Reporte Final

ICI2241-1 – Análisis y Diseño de Algoritmos

José Lara Arce

▼ **Problema del Laberinto Inteligente**

Imagina que te encuentras en un laberinto bidimensional representado como una matriz. La matriz está compuesta por celdas, donde cada celda puede ser un pasillo (representado por 0) o una pared (representada por 1). El objetivo es encontrar el camino más corto desde la entrada del laberinto hasta la salida.

El laberinto tiene una entrada y una salida definidas en la matriz. La entrada se encuentra en una celda específica de la primera fila (fila 0) y la salida en una celda de la última fila (última fila de la matriz). Puedes moverte solo a las celdas adyacentes (izquierda, derecha, arriba, abajo), siempre y cuando no te encuentres con una pared.

▼ **Definición del problema**

▼ Entradas

- Una matriz laberinto de tamaño MxN (M filas y N columnas) que representa el laberinto bidimensional.
- Dos pares de coordenadas (entrada_fila, entrada_col) y (salida_fila, salida_col) que indican la ubicación de la entrada y la salida del laberinto, respectivamente.

▼ Salida

- Si existe un camino válido desde la entrada hasta la salida del laberinto, la salida será la distancia mínima (número de pasos) para recorrer ese camino.
- Si no existe un camino válido desde la entrada hasta la salida, la salida será -1 para indicar que no hay un camino posible.

▼ Restricciones

- La matriz laberinto estará compuesta por números enteros 0 y 1. Un 0 representa un pasillo que se puede atravesar, y un 1 representa una pared que bloquea el paso.
- La entrada y salida deben estar ubicadas en celdas válidas del laberinto, es decir, deben ser coordenadas dentro de la matriz y deben ser pasillos (valores 0).
- Es importante mencionar que el laberinto debe tener un camino válido desde la entrada hasta la salida, sin celdas aisladas o inalcanzables.

▼ **Diseño del algoritmo**

▼ Algoritmo para resolver el Laberinto Inteligente

Para resolver el problema del Laberinto Inteligente, podemos utilizar el algoritmo de búsqueda en anchura (BFS). Este algoritmo recorre el laberinto de manera ordenada, explorando primero todos los nodos vecinos a la entrada y, a medida que avanza, expande la búsqueda en todas las direcciones posibles hasta alcanzar la salida.

El algoritmo sigue los siguientes pasos:

1. Crear una cola vacía y agregar la celda de entrada del laberinto a la cola.
2. Crear una matriz de booleanos del mismo tamaño que el laberinto para marcar las celdas ya visitadas. A medida que el algoritmo BFS avanza y explora nuevas celdas, actualizaremos esta matriz de booleanos para marcar las celdas que ya han sido visitadas, las marca como visitadas estableciendo sus correspondientes entradas en la matriz "visitado" a `True`. evitando así recorridos innecesarios y optimizando la búsqueda del camino más corto desde la entrada hasta la salida del laberinto.
3. Mientras la cola no esté vacía:
   a. Sacar una celda de la cola y marcarla como visitada.
   b. Verificar si la celda es la salida del laberinto. Si es así, se ha encontrado el camino más corto y terminar.

c. En caso contrario, expandir la búsqueda a las celdas adyacentes no visitadas que sean pasillos (0) y agregarlas a la cola.

4. **Actualización de distancias:**

El algoritmo BFS también se utiliza para encontrar el camino más corto desde un punto de inicio hasta un destino en un grafo. Para lograrlo, se utiliza una matriz llamada "distancias". Al comienzo, esta matriz se inicializa con valores altos o infinitos para todas las celdas, excepto para la celda de inicio que se inicializa con distancia cero. A medida que el algoritmo avanza, se actualizan las distancias de las celdas adyacentes a la celda actual visitada. La distancia se actualiza sumando uno a la distancia de la celda actual. Esto garantiza que estemos calculando la distancia mínima desde la celda de inicio hasta cada celda en el laberinto.

