ESCUELA POLITÉCNICA NACIONAL Estructuras de Datos y Algoritmos II – Computación	INFORME No.
Nombre: Luis Enrique Perez Señalin	4

Ch2 – Grafos

Tabla de Contenidos

1. Objetivos

Utilizar el algoritmo de BFS (Breadth-first-Search) para encontrar un nodo en un grafo y para crear un arbol BFS para encontrar la ruta más corta a un nodo

2. Introducción

En este capítulo veremos el primer algoritmo de grafo. Se llama Breadth-first Search (BFS). • BFS le permite encontrar la distancia más corta entre dos nodos. • Puede usar BFS para: • Escribir un juego de damas que calcule la menor cantidad de movimientos hacia la victoria • Escribir un corrector ortográfico (la menor cantidad de ediciones de su falta de ortografía a una palabra real; por ejemplo, READED-> READER es una edición) • Encontrar el médico más cercano a usted en su red. • Los algoritmos de grafos son algunos de los algoritmos más útiles

3. EJERCICIOS PLANTEADOS Y/O PROGRAMAS IMPLEMENTADOS

1 Dado el siguiente grafo, implemente el algortimo BSF donde se responda a las preguntas: 1. Existe un path desde S hasta F? 2. Cuál es esa ruta?

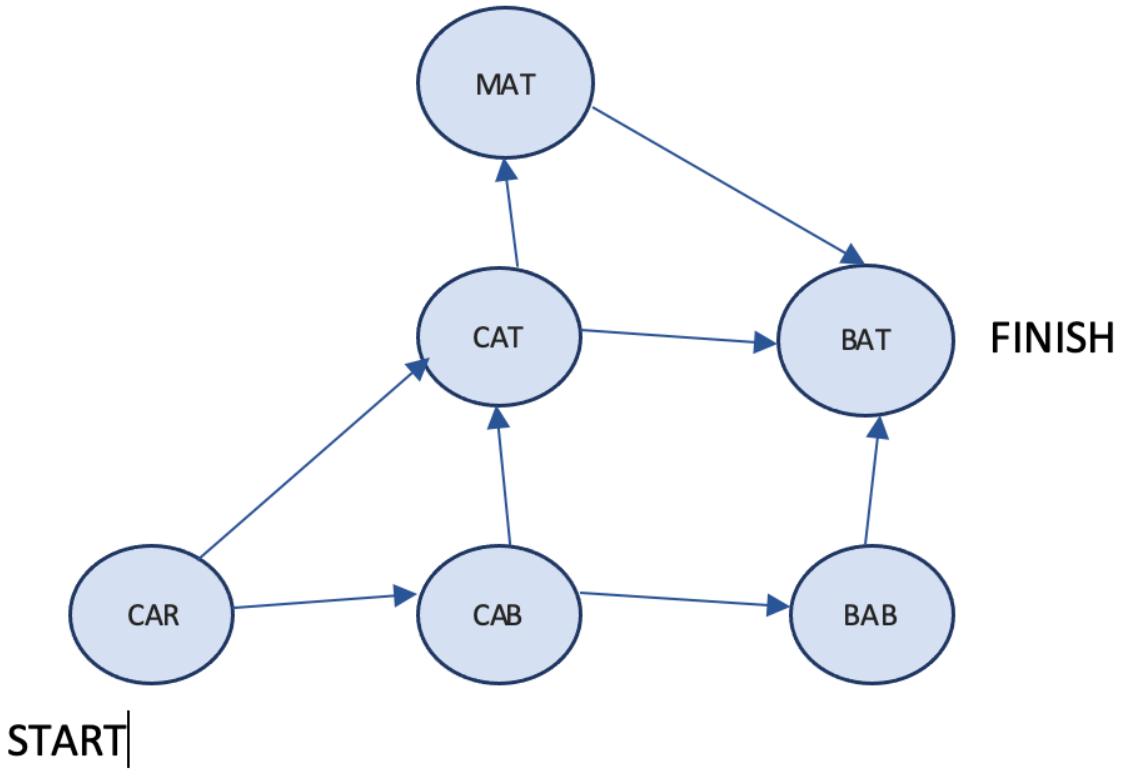

Figura 1: grafo 1, imagen_muestra

Vamos a poner el grafo en código usando un diccionario en python

```
graph = {}
graph['car'] = ["cat","cab"]
graph['cat'] = ["mat", "bat"]
graph['cab'] = ["bab", "cat"]
graph['mat'] = ["bat"]
graph['bat'] = []
graph['bab'] = ["bat"]
```

