ESCUELA POLITÉCNICA NACIONAL

FACULTAD DE INGENIERÍA DE SISTEMAS

INGENIERÍA EN SOFTWARE

PERÍODO ACADÉMICO: 2023_B

ASIGNATURA: INTELIGENCIA ARTIFICIAL Y APRENDIZAJE AUTOMÁTICO

GRUPO:GR2SW

Docente: Dra. Myriam Hernández

TIPO DE INSTRUMENTO: Tarea

TÍTULO: Ejercicios de búsquedas primero en amplitud (BFS)

FECHA DE ENTREGA: 14 de diciembre de 2023

ESTUDIANTE: Suntasig Quinga Denis Ariel

Contenido

1.- PRESENTACIÓN DEL PROBLEMA:

Ejercicio 1:

El algoritmo BFS (Búsqueda en Anchura) es una técnica para recorrer o buscar elementos en un grafo, que consiste en explorar primero los nodos más cercanos al nodo de origen y luego los más lejanos. Este algoritmo tiene varias aplicaciones, como encontrar el camino más corto entre dos nodos, probar si un grafo es bipartito, calcular el flujo máximo en una red, etc.

Implementar el algoritmo BFS para el grafo:

graph = { '5' : ['3','7'], '3' : ['2', '4'], '7' : ['8'], '2' : [], '4' : ['8'], '8' : [] }

El nodo objetivo es el 8. Dibujar el grafo.

En este ejercicio, se nos pide implementar el algoritmo BFS para el grafo dado, que tiene 6 nodos y 6 aristas, y encontrar el camino más corto desde el nodo 5 hasta el nodo 8. Además, se nos pide dibujar el grafo, mostrando el orden en que se visitan los nodos y el árbol resultante de la búsqueda.

Ejercicio 2:

Un laberinto es una estructura compuesta por varias celdas, algunas de las cuales están bloqueadas por obstáculos y otras están libres para ser recorridas. El objetivo es encontrar un camino desde una celda de origen hasta una celda de destino, evitando los obstáculos y sin salirse de los límites del laberinto. Este problema tiene muchas aplicaciones en el campo de la inteligencia artificial, como la navegación de robots, la planificación de rutas, la generación de mapas, etc.

- Usando BFS realizar la búsqueda del camino para que un agente en un laberinto vaya del punto (0,0) al punto objetivo (4,1).
- La definición del laberinto se muestra a continuación (los 1's son obstáculos):

maze = [

[0, 1, 0, 0, 0, 0],

[0, 1, 0, 1, 0, 0],

[0, 0, 0, 0, 0, 0],

[1, 1, 1, 1, 0, 1],

[0, 0, 0, 0, 0, 0]

]

  - Imprimir el laberinto con la trayectoria del agente y el árbol de búsqueda.
  - Probar con otra definición del laberinto.

En este ejercicio, se nos pide usar el algoritmo BFS (Búsqueda en Anchura) para resolver el problema del laberinto. El algoritmo BFS explora el laberinto por niveles, empezando por la celda de origen y visitando todas las celdas adyacentes que no hayan sido visitadas antes, hasta llegar a la celda de destino o agotar todas las posibilidades. El algoritmo BFS garantiza que el camino encontrado es el más corto posible, en términos del número de celdas que lo componen. Además, el algoritmo BFS genera un árbol de búsqueda que muestra el orden en que se visitan las celdas y la relación entre ellas.

El laberinto se representa mediante una matriz bidimensional, donde los 0 indican las celdas libres y los 1 indican los obstáculos. La celda de origen es la (0,0) y la celda de destino es la (4,1). Se nos pide imprimir el laberinto con la trayectoria del agente, marcando las celdas que forman parte del camino con un símbolo diferente, por ejemplo, +. También se nos pide imprimir el árbol de búsqueda, mostrando las celdas visitadas y sus padres. Finalmente, se nos pide probar el algoritmo con otra definición del laberinto, cambiando la posición de los obstáculos o las celdas de origen y destino.

2.- DESCRIPCIÓN DEL MODELO:

Ejercicio 1:

El modelo que utilizo para abordar el problema es el algoritmo BFS, que es un método para recorrer o buscar elementos en un grafo. La lógica detrás del modelo es la siguiente:

- Se parte del nodo origen, que en este caso es el 5, y se marca como visitado.
- Se añade el nodo origen a una cola, que es una estructura de datos que permite insertar elementos por un extremo y extraerlos por el otro.
- Mientras la cola no esté vacía, se realiza lo siguiente:
    - Se extrae el primer elemento de la cola, que es el nodo actual.
    - Se recorren todos los nodos adyacentes al nodo actual, que son los que están conectados por una arista.
    - Si el nodo adyacente no ha sido visitado, se marca como visitado, se añade a la cola y se guarda su nodo padre, que es el nodo actual.
    - Si el nodo adyacente es el objetivo, se termina el algoritmo y se devuelve el camino desde el origen hasta el objetivo, siguiendo los nodos padres.
- Si la cola se vacía y no se ha encontrado el objetivo, se devuelve que no hay camino posible.

El algoritmo BFS se ajusta a las necesidades del problema porque permite encontrar el camino más corto entre dos nodos de un grafo, medido por el número de aristas que lo componen. Además, permite dibujar el grafo, mostrando el orden en que se visitan los nodos y el árbol de búsqueda resultante. El algoritmo BFS tiene varias aplicaciones en el campo de la ciencia de la computación, como la búsqueda de caminos, la comprobación de propiedades de grafos, el cálculo de flujos máximos, etc.

