



Instituto Politécnico Nacional
Escuela Superior de Física y Matemáticas
Licenciatura en Matemáticas Algorítmicas



Teoría de grafos

Porfirio Damián Escamilla Huerta

Omar Porfirio García

Carlos David Zamora Gutierrez



Teoría de Grafos | Práctica 2

Busqueda en anchura

Escamilla Porfirio

Porfirio Omar

Zamora Carlos

Agosto 2022

1 Impresión

La función `dibuja_matriz(M)` recorre una matriz `M` e imprime las entradas $m_{i,j}$ en el medio de una cadena de longitud tres.

```
1 def dibuja_matriz(M):
2     for i in range(len(M)):
3         print('[', end='')
4         for j in range(len(M[i])):
5             print('{:~3n}'.format(M[i][j]), end='')
6         print(']')
```

Listing 1: Función `dibuja_matriz(M)`

2 Eliminación de bucles (loops)

La función `elimina_bucles(M)` recorre una matriz `M` y le asigna el valor 0 a las entradas $m_{i,i}$, entradas que representan bucles de longitud uno.

```
1 def elimina_bucles(M):
2     n = len(M)
3     for i in range(n):
4         M[i][i] = 0
```

Listing 2: Función `elimina_bucles(M)`

3 Vértices

La función `crea_vértices(M)` recorre una matriz `M`, lista los vértices adyacentes y convierte las entradas $m_{i,j}$ en un diccionario que contiene el número de vértice, vértices adyacentes y un estado que indica si está conectado o no, para listarlas y devolver la lista de los vértices.

```
1 def crea_vertices(M):
2     n = len(M)
3     vertices = []
4     vecinos = []
5     vecinos_vj = []
6     for i in range(n):
7         for j in range(n):
8             if M[i][j] != 0:
9                 vecinos_vj.append(j + 1)
10            vecinos.append(dict([("vecinos", vecinos_vj.copy())]))
11            vecinos_vj.clear()
12        for i in range(n):
13            vertices.append(dict([("vertex", i + 1), ("vecinos", vecinos[i]), ("conectado", False)]))
14        return vertices
```

Listing 3: Función `crea_vértices(M)`

4 Entrada

La función `lee_matriz(M)` solicita el número de vértices del grafo, lee una matriz triangular inferior ($n \times n$), es decir las entradas $m_{i,j}$ con $i \geq j$, donde las entradas $m_{i,j}$ con $i = j$ deben ser pares, pues la diagonal representa a los bucles (loops), y luego le asigna el valor de las entradas $m_{i,j}$ a las entradas $m_{j,i}$, para asegurar que sea una matriz de adyacencia (debe ser simétrica). Retorna la matriz generada.

```

1 def lee_matriz():
2     n = int(input("Ingrese el numero de vertices del grafo: "))
3     M = [[-1] * n for i in range(n)]
4     for i in range(n):
5         for j in range(i + 1):
6             if i == j:
7                 M[i][j] = valida_entrada(i, j, True)
8             else:
9                 M[i][j] = valida_entrada(i, j, False)
10            if i > j:
11                M[j][i] = M[i][j]
12    return M

```

Listing 4: Función lee_matriz()

4.1 Validación

La función valida_entrada(M) valida que los valores leídos sean numéricos y que las entradas de la diagonal sean números pares. Así como que las opciones elegidas sean correctas.

```

1     resul: int = 1
2     if flag == 1:
3         while resul % 2 != 0:
4             while True:
5                 try:
6                     resul = int(input(
7                         "Ingrese la entrada M[{0}][{1}] (Esta debe ser
8                         par): ".format(str(i + 1), str(j + 1))))
9                     break
10                except:
11                    print("No es un valor valido!")
12            return resul
13        elif flag == 0:
14            while True:
15                try:
16                    resul = int(input("Ingrese la entrada M[{0}][{1}]: ".
17                        format(str(i + 1), str(j + 1))))
18                    return resul
19                except:
20                    print("No es un valor valido!")
21        elif flag == 2:
22            while True:
23                try:
24                    resul = int(input("Ingrese un numero 1 si desea
25                        aplicar busqueda en anchura y un 2 si desea aplicar busqueda en
26                        profundidad: "))
27                    if 0 < resul and resul < 3:
28                        return resul
29                    else:
30                        raise Exception("No es un valor valido!")
31                except:
32                    print("No es un valor valido!")
33        elif flag == 3:
34            while True:
35                try:
36                    resul = int(input("Ingrese un numero 1 si desea buscar
37                        otro arbol y un 2 si desea salir: "))

```

```

33         if resul == 1:
34             return True
35         if resul == 2:
36             return False
37         else:
38             raise Exception("No es un valor valido!")
39     except:
40         print("No es un valor valido!")

```

Listing 5: Función valida_entrada()

5 Búsqueda en anchura

La función `generar_arbol_anchura(M)` genera una matriz de adyacencia de un árbol generador dada una matriz `M`, a partir del algoritmo de búsqueda por anchura. Esta función, recorre la matriz de adyacencia hasta conectar a todos los vértices, retornando la matriz de adyacencia del árbol encontrado.

```

1  def generar_arbol_anchura(M):
2      conectados = [0]
3      n = len(M)
4      MA = [[0 for i in range(n)] for i in range(n)]
5      while len(conectados) != n:
6          for c in conectados:
7              idx=0
8              for i in M[c]:
9                  if i != 0 and idx not in conectados:
10                     conectados.append(idx)
11                     MA[c][idx] = 1
12                     MA[idx][c] = 1
13                     idx += 1
14     return MA

```

Listing 6: Función generar_arbol_anchura(M)

6 Búsqueda en profundidad

La función `generar_arbol_profundidad(M)` genera una matriz de adyacencia de un árbol generador dada una matriz `M`, a partir del algoritmo de búsqueda por profundidad. Esta función, hace uso de una pila auxiliar para conectar un vértice adyacente al anterior.

```

1  def generar_arbol_profundidad(M):
2      V = set() # Conjunto de vertices ya conectados
3      Vaux = set() # Conjunto de vertices auxiliar
4      vecino = 0 # Vecino mas pequeno
5      Pila = [] # Pila auxiliar
6      n = len(M) # Numero de vertices
7      MA = [[0 for i in range(n)] for i in range(n)] # Matriz de
adyacencia
8      vertices = crea_vertices(M)
9      Pila.append(1)
10     V.add(1)
11     while len(Pila) != 0: # Acaba cuando la pila esta vacia
12         u = Pila[-1] # Tomamos el ultimo elemento de la pila
13         Vaux = set(vertices[u - 1].get("vecinos")) # Obtenemos el
conjunto de vertices no conectados a u
14         if len(Vaux - V) > 0: # Si hay al menos un vecino no
conectado a u

```

```

15         vecino = min(Vaux - V) # Tomamos el vertice no
conectado mas pequeno
16         MA[u - 1][vecino - 1] = 1 # Los conectamos
17         MA[vecino - 1][u - 1] = 1
18         if vecino not in Pila: # Si el vecino no estaba en la
pila, lo agregamos
19             Pila.append(vecino)
20             V.add(vecino) # Agregamos al vecino al conjunto de
vertices ya conectados
21         else:
22             Pila.remove(u) # Si u no tiene vecinos sin conectar,
lo quitamos de la pila
23             Pila.clear()
24             V.clear()
25             Vaux.clear()
26         return MA

```

Listing 7: Función generar_arbol_profundidad(M)