FACULTAD DE CIENCIAS
INTELIGENCIA ARTIFICIAL

Agentes inteligentes en laberintos

Equipo: `Skynet Scribes`

Número de practica: `02`

Carlos Daniel Cortés Jiménez
420004846

Sarah Sophía Olivares García
318360638

Marco Silva Huerta
316205326

Juan Daniel Barrera Holan
417079372

Laura Itzel Tinoco Miguel
316020189

Profesora: Cecilia Reyes Peña
Ayudante teoría: Karem Ramos Calpulalpan
Ayudante laboratorio: Tania Michelle Rubí Rojas

Semestre 2024-2

Fecha de entrega:
21 de Febrero del 2024

1. Backtracking

Backtracking es una técnica algorítmica para hacer una búsqueda exhaustiva y sistemática por todas las configuraciones posibles del espacio de búsqueda del problema, para encontrar el resultado definido por el problema. El término *backtrack* fue acuñado por primera vez por el matemático estadounidense D. H. Lehmer en la década de 1950.

En general, la forma de actuar consiste en elegir una alternativa del conjunto de opciones en cada etapa del proceso de resolución, y si esta elección no funciona (no nos lleva a ninguna solución), la búsqueda vuelve al punto donde se realizó esa elección, e intenta con otro valor. Cuando se han agotado todos los posibles valores en ese punto, la búsqueda vuelve a la anterior fase en la que se hizo otra elección entre valores. Si no hay más puntos de elección, la búsqueda finaliza.

Se suele aplicar en la resolución de un gran número de problemas, muy especialmente en los de decisión y optimización:

- **Problemas de decisión:** Búsqueda de las soluciones que satisfacen ciertas restricciones.

  Ejemplo: Problema de las N-Reinas.

- **Problemas de optimización:** Búsqueda de la mejor solución en base a una función objetivo.

  Ejemplo: Problema de la mochila.

Sus ventajas y desventajas son:

- **Ventajas:**
  - Si existe una solución, la calcula.
  - Es un esquema sencillo de implementar.
  - Adaptable a las características especificas de cada problema.
- **Desventajas:**
  - Coste exponencial en la mayoría de los casos.
  - Si el espacio de búsqueda es infinito, la solución, aunque exista, no se encontrará nunca.
  - Por termino medio consume mucha memoria al tener que almacenar las llamadas recursivas.

2. Algoritmos similares

Algunos algoritmos que utilizan estrategias similares a Backtracking son:

- **Branch and Bound:** Este realiza un recorrido sistemático del árbol de estados de un problema, si bien ese recorrido no tiene por qué ser en profundidad, como sucedía en backtracking: usaremos una estrategia de ramificación. Aquí se utilizan técnicas de poda para poder eliminar aquellos nodos que no lleven a soluciones óptimas.
- **Depth-First Search (DFS):** Este es un algoritmo de búsqueda no informada, lo que hace primero es proporcionar los pasos para atravesar todos los nodos de un gráfico sin repetir ningún nodo.

- **Breadth-First Search (BFS):**Es un algoritmo para atravesar o buscar estructuras de datos de árboles. Comienza en la raíz del árbol (o algún nodo arbitrario, en ocasiones denominado como:clave de búsqueda) y explora primero los nodos vecinos antes de pasar a los vecinos del siguiente nivel.
- **A* (A estrella):** A* combina la búsqueda heurística con la búsqueda informada para encontrar la solución más eficiente en problemas de ruta. Aunque difiere del backtracking, comparten el objetivo de encontrar soluciones en un espacio de búsqueda.

