



UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

Práctica No. 1

ALUMNOS

López Diego Gabriela - 318243485
San Martín Macías Juan Daniel - 318181637
Martínez Hidalgo Paola Mildred - 319300217

PROFESOR

Mauricio Riva Palacio Orozco

AYUDANTES

Alan Alexis Martínez López
Yael Antonio Calzada Martín

ASIGNATURA

COMPUTACIÓN DISTRIBUIDA

Fecha de entrega: 03 de Septiembre del 2024

Introducción

En esta práctica se aborda la implementación del algoritmo de búsqueda en anchura (**Breadth-First Search, BFS**) aplicado a gráficas. Este algoritmo es fundamental en el campo de la computación distribuida, ya que permite explorar de manera eficiente todos los nodos de una gráfica, asegurando que se visiten todos los vértices alcanzables desde un nodo inicial.

La práctica se divide en dos partes principales: la primera consiste en la implementación estándar del algoritmo BFS para recorrer una gráfica dada; la segunda, que otorga un punto extra, permite al usuario seleccionar un nodo de inicio específico desde el cual se realizará el recorrido, proporcionando mayor flexibilidad y aplicabilidad en escenarios más complejos.

Primera Implementación

En éste primer entregable (*Script BFS.py*) implementamos el algoritmo de búsqueda en anchura (BFS) para recorrer un árbol representado como una gráfica. El propósito principal de este código es demostrar cómo BFS explora todos los nodos de una gráfica conectada de manera sistemática, garantizando que cada nodo se visite exactamente una vez.

La siguiente gráfica se representa mediante un diccionario en Python, donde las llaves son los nodos y los valores son listas de nodos adyacentes (vecinos). Por ejemplo, el nodo 'A' tiene como vecinos a 'B', 'C', 'D', y 'E'. Continuando con esta lógica representamos la gráfica dada (figura 1) como lo muestra el código de la figura 2.

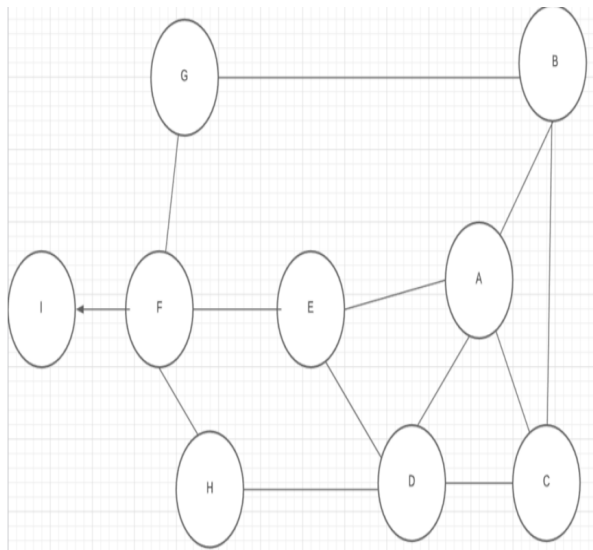


Figura 1: Gráfica

```
grafica = {  
    'A': ['B', 'C', 'D', 'E'],  
    'B': ['A', 'G'],  
    'C': ['A', 'D'],  
    'D': ['A', 'C', 'H'],  
    'E': ['A', 'F'],  
    'F': ['E', 'H', 'I'],  
    'G': ['B'],  
    'H': ['D', 'F'],  
    'I': ['F']  
}
```

Figura 2: Representación en python

La función *bfs* es la implementación central del algoritmo, este recibe tres argumentos:

- **grafica:** La gráfica que se recorrerá.
- **nodo_raiz:** El nodo desde el cual comienza el recorrido.
- **nodos_visitados:** Una lista que almacena los nodos que ya han sido visitados para evitar ciclos o visitas redundantes.

El algoritmo comienza añadiendo el *nodo_raiz* a una cola y marcándolo como visitado. Luego, mientras la cola no esté vacía, se extrae el primer nodo y se exploran todos sus vecinos. Si un vecino no ha sido visitado, se agrega a la cola y se marca como visitado. Este proceso continúa hasta que todos los nodos alcanzables desde el nodo raíz hayan sido visitados.