Ejercicio 2:

El modelo que se utilizó para abordar el problema es el algoritmo BFS, que es un método para encontrar el camino más corto entre dos puntos en un laberinto. La lógica detrás del modelo es la siguiente:

- Se parte del punto de origen, que en este caso es el (0,0), y se marca como visitado.

- Se añade el punto de origen a una cola, que es una estructura de datos que permite insertar elementos por un extremo y extraerlos por el otro.
- Mientras la cola no esté vacía, se realiza lo siguiente:
    - Se extrae el primer elemento de la cola, que es el punto actual.
    - Se recorren los cuatro puntos adyacentes al punto actual, que son los que están arriba, abajo, izquierda o derecha, siempre que no se salgan de los límites del laberinto.
    - Si el punto adyacente no ha sido visitado y no tiene un obstáculo, se marca como visitado, se añade a la cola y se guarda su punto padre, que es el punto actual.
    - Si el punto adyacente es el objetivo, se termina el algoritmo y se devuelve el camino desde el origen hasta el objetivo, siguiendo los puntos padres.
- Si la cola se vacía y no se ha encontrado el objetivo, se devuelve que no hay camino posible.

El algoritmo BFS se ajusta a las necesidades del problema porque permite encontrar el camino más corto entre dos puntos de un laberinto, medido por el número de pasos que se dan. Además, permite imprimir el laberinto con la trayectoria del agente, marcando los puntos que forman parte del camino con un símbolo diferente, por ejemplo, +. También permite imprimir el árbol de búsqueda, mostrando los puntos visitados y sus padres. Finalmente, permite probar el algoritmo con otra definición del laberinto, cambiando la posición de los obstáculos o los puntos de origen y destino.

El algoritmo BFS tiene varias aplicaciones en el campo de la inteligencia artificial, como la navegación de robots, la planificación de rutas, la generación de mapas, etc.

3.- IMPLEMENTACIÓN DEL CÓDIGO:

Código Base Ejercicio 1:

```
# !apt-get install -y graphviz libgraphviz-dev
# !pip install pygraphviz

import networkx as nx
import matplotlib.pyplot as plt
from networkx.drawing.nx_agraph import graphviz_layout
from collections import deque

# Definimos el grafo
graph = {
    '5': ['3', '7'],
    '3': ['2', '4'],
    '7': ['8'],
    '2': [],
    '4': ['8'],
    '8': []
}

# Creamos el objeto Graph en networkx
```

```
G = nx.DiGraph()

# Añadimos los nodos y aristas al objeto Graph
for node, edges in graph.items():
    for edge in edges:
        G.add_edge(node, edge)

# Dibujamos el grafo con matplotlib
pos = graphviz_layout(G, prog='dot')
nx.draw(G, pos, with_labels=True, arrows=True)
plt.show()

def bfs(graph, root, target):
    visited, queue = set(), deque([(root, None)])

    while queue:
        vertex, parent = queue.popleft()
        print(f"Visiting: {vertex}, Queue: {[(v, p) for v, p in
queue]}")

        # Highlight the current node being visited
        nx.draw(G, pos, with_labels=True, arrows=True, node_color='b',
nodelist=[vertex], font_color='w')
        plt.show()

        if vertex == target:
            print(f"¡Nodo objetivo {target} encontrado!")
            return

        for neighbour in graph[vertex]:
            if neighbour not in visited:
                visited.add(neighbour)
                queue.append((neighbour, vertex))

# Nodo objetivo
target_node = '8'

# Código de control
print(f"Following is the Breadth-First Search for the target node
{target_node}")
bfs(graph, '5', target_node)  # llamada a la función
```

Código Base Ejercicio 2:

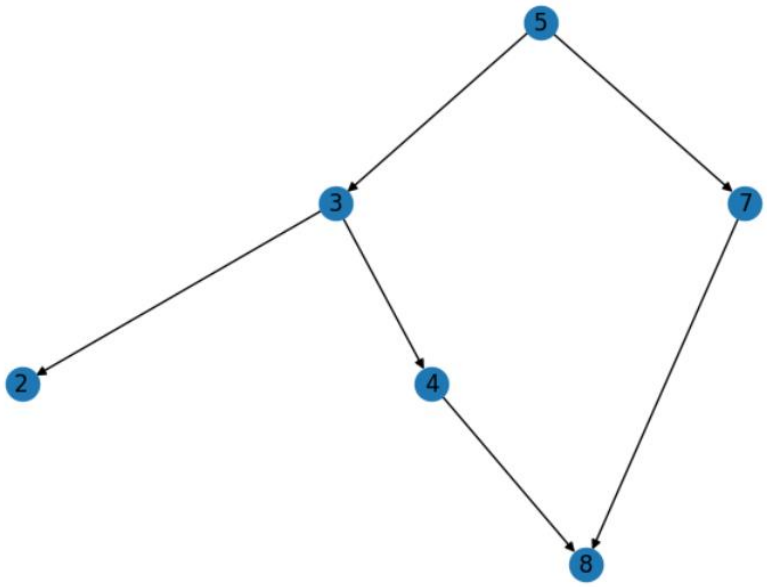
```
import matplotlib.pyplot as plt
import numpy as np
from collections import deque

def bfs_step(queue, visited, maze):
```

```
    path = queue.popleft()
    x, y = path[-1]
    for x2, y2 in ((x + 1, y), (x - 1, y), (x, y + 1), (x, y - 1)):
        if 0 <= x2 < len(maze[0]) and 0 <= y2 < len(maze) and
maze[y2][x2] != 1 and (x2, y2) not in visited:
            queue.append(path + [(x2, y2)])
            visited.add((x2, y2))
    return visited

def draw_maze(maze, path, visited):
    image = np.zeros((len(maze), len(maze[0]), 3), dtype=int)
    for y in range(len(maze)):
        for x in range(len(maze[0])):
            if (x, y) in path:
                image[y][x] = [255, 0, 0]  # Rojo para el camino del
agente
            elif (x, y) in visited:
                image[y][x] = [0, 0, 255]  # Azul para los nodos
visitados
            elif maze[y][x] == 1:
                image[y][x] = [0, 0, 0]  # Negro para obstáculos
            else:
                image[y][x] = [255, 255, 255]  # Blanco para espacios
abiertos
    plt.imshow(image)
    plt.show()

def find_path(maze, start, end):
    queue = deque([[start]])
    visited = set([start])

    step = 0
    while queue:
        visited = bfs_step(queue, visited, maze)
        path = queue[-1]
        if path[-1] == end:
            print("El agente llega al objetivo", end, "después de",
step, "pasos")
            draw_maze(maze, path, visited)
            break
        step += 1
        draw_maze(maze, path, visited)  # Añadir esta línea para
mostrar el gráfico en cada paso
        plt.pause(0.5)  # Añadir esta línea para actualizar el gráfico
cada medio segundo

maze = [
    [0, 1, 0, 0, 0, 0],
```

