Soluciones - práctico 5

Grafos

Ejercicio 1

1. Cuando existe una arista desde un nodo x a un nodo y, se puede ver en la matriz un 1 en la fila x, columna y. En caso contrario hay un 0 en esa posición.

Grafo (a)

nodo	1	2	3	4	5	6	7
1	0	1	1	0	0	1	0
2	1	0	1	0	0	0	1
3	1	1	0	1	0	1	0
4	0	0	1	0	0	0	1
5	0	0	0	0	0	1	1
6	1	0	1	0	1	0	0
7	0	1	0	1	1	0	0

Grafo (b)

nodo	1	2	3	4	5	6
1	0	0	0	0	1	0
2	1	0	0	1	0	0
3	0	1	0	0	0	1
4	0	0	1	0	1	1
5	1	0	0	0	0	0
6	1	1	0	0	1	0

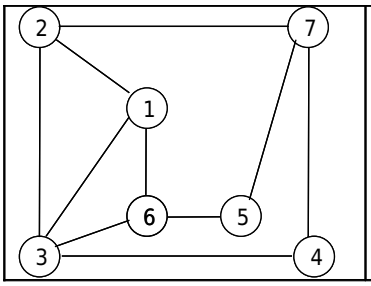
,

Ejercicio 4

Las recorridas DFS y BFS numeran los vértices de un Grafo, partiendo de un Vértice dado.

a) Constuya las recorridas DFS y BFS para los grafos del ejercicio 1 a partir del nodo número 6.

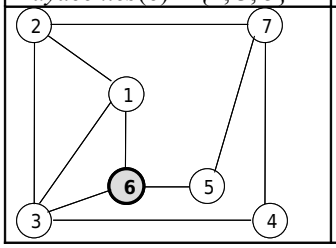
En este caso se muestra únicamente para el primer grafo.



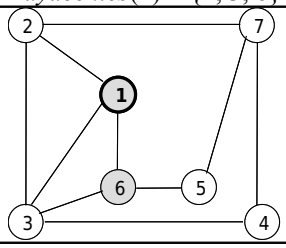
Adyacentes(1) = {2, 3, 6}
Adyacentes(2) = {1, 3, 7}
Adyacentes(3) = {1, 2, 4, 6}
Adyacentes(4) = {3, 7}
Adyacentes(5) = {6, 7}
Adyacentes(6) = {1, 3, 5}
Adyacentes(7) = {2, 4, 5}

DFS

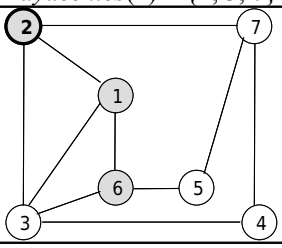
Visit = {0, 0, 0, 0, 0, **1**, 0}
Rec = {**6**}
Adyacentes(6) = {**1**, 3, 5}



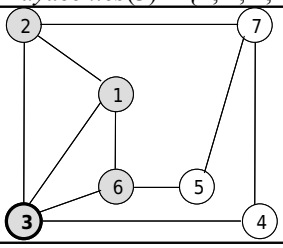
Visit = {**1**, 0, 0, 0, 0, 1, 0}
Rec = {6, **1**}
Adyacentes(1) = {**2**, 3, 6}



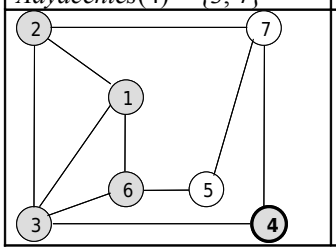
Visit = {1, **1**, 0, 0, 0, 1, 0}
Rec = {6, 1, **2**}
Adyacentes(2) = {1, **3**, 7}



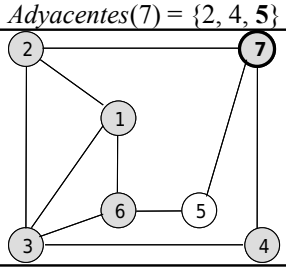
Visit = {1, 1, **1**, 0, 0, 1, 0}
Rec = {6, 1, 2, **3**}
Adyacentes(3) = {1, 2, **4**, 6}



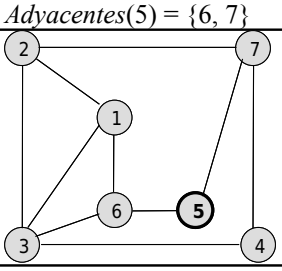
Visit = {1, 1, 1, **1**, 0, 1, 0}
Rec = {6, 1, 2, 3, **4**}
Adyacentes(4) = {3, **7**}