3. Pseudocódigo

Vamos a desarrollar una idea en palabras simples sobre como es la implementación del algoritmo de backtracking para la resolución de un laberinto

```
Función ResolverLaberinto(cuadrado_actual):
    Si el cuadrado_actual está en la salida del laberinto, significa que
     hemos encontrado una solución y devolvemos VERDADERO.
    Si el cuadrado_actual ya está marcado, eso significa que hemos
     intentado este camino antes y no queremos repetirlo, así que
     devolvemos FALSO.
    Marcamos el cuadrado_actual como visitado para indicar que
     estamos explorando este camino.
    Para cada una de las cuatro direcciones posibles
    (izquierda, bajo, arriba, derecha):
        Si no hay una pared en la dirección actual:
            Movemos un paso en esa dirección desde el cuadrado_actual.
            Intentamos resolver el laberinto desde el nuevo cuadrado
             haciendo una llamada recursiva a
             la función ResolverLaberinto.
            Si esta llamada retorna VERDADERO, significa que hemos
             encontrado una solución, así que devolvemos VERDADERO.
    Si ninguna de las cuatro direcciones nos lleva a una solución,
     desmarcamos el cuadrado_actual y devolvemos FALSO para
     indicar que este camino no nos llevó a ninguna solución.

La parte crucial de este algoritmo es el bucle for, que nos
 permite explorar cada una de las posibles direcciones
 desde el cuadrado_actual.
Para cada dirección posible (izquierda, bajo, arriba, derecha):
    Si no hay una pared en esa dirección:
        Intentamos resolver el laberinto desde el
         cuadrado adyacente en esa dirección.
        Si encontramos una solución desde ese cuadrado
         adyacente, significa que también tenemos una
         solución desde el cuadrado_actual,
         así que retornamos VERDADERO.
```

La idea de esto fue basada en el siguiente pdf.

4. Documentación

4.1. Forma de ejecutar

- Asegurarnos de tener instalada una de las siguientes versiones de python: [ope]

```
skynet: $ python --version
skynet: $  Python 3.7
skynet: $  Python 3.8
skynet: $  Python 3.9
skynet: $  Python 3.10
```

  Esto es para no tener problemas a la hora de trabajar con `gym`

- Vamos a la carpeta donde se encuentra el archivo `laberinto.py`

```
skynet: $ cd Practica02/src
skynet: Practica02/src $  ls
laberinto.py
```

- Instalar `gym`

```
skynet: Practica02/src $  pip install gym
```

- Para ejecutar usaremos el siguiente comando

```
skynet: Practica02/src $  python laberinto.py
```

4.2. Código

OpenAI gym

Gym es una biblioteca Python de código abierto para desarrollar y comparar algoritmos de aprendizaje por refuerzo al proporcionar una API estándar para comunicarse entre algoritmos y entornos de aprendizaje, así como un conjunto estándar de entornos compatibles con esa API. [ope]

Spaces: Define el formato válido de los espacios de observación y accion para un entorno.

Los problemas de aprendizaje por refuerzo consisten en el agente y el entorno. El entorno proporciona retroalimentación al agente para que pueda aprender qué accion es apropiada para un estado específico.

4.3. Cambios realizados

Definición del agente

Haciendo uso de la clase **Maze** que hereda de `gym` construimos nuestra clase para el `Agente` y el laberinto. Almacenando al mismo laberinto por su altura y anchura junto con las lineas de movimiento con todos los estados posibles dentro del laberinto, también se inicializa un posicion para el agente y una operación que nos indica si se esta ejecutando o a finalizado.

```python
# Representar el entorno del laberinto
class Maze(gym.Env):

    def __init__(self, maze):
        super(Maze, self).__init__()
        # Almacena el laberinto
        self.maze = maze
        # Almacena la altura del laberinto
        self.height = len(maze)
        # Almacena la anchura del laberinto
        self.width = len(maze[0])
        # Espacio de accion del agente: arriba, abajo, izquierda, derecha
        self.action_space = spaces.Discrete(4)
        # Espacio de observacion (posibles estados que el agente puede observar del entorno)
        self.observation_space = spaces.Discrete(self.height * self.width)
        self.agent_pos = None  # Posicion inicial
        self.done = False  # Indica si el episodio ha terminado (interacciones entre el agente y el entorno )
```

Listing 1: Definición del agente

El método `reset` básicamente reinicia el laberinto con la finalidad de que en cada llamada el agente se encuentre en su poción inicial correcta. Lo hace llamando al agente y asignándole la posición y terminando la interacción agente laberinto.