Creamos una función personalizada que nos retornará una función que comparará los "vecino" del nodo pasado como argumento y los comprobará con la variable "nodo_correct" que pasamos para que esta función la construya

```
def fun_nodo_is_correct(nodo_correct):
    return lambda nodo: nodo_correct in graph[nodo]
```

Usamos la función search con una cola

```
from collections import deque

def search(name,nodo_is_correct):
    search_queue = deque()
    search_queue += graph[name]
    searched = []
    while search_queue:
        nodo = search_queue.popleft()
        if not nodo in searched:
            if nodo_is_correct(nodo):
                print(nodo,"have a bat node")
                return True
            else:
                search_queue += graph[nodo]
                searched.append(nodo)
    return False
```

Guardamos la función de retorno creada mandando el nodo “bat” para que sea encontrado y llamamos la función search con el nodo “car” como inico

```
nodo_is_correct = fun_nodo_is_correct("bat")
search('car',nodo_is_correct)
```

```
cat have a bat node
```

```
True
```

1.Existe un path desde S hasta F? **Respuestas:** Existen multiples caminos, hay 5 en total, pero existe un camino más corto que el resto.

2.Cuál es esa ruta? **Respuesta:** La ruta más corta es: “car” -> “cat” -> “bat”

2 Genere el árbol BFS (BFS Tree) para el siguiente grafo G, sabiendo que el nodo de inicio es 0. Ese sería el componente conectado que contiene al nodo 0. Implemente el algoritmo BFS y use las estructuras auxiliares necesarias.

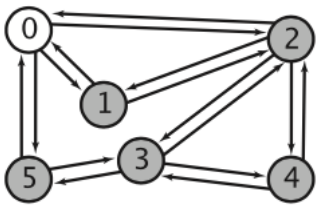

Figura 2: grafo 2, imagen_muestra

```
graph2 = {}
graph2['0'] = ["1","2","5"]
graph2['1'] = ["2", "0"]
graph2['2'] = ["1", "0", "3","4"]
graph2['5'] = ["3","0"]
graph2['3'] = ["5","2","4"]
graph2['4'] = ["3","2"]
```

La función principal para crear el arbol bfs, funciona de forma parecida al anterior cóidigo, solo que este tiene un fin distinto, no encuentra un vecino que tenga una arista hacia un nodo buscado, sino que retorna un diccionario que contiene el par de clave valor tipo: nodo: padre.

La forma que funciona es usando los arrays de tree(arbol de nodo-padre), el diccionario "visited"(guardara temporalmenete el nodo con su padre) y la cola "queue" va a iterar en cada nodo y guardará su padre correspondiente

primero al nodo "start" le pondremos de padre "None", luego haremos un ciclo while, extraeremos el nodo de la cola, e iteraremos con un for en cada vecino, verificamos si el vecino no ha sido visitado antes, y si no es el caso, guardaremos el nodo actual, osea del cual se extrajo los vecinos como valor "padre" y la clave será el vecino actual, y así sucesivamente

Cabe aclarar que usando el debugeador de python, el diccionario tree y visited van o deben terminar siendo iguales, solo para mayor claridad se usan por separados.

```
def bfs(graph, start):
  visited = {start: None}
  queue = [start]
  tree = {}

  while queue:
    current_node = queue.pop(0)
    tree[current_node] = visited[current_node]

    for neighbor in graph[current_node]:
      if neighbor not in visited:
        visited[neighbor] = current_node
        queue.append(neighbor)

  return tree
```

```
#Asignamos el nodo "Start"
start_node = '0'

#LLamamos a la función pasando el grafo y el nodo inicial
```

```python
bfs_tree = bfs(graph2, start_node)

#Imprimimos el valor del arbol, extrayendo clave valor usando .items()
print("Árbol BFS:")
# Iteramos los nodos con los padres del arbol ya ordenado
for node, parent in sorted(bfs_tree.items()):
  print(f"{node} -> {parent}")
```

```
Árbol BFS:
0 -> None
1 -> 0
2 -> 0
3 -> 2
4 -> 2
5 -> 0
```

4. Conclusiones

El algoritmo de BFS para encontrar el nodo que tiene un vecino específico, y para crear el arbol con la ruta más corta funcionan de forma eficinete y sencilla, permitiendonos búsquedas eficientes y rutas cortas para visualizar el grafo.

5 .Referencias bibliográficas

Todo el contenido usado para este informe fue proporcionado dentro de los pdfs.

6 .Declaración de uso de ia

Se ha usado la inteligencia artificual para comprender y crear correctamente arboles BFS junto con el uso del debugger de python para entender correctamente el funcionamento del algoritmo