5. Si se recorren todas las celdas posibles sin encontrar la salida, no hay un camino válido, retornando el valor -1.

▼ Ejemplo

Teniendo el siguiente laberinto:

```
laberinto =
[
  [0, 1, 0, 0],
  [0, 0, 0, 1],
  [1, 1, 0, 0],
  [0, 1, 0, 0],
]

visitado =
[
    [False, False, False, False],
    [False, False, False, False],
    [False, False, False, False],
    [False, False, False, False],
]
entrada_fila, entrada_col = 0, 0
salida_fila, salida_col = 3, 3
```

En este caso, la **entrada** está en la coordenada `(0, 0)` y la **salida** en la coordenada `(3, 3)`. La representación visual del laberinto sería la siguiente:

0	1	0	0
0	0	0	1
1	1	0	0
0	1	0	0

Donde:

- `0` representa un pasillo que se puede atravesar.
- `1` representa una pared que bloquea el paso.

Ejecutando el algoritmo de búsqueda en anchura sobre este laberinto, se explorarán las celdas desde la entrada hasta la salida. Durante la exploración, el algoritmo marcará el camino más corto desde la entrada hasta cada celda visitada. El proceso podría verse así:

0	1	0	0
1	2	3	1
1	1	4	5
0	1	5	6

En esta representación, se han marcado los números desde `2` hasta `6` en las celdas visitadas, siguiendo el camino más corto desde la entrada hasta la salida. Los números indican el orden en el que se visitaron las celdas. Por lo tanto, en este ejemplo, la distancia mínima desde la entrada hasta la salida es de `6` pasos. En consecuencia, la salida del programa sería:

```
El camino más corto desde la entrada hasta la salida es de 6 pasos.
```

▼ Código

```
from collections import deque
from matplotlib.ticker import ScalarFormatter
import time
import matplotlib.pyplot as plt
import random

class Laberinto:
  def __init__(self, laberinto, coordenadaEntrada, coordenadaSalida):
    self.laberinto = laberinto
    self.entrada = coordenadaEntrada
    self.salida = coordenadaSalida
##

def generarLaberinto(filas, columnas):
  laberinto = [[0 for _ in range(columnas)] for _ in range(filas)]
  probabilidadInsercionPared = 0.2

  for fila in range(filas):
    for columna in range(columnas):
      if random.random() < probabilidadInsercionPared:
        laberinto[fila][columna] = 1

  return laberinto
##

def movimientoValido(laberinto, visitado, fila, columna):
    filas, columnas = len(laberinto), len(laberinto[0])
    return 0 <= fila < filas and 0 <= columna < columnas and laberinto[fila][columna] == 0 and not visitado[fila][columna]

##
def encontrarCaminoMasRapido(laberinto, filaInicial, columnaInicial, filaFinal, columnaFinal):
    filas, columnas = len(laberinto), len(laberinto[0])
    visitado = [[False for _ in range(columnas)] for _ in range(filas)]
    cola = deque([(filaInicial, columnaInicial, 0)])
    operaciones = 0

    while cola:
        filaActual, columnaActual, distancia = cola.popleft()
        operaciones += 1
        visitado[filaActual][columnaActual] = True

        if filaActual == filaFinal and columnaActual == columnaFinal:
            return distancia, operaciones

        movimientos = [(0, 1), (1, 0), (0, -1), (-1, 0)]

        for filaMov, columnaMov in movimientos:
            filaSig, columnaSig = filaActual + filaMov, columnaActual + columnaMov
            if movimientoValido(laberinto, visitado, filaSig, columnaSig):
                cola.append((filaSig, columnaSig, distancia + 1))

    return -1, operaciones  # Se retorna -1 si es que no se consigue encontrar un camino válido para el laberinto.

def evaluarDesempeno(tamanosMatrices):
    listaOperaciones = []

    for tamano in tamanosMatrices:
        sizeFilas, sizeColumnas = tamano, tamano

        laberinto = generarLaberinto(sizeFilas, sizeColumnas)

        entradaFila, entradaColumna = 0, 0  # Entrada en la esquina superior izquierda
        salidaFila, salidaColumna = sizeFilas - 1, sizeColumnas - 1  # Salida en la esquina inferior derecha

        if laberinto[entradaFila][entradaColumna] == 1:
            entradaColumna = 1

        salidaFila, salidaColumna = sizeFilas - 1, sizeColumnas - 1
        if laberinto[salidaFila][salidaColumna] == 1:
            salidaColumna = sizeColumnas - 2

        distanciaMinima, numOperaciones = encontrarCaminoMasRapido(laberinto, entradaFila, entradaColumna, salidaFila, salidaColumn
        listaOperaciones.append(numOperaciones)

    plt.plot(tamanosMatrices, listaOperaciones, marker='o')
    plt.xlabel('Tamaño de Matriz (N x N)')
    plt.ylabel('Cantidad de Operaciones')
```

```
    plt.title('Operaciones vs Tamaño de Matriz')
    plt.show()

tamanosMatrices = [2, 3, 4, 5, 6, 7, 8, 9, 10]
evaluarDesempeno(tamanosMatrices)
```