Movimientos del agente

```python
    # Reinicia el entorno a su estado inicial y devuelve la posicion inicial del agente.
    def reset(self):
        # Se busca la posicion de inicio en el laberinto
        self.agent_pos = self.find_start()
        # Reinicia el estado de finalizacion del episodio
        self.done = False
        return self.agent_pos
```

Listing 2: metodo reset

El método `step` (similar al método *mover*) es el encargado de hacer que nuestro agente tome las direcciones arriba, abajo, izquierda, derecha. Primero comprobando el valor de self.done para saber si ha terminado, si es así, regresa a la posición actual del agente sin realizar ningún movimiento adicional. Se toman las coordenadas x, y, que son las actuales del agente, con *action* se verifica que dirección quiere tomar, si es valida actualiza la posición del agente y vuelve verificar pero esta vez si es que la poción es la de salida marcando un True. Finalmente, regresa la nueva posición del agente, una recompensa de 0, el estado de finalización del episodio y un diccionario vacío.

```python
    # Toma una accion y se mueve un paso en el entorno.
    def step(self, action):
        if self.done:
            # Si el episodio ya ha terminado, regresamos a la posicion actual
            return self.agent_pos, 0, True, {}

        x, y = self.agent_pos

        # Movimientos posibles: arriba, abajo, izquierda, derecha
        if action == 0 and x > 0 and self.maze[x - 1][y] != 1:
```

```
            print("Movimiento: arriba")
            x -= 1
        elif action == 1 and x < self.height - 1 and self.maze[x + 1][y] != 1:
            print("Movimiento: abajo")
            x += 1
        elif action == 2 and y > 0 and self.maze[x][y - 1] != 1:
            print("Movimiento: izquierda")
            y -= 1
        elif action == 3 and y < self.width - 1 and self.maze[x][y + 1] != 1:
            print("Movimiento: derecha")
            y += 1

        self.agent_pos = (x, y)

        # Verifica si la posicion en la que se encuentra es la salida
        if self.maze[x][y] == "S":
            self.done = True

        return self.agent_pos, 0, self.done, {}
```

Listing 3: metodo step

Con este método vamos iterando por lo alto y ancho del laberinto para la búsqueda de la poción inicial del agente, ya que nos adelantamos a que no siempre será la poción $(0,0)$, de esta forma el agente inicia en la poción correcta el laberinto cada vez que se llama a `reset`, además garantiza que que exista un punto de inicio.

```
    # Encuentra la posicion inicial del agente (entrada del laberinto).
    def find_start(self):
        for i in range(self.height):
            for j in range(self.width):
                if self.maze[i][j] == "E":
                    return (i, j)
        raise ValueError("No se pudo encontrar el punto de inicio E en el laberinto")
```

Listing 4: metodo find start

El algoritmo

Ahora si, la carne dentro del código, la salida usando backtracking. el método `solve` comienza por definir los 4 movimientos del agente por las tuplas de las coordenadas

```
# Encuentra la salida usando backtracking
    def solve(env):
        # arriba, abajo, izquierda, derecha
        actions = [(0, -1), (0, 1), (-1, 0), (1, 0)]
```

Listing 5: Movimientos del agente

Ya definidos los movimientos pasamos a verificar que nuestro agente no traspasará muros o hará un recorridos al puro estilo pacman:

- Primero verificamos si las coordenadas están fuera de los límites del laberinto.
- Si no hemos topado con un muro
- Si el agente ya ha pasado por esta celda en la ruta actual `((x, y) in path)`

```
        def backtrack(x, y, path):
            if x < 0 or x >= env.height or y < 0 or y >= env.width or env.maze[x][y] ==
    1 or (x, y) in path:
                return False

            path.append((x, y))
```

Listing 6: Validación de movimientos

Después vamos a verificar si la poción es la salida. Si es así imprimimos que se ha encontrado y se devuelve True para indicar que se ha encontrado la salida.

Recursivamente llamamos a `backtrack` para iterar sobre todos los movimientos posibles en la posición actual. Busca los caminos y si no llega a la salida retrocede con el `path.pop()`. Y como antes lo explicábamos en el método buscamos la poción inicial, la usamos para encontrar la x y y y se las pasamos al método `backtrack`, de esta forma de itera para ir imprimiendo la ruta que siguió el agente o si no hay SOLUCION nos lo hace saber el programa.

```
            if env.maze[x][y] == "S":
                print("Encontre la salida :D")
                return True

            for action in range(env.action_space.n):
                if backtrack(x + actions[action][0], y + actions[action][1], path):
                    return True

            path.pop()
            return False

        start_x, start_y = env.find_start()
        path = []
        if backtrack(start_x, start_y, path):
            print("La ruta es:")
            for i, pos in enumerate(path):
                x, y = pos
                direction = ""
                if i > 0:
                    # Comparar la posicion actual con la anterior para determinar la
    direccion
                    prev_pos = path[i - 1]
                    if x < prev_pos[0]:
                        direction = "arriba"
                    elif x > prev_pos[0]:
                        direction = "abajo"
                    elif y < prev_pos[1]:
                        direction = "izquierda"
                    elif y > prev_pos[1]:
                        direction = "derecha"
                print(f"Posicion: {pos}, Direccion: {direction}")
        else:
            print("iiiiii no encontre la salida")
```

