**Introducción:**

En este laboratorio, nos sumergiremos en el fascinante mundo de los algoritmos de búsqueda en profundidad (DFS) y en anchura (BFS). Estos algoritmos son herramientas fundamentales en la resolución de una amplia gama de problemas, desde la navegación en redes hasta la planificación de rutas óptimas.

El objetivo principal es construir un prototipo funcional que aborde un escenario de la vida real utilizando estos algoritmos. Nos enfrentaremos a preguntas como: ¿cómo encontrar la ruta más corta para llegar de un punto A a un punto B? ¿O cómo explorar un laberinto para encontrar la salida más rápida?

En este informe, presentaremos nuestros diseños y soluciones para resolver estos desafíos. Exploraremos tanto el algoritmo DFS, que se sumerge profundamente en un problema antes de retroceder, como el algoritmo BFS, que explora todas las posibles soluciones a la vez en busca de la óptima.

Además, adjuntaremos casos de prueba del prototipo para demostrar su funcionalidad y eficacia en situaciones variadas.

**Estrategia:**

La estrategia propuesta se basa en la transformación de un laberinto generado aleatoriamente en un grafo, donde cada celda del laberinto se convierte en un vértice y las conexiones entre celdas adyacentes representan las relaciones en el grafo, excluyendo las conexiones bloqueadas por paredes. Este proceso establece una representación estructurada del laberinto que permite la aplicación de algoritmos de búsqueda. Posteriormente, se identifica un objetivo dentro del laberinto, determinado por la presencia de un número específico (target\_number) en una celda. Se emplean los algoritmos de búsqueda BFS y DFS para encontrar caminos óptimos desde la celda que contiene el objetivo hacia todas las demás celdas que también poseen dicho número. BFS garantiza la búsqueda del camino más corto y eficiente, mientras que DFS explora exhaustivamente las rutas posibles. Finalmente, se compara la eficacia y la longitud de los caminos encontrados por ambos algoritmos para determinar la solución óptima en términos de eficiencia y recursos empleados. Esta estrategia proporciona un enfoque sistemático y eficiente para resolver el problema de encontrar un camino óptimo en un laberinto generado aleatoriamente.

**Entradas:**

Rows: número de filas del laberinto

Cols: número de columnas del laberinto

Max\_number: el número máximo que puede haber en una celda de el laberinto.

Target\_number: número que se quiero encontrar en el laberinto

**Salida:**

El camino óptimo desde una celda cualquiera que contiene el target\_number, hasta las celdas con este número utilizando las búsquedas por anchura y por profundidad.

**Casos de prueba:**

|  |  |  |
| --- | --- | --- |
| Entrada | Justificación | Salida |
| (1,2,3,4) | Matriz no cuadrada, número máximo más pequeño que el número a encontrar | NULL |
| (5,5,6,6) | Target\_number igual al numero máximo, matriz cuadrada | 24 caminos desde todas las celdas usando BFS  Y 22 usando DFS |
| (5,5,4,3) | Matriz cuadrada, numero máximo menor que la cantidad de filas de la matriz, target\_number menor que el numero maximo | 19 caminos desde todas las celdas usando BFS  Y 19 usando DFS |

**Implementación:**

**import** **random**

WHITE = "white"

BLACK = "black"

GRAY = "gray"

**import** **math**

**class** **Graph**:

**def** **\_\_init\_\_**(self, vertexes, relations, directed=True, view=True):

self.directed = directed

self.view = view

self.vertexes = vertexes

self.relations = relations

self.\_buildAdjList()

self.\_buildEncoding()

self.\_buildAdjMatrix()

**def** **\_buildAdjMatrix**(self):

self.adjMat = [[**0** **for** v **in** range(len(self.vertexes))] **for** v **in** range(len(self.vertexes))]

**for** relation **in** self.relations:

row, col = self.encoder[relation[**0**]], self.encoder[relation[**1**]]

self.adjMat[row][col] = **1**

**def** **\_buildEncoding**(self):

self.encoder, self.decoder = {}, {}

index = **0**

**for** v **in** self.vertexes:

self.encoder[v] = index

self.decoder[index] = v

index += **1**

**def** **\_buildAdjList**(self):

self.adjList = {}

**for** v **in** self.vertexes:

self.adjList[v] = []

**for** relation **in** self.relations:

self.adjList[relation[**0**]].append(relation[**1**])

**def** **\_buildRelation**(self, e):

**if** self.directed:

self.relations = e

**else**:

self.relations = set()

**for** el **in** e:

self.relations.add(el)

self.relations.add((el[**1**], el[**0**]))

**def** **\_init\_**(self, v, e, directed=True, view=True):

self.directed = directed

self.view = view

self.vertexes = v

self.\_buildRelation(e)

self.\_buildAdjList()

self.\_buildEncoding()

self.\_buildAdjMatrix()

**def** **getAdjMatrix**(self):

**return** self.adjMat

**def** **getAdjList**(self):

**return** self.adjList

**def** **\_buildVProps**(self, source=None):

self.v\_props = {}

**for** v **in** self.vertexes:

self.v\_props[v] = {

'color': WHITE,

'distance': math.inf,

'parent': None

}

**if** source **is** **not** None:

self.v\_props[source] = {

'color': GRAY,

'distance': **0**,

'parent': None

}

**def** **\_getNeighborsAdjList**(self, vertex):

**return** self.adjList[vertex]

**def** **\_getNeighborsMatAdj**(self, vertex):

neighbors = []