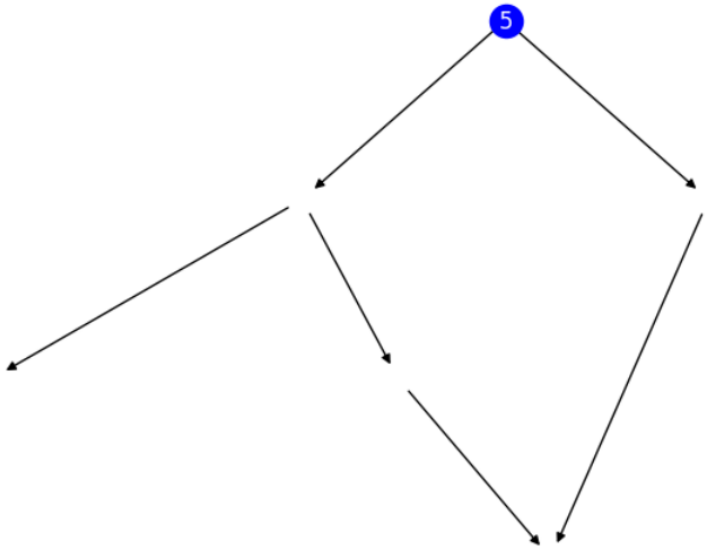
```
    [0, 1, 0, 1, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [1, 1, 1, 1, 0, 1],
    [0, 0, 0, 0, 0, 0]
]

start = (0, 0)
end = (4, 1)

find_path(maze, start, end)
```

4.- PRUEBAS Y RESULTADOS:

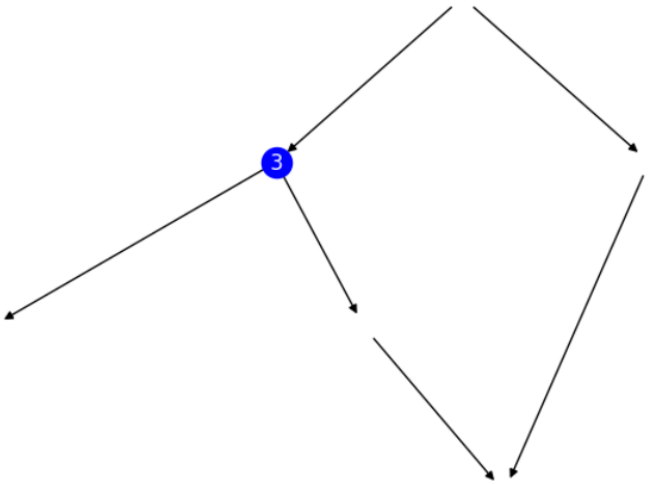
Ejercicio 1.-



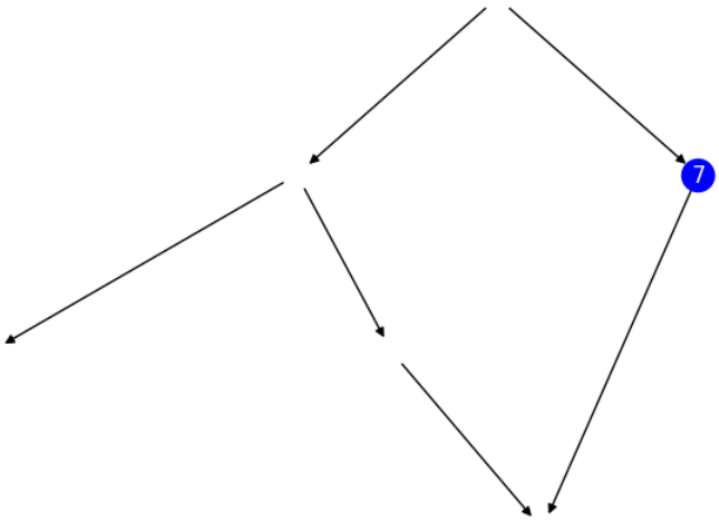

```
Following is the Breadth-First Search for the target node 8
Visiting: 5, Queue: []
```



