Universidad de Concepción

Universidad de Concepción

(503356-1)

Inteligencia Artificial

Profesor Julio Godoy Del Campo

Tarea 1: Laberinto Saltarín

Autor:
Daniel Soto Salgado - 2021750673

Concepción, 2025

1. Introducción

En este informe se presenta el desarrollo de algoritmos de búsqueda como lo son Depth-First Search (DFS) y Búsqueda de Costo Uniforme (UCS), mediante la implementación de agentes los cuales se encargan de hallar una solución. El objetivo principal es encontrar el camino más corto desde una posición inicial hasta una posición final dentro de una matriz, considerando restricciones específicas y visualizando el proceso de búsqueda en tiempo real mediante la biblioteca PyGame. A lo largo del informe, se detallan los pasos seguidos para la implementación, los desafíos encontrados durante el desarrollo y las soluciones adoptadas para garantizar la correcta ejecución del algoritmo y su visualización.

2. Depth-First Search

2.1. Estructura del Algoritmo DFS

El algoritmo DFS se implementa mediante una función recursiva que explora las celdas adyacentes de la matriz en profundidad. La función `dfs` tiene los siguientes parámetros principales:

- `matriz`: La matriz en la que se realiza la búsqueda.
- `filaInicio, columnaInicio`: Coordenadas de la celda actual.
- `visitados`: Un conjunto que rastrea las celdas ya visitadas para evitar ciclos.
- `filaFinal, columnaFinal`: Coordenadas de la celda objetivo.
- `pasos`: Número de pasos acumulados desde el inicio hasta la celda actual.
- `caminoMinimo`: Una lista que contiene:
  - `caminoMinimo[0]`: El número mínimo de pasos encontrados hasta ahora.
  - `caminoMinimo[1]`: El camino correspondiente al mínimo número de pasos.
- `caminoActual`: Una lista que rastrea el camino actual durante la recursión.

2.2. Lógica del Algoritmo

2.2.1. Condiciones base

```
if (filaInicio < 0 or filaInicio >= len(matriz) or
    columnaInicio < 0 or columnaInicio >= len(matriz[0]) or
    (filaInicio, columnaInicio) in visitados):
    return False

if pasos >= caminoMinimo[0]:
    return False
```

- Se verifica si la celda actual está fuera de los límites de la matriz o ya fue visitada.

- Si los pasos acumulados son mayores o iguales al mejor camino encontrado hasta ahora, se detiene la exploración (poda).

2.2.2. Marcar la celda como visitada

```
visitados.add((filaInicio, columnaInicio))
caminoActual.append((filaInicio, columnaInicio))
```

- La celda actual se agrega al conjunto de celdas visitadas y al camino actual.

2.2.3. Verificar si se alcanzó el objetivo

```
# Verificar si se encontro el valor final
if (filaInicio, columnaInicio) == (filaFinal, columnaFinal):
    if pasos < caminoMinimo[0]:
        caminoMinimo[0] = pasos
        caminoMinimo[1] = list(caminoActual)
    visitados.remove((filaInicio,columnaInicio))
    caminoActual.pop()
    return True
```

- **Verificar si se alcanzó el objetivo:** Se compara la posición actual con la posición objetivo. Si son iguales, significa que se ha llegado al nodo final.
- **Actualizar el camino mínimo:** Si el número de pasos realizados (`pasos`) es menor que el número de pasos almacenado en `caminoMinimo[0]`, se actualiza:
  - `caminoMinimo[0]`: Se guarda el nuevo número mínimo de pasos.
  - `caminoMinimo[1]`: Se copia el contenido de `caminoActual` para registrar el camino más corto encontrado hasta el momento.
- **Retroceder en la recursión (backtracking):** Se eliminan las referencias a la celda actual para permitir que otros caminos puedan explorarla:
  - `visitados.remove((filaInicio, columnaInicio))`: Se elimina la celda actual del conjunto de celdas visitadas.
  - `caminoActual.pop()`: Se elimina la celda actual del camino actual.
- **Finalizar la búsqueda en este camino:** Se retorna `True` para indicar que se alcanzó el objetivo en este camino.