Listing 7: algoritmo backtracking

Vamos a ver más de cerca esta linea: Tenemos una llamada recursiva a `backtrack`, y nos posicionamos en x y y coordenadas actuales donde `actions` hace los movimientos correspondientes al momento de hacer volver a llamar al método le estamos pasamos esas coordenadas y la lista de posiciones visitadas hasta el momento. La condicional verifica la llamada, pues si es True ya se encontró la salida pero si devuelve un False, no ha encontrado la salida y lo que hace es retroceder eliminando la posición de la lista.

```
    if backtrack(x + actions[action][0], y + actions[action][1], path):
```

Listing 8: algoritmo backtracking

Código extra

Como extra hemos añadido 5 laberintos para ejecutar nuestras pruebas

```
    # Laberinto 01 CON SOLUCION
    laberinto01 = [
    ["E", 0, 1, 0,   0, 0, 0, 0],
    [  0, 0, 1, 0,   1, 0, 1, 0],
    [  0, 0, 1, 0, "S", 0, 1, 1],
    [  0, 0, 0, 0,   1, 1, 0, 0],
    [  1, 0, 1, 1,   1, 1, 1, 0],
    [  0, 0, 0, 0,   0, 0, 1, 0],
    [  0, 1, 1, 1,   1, 1, 1, 0],
    [  0, 0, 0, 0,   0, 0, 0, 0]
]
```

Listing 9: laberinto01

```
    # Laberinto 02 CON SOLUCION
    laberinto02 = [
        ["E", 0, 0,   0, 0, 0, 0, 0],
        [  0, 0, 1,   0, 1, 0, 1, 0],
        [  0, 0, 1,   0, 1, 0, 1, 1],
        [  0, 0, 0,   0, 1, 1, 1, 0],
        [  1, 0, 1,   1, 1, 1, 1, 0],
        [  0, 0, 0,   0, 0, 0, 0, 1],
        [  0, 1, 1,   1, 0, 0, 0, 0],
        [  0, 0, 0, "S", 0, 0, 1, 1]
    ]
```

Listing 10: laberinto02

```
    # Laberinto 03 SIN SOLUCION
    # No hay camino
    laberinto03 = [
        ["E", 1],
        [1, "S"]
    ]
```

Listing 11: laberinto03

```
    # Laberinto 04 SIN SOLUCION
    # No hay salida
    laberinto04 = [
        ["E", 0, 0, 0, 0, 0, 0, 0],
        [  0, 0, 1, 0, 1, 0, 1, 0],
        [  0, 0, 1, 0, 1, 0, 1, 1],
        [  0, 0, 0, 0, 1, 1, 1, 0],
        [  1, 0, 1, 1, 1, 1, 1, 0],
        [  0, 0, 0, 0, 0, 0, 0, 1],
        [  0, 1, 1, 1, 0, 0, 0, 0],
        [  0, 0, 0, 1, 0, 0, 1, 1]
    ]
```