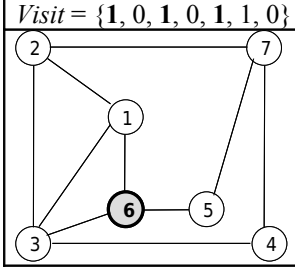
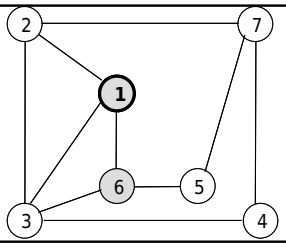
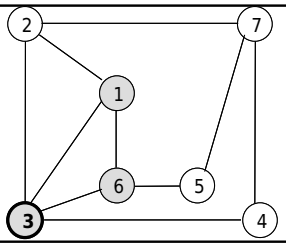
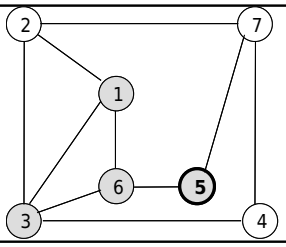
Visit = {1, 1, 1, 1, 0, 1, **1**}
Rec = {6, 1, 2, 3, 4, **7**}
Adyacentes(7) = {2, 4, **5**}

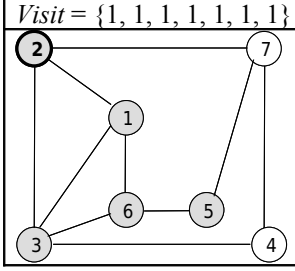
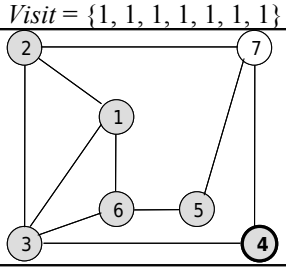
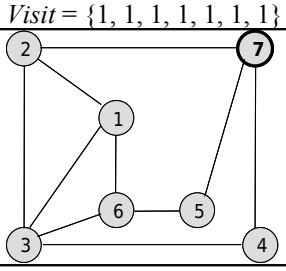


Visit = {1, 1, 1, 1, **1**, 1, 1}
Rec = {6, 1, 2, 3, 4, 7, **5**}
Adyacentes(5) = {6, 7}



BFS

Visit = {0, 0, 0, 0, 0, **1**, 0} *Prox* = {**6**} *Rec* = {**6**} *Adyacentes*(6) = {1, 3, 5} *Prox* = {**1**, **3**, **5**} *Visit* = {**1**, 0, **1**, 0, **1**, 1, 0}	*Prox* = {**1**, 3, 5} *Rec* = {6, **1**} *Adyacentes*(1) = {2, 3, 6} *Prox* = {3, 5, **2**} *Visit* = {1, **1**, 1, 0, 1, 1, 0}	*Prox* = {**3**, 5, 2} *Rec* = {6, 1, **3**} *Adyacentes*(3) = {1, 2, 4, 6} *Prox* = {5, 2, **4**} *Visit* = {1, 1, 1, **1**, 1, 1, 0}	*Prox* = {**5**, 2, 4} *Rec* = {6, 1, 3, **5**} *Adyacentes*(5) = {6, 7} *Prox* = {2, 4, **7**} *Visit* = {1, 1, 1, 1, 1, 1, **1**}
			

Prox = {**2**, 4, 7} *Rec* = {6, 1, 3, 5, **2**} *Adyacentes*(2) = {1, 3, 7} *Prox* = {4, 7} *Visit* = {1, 1, 1, 1, 1, 1, 1}	*Prox* = {**4**, 7} *Rec* = {6, 1, 3, 5, 2, **4**} *Adyacentes*(4) = {3, 7} *Prox* = {7} *Visit* = {1, 1, 1, 1, 1, 1, 1}	*Prox* = {**7**} *Rec* = {6, 1, 3, 5, 2, 4, **7**} *Adyacentes*(7) = {2, 4, 5} *Prox* = {} *Visit* = {1, 1, 1, 1, 1, 1, 1}
		

Ejercicio 5

Programa principal:

```
Lista masLargo (Grafo g, int nodoInicial)
{
    Lista caminoMasLargo = VaciaLista (), caminoActual = VaciaLista ();
    int costoMaximo = 0;
    CaminoMasLargo (g, nodoInicial, caminoMasLargo, caminoActual,
           costoMaximo, 0);
    DestruirLista (caminoActual);
    return caminoMasLargo;
}
```