Listing 1: Algoritmo BFS

```
# Algoritmo BFS
def bfs(grafica, nodo_raiz, nodos_visitados):
    #Agregamos el nodo raiz a la cola y lo marcamos como visitado
    cola.put(nodo_raiz)
    nodos_visitados.append(nodo_raiz)

    while not cola.empty():
        nodo_actual = cola.get()
        #Vamos construyendo el recorrido
        print(nodo_actual, end = "-")

        #Revisamos los nodos vecinos del nodo actual
        for vecino in grafica[nodo_actual]:
            #Si este aun no ha sido visitado,
            #lo agregamos a la cola y lo marcamos como visitado
            if vecino not in nodos_visitados:
                nodos_visitados.append(vecino)
                cola.put(vecino)
```

La decisión de utilizar una cola si bien está influenciada por el hint consideramos que es clave para **BFS**, ya que garantiza que se visiten primero los nodos más cercanos al nodo raíz antes de explorar los nodos más alejados.

Para esta implementación específica, elegimos un enfoque simple y directo que sigue el pseudocódigo tradicional de BFS. Decidimos representar la gráfica como un diccionario porque esta estructura permite un acceso rápido a los nodos y sus vecinos, lo cual es esencial para la eficiencia del algoritmo. Además, se implementó una lista de nodos visitados para prevenir el recorrido cíclico y asegurar que cada nodo se procese una sola vez. Esto es especialmente importante en gráficas con ciclos, donde un enfoque sin control de visitas podría llevar a bucles infinitos.

Por lo que al ejecutar el algoritmo se muestra de esta manera:

```
1 Practica 01: Implementación del algoritmo de recorrido BFS (Breadth First Search)
2 Curso: Computación distribuida 2025-1
3 Fecha de entrega: 03/Sep/24
4 Equipo:
5 - López Diego Gabriela
6 - San Martín Nicolás Juan Daniel
7 - Martínez Hidalgo Paola Mildred
8
9 import queue
10
11
12 grafica = {
13     'A': ['B', 'C', 'D', 'E'],
14     'B': ['A', 'C', 'G'],
15     'C': ['A', 'B', 'D'],
16     'D': ['H', 'E', 'A', 'C'],
17     'E': ['A', 'D', 'F'],
18     'F': ['G', 'E', 'H', 'I'],
19     'G': ['F', 'B'],
20     'H': ['F', 'D'],
21     'I': ['F']
22 }
23
24 # Algoritmo BFS
25 def bfs(grafica, nodo_raiz, nodos_visitados):
26     #Agregamos el nodo raiz a la cola y lo marcamos como visitado
27     cola.put(nodo_raiz)
28     nodos_visitados.append(nodo_raiz)
29
30     while not cola.empty():
31         nodo_actual = cola.get()
32         #Vamos construyendo el recorrido
33         print(nodo_actual, end = "-")
34
35         #Revisamos los nodos vecinos del nodo actual
36         for vecino in grafica[nodo_actual]:
37             #Si este aun no ha sido visitado, lo agregamos a la cola y lo marcamos como visitado
```