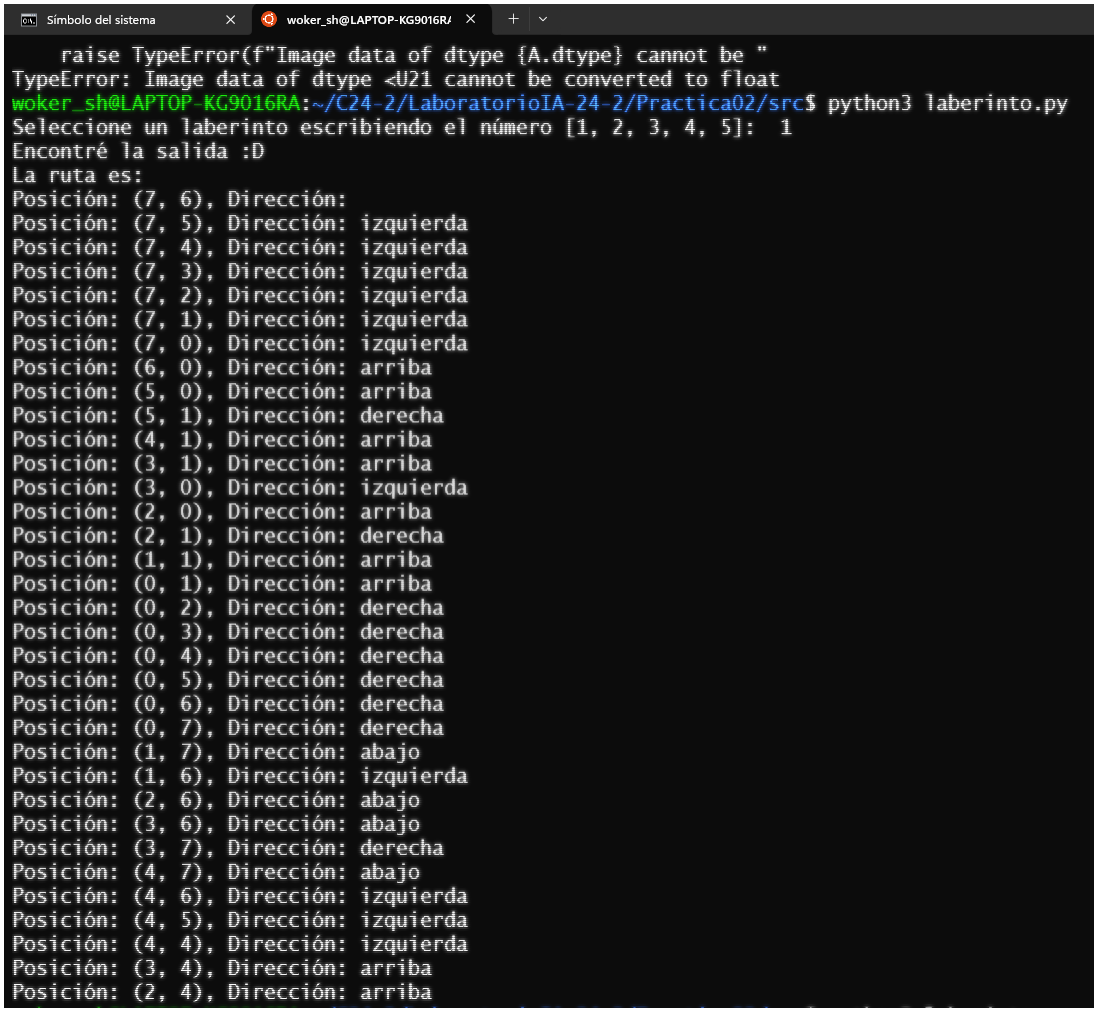
Listing 12: laberinto04

```
    # Laberinto 05 SIN SOLUCION
    # No hay entrada
    laberinto05 = [
        [ 1, 0, 1, 0,   0, 0, 0, 0],
        [ 0, 0, 1, 0,   1, 0, 1, 0],
        [ 0, 0, 1, 0, "S", 0, 1, 1],
        [ 0, 0, 0, 0,   1, 1, 0, 0],
        [ 1, 0, 1, 1,   1, 1, 1, 0],
        [ 0, 0, 0, 0,   0, 0, 1, 0],
        [ 0, 1, 1, 1,   1, 1, 1, 0],
        [ 0, 0, 0, 0,   0, 0, 0, 0]
    ]
```