Algoritmo recursivo para encontrar el camino más largo en un grafo dirigido acíclico:

```
void CaminoMasLargo (Grafo g, int nodoInicial, Lista & caminoMasLargo,
    Lista caminoActual, int & costoMaximo, int costoActual)
{
    if (costoMaximo < costoActual)
    {
           DestruirLista (caminoMasLargo);
           caminoMasLargo = CopiarLista (caminoActual);
           costoMaximo = costoActual;
    }
    Lista adyacentes = AdyacentesGrafo (g, nodoInicial);
    while (!EsVaciaLista (adyacentes))
    {
           Vertice v = PrimeroLista (adyacentes);
           adyacentes = RestoLista (adyacentes);
           InsertarLista (caminoActual, v);
           CaminoMasLargo (g, v, caminoMasLargo, caminoActual, costoMaximo,
                  costoActual + v.costo);
           // v.costo es el costo de la arista que va desde nodoInicial a v
    }
}
```

Ejercicio 6

Usamos una cola para guardar vértices con grado de entrada cero.
Los grados de entrada variarían a medida que corre el algoritmo de la siguiente forma:

Todo		3	3	3	3	0
Case		2	1	1	0	-
Data warehouse	1	1	1	0	-	-
Admin BD		2	1	1	0	-
Help case	0	-	-	-	-	
BD		6	2	0	-	-
Instalador	0	-	-	-	-	
Help adm.	0	-	-	-	-	
Doc BD		0	-	-	-	-
Serv		3	0	-	-	-
ODBC		0	-	-	-	-
Con. por red		0	-	-	-	-
Cliente		2	0	-	-	-
Loader		0	-	-	-	-
Otros		0	-	-	-	-

Por lo que una posible ordenación sería:

Help case
Instalador
Help adm.
Doc BD
ODBC
Con. Por red
Loader
Otros
Serv
Cliente
BD
Case
Data warehouse
Admin BD
Todo

Ejercicio 7

Un camino euleriano es aquel camino que pasa una vez por cada arista del grafo, comenzando y terminando posiblemente en vértices diferentes.

Grafos no dirigidos

a) El grafo debe contener 0 ó 2 vértices de grado impar.

```
bool camino_euler (Grafo g)
{
    lista l = obtener_vertices (g);
    int grado_impar = 0;
    while (!es_vacia (l) && grado_impar <= 2)
    {
           lista ady = obtener_adyacentes (g, primero (l));
           l = resto (l);
           if (largo (ady) % 2 != 0)
                 grado_impar++;
    }
    return (grado_impar == 0 || grado_impar == 2);
}
```

b) Comparación entre representaciones (análisis de orden):

Considérese |V| = cantidad de vértices de *g*

Para el peor caso se tiene un grafo completo que contiene un camino de euler. En este caso, el ciclo `while` recorre todos los vértices del grafo.

Grafo implementado con listas de adyacencia.

Se obtiene la lista de adyacentes de cada nodo en tiempo constante. Se considera además, que estás tienen una operación que retorna su largo sin recorrerla. Las restantes operaciones del ciclo se ejecutan también en tiempo constante, por tanto, el orden del tiempo de ejecución de la operación es O(|V|) para el peor caso.

Grafo implementado con matriz de adyacencias.

En un grafo completo, generar una lista de adyacencia a partir de una matriz de adyacencias tiene costo |V|, entonces, el orden del tiempo de ejecución de la operación es $O(|V|^2)$ para el peor caso.

Grafos dirigidos

a) El grafo debe contener vértices con estas características:

- 1 vértice de (grado de entrada = 1 + grado de salida)
- 1 vértice de (grado de entrada + 1 = grado de salida)

O de lo contrario, todos los vértices deben tener igual grado de entrada y de salida.

En este caso, el algoritmo se simplifica si el TAD Grafo contiene la operación

```
lista obtener_incidentes (Grafo g, int v);
```

El código queda de la siguiente manera:

```
bool camino_euler (Grafo g)
{
    lista l = obtener_vertices (g);
    bool inicio = false; // indica si se encuentra un vertice de inicio
    bool fin = false; // indica si se encuentra un vertice de fin
    bool falla = false; // indica si se encuentra un vertice
                        // que no permite construir un camino euleriano
    while (!es_vacia (l) && !falla)
    {
           lista ady = obtener_adyacentes (g, primero (l));
           lista inc = obtener_incidentes (g, primero (l));
           l = resto (l);
           if (largo (ady) == largo (inc) + 1)
                 inicio = true;
           else if (largo (ady) + 1 == largo (inc))
                 fin = true;
           else if (largo (ady) != largo (inc))
                 falla = true;
    }
    return (inicio && fin || !(inicio || fin));
}
```

b) Considerando que la operación comentada en la parte (a) (grafos dirigidos) se ejecuta en O(1) en el peor caso y que el largo de la lista de incidentes también se obtiene en tiempo constante, se logran los mismos tiempos de ejecución que para grafos no dirigidos.