▼ Operaciones clave

Considerando el algoritmo presentado en la preclase n°1, su tiempo de ejecución del algoritmo BFS para resolver el laberinto está influenciado por varias operaciones clave que ocurren dentro del bucle principal y en otras partes del código. Aquí están las operaciones clave que determinan el tiempo de ejecución:

▼ **Bucle principal**

El algoritmo BFS está contenido en un bucle principal que itera mientras la cola no esté vacía. Este bucle es la esencia del algoritmo y su complejidad depende del número de celdas exploradas y de la estructura del laberinto. En cada iteración, se saca una celda de la cola. Si asumimos que cada celda se agrega y saca una vez, esto se repetirá un número total de veces igual al número total de celdas en el laberinto, es decir, M * N.

▼ **Exploración de vecinos**

En cada iteración del bucle principal, se exploran los vecinos de la celda actual. Esto implica realizar operaciones de acceso y verificación en la matriz del laberinto y la matriz de visitados. En el peor caso, se explorarán los cuatro vecinos. Por lo tanto, esta operación se realizará hasta 4 veces para cada celda explorada.

La exploración de vecinos implica las siguientes operaciones:

- Comprobar la validez del movimiento en el laberinto: `movimientoValido()`
- Acceso a la posición actual y los vecinos: `laberinto[fila][columna]`
- Marcar la matriz como visitada: `visitado[fila][columna]`

▼ **Operaciones de cola**

Dentro del bucle principal, se realizan operaciones de cola para agregar y sacar elementos:

- Agregar un elemento a la cola: `cola.append((filaSig, columnaSig, distancia + 1))`
- Sacar un elemento de la cola: `filaActual, columnaActual, distancia = cola.popleft()`

En cada iteración, se agregan celdas a la cola y se sacan de la cola. En total, esto dependerá del número de celdas en el laberinto y cuántas veces se exploran.

▼ **Verificación de condición de salida**

La verificación de la condición de salida (si se alcanzó la celda de salida) se realizará en cada iteración del bucle, lo que resultará en M * N veces en el peor caso.

En cada iteración del bucle, se verifica si la celda actual es la celda de salida. Esto implica comparaciones de igualdad entre las coordenadas de la celda actual y la celda de salida.