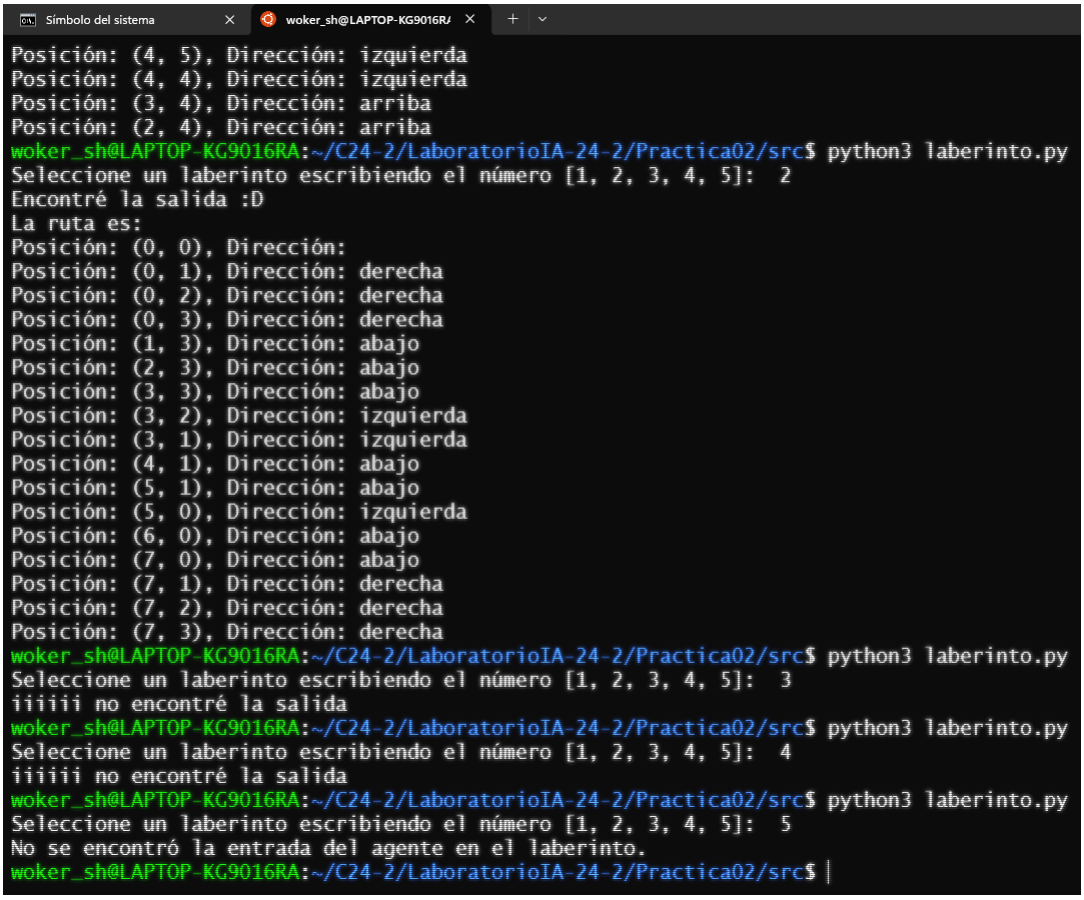
Listing 13: laberinto05

Pruebas en terminal

Ejecución laberinto01



Ejecución laberinto02 - laberinto05



Hemos realizado pruebas con dos entradas y dos salidas dentro del laberinto, los resultados son que si hay solución nos devuelve el recorrido que encuentre primero, es decir, la entrada que encuentre primero es la que toma así como la salida.

Referencias

[KT05] John Kleinberg y Eva Tardos. *Algorithm Algorithm Design*. Addison Addison-Wesley, 2005.

[Fig12] Jhosimar George Arias Figueroa. *Algorithms and More*. https://jariasf.wordpress.com/2012/03/02/algoritmo-de-busqueda-depth-first-search-parte-1/. 2012.

[RN16] Stuart Russell y Peter Norvig. *Inteligencia Artificial Un Enfoque moderno*. 2nd. Pearson Prentice Hall, 2016.

[Pan21] Thanakorn Panyapiang. *Developing Reinforcement Learning Environment Using OpenAI Gym*. https://medium.com/geekculture/developing-reinforcement-learning-environment-using-openai-gym-f510b0393eb7. 2021.

[RAN21] ASHISH RANA. *Introduction: Reinforcement Learning with OpenAI Gym*. https://towardsdatasci
com/reinforcement-learning-with-openai-d445c2c687d2. 2021.

[Ref21] Programación Refactoriza. *Backtracking*. https://docs.jjpeleato.com/algoritmia/backtracking. 2021.

[Edu] Education-wiki. *DFS Algorithm*. https://es.education-wiki.com/7857184-dfs-algorithm.

[ope] openai. *gym*. https://github.com/openai/gym?tab=readme-ov-file.

[Tec] Techiedelight. *Búsqueda en amplitud (BFS): implementación iterativa y recursiva*. https://www.techiedelight.com/es/breadth-first-search/.