Informe Semana 8

Marlon Zurita

2024-11-15

Tabla de Contenidos

Ordenamiento topológico

1. Objetivos

- Describir y analizar el algoritmo de orden topológico implementado, identificar las estructuras de datos utilizadas y explicar su papel en el algoritmo.
- Investigar y discutir los desafíos asociados con grafos con ciclos en el algoritmo.

2. Introducción

Este informe se centra en el análisis del algoritmo de orden topológico implementado en el código proporcionado. El orden topológico es una técnica fundamental en teoría de grafos, especialmente en la resolución de problemas que involucran tareas que deben ser ejecutadas en un orden específico y que dependen de tareas anteriores. Este algoritmo es esencial en

diversas áreas de la informática, como la planificación de tareas y la gestión de dependencias en sistemas complejos (Cake 2024).

A lo largo de este informe, se explora en detalle la lógica subyacente del algoritmo y se analiza su funcionamiento cuando existen ciclos en el grafo (Geeks 2024).

3. Ejercicios planteados

Aplique el algoritmo de ordenamiento topológico en el siguiente ejemplo:

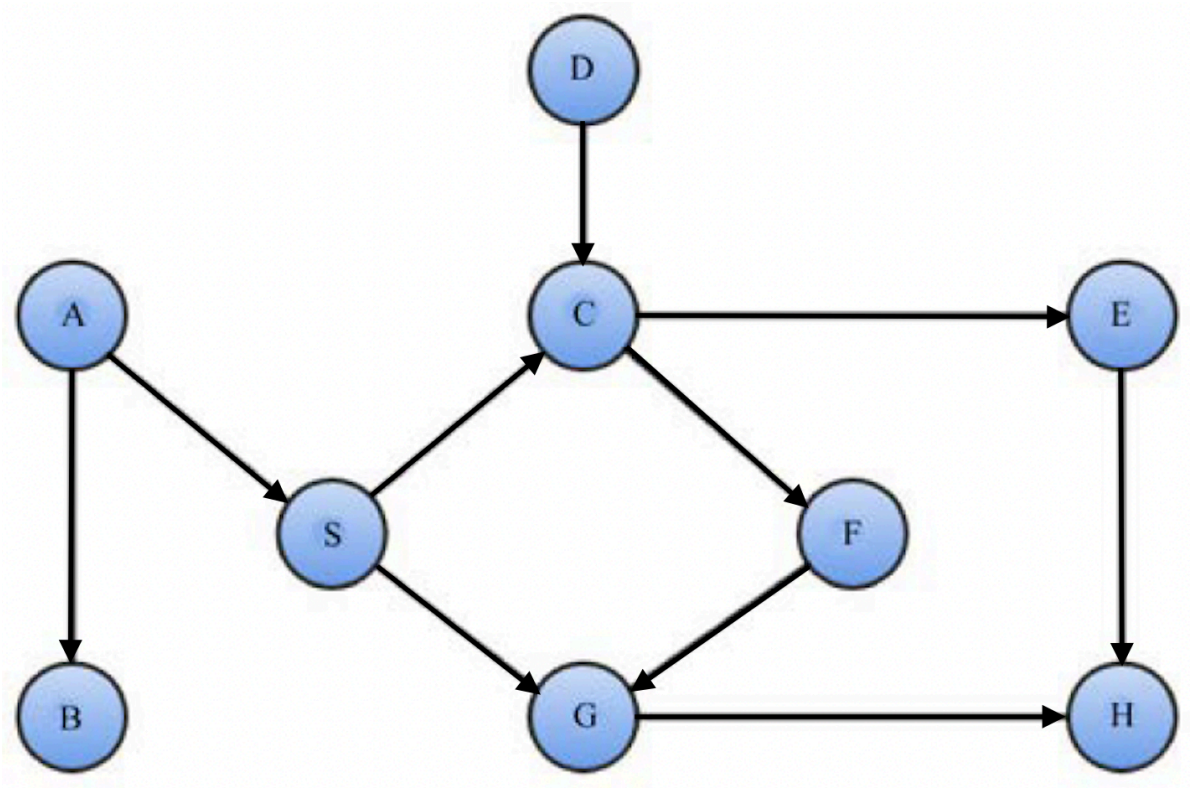

Figura 1: Ejemplo 1

```
def indegree(grafo):
    # crea un diccionario para almacenar la cuenta del indegree para cada nodo
    indegree_count = {nodo: 0 for nodo in grafo}

    # itera cada nodo en el grafo
    for nodo in grafo:
```

```
        # itera cada vecino en el nodo
        for vecino in grafo[nodo]:
            # incrementa la cuenta del indegree por uno
            indegree_count[vecino] += 1
    # retorna el indegree calculado para cada nodo
    return indegree_count

def topological_sort(grafo):
    # lista para el ordenamiento topológico del grafo
    top_sorted = []
    # lista para almacenar los nodos con indegree = 0
    ready = []
    # diccionario para almacenar la cuenta del indegree para cada nodo
    incount = indegree(grafo)

    # itera cada nodo en el grafo
    for nodo in grafo.keys():
