posY - 1][posX] = n
            queue.append([posX, posY - 1, n])

    return arreglo

arreglo = []
for i in range(N):
    row = []
    for j in range(N):
        if laberinto[i][j] == 0:
            row.append(-1)
        else:
            row.append(-2)
    arreglo.append(row)

arreglo = floodFill(N - 1, 0, arreglo)

def navegar(arreglo):
    x = 0
    y = N - 1
    n = arreglo[y][x]
    camino = [[x, y]]
    while n != 0:
        vecinos = []
        if valid2(x + 1, y, arreglo) and arreglo[y][x + 1] < n:
            vecinos.append([x + 1, y, arreglo[y][x + 1]])
        if valid2(x - 1, y, arreglo) and arreglo[y][x - 1] < n:
            vecinos.append([x - 1, y, arreglo[y][x - 1]])
        if valid2(x, y + 1, arreglo) and arreglo[y + 1][x] < n:
            vecinos.append([x, y + 1, arreglo[y + 1][x]])
        if valid2(x, y - 1, arreglo) and arreglo[y - 1][x] < n:
            vecinos.append([x, y - 1, arreglo[y - 1][x]])
```

```
        if not vecinos:
            return camino
        vecinos.sort(key=lambda x: x[2])
        x = vecinos[0][0]
        y = vecinos[0][1]
        n = vecinos[0][2]
        camino.append([x, y])
    return camino

camino = navegar(arreglo)
```