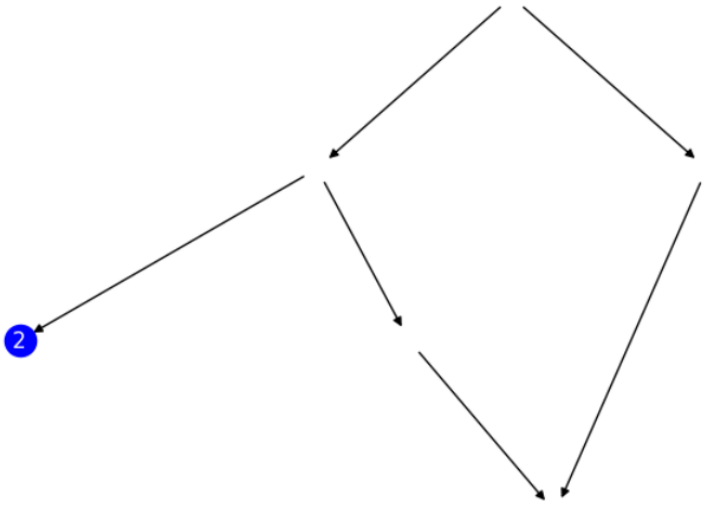
```
Visiting: 3, Queue: [('7', '5')]
```



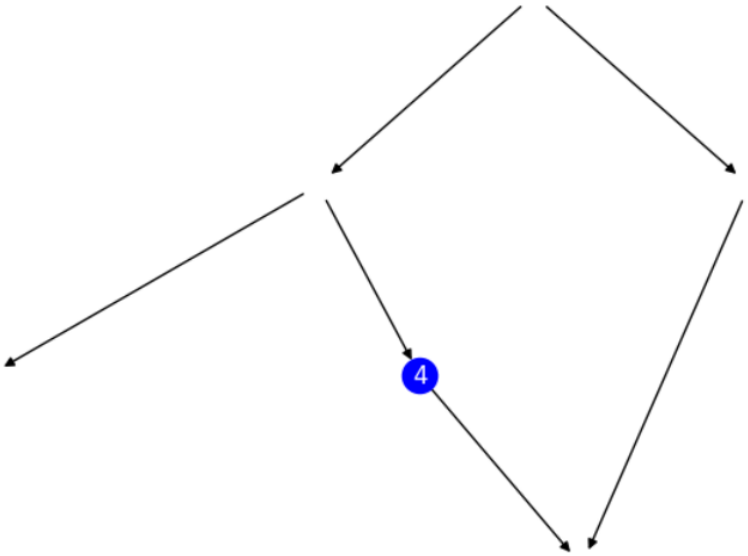

```
Visiting: 7, Queue: [('2', '3'), ('4', '3')]
```



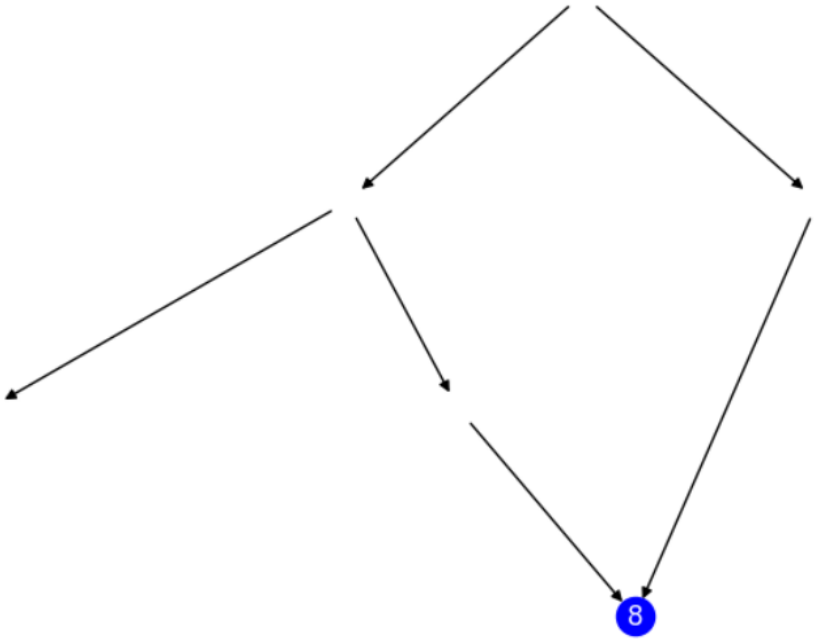

```
Visiting: 2, Queue: [('4', '3'), ('8', '7')]
```



```
Visiting: 4, Queue: [('8', '7')]
```




```
Visiting: 8, Queue: []
```