2.2.4. Explorar las direcciones posibles

```
valorActual = matriz[filaInicio][columnaInicio]
direcciones = [(-valorActual, 0), (valorActual, 0), (0, -valorActual), (0, valorActual)]
for df, dc in direcciones:
    dfs(matriz, filaInicio + df, columnaInicio + dc, visitados, filaFinal, columnaFinal, pasos + 1, caminoMinimo, caminoActual)
```

- Se calculan las direcciones posibles basadas en el valor de la celda actual (`valorActual`).
- Para cada dirección, se llama recursivamente a `dfs` con las nuevas coordenadas y el número de pasos incrementado en 1.

2.2.5. Deshacer cambios al retroceder

```
visitados.remove((filaInicio, columnaInicio))
caminoActual.pop()
return False
```

- Después de explorar todas las direcciones, se eliminan la celda actual del conjunto de visitados y del camino actual para permitir otras exploraciones.
- La función retorna `False` si no se encuentra un camino válido desde esta celda.

2.3. Recursividad en DFS

Dado que la implementación del DFS estándar no garantiza óptimos, para lograrlo se utiliza recursividad. Cada llamada a `dfs` explora una celda y luego llama a sí misma para explorar las celdas vecinas. Esto crea un árbol de llamadas recursivas que explora todas las rutas posibles desde la celda inicial hasta la celda final.

2.3.1. Cómo se garantiza el camino óptimo

- **Poda por subóptimos:** La condición `if pasos >= caminoMinimo[0]` asegura que no se exploren caminos que ya son peores que el mejor encontrado hasta ahora.
- **Actualización del camino mínimo:** Cada vez que se alcanza la celda objetivo, se verifica si el número de pasos es menor que el mejor encontrado. Si es así, se actualiza el camino mínimo.

3. Búsqueda de Costo-Uniforme

La función `ucs` implementa el algoritmo de búsqueda de costo uniforme, que encuentra el camino de menor costo desde una celda inicial hasta una celda final en una matriz. Cabe destacar que en esta implementación se trabajó bajo dos supuestos, los cuales nos llevan a costos acumulados

distintos. Si bien el agente que trabaje con este algoritmo debe hallar la solución al problema con el camino más corto, dicho camino puede calcularse de dos maneras:

- **Costo basado en el valor de la celda:** En esta implementación, el costo acumulado se incrementa según el valor de la celda visitada. Esto significa que las celdas con valores más altos representan un mayor "peso.en el cálculo del costo total, lo que puede influir en la selección del camino óptimo. Este enfoque es útil cuando el valor de la celda tiene un significado específico, como el tiempo, la energía o algún recurso consumido.
- **Costo constante por movimiento:** En esta variante, el costo acumulado se incrementa en 1 por cada movimiento, independientemente del valor de la celda. Este enfoque considera que todos los movimientos tienen el mismo costo, lo que resulta en un cálculo basado únicamente en la distancia en términos de pasos.

Para efectos de este informe se decidió dar una descripción al algoritmo que se basa en el costo de las celdas.

3.1. Inicialización

- Se inicializan:
  - `visitados`: Diccionario para almacenar el costo mínimo de cada celda visitada.
  - `priorityQueue`: Cola de prioridad para explorar celdas en orden de costo acumulado.
- La celda inicial se agrega a la cola con costo 0.

3.2. Bucle principal

- Mientras la cola no esté vacía:
  - Extraer la celda con el menor costo acumulado.
  - Si ya fue visitada con un costo menor o igual, omitirla.

```
while priorityQueue:
    #Extraer el nodo con el menor costo acumulado
    costoAcumulado, fila, columna, camino = heapq.heappop(priorityQueue)

    #Si ya se visito esta celda con un costo menor o igual, omitirla
    if (fila, columna) in visitados and visitados[(fila, columna)] <=
        costoAcumulado:
         continue

    visitados[(fila, columna)] = costoAcumulado
    camino = camino + [(fila, columna)]  #Actualizar el camino
```

Extracción y procesamiento de nodos en UCS

3.3. Verificar si se alcanzó el objetivo

Si la celda actual es la celda objetivo, el algoritmo retorna:

- El número de pasos necesarios para llegar al objetivo
- El costo acumulado total.
- El camino óptimo como una lista de coordenadas.

```
        if (fila, columna) == (filaFinal, columnaFinal):
        #Retornar pasos, costo y el camino optimo
        return len(camino) - 1, costoAcumulado, camino
```