        # si el indegree de un nodo es 0, añadir a la lista ready
        if incount[nodo] == 0:
            ready.append(nodo)
    print('Indegree:', incount)
    print('READY:', ready)

    # procesa los nodos con indegree 0 hasta que todos sean procesados
    while len(ready) > 0:
        # saca un nodo con indegree 0 de la lista ready
        nodo = ready.pop()
        print('Nodo para usarse:', nodo)

        # añade el nodo a la lista de nodos ordenados topológicamente
        top_sorted.append(nodo)

        # actualiza el indegree de los nodos vecinos
        for vecino in grafo[nodo]:
            print(vecino, ': ', incount[vecino], '- 1')
            incount[vecino] -= 1
            # si el indegree se vuelve 0, añade el nodo a la lista ready
            if incount[vecino] == 0:
                ready.append(vecino)
        print('READY:', ready)

    # retorna el orden topológico
```

```
    return top_sorted

grafo = {'A': ['B','S'],
         'B': [],
         'C': ['E','F'],
         'D': ['C'], 'E': ['H'],
         'F':['G'],
         'G':['H'],
         'H':[],
         'S':['C','G']}

topological_sort(grafo)
```

```
Indegree: {'A': 0, 'B': 1, 'C': 2, 'D': 0, 'E': 1, 'F': 1, 'G': 2, 'H': 2, 'S': 1}
READY: ['A', 'D']
Nodo para usarse: D
C :  2 - 1
READY: ['A']
Nodo para usarse: A
B :  1 - 1
S :  1 - 1
READY: ['B', 'S']
Nodo para usarse: S
C :  1 - 1
G :  2 - 1
READY: ['B', 'C']
Nodo para usarse: C
E :  1 - 1
F :  1 - 1
READY: ['B', 'E', 'F']
Nodo para usarse: F
G :  1 - 1
READY: ['B', 'E', 'G']
Nodo para usarse: G
H :  2 - 1
READY: ['B', 'E']
Nodo para usarse: E
H :  1 - 1
READY: ['B', 'H']
Nodo para usarse: H
READY: ['B']
Nodo para usarse: B
```

```
READY: []
```

```
['D', 'A', 'S', 'C', 'F', 'G', 'E', 'H', 'B']
```

¿Qué sucede si el grafo dirigido G dado tiene un ciclo? Pruebe el algoritmo de ordenamiento topológico con un grafo dirigido que tenga un ciclo y explique el resultado de lo que retorna.

Para la demostración se usará el siguiente grafo, tiene un ciclo.