**for** i, v **in** enumerate(self.adjMat[vertex]):

**if** v == **1**:

neighbors.append(self.decoder[i])

**return** neighbors

**def** **getNeighbors**(self, vertex):

**if** self.view:

**return** self.\_getNeighborsAdjList(vertex)

**return** self.\_getNeighborsMatAdj(vertex)

**def** **bfs**(self, source):

self.\_buildVProps(source)

queue = [source]

**while** len(queue) > **0**:

u = queue.pop(**0**)

**for** neighbor **in** self.getNeighbors(u):

**if** self.v\_props[neighbor]['color'] == WHITE:

self.v\_props[neighbor]['color'] = GRAY

self.v\_props[neighbor]['distance'] = self.v\_props[u]['distance'] + **1**

self.v\_props[neighbor]['parent'] = u

queue.append(neighbor)

self.v\_props[u]['color'] = BLACK

**return** self.v\_props

**def** **dfs**(self):

self.\_buildVProps()

time = **0**

**for** v **in** self.vertexes:

**if** self.v\_props[v]['color'] == WHITE:

time = self.dfs\_visit(v, time)

**return** self.v\_props

**def** **dfs\_visit**(self, vertex, time):

time = time + **1**

self.v\_props[vertex]['distance'] = time

self.v\_props[vertex]['color'] = GRAY

**for** neighbor **in** self.getNeighbors(vertex):

**if** self.v\_props[neighbor]['color'] == WHITE:

self.v\_props[neighbor]['parent'] = vertex

time = self.dfs\_visit(neighbor, time)

self.v\_props[vertex]['color'] = BLACK

time = time + **1**

self.v\_props[vertex]['final'] = time

**return** time

**def** **printVProps**(v\_props):

**print**("===================== Resultados =======================")

**for** v **in** v\_props.keys():

v\_props[v]['path'] = '-->'.join(map(str, getPath(v, v\_props)))

**print**("Camino más óptimo desde", str(v), ":", v\_props[v])

**def** **printAdjMatrix**(graph):

**print**("===================== Matriz de Adyacencia =======================")

adjMat = graph.getAdjMatrix()

**for** row **in** adjMat:

**print**(' '.join(list(map(str, row))))

**def** **getPath**(vertex, v\_props):

path = [vertex]

current = vertex

**while**(v\_props[current]['parent'] **is** **not** None):

path.insert(**0**, v\_props[current]['parent'])

current = v\_props[current]['parent']

**return** path

**def** **printAdjList**(graph):

**print**("===================== Lista de Adyacencia =======================")

adjList = graph.getAdjList()

**for** v **in** adjList.keys():

**print**(str(v), list(map(str, adjList[v])))

**def** **generate\_labyrinth**(rows, cols, max\_number):

labyrinth = []

**for** i **in** range(rows):

row = []

**for** j **in** range(cols):

row.append(random.randint(**0**, max\_number))

labyrinth.append(row)

**return** labyrinth

**def** **convert\_to\_graph**(labyrinth):

vertexes = []

relations = []

rows = len(labyrinth)

cols = len(labyrinth[**0**])

**for** i **in** range(rows):

**for** j **in** range(cols):

**if** labyrinth[i][j] != **0**: # Si no es una pared

vertexes.append((i, j))

# Verificar vecinos para agregar relaciones

**if** i > **0** **and** labyrinth[i - **1**][j] != **0**: # Vecino superior

relations.append(((i, j), (i - **1**, j)))

**if** i < rows - **1** **and** labyrinth[i + **1**][j] != **0**: # Vecino inferior

relations.append(((i, j), (i + **1**, j)))

**if** j > **0** **and** labyrinth[i][j - **1**] != **0**: # Vecino izquierdo

relations.append(((i, j), (i, j - **1**)))

**if** j < cols - **1** **and** labyrinth[i][j + **1**] != **0**: # Vecino derecho

relations.append(((i, j), (i, j + **1**)))

**return** Graph(vertexes, relations)

**def** **main**():

rows = **5**

cols = **5**

max\_number = **6** # Número máximo en una celda

target\_number = **6** # Número a encontrar en el laberinto

laberinto = generate\_labyrinth(rows, cols, max\_number)

**print**("Laberinto generado:")

**for** row **in** laberinto:

**print**(row)

graph = convert\_to\_graph(laberinto)

**print**("**\n**Usando BFS para encontrar el camino más óptimo al número", target\_number)

**for** v **in** graph.vertexes:

**if** laberinto[v[**0**]][v[**1**]] == target\_number:

bfs\_result = graph.bfs(v)

printVProps(bfs\_result)

**break**

**print**("**\n**Usando DFS para encontrar el camino más óptimo al número", target\_number)

**for** v **in** graph.vertexes:

**if** laberinto[v[**0**]][v[**1**]] == target\_number:

dfs\_result = graph.dfs()

printVProps(dfs\_result)

**break**

**if** \_\_name\_\_ == "\_\_main\_\_":

main()