DAA

Proyecto Final

Ejercicio 1- 440 D. Berland Federalization

Autor:

Karen Dianelis Cantero López (C411) karen.canterolopez@gmail.com

2024

Contents

1 Definición formal del problema

Recently, Berland faces federalization requests more and more often. The proponents propose to divide the country into separate states. Moreover, they demand that there is a state which includes exactly k towns.

Currently, Berland has n towns, some pairs of them are connected by bilateral roads. Berland has only n-1 roads. You can reach any city from the capital, that is, the road network forms a tree.

The Ministry of Roads fears that after the reform those roads that will connect the towns of different states will bring a lot of trouble.

Your task is to come up with a plan to divide the country into states such that:

- each state is connected, i.e. for each state it is possible to get from any town to any other using its roads (that is, the roads that connect the state towns),
- there is a state that consisted of exactly k cities,
- the number of roads that connect different states is minimum.

1.0.1 Input

The first line contains integers n, k ($1 \leq k \leq n \leq 400$). Then follow n-1 lines, each of them describes a road in Berland. The roads are given as pairs of integers $x_i, y_i (1 \leq x_i, y_i \leq n; x_i \neq y_i)$ — the numbers of towns connected by the road. Assume that the towns are numbered from 1 to n.

1.0.2 Output

The the first line print the required minimum number of "problem" roads t. Then print a sequence of t integers — their indices in the found division. The roads are numbered starting from 1 in the order they follow in the input. If there are multiple possible solutions, print any of them.

If the solution shows that there are no "problem" roads at all, print a single integer 0 and either leave the second line empty or do not print it at all.

2 Reformulando el problema

Divida el árbol en una cantidad de subconjuntos (diferentes) conectados de nodos (o subárboles) en el árbol, con al menos uno de los subárboles teniendo exactamente K nodos. Luego, se debe mostar la cantidad de aristas que conectan los diferentes subárboles y los números de aristas.

3 Solución

3.1 Explicación

3.1.1 Algoritmo

El objetivo del algoritmo es resolver un problema de partición de un árbol en subárboles conectados, asegurando que uno de los subárboles tenga exactamente k nodos, minimizando el numero de aristas que conectan estos subárboles.

Para esto, utilizaremos un enfoque basado en programación dinámica (DP) y una búsqueda en profundidad (DFS) para explorar el árbol.

1. El algoritmo utiliza un recorrido en profundidad (DFS) para visitar cada nodo del árbol, comenzando desde un nodo raíz. A medida que se avanza en el árbol, se va acumulando información sobre los subárboles que se pueden formar a partir de cada nodo.

2. Para cada nodo, utilizaremos una tabla DP para registrar el costo de formar un subarbol con tamaño i a partir del nodo en el que estemos parados

   De igual modo utilizaremos una tabla DPR para almacenar la numeración de las carreteras que se utilizan para construir la solución. Esta información es necesaria guardarla ya que es lo que se pide en el ejercicio.

3. Cada vez que el DFS visita un nodo, combina los resultados de sus hijos para crear subárboles más grandes. Por ejemplo, si el nodo hijo puede formar un subárbol con j nodos, el algoritmo intentará combinar este subárbol con los subárboles ya formados en el nodo actual para formar subárboles de tamaño $i + j$ nodos. Mientras se combinan los subárboles, el algoritmo selecciona las combinaciones que minimizan la cantidad de aristas utilizadas para conectar los subárboles. Este proceso es repetido para todas las combinaciones posibles de nodos y tamaños.

4. Durante el proceso de combinación de subárboles, se actualizan ambas tablas DP y DPR.

5. Una vez que se ha recorrido todo el árbol, el algoritmo habrá encontrado la manera óptima de dividir el árbol en subárboles conectados, con uno de esos subárboles teniendo exactamente K nodos.

   Finalmente, se imprime el número de aristas que conectan estos subárboles y cuáles son esas aristas.

Para una información detallada sobre el código y la solución ver el archivo 'problem.md'

4 Demostración de correctitud

4.1 Subestructura óptima

Vamos a demostrar que el problema tiene la propiedad de **subestructura óptima** por reducción al absurdo.

1. Supongamos que el problema no tiene una subestructura óptima: Suponemos que la solución al problema de dividir el árbol entero en estados (donde un estado tiene exactamente k nodos y minimiza la cantidad de caminos entre estados) no puede construirse de manera óptima a partir de soluciones a subproblemas más pequeños (es decir, dividiendo los subárboles en estados de manera óptima).

   En otras palabras, asumimos que la solución óptima para el árbol entero no depende de las soluciones óptimas de sus subárboles.

2. Consideremos un árbol T con raíz en el nodo u, y sean $T_{v_1}, T_{v_2}, \ldots, T_{v_m}$ los subárboles con raíz en los hijos de u. La solución óptima para T implica dividir este árbol en subárboles conectados (estados) de manera que un estado tenga exactamente k nodos, y se minimice la cantidad de aristas que conectan diferentes estados.

   Según nuestra suposición, la solución óptima para T no está compuesta por las soluciones óptimas para los subproblemas que involucran a $T_{v_1}, T_{v_2}, \ldots, T_{v_m}$.

   Esto significa que hay una manera de resolver los subproblemas $T_{v_1}, T_{v_2}, \ldots, T_{v_m}$ que no es óptima, pero que cuando se combina, conduce a una solución óptima para T.

3. Contradicción: Esta implicación es contradictoria porque:

   Si una solución para un subárbol T_{v_i} no es óptima, introduce aristas innecesarias que conectan diferentes estados dentro del subárbol.

   Cuando combinamos estas soluciones no óptimas de los subárboles $T_{v_1}, T_{v_2}, \ldots T_{v_m}$, el número total de aristas entre estados para todo el árbol T también incluirá estas aristas innecesarias.

   Por lo tanto, si los subproblemas se resuelven de manera no óptima, el número de aristas que conectan diferentes estados para todo el árbol T aumentará, lo que contradice la suposición de que la solución para T es óptima (es decir, que minimiza la cantidad de aristas entre estados).

Dado que nuestra suposición nos lleva a una contradicción, podemos concluir que la suposición original es falsa. Por lo tanto, el problema sí tiene la propiedad de subestructura óptima.

Esto significa que la solución óptima del problema está compuesta de soluciones óptimas de sus subproblemas.

4.2 Subproblemas superpuestos

Recordemos que el algoritmo que seguimos para resolver el problema está enfocado en:

1. Resolver subproblemas para los subárboles
2. Combinar las soluciones a estos subproblemas para obtener la solución del problema general

Para cada nodo u en el árbol, intentamos dividir su subárbol en componentes conectados (estados) de múltiples maneras. Específicamente, intentamos asignar diferentes números de nodos a diferentes subárboles, y para cada posible asignación de nodos, resolvemos recursivamente los subproblemas para los hijos.

Sea $OPT(T_u, x)$ el número mínimo de aristas que conectan diferentes estados cuando particionamos el subárbol con raíz en el nodo u de tal manera que uno de los estados contenga exactamente x nodos

Para demostrar que el problema tiene subproblemas superpuestos, necesitamos demostrar que el mismo subproblema se resuelve varias veces durante este proceso recursivo. En este caso es evidente ya que:

1. La estructura de árbol conduce a cálculos repetitivos de subárboles en diferentes llamadas recursivas. Por ejemplo, tenemos un nodo u con múltiples hijos. Para cada subárbol hijo T_{v_i}, calculamos recursivamente la cantidad de formas de dividirlo en estados conectados para diferentes cantidades de nodos (desde 1 hasta el tamaño del subárbol).

   Esto significa que podemos resolver el mismo subproblema varias veces:

   - Resolvemos $OPT(T_{v_i}, x)$ al considerar cómo particionar el subárbol con raíz en u
   - También podemos resolver $OPT(T_{v_i}, x)$ nuevamente al considerar cómo particionar el subárbol con raíz en un nodo diferente, como uno de los ancestros de u u otro nodo en el árbol.

   Por lo tanto, el mismo subproblema se puede resolver varias veces en diferentes llamadas recursivas.

4.2.1 Ejemplo de subproblemas superpuestos

Supongamos que tenemos un árbol donde el nodo u tiene dos hijos v_1, v_2.

Para resolver $OPT(T_u, k)$, necesitamos decidir cómo dividir los k nodos entre los subárboles con raíz en v_1 y v_2. Para cada división posible, calculamos $OPT(T_{v_1}, x_1)$ y $OPT(T_{v_2}, x_2)$, donde $x_1 + x_2 = k - 1$ (ya que u toma un nodo).

Ahora, supongamos que nos movemos hacia arriba un nivel para resolver $OPT