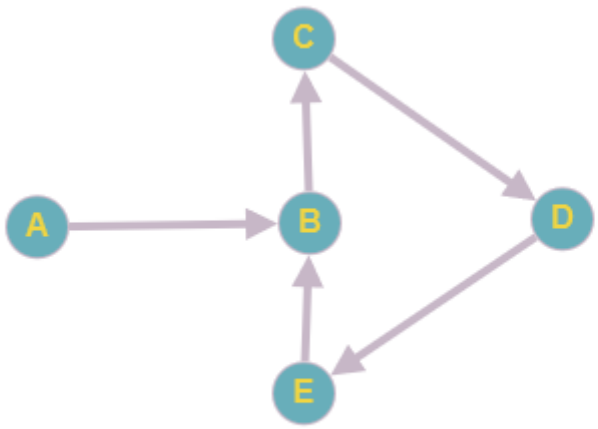

Figura 2: Ejemplo 2

Cuando se usa este algoritmo en un grafo, no existe una comprobación para verificar que el grafo sea acíclico. La salida puede entrar en un bucle infinito o bien puede retornar una lista que no corresponde a un ordenamiento topológico.

Para este grafo, el algoritmo retorna [‘A’] porque es el único nodo procesado al ser iniciado con un indegree de 0. Luego el algoritmo procesa B y reduce su indegree en 1, pero no llega a 0 (el indegree de B pasó de 2 a 1) y no existen más nodos con indegree de 0, por lo que el algoritmo termina prematuramente.

```python
def indegree(grafo):
    # crea un diccionario para almacenar la cuenta del indegree para cada nodo
    indegree_count = {nodo: 0 for nodo in grafo}
```

```
    # itera cada nodo en el grafo
    for nodo in grafo:
        # itera cada vecino en el nodo
        for vecino in grafo[nodo]:
            # incrementa la cuenta del indegree por uno
            indegree_count[vecino] += 1
    # retorna el indegree calculado para cada nodo
    return indegree_count

def topological_sort(grafo):
    # lista para el ordenamiento topológico del grafo
    top_sorted = []
    # lista para almacenar los nodos con indegree = 0
    ready = []
    # diccionario para almacenar la cuenta del indegree para cada nodo
    incount = indegree(grafo)

    # itera cada nodo en el grafo
    for nodo in grafo.keys():
        # si el indegree de un nodo es 0, añadir a la lista ready
        if incount[nodo] == 0:
            ready.append(nodo)
    print('Indegree:', incount)
    print('READY:', ready)

    # procesa los nodos con indegree 0 hasta que todos sean procesados
    while len(ready) > 0:
        # saca un nodo con indegree 0 de la lista ready
        nodo = ready.pop()
        print('Nodo para usarse:', nodo)

        # añade el nodo a la lista de nodos ordenados topológicamente
        top_sorted.append(nodo)

        # actualiza el indegree de los nodos vecinos
        for vecino in grafo[nodo]:
            print(vecino, ': ', incount[vecino], '- 1')
            incount[vecino] -= 1
            # si el indegree se vuelve 0, añade el nodo a la lista ready
            if incount[vecino] == 0:
                ready.append(vecino)
        print('READY:', ready)
```

```
    # retorna el orden topológico
    return top_sorted

grafo = {'A': ['B'], 'B': ['C'], 'C': ['D'], 'D': ['E'], 'E': ['B']}

topological_sort(grafo)
```

```
Indegree: {'A': 0, 'B': 2, 'C': 1, 'D': 1, 'E': 1}
READY: ['A']
Nodo para usarse: A
B :  2 - 1
READY: []


['A']
```

4. Conclusiones

- El algoritmo de ordenamiento topológico resulta eficaz en situaciones donde el grafo es acíclico. La capacidad de generar un orden lineal de nodos basado en dependencias direccionales es valiosa en escenarios como la planificación de proyectos y la compilación de código.
- Sin embargo, se observa que la implementación actual presenta limitaciones en el manejo de grafos con ciclos. La falta de una estrategia robusta para identificar y tratar ciclos puede afectar la precisión de los resultados y conducir a una terminación prematura en presencia de dependencias circulares, o bien la entrada en un bucle infinito.
- La inclusión de un mecanismo de detección y manejo de ciclos más sofisticado podría mejorar significativamente la aplicabilidad del algoritmo, permitiendo su uso en un número mayor de escenarios prácticos.

5. Referencias

Cake, Interview. 2024. «Topological sort». https://www.interviewcake.com/concept/java/topological-sort.

Geeks, Geeks for. 2024. «Topological sorting». https://www.geeksforgeeks.org/topological-sorting/.