Tecnológico de Monterrey
Escuela de Ingeniería y Ciencias

Campus Sonora Norte

Act 4.2 - Grafos: Algoritmos complementarios

Curso:

Programación de estructuras de datos y algoritmos fundamentales

(TC1031)

Estudiante:

Daniel Alfredo Barreras Meraz A01254805

Docente:

Baldomero Olvera Villanueva

Fecha de entrega:

12 de noviembre de 2023

Código fuente

- Adjunto en el archivo zip.

Casos de prueba

Caso de prueba	**Entrada**	**Resultados**		
		isTree	**topologicalSort**	**bitPartiteGraph**
1	{{0, 1}, {1, 3}, {3, 4}, {4, 2}}	Sí	0 1 3 4 2	Sí
2	{{0, 1}, {1, 2}, {2, 4}, {4, 3}, {3, 0}}	No	0 1 2 4 3	No
3	{{0, 1}, {1, 2}, {0, 3}, {1, 4}, {2, 5}, {3, 4}, {4, 5}}	No	0 1 2 5 4 3	Sí
4	{{0, 1}, {1, 3}, {0, 2}, {2, 3}, {3, 4}, {3, 5}, {4, 6}, {5, 6}}	No	0 1 3 4 6 5 2	Sí

Comprobación de casos de prueba

```
Matriz de adyacencia:
0: 0 1 0 0 0
1: 1 0 0 1 0
2: 0 0 0 0 1
3: 0 1 0 0 1
4: 0 0 1 1 0

Lista de adyacencia:
0: 1
1: 0 3
2: 4
3: 1 4
4: 3 2
El grafo 1 es un árbol: Sí
El grafo 1 es bipartito: Sí
Orden topológico del grafo 1: 0 1 3 4 2
```

```
Matriz de adyacencia:
0: 0 1 0 1 0
1: 1 0 1 0 0
2: 0 1 0 0 1
3: 1 0 0 0 1
4: 0 0 1 1 0

Lista de adyacencia:
0: 1 3
1: 0 2
2: 1 4
3: 4 0
4: 2 3
El grafo 2 es un árbol: No
El grafo 2 es bipartito: No
Orden topológico del grafo 2: 0 1 2 4 3
```

```
Matriz de adyacencia:
0: 0 1 0 1 0 0
1: 1 0 1 0 1 0
2: 0 1 0 0 0 1
3: 1 0 0 0 1 0
4: 0 1 0 1 0 1
5: 0 0 1 0 1 0

Lista de adyacencia:
0: 1 3
1: 0 2 4
2: 1 5
3: 0 4
4: 1 3 5
5: 2 4
El grafo 3 es un árbol: No
El grafo 3 es bipartito: Sí
Orden topológico del grafo 3: 0 1 2 5 4 3
```

```
Matriz de adyacencia:
0: 0 1 1 0 0 0 0
1: 1 0 0 1 0 0 0
2: 1 0 0 1 0 0 0
3: 0 1 1 0 1 1 0
4: 0 0 0 1 0 0 1
5: 0 0 0 1 0 0 1
6: 0 0 0 0 1 1 0

Lista de adyacencia:
0: 1 2
1: 0 3
2: 0 3
3: 1 2 4 5
4: 3 6
5: 3 6
6: 4 5
El grafo 4 es un árbol: No
El grafo 4 es bipartito: Sí
Orden topológico del grafo 4: 0 1 3 4 6 5 2
```

Complejidad temporal

loadGraph: O(n^2) - Esta función carga los arcos del grafo y los almacena en una Matriz de Adyacencia y en una Lista de Adyacencia. Como cada vértice puede estar conectado con todos los demás, en el peor de los casos, la función debe recorrer todos los pares de vértices, lo que da una complejidad de tiempo de O(n^2).

isTree: O(n + m) - La función isTree examina si un Grafo Dirigido Acíclico (DAG) es un árbol. Para hacer esto, puede necesitar recorrer todos los vértices y arcos del grafo, lo que resulta en una complejidad de tiempo de O(n + m).

topologicalSort: O(n + m) - La función topologicalSort lleva a cabo una ordenación topológica en un grafo. Este proceso puede implicar visitar cada vértice y arco en el grafo, lo que nos lleva a una complejidad de tiempo de O(n + m).

bipartiteGraph: O(n + m) - La función bipartiteGraph verifica si un Grafo Dirigido Acíclico (DAG) puede ser representado como un grafo bipartito. Este proceso puede requerir un recorrido por todos los vértices y arcos del grafo, lo que da como resultado una complejidad de tiempo de O(n + m).