▼ **Iniciación y creación de estructuras de datos**

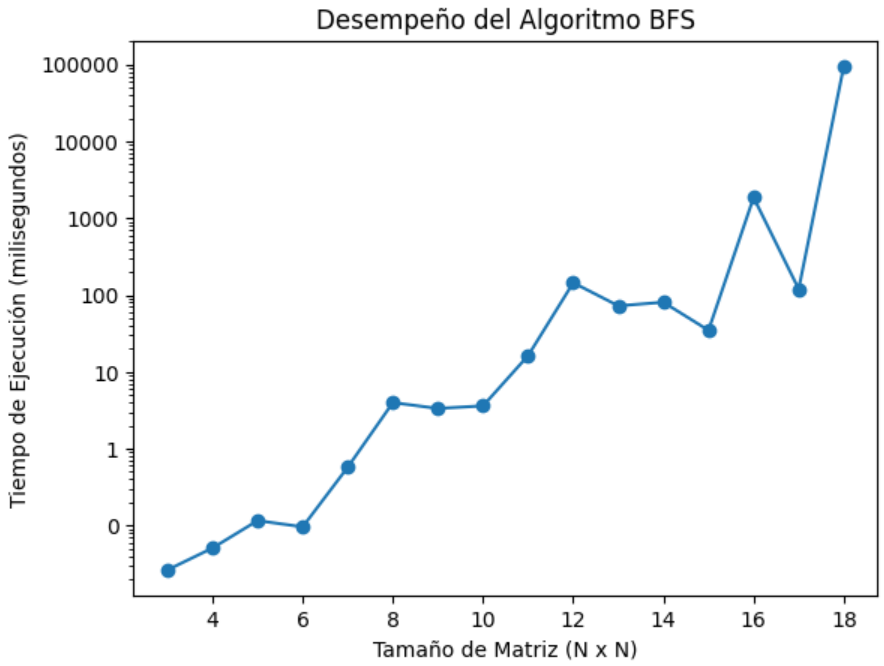
Las estructuras de datos como la matriz de visitados y la cola se crean una sola vez al principio, por lo que estas operaciones tienen un impacto constante en la complejidad temporal.

Antes de iniciar el bucle principal, se realizan operaciones para crear y llenar estructuras de datos como la matriz de visitados y la cola inicial.

▼ **Complejidad temporal**

La complejidad del algoritmo BFS depende principalmente del número total de celdas exploradas y de la cantidad de vecinos que se deben explorar en cada iteración del bucle. Si el laberinto tiene M filas y N columnas, y el número total de celdas es M * N, por lo tanto la complejidad típica del algoritmo BFS es del orden O(M * N), ya que cada celda se explorará al menos una vez. Esto refleja con precisión el comportamiento del algoritmo, ya que su tiempo de ejecución depende linealmente del número total de celdas en el laberinto.

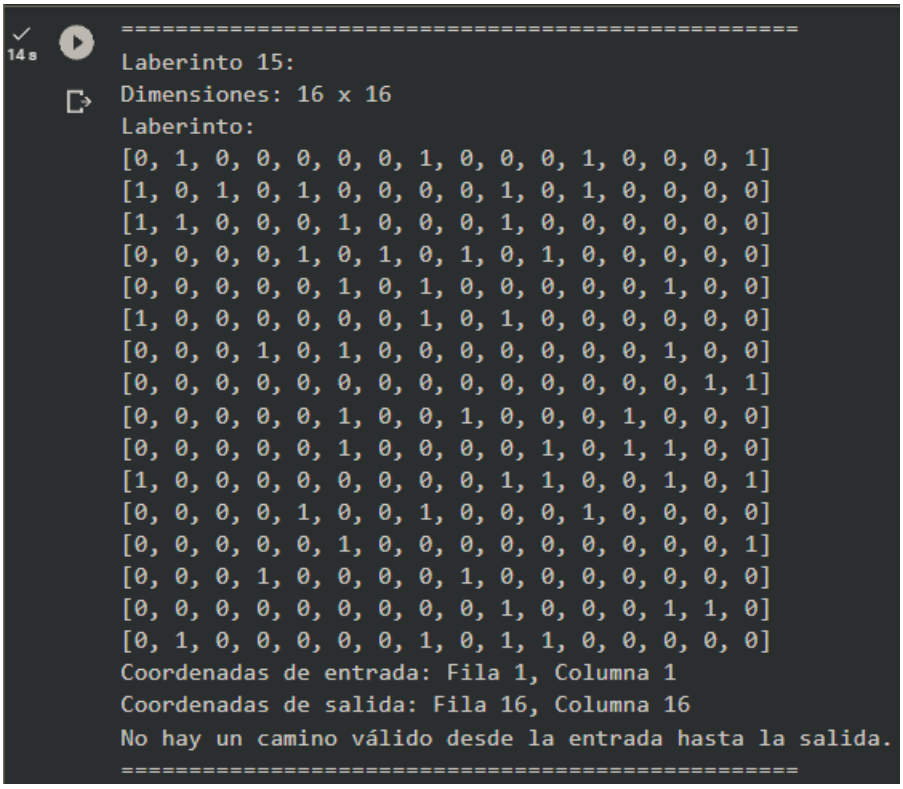
▼ Desempeño del algoritmo Breadth-first search (BFS) para resolver laberintos generados automáticamente de la manera más rápida posible



