UNIVERSIDAD DEL VALLE
FACULTAD DE INGENIERÍA
Escuela de Ingeniería de Sistemas y Computación
Curso: Inteligencia Artificial
Docentes: Oscar Bedoya
Integrantes: Johan Sebastían Tombe Campo, César Alejandro Grijalba Zúñiga & Gilberth Banguero

---

Proyecto 1

Algoritmo búsqueda por amplitud

La búsqueda en amplitud (BFS) es un algoritmo que permite atravesar o buscar estructuras de datos de árboles o grafos. Comienza en la raíz del árbol o algún nodo arbitrario de un grafo, y explora primero los nodos vecinos antes de pasar a los vecinos del siguiente nivel. En otras palabras, la búsqueda en amplitud explora los vértices en el orden de su distancia desde el vértice de origen, donde la distancia es la longitud mínima de un camino desde el vértice de origen hasta el nodo.

En este caso no se utiliza heurística ya que no se considera el costo del camino, si no que se explora el grafo por niveles.

A continuación, se muestra la implementación del algoritmo de búsqueda por amplitud

Python

```python
def bfs(maze, start, goals):
    queue = deque()
    way = []
    visited = set()
    cont_goals = 0
    goal_one = goals[0]
    goal_two = goals[1]
    queue.append(start)
    while queue:
        print("\n")
        n = queue.popleft()
        print(f"Popping {n}"+'='+str(maze[n[0]][n[1]]))
```

```
        way.append(n)
        print(f"Is {n} my goal?")
        if (n == goal_one) or (n == goal_two):
            print("DONE!")
            cont_goals = cont_goals+1
        if cont_goals == 2:
            print(f"Way: {way}")
            return way
        print(f"No.")
        next_steps = find_next(n, maze)
        print(f"Next Steps: {next_steps}")
        for x in next_steps:
            print(f"Evaluating {x}")
            if x in visited:
                print(f"{x} already visited, skipping...")
                continue
            visited.add(x)
            queue.append(x)
            print(f"Visited nodes: {visited}")
            print(f"Queue: {queue}")
```

La función recibe una matriz **maze** una posición inicial **start** y una lista de posiciones objetivo **goals**. La función devuelve una lista way que contiene el camino desde start hasta el primer objetivo, seguido del camino desde el primer objetivo hasta el segundo objetivo, si ambos objetivos son alcanzables desde start. Si uno de los objetivos no es alcanzable, la función devuelve el camino al objetivo alcanzado. La cola **queue** almacena los nodos que están pendientes de exploración y la variable **visited** almacena los nodos que ya han sido explorados.
La función **find_next (no mostrada en el código)** devuelve una lista de las posiciones vecinas a una posición dada en la matriz maze.

Algoritmo de búsqueda por profundidad

La búsqueda en profundidad es un algoritmo de búsqueda no informada que consiste en visitar sucesivamente el primer hijo no visitado del nodo actual, y si no hay más hijos por

visitar, volver al nodo padre y continuar con el siguiente hijo no visitado. Este proceso se repite hasta encontrar el nodo objetivo o hasta recorrer todo el árbol sin éxito.

La heurística de búsqueda en profundidad se refiere a técnicas que se pueden utilizar para reducir el árbol de búsqueda y mejorar la eficiencia de la búsqueda en profundidad en la práctica. La heurística es un método para resolver problemas que no garantiza la solución, pero que en general funciona bien.

En este caso mejoramos la eficiencia evitando que entre en ciclos esto se logra llevasndo un registro de los nodos visitados, asi, no se volveran a visitar los nodos que nos puedan generar ciclos.

A continuación, se muestra la implementación del algoritmo de búsqueda por profundidad

Python

```python
def dfs(maze, start, goals):
    print("longitud de maze"+str(len(maze)))
    stack = []
    way =[]
    visited = set()
    cont_goals=0
    goal_one = goals[0]
    goal_two = goals[1]
    stack.append(start)
    while stack:
        print("\n")
        n = stack.pop()
        print(f"Popping {n}"+'='+str(maze[n[0]][n[1]]))
        way.append(n)
        print(f"Is {n} my goal?")
        if (n == goal_one) or (n == goal_two):
            print("DONE!")
            cont_goals = cont_goals+1
        if cont_goals == 2:
            print(f"Way: {way}")
            way_global=way
            return way
```

```
        print(f"No.")
        next_steps = find_next(n, maze)
        print(f"Next Steps: {next_steps}")
        for x in next_steps:
            print(f"Evaluating {x}")
            if x in visited:
                print(f"{x} already visited, skipping...")
                continue
            visited.add(x)
            stack.append(x)
            print(f"Visited nodes: {visited}")
            print(f"Stack: {stack}")
```

La función recibe una matriz **maze** una posición inicial **start** y una lista de posiciones objetivo **goals**. La función devuelve una lista way que contiene el camino desde start hasta el primer objetivo, seguido del camino desde el primer objetivo hasta el segundo objetivo, si ambos objetivos son alcanzables desde start. Si uno de los objetivos no es alcanzable, la función devuelve el camino al objetivo alcanzado. La pila **stack** almacena los nodos que están pendientes de exploración y la variable **visited** almacena los nodos que ya han sido explorados.
La función **find_next (no mostrada en el código)** devuelve una lista de las posiciones vecinas a una posición dada en la matriz maze.

Bibliografía

- https://es.wikipedia.org/wiki/B%C3%BAsqueda_en_profundidad
- https://www.techiedelight.com/es/breadth-first-search/
- https://www.cs.us.es/cursos/ia1/temas/tema-04.pdf