Construcción de un Árbol según Distancias entre Vértices

Nombre del Autor

September 23, 2024

Explicación del problema

Se nos pide construir un árbol que cumpla con dos condiciones específicas relacionadas con la distancia entre pares de vértices:

- El número de pares de vértices con una distancia par entre ellos es igual a x.
- El número de pares de vértices con una distancia impar entre ellos es igual a y.

Un árbol es un grafo conectado y no dirigido que tiene n vértices y $n-1$ aristas. La distancia entre dos vértices se define como el número de aristas en el camino simple único que los conecta. Los pares de vértices se consideran como pares ordenados, lo que significa que para cada par (u, v), tanto (u, v) como (v, u) son considerados.

Solución trivial (fuerza bruta)

La solución más simple sería generar todos los posibles árboles con un número dado de vértices y contar los pares de vértices que cumplen las condiciones dadas. Este enfoque consiste en:

1. Generar todos los árboles posibles con n vértices.
2. Para cada árbol generado, calcular la distancia entre todos los pares de vértices.
3. Contar cuántos pares tienen una distancia par y cuántos tienen una distancia impar.
4. Verificar si estos conteos coinciden con x e y.

Correctitud del algoritmo

Este enfoque es correcto porque explora todas las configuraciones posibles y cuenta exhaustivamente las distancias. Sin embargo, dado que el número de árboles crece exponencialmente con el número de vértices, este método no es eficiente.

Complejidad temporal

La complejidad temporal es muy alta debido a la generación de todos los árboles y el cálculo de distancias:

- Generación de árboles: Aproximadamente $O(2^n)$ para árboles no etiquetados.
- Cálculo de distancias: Para cada árbol, calcular las distancias toma $O(n^2)$.

Por lo tanto, la complejidad total puede ser considerada como $O(2^n \cdot n^2)$.

Ejemplo de código (fuerza bruta)

```
import itertools
from collections import defaultdict, deque


def generate_trees(n):
"""Genera todos los árboles no etiquetados con n vértices."""
if n == 1:
yield {0: []}
return

vertices = list(range(n))
# Generar todas las combinaciones de aristas posibles
for edges in itertools.combinations(itertools.combinations(vertices, 2), n - 1):
adjacency_list = defaultdict(list)

# Construir la lista de adyacencia
for u, v in edges:
adjacency_list[u].append(v)
adjacency_list[v].append(u)

# Verificar si es un árbol
if is_tree(adjacency_list, n):
yield adjacency_list


def is_tree(adjacency_list, n):
```

```
"""Verifica si el grafo es un árbol (conectado y sin ciclos)."""
visited = set()
queue = deque([0])  # Comenzamos desde el nodo 0

while queue:
node = queue.popleft()
if node not in visited:
visited.add(node)
for neighbor in adjacency_list[node]:
if neighbor not in visited:
queue.append(neighbor)

return len(visited) == n and all(len(adjacency_list[v]) > 0 for v in range(n))


def calculate_distance(tree, u, v):
"""Calcula la distancia entre dos vértices en el árbol usando BFS."""
if u == v:
return 0

queue = deque([(u, 0)])  # (nodo actual, distancia)
visited = set()

while queue:
current_node, distance = queue.popleft()

if current_node == v:
return distance

visited.add(current_node)

for neighbor in tree[current_node]:
if neighbor not in visited:
queue.append((neighbor, distance + 1))

return float(
"inf"
)  # En caso de que no se encuentre (no debería suceder en un árbol)


def count_pairs(tree):
"""Cuenta los pares de vértices con distancias par e impar."""
distances = defaultdict(int)
vertices = list(tree.keys())

for u in vertices:
```

```
for v in vertices:
distance = calculate_distance(tree, u, v)
distances[distance % 2] += 1  # 0 para par, 1 para impar

return distances[0], distances[1]


def find_tree_with_distances(x, y):
"""Encuentra un árbol que cumpla con las condiciones de pares de distancias."""
total_pairs = x + y
n = int(total_pairs**0.5)  # Calcular el número de vértices

if n * n != total_pairs:  # Asegurarse de que sea un cuadrado perfecto
return None

for tree in generate_trees(n):
even_count, odd_count = count_pairs(tree)
if even_count == x and odd_count == y:
return tree
return None


# Ejemplo de uso
x = 221  # Pares con distancia par
y = 220  # Pares con distancia impar

result = find_tree_with_distances(x, y)
if result:
print("Árbol encontrado:", result)
else:
print("No se encontró ningún árbol que cumpla las condiciones.")
```

Solución óptima

Una solución más eficiente consiste en entender cómo se distribuyen los pares según su paridad. En un árbol bipartito (que es lo que se obtiene al colorear un árbol), los vértices se pueden dividir en dos conjuntos:

- Conjunto A (vértices en niveles pares)
- Conjunto B (vértices en niveles impares)

La cantidad total de pares se puede calcular como:
- Pares con distancia par: Se forman al elegir dos vértices del mismo conjunto. - Pares con distancia impar: Se forman al elegir un vértice del conjunto A y otro del conjunto B.

Correctitud del algoritmo

Esta solución es correcta porque utiliza la propiedad estructural del árbol bipartito para contar eficientemente los pares según su paridad sin necesidad de generar todos los árboles.

Complejidad temporal

La complejidad temporal se reduce significativamente a:
- Determinar el tamaño de cada conjunto: $O(n)$. - Contar los pares: $O(1)$.
Por lo tanto, la complejidad total es $O(n)$.

Ejemplo de código en Python (óptimo)

```
def find_tree_structure(x, y):
# Iterar sobre posibles valores para a
for kA in range(int((x + y) ** 0.5) + 1):
# Calcular b usando la fórmula derivada
kB_squared = x - kA**2
if kB_squared < 0:
continue

kB = int(kB_squared**0.5)

if kA**2 + kB**2 == x and 2 * kA * kB == y:
return (kA, kB)

return None
```

Conclusión

Este reporte presenta dos enfoques para resolver el problema de construir un árbol basado en las distancias entre sus vértices. La solución trivial explora todas las configuraciones posibles pero es ineficiente para grandes valores de n. En contraste, la solución óptima aprovecha la estructura bipartita del árbol para contar pares eficientemente.