3.4. Generar vecinos

- Para cada vecino válido (dentro de los límites de la matriz):
  - Calcular el nuevo costo acumulado.
  - Se generan las direcciones posibles (arriba, abajo, izquierda, derecha) basadas en el valor de la celda actual `ValorActual`.
  - Para cada vecino válido, se calcula el nuevo costo acumulado y se agrega a la cola de prioridad si no ha sido visitado o si el nuevo costo es menor que el registrado previamente.
  - Cabe destacar que la unica variante de implementación al usar coste constante basado en los movimientos es cambiar la forma en la cual se calcula el nuevo costo.

```
# Generar vecinos
    valorActual = matriz[fila][columna]
    direcciones = [(-valorActual, 0), (valorActual, 0), (0, -
        valorActual), (0, valorActual)]
    for df, dc in direcciones:
        nuevaFila, nuevaColumna = fila + df, columna + dc
        #Verificar si el vecino esta dentro de los limites de la
            matriz
        if 0 <= nuevaFila < filas and 0 <= nuevaColumna < columnas:
            nuevoCosto = costoAcumulado + valorActual
            #Metodo de coste constante
            #nuevoCosto = costoAcumulado + 1
            #Agregar el vecino a la cola si no ha sido visitado o si
                tiene un costo menor
            if (nuevaFila, nuevaColumna) not in visitados or
               nuevoCosto < visitados[(nuevaFila, nuevaColumna)]:
                heapq.heappush(priorityQueue, (nuevoCosto, nuevaFila
                    , nuevaColumna, camino))
return -1, -1, []  #Si no se encuentra un camino
```

Generación de vecinos en UCS

- Si no encuentra un camino, retorna -1, -1, []

4. Interfaz Gráfica

4.1. Dibujar la matriz

Esta función es responsable de renderizar la matriz en la ventana principal. Cada celda de la matriz se dibuja como un rectángulo con un color específico, dependiendo de su estado.

4.2. Visualizar Camino Mínimo

Esta función muestra el camino mínimo paso a paso en la matriz:

- **Inicialización**
  - Configura la ventana de PyGame con dimensiones basadas en el tamaño de la matriz.
  - Define un botón para avanzar a la siguiente matriz.
- **Dibujo paso a paso:**
  - Itera sobre los pasos del camino mínimo (`caminoMinimo`).
  - En cada paso:
    - Se actualiza la matriz para mostrar el camino hasta ese punto.
    - Se muestra un contador de pasos en la parte inferior de la ventana.
    - Se introduce un retraso (`pygame.time.delay(1000)`) para controlar la velocidad de visualización.

4.3. Menú de Selección de Algoritmo

Esta función presenta un menú inicial para que el usuario seleccione el algoritmo a utilizar (**DFS** o **UCS**):

- **Botones:**
  - Se crean dos botones (`botonDFS` y `botonUCS`) con dimensiones y posiciones específicas.
  - Cada botón tiene un texto que indica el algoritmo correspondiente.

5. Funcionamiento

Todo ocurre en el algoritmo `main.py`, donde en primera instancia se lee el **input** que es dado por un archivo `.txt` el cual brinda las matrices en el formato correspondiente a explorar para luego ser procesadas por el algoritmo a elección.

6. Ejemplo de Input y Output

Input	Output
5 5 0 0 4 4 2 3 1 1 2 2 1 3 1 1 1 2 1 2 1 3 1 1 3 2 2 2 1 1 3 4 4 0 0 3 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 0 0 2 2 1 2 3 3 1 1 2 1 2 0	Procesando matriz 1: Matriz: Buscando el camino más corto hacia el valor final con DFS ... Se llegó a destino en: 5 pasos. Camino mínimo: (0, 0) (2, 0) (2, 1) (4, 1) (4, 3) (4, 4) Procesando matriz 2: Matriz: Buscando el camino más corto hacia el valor final con DFS ... No se encontró un camino hacia el valor final. Procesando matriz 3: Matriz: Buscando el camino más corto hacia el valor final con DFS ... Se llegó a destino en: 3 pasos. Camino mínimo: (0, 0) (0, 1) (2, 1) (2, 2)