Output: A B C D E H F I

Figura 3: Ejecución del algoritmo

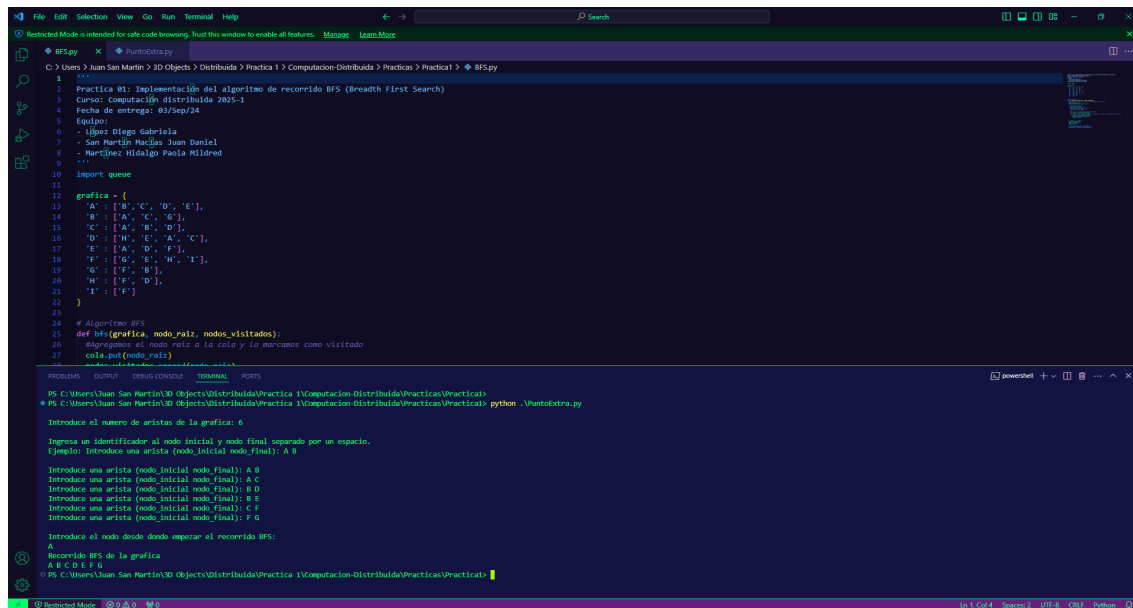
Punto EXTRA

A diferencia de la implementación anterior en esta variante, se permite al usuario definir la gráfica mediante la entrada de aristas y elegir el nodo de inicio para el recorrido. Las aristas se ingresan en el formato "nodo_inicial nodo_final", y se actualiza la representación de la gráfica de manera dinámica, es decir, se solicita el número de aristas y se ingresa cada arista, actualizando la gráfica, ya con esta información se recoge el nodo raíz desde el cual iniciar el recorrido BFS.

Al igual que en el script anterior, la cola se utiliza para mantener el orden de los nodos a ser explorados, asegurando que los nodos más cercanos al nodo raíz sean procesados antes que los nodos más alejados. La gran diferencia (el punto extra) es que permite al usuario definir la gráfica y el nodo de inicio, permitiendo probar el algoritmo en diferentes escenarios sin necesidad de modificar el código.

Este script se inicializa un diccionario vacío, este diccionario se llenará dinámicamente con los datos proporcionados por el usuario los cuales llevan la siguiente secuencia.