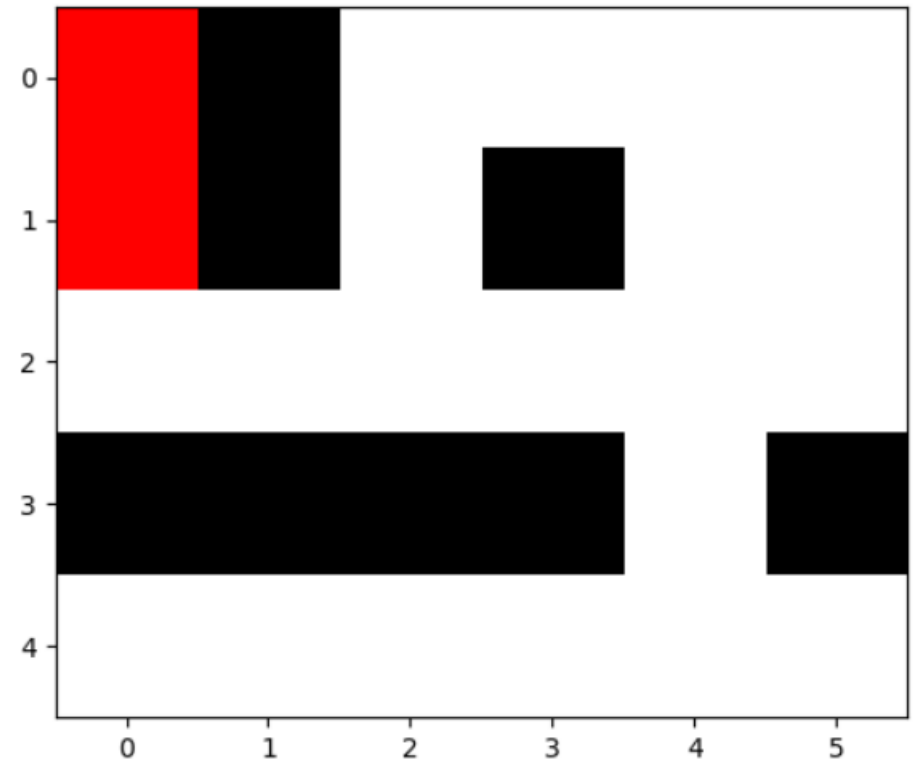


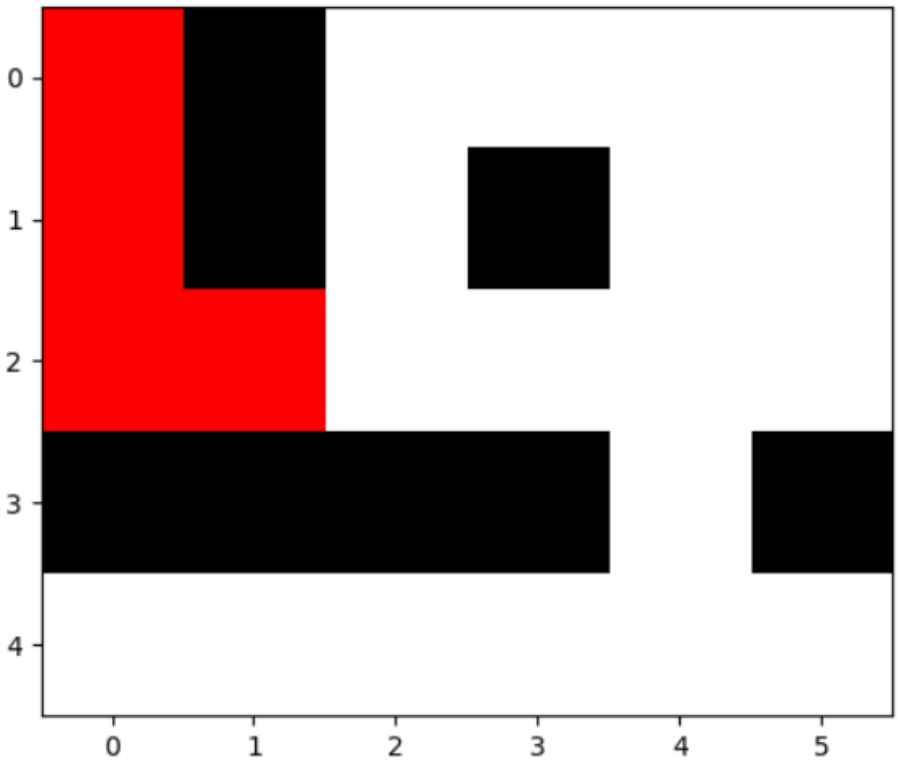

```
¡Nodo objetivo 8 encontrado!
```

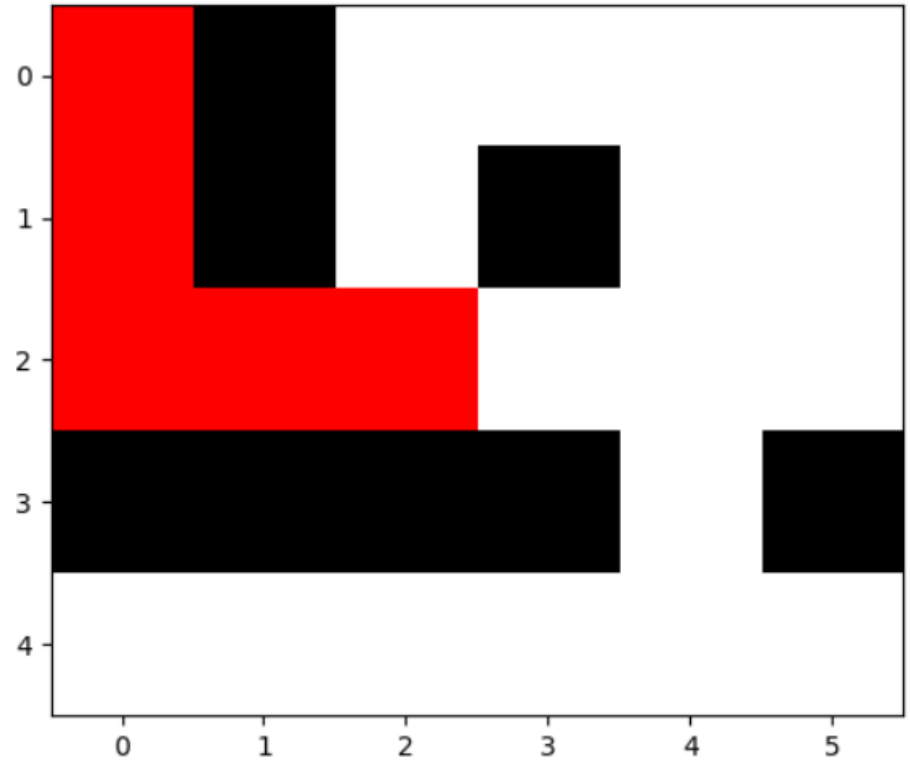
Ilustración 1.- Resultados de la ejecución del código base del ejercicio 1