El gráfico generado por la librería “matplotlib” muestra cómo el tiempo de ejecución del algoritmo BFS aumenta a medida que se incrementan las dimensiones de la matriz del laberinto. Se observa que a medida que el tamaño de la matriz crece, el tiempo necesario para encontrar la distancia mínima entre la entrada y la salida también aumenta. Es importante destacar que las limitaciones de recursos de la plataforma de ejecución influyeron en la capacidad de probar matrices con dimensiones superiores a 18x18. Google Colab, la herramienta utilizada para el desarrollo y evaluación del algoritmo, presentó restricciones que impidieron la prueba de tamaños de matriz más grandes. A pesar de estas limitaciones, el gráfico refleja claramente la tendencia esperada de un aumento gradual en el tiempo de ejecución a medida que se consideran laberintos con mayores dimensiones. Demostrando que efectivamente la complejidad temporal es O(M * N) siendo M las filas y N las columnas. En nuestros casos de testeo solo lo hicimos con matrices cuadradas, pero el resultado es el mismo si es que no lo son.

Es importante notar que en algunas instancias, el gráfico puede mostrar valores más bajos de tiempo de ejecución. Esto puede ser atribuido al hecho de que en ciertos casos, el algoritmo no logra encontrar un camino válido entre la entrada y la salida del laberinto. Cuando esto ocurre, la ejecución del algoritmo termina prematuramente y en menos tiempo en comparación con los casos en los que se encuentra un camino. Estas inconsistencias se deben a la naturaleza aleatoria de los laberintos generados y la posibilidad de que no haya una ruta viable entre la entrada y la salida. A pesar de estas discrepancias, el patrón general de aumento en el tiempo de ejecución con dimensiones más grandes del laberinto sigue siendo evidente en el gráfico.

Por ejemplo:



En este caso no hay un camino posible para el algoritmo que pueda resolver, por lo que termina la ejecución abruptamente y se consigue un menor tiempo de cómputo.

En un caso donde el algoritmo hizo el trabajo de cómputo completo se vería de esta forma:

```
==================================================
Laberinto 17:
Dimensiones: 18 x 18
Laberinto:
[0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0]
[0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
[1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0]
[0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0]
[0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1]
[0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1]
[0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0]
[0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1]
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Coordenadas de entrada: Fila 1, Columna 1
Coordenadas de salida: Fila 18, Columna 18
Distancia mínima: 34
==================================================
```

▼ **Mejoras**

Agregué la información sobre las restricciones de tamaño de la matriz laberinto y cómo se validan las coordenadas de entrada y salida en el código proporcionado. Esto asegura que las coordenadas de entrada y salida estén dentro de los límites de la matriz y se ajusten en caso de que haya paredes en esas celdas. Además, he corregido una pequeña imprecisión en la asignación de las coordenadas de salida, asegurando que se muevan a la columna anterior si hay una pared en la esquina inferior derecha del laberinto. Esto proporciona una mayor claridad en la validación de las coordenadas de entrada y salida.

También agregué más detalles al algoritmo para explicar cómo se maneja el proceso de marcar celdas visitadas y cómo se actualizan las distancias en la matriz de distancias. Esto ayudará a los lectores a comprender mejor el funcionamiento interno del algoritmo y cómo se utiliza la matriz de visitados para garantizar que no se recorran celdas ya visitadas. También he agregado una breve descripción de la matriz de visitados y cómo se inicializa en cada iteración.