1. Se solicita al usuario que ingrese el número total de aristas que tendrá la gráfica.
2. Se instruye al usuario para que introduzca cada arista en el formato "nodo_inicial nodo_final". El script lee estas entradas en un bucle y actualiza el diccionario.
 - a) Si un nodo no existe en la gráfica, se agrega como una clave en el diccionario con una lista vacía de vecinos.
 - b) Se añaden los nodos finales a las listas de vecinos de los nodos iniciales y viceversa, asegurando que la gráfica sea bidireccional.
3. Después de construir la gráfica, se solicita al usuario que introduzca el nodo desde el cual comenzar el recorrido BFS.



```
1 # Practica 01: Implementación del algoritmo de recorrido BFS (Breadth First Search)
2 # Curso: Computación distribuida 2025-1
3 # Fecha de entrega: 30/sep/24
4 # Equipo:
5 # - López Diego Gabriel
6 # - San Martín Macías Juan Daniel
7 # - Martínez Hidalgo Paola Mildred
8 #
9 import queue
10
11 # Gráfica
12 grafica = {
13     'A': ['B', 'C', 'D', 'E'],
14     'B': ['A', 'C', 'G'],
15     'C': ['A', 'B', 'D'],
16     'D': ['B', 'C', 'A', 'C'],
17     'E': ['A', 'D', 'F'],
18     'F': ['G', 'E', 'H', 'I'],
19     'G': ['B', 'D'],
20     'H': ['F', 'D'],
21     'I': ['F']
22 }
23
24 # Algoritmo BFS
25 def bfs(grafica, nodo_raiz, nodos_visitados):
26     # Agregamos el nodo raíz a la cola y lo marcamos como visitado
27     cola.put(nodo_raiz)
28     nodos_visitados.add(nodo_raiz)
29
30     while not cola.empty():
31         nodo = cola.get()
32         for vecino in grafica[nodo]:
33             if vecino not in nodos_visitados:
34                 cola.put(vecino)
35                 nodos_visitados.add(vecino)
36
37     return nodos_visitados
38
39 # Ejemplo de uso
40 if __name__ == '__main__':
41     # Solicitamos el número de aristas
42     n_aristas = int(input("Introduce el número de aristas de la gráfica: "))
43
44     # Solicitamos el nodo inicial y nodo final separados por un espacio.
45     # Ejemplo: Introduce una arista (nodo_inicial nodo_final): A B
46     for i in range(n_aristas):
47         arista = input(f"Introduce una arista (nodo_inicial nodo_final): ")
48         partes = arista.split()
49         if len(partes) == 2:
50             nodo_inicial, nodo_final = partes
51             grafica[nodo_inicial].append(nodo_final)
52             grafica[nodo_final].append(nodo_inicial)
53
54     # Solicitamos el nodo desde donde empezar el recorrido BFS:
55     nodo_raiz = input("Introduce el nodo desde donde empezar el recorrido BFS: ")
56
57     # Ejecutamos el algoritmo BFS
58     nodos_visitados = bfs(grafica, nodo_raiz, set())
59     print("Recorrido BFS de la gráfica")
60     print(nodos_visitados)
```

Terminal output:

```
PS C:\Users\Juan San Martín\3D Objects\Distibuida\Practica 1\Computacion-Distribuida\Practicas\Practica1>
PS C:\Users\Juan San Martín\3D Objects\Distibuida\Practica 1\Computacion-Distribuida\Practicas\Practica1> python .\PuntoExtra.py
Introduce el número de aristas de la gráfica: 6
Introduce una arista (nodo_inicial nodo_final): A B
Introduce una arista (nodo_inicial nodo_final): A C
Introduce una arista (nodo_inicial nodo_final): B D
Introduce una arista (nodo_inicial nodo_final): B E
Introduce una arista (nodo_inicial nodo_final): C F
Introduce una arista (nodo_inicial nodo_final): F G
Introduce el nodo desde donde empezar el recorrido BFS:
A
Recorrido BFS de la gráfica
A B C D E F G
```

Figura 4: Ejemplo de uso

En esta ejecución formamos la siguiente gráfica

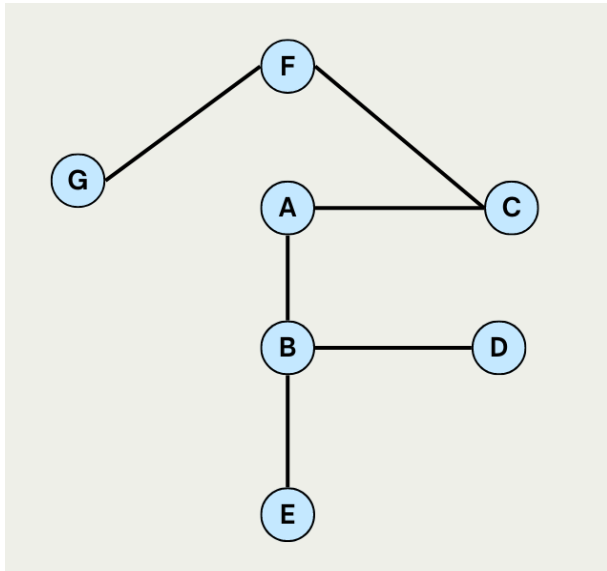


Figura 5: Gráfica

```
grafica = {  
    'A': ['B', 'C']  
    'B': ['A', 'D', 'E']  
    'C': ['A', 'F']  
    'D': ['B']  
    'E': ['B']  
    'F': ['C', 'G']  
    'G': ['F']  
}
```

Figura 6: Representación en python