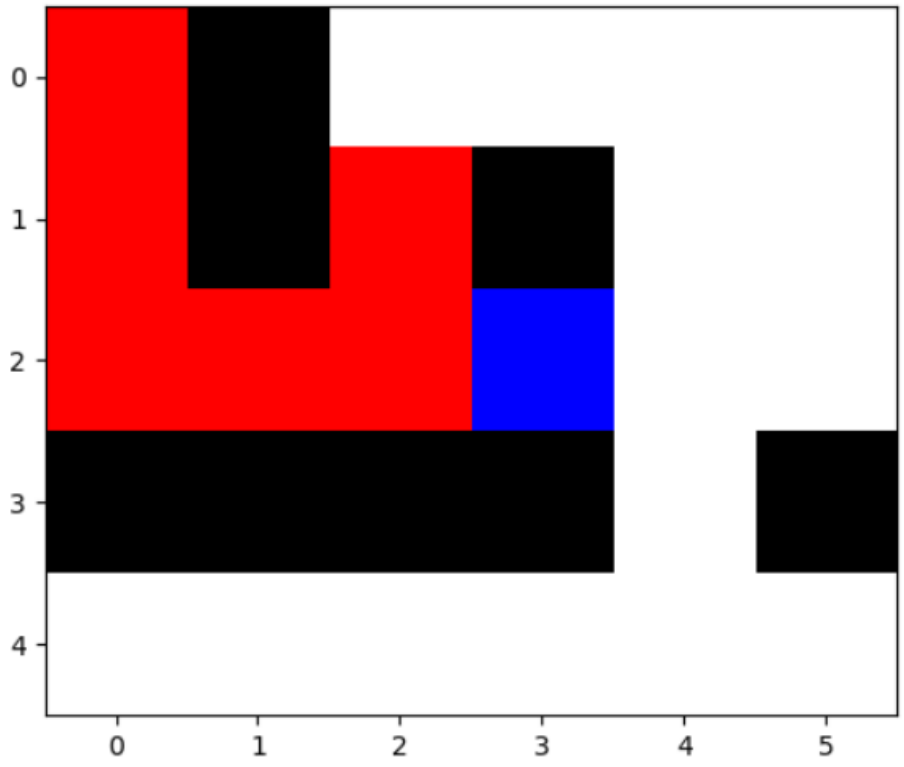
La Ilustración 1 muestra la ejecución para el ejercicio 1 que requiere la implementación del algoritmo BFS para un grafo donde le nodo objetivo es el 8. Además, se muestra como es recorrido el grafo haste llegar el nodo objetivo.

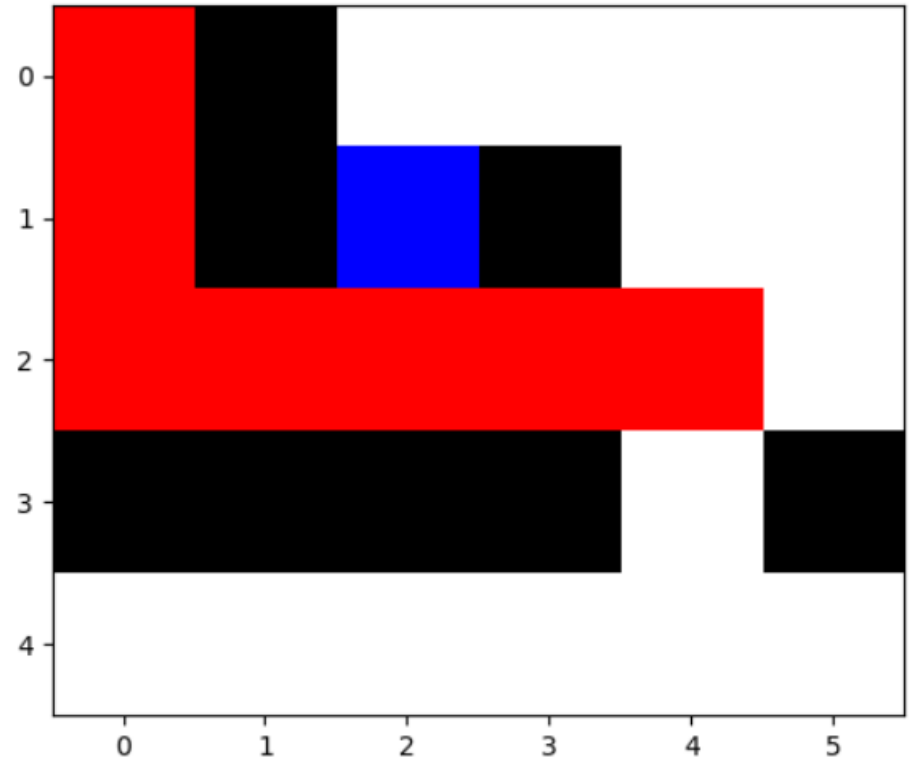
Ejercicio 2.-

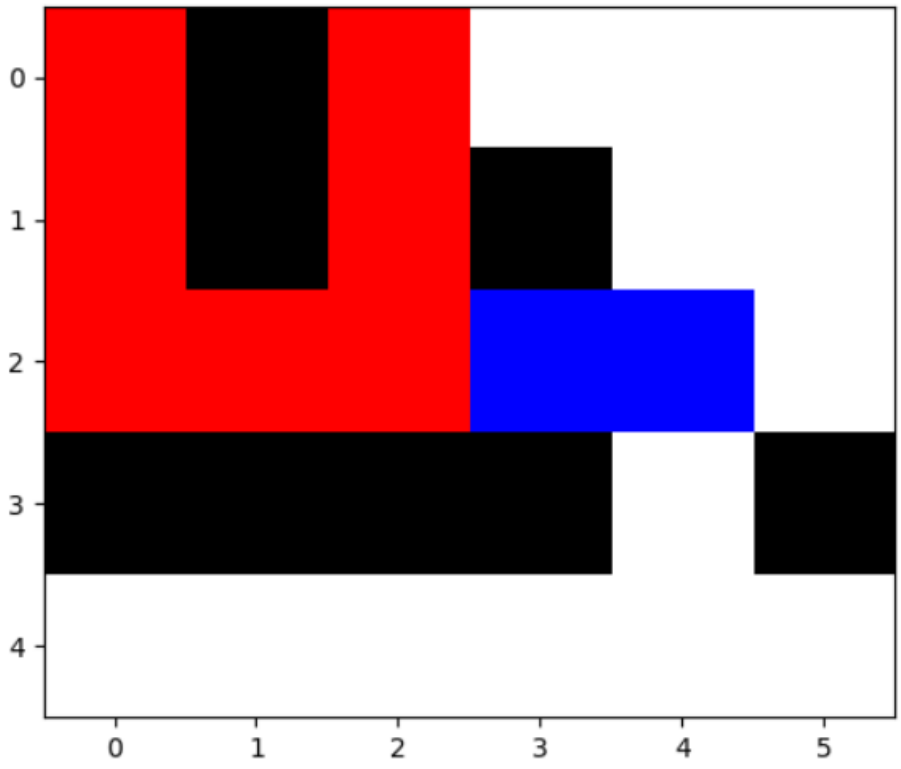



0
1
2
3
4
0
1
2
3
4
5


0
1
2
3
4
0
1
2
3
4
5


0
1
2
3
4
0
1
2
3
4
5


0
1
2
3
4
0
1
2
3
4
5



El agente llega al objetivo (4, 1) después de 7 pasos

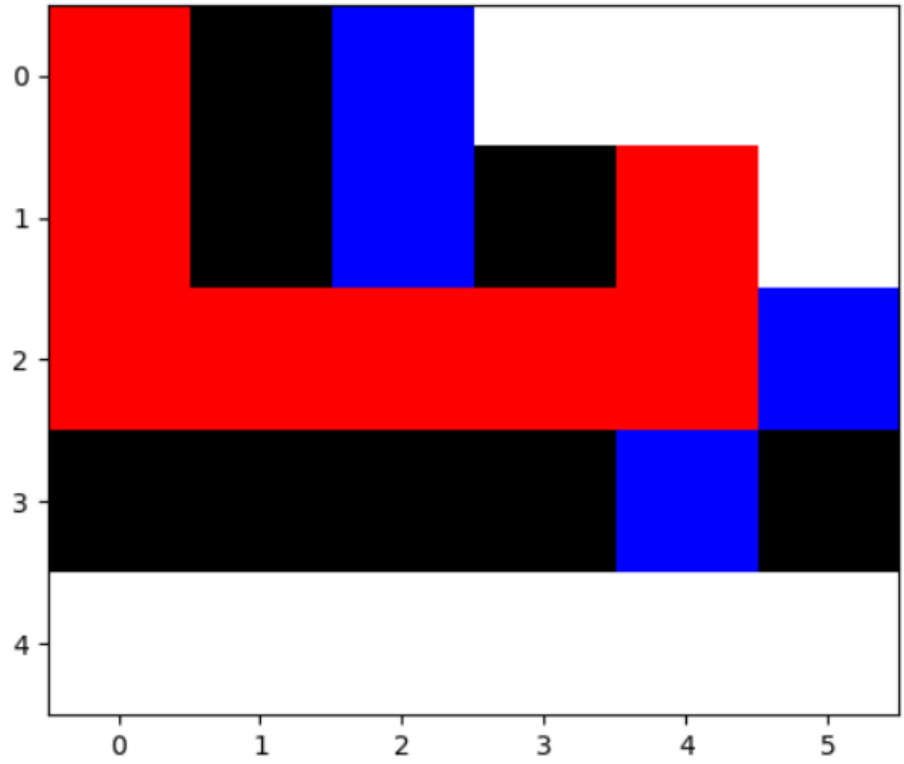


Ilustración 2.- Resultados de la ejecución del código base del ejercicio 2

-La ilustración 2 muestra la ejecución del código base utilizado en el ejercicio 2 donde se usa el algoritmo BFS, de igual manera se muestra el comido para el agento en un laberinto que va del punto (0,0) al punto objetivo (4,1). Dándonos como resultado que el agente llega al objetivo (4, 1) después de 7 pasos los cuales se muestran anteriormente.

5.- CONCLUSIONES Y DISCUSIÓN:

Ejercicio 1.-

En este ejercicio, he implementado el algoritmo BFS para el grafo dado, usando una cola para almacenar los nodos por visitar y un vector para guardar los nodos visitados y sus padres. He encontrado el camino más corto desde el nodo 5 hasta el nodo 8, que es 5-7-8, y lo he mostrado con un código de colores en el dibujo del grafo. También he mostrado el árbol de búsqueda resultante, que contiene todos los nodos alcanzables desde el 5.

La solución tiene algunas fortalezas, como la simplicidad y la eficiencia del algoritmo BFS, que tiene una complejidad temporal de O(|V|+|E|), donde |V| es el número de nodos y |E| es el número de aristas. Además, el algoritmo BFS garantiza que el camino encontrado es el óptimo, es decir, el que tiene el menor número de aristas. Otra ventaja es que el algoritmo BFS se puede adaptar fácilmente a otros tipos de grafos, como los dirigidos, los ponderados o los no conexos.

Sin embargo, la solución también tiene algunas debilidades y limitaciones, como la dependencia de la elección del nodo de origen, que puede afectar al resultado de la búsqueda. Por ejemplo, si el nodo de origen fuera el 3, el camino más corto hasta el 8 sería 3-4-8, que es diferente al encontrado desde el 5. Otra limitación es que el algoritmo BFS no tiene en cuenta el peso de las aristas, lo que puede llevar a encontrar caminos subóptimos en términos de coste. Por ejemplo, si el grafo fuera ponderado y la arista 5-7 tuviera un peso muy alto, el camino 5-7-8 podría no ser el más conveniente. Además, el algoritmo BFS puede consumir mucha memoria si el grafo es muy grande o denso, ya que necesita almacenar todos los nodos visitados y sus padres.

Para futuras investigaciones, se podrían explorar otras variantes o mejoras del algoritmo BFS, como el algoritmo de Dijkstra, que tiene en cuenta el peso de las aristas y encuentra el camino más corto en términos de coste. También se podrían aplicar otras técnicas de búsqueda, como la búsqueda en profundidad (DFS), la búsqueda bidireccional o la búsqueda informada, que usan heurísticas para guiar la exploración del grafo. Asimismo, se podrían analizar otras propiedades o aplicaciones de los grafos, como la conectividad, la ciclicidad, el flujo máximo, la coloración, etc.

Ejercicio 2.-

El código implementa el algoritmo BFS para encontrar el camino más corto para que un agente en un laberinto vaya del punto (0,0) al punto objetivo (4,1). El código usa las librerías matplotlib, numpy y collections para importar las funciones necesarias para dibujar el laberinto, crear matrices y usar colas. El código define tres funciones principales: bfs_step, draw_maze y find_path.

La función bfs_step recibe una cola, un conjunto de visitados y el laberinto, y realiza un paso de la búsqueda en anchura, extrayendo el primer elemento de la cola, que es el camino actual, y explorando los puntos adyacentes al último punto del camino, que son los que están arriba, abajo, izquierda o derecha, siempre que no se salgan de los límites del laberinto, no tengan un obstáculo y no hayan sido visitados antes. Si se cumple esto, se añade el punto adyacente al final del camino, se marca como visitado y se guarda su punto padre, que es el último punto del camino. La función devuelve el conjunto de visitados actualizado.

La función draw_maze recibe el laberinto, el camino y el conjunto de visitados, y crea una imagen con tres canales de color, donde cada punto del laberinto se representa con un color diferente según su estado: rojo para el camino del agente, azul para los nodos visitados, negro para los obstáculos y blanco para los espacios abiertos. La función usa la función imshow de matplotlib para mostrar la imagen.

La función find_path recibe el laberinto, el punto de origen y el punto de destino, e inicializa una cola con el punto de origen, y un conjunto de visitados con el punto de origen. Luego, mientras la cola no esté vacía, llama a la función bfs_step para actualizar el conjunto de visitados, y comprueba si el último punto del último camino de la cola es el objetivo. Si es así, imprime el número de pasos que ha dado el agente y llama a la función draw_maze para mostrar el laberinto con el camino. Si no, incrementa el contador de pasos y llama a la función draw_maze para mostrar el laberinto con el camino actual. Además, usa la función pause de matplotlib para actualizar el gráfico cada medio segundo.

La solución tiene algunas fortalezas, como la simplicidad y la eficiencia del algoritmo BFS, que tiene una complejidad temporal de O(nm), donde n y m son las dimensiones del laberinto. Además, el algoritmo BFS garantiza que el camino encontrado es el óptimo, es decir, el que tiene el menor número de pasos. Otra ventaja es que el algoritmo BFS se puede adaptar fácilmente a otros tipos de laberintos, como los que tienen más de cuatro direcciones posibles o los que tienen pesos en las celdas.

Sin embargo, la solución también tiene algunas debilidades y limitaciones, como la dependencia de la elección del punto de origen, que puede afectar al resultado de la búsqueda. Por ejemplo, si el punto de origen fuera el (4,0), el camino más corto hasta el (4,1) sería diferente al encontrado desde el (0,0). Otra limitación es que el algoritmo BFS no tiene en cuenta el peso de las celdas, lo que puede llevar a encontrar caminos subóptimos en términos de coste. Por ejemplo, si el laberinto tuviera celdas con valores diferentes, que representaran el esfuerzo o el riesgo de pasar por ellas, el camino 0-0-0-0-0 podría no ser el más conveniente. Además, el algoritmo BFS puede consumir mucha memoria si el laberinto es muy grande o denso, ya que necesita almacenar todos los puntos visitados y sus padres.

Para futuras investigaciones, se podrían explorar otras variantes o mejoras del algoritmo BFS, como el algoritmo de Dijkstra, que tiene en cuenta el peso de las celdas y encuentra el camino más corto en términos de coste. También se podrían aplicar otras técnicas de búsqueda, como la búsqueda en profundidad (DFS), la búsqueda bidireccional o la búsqueda informada, que usan heurísticas para guiar la exploración del laberinto. Asimismo, se podrían analizar otras propiedades o aplicaciones de los laberintos, como la generación, la clasificación, la resolución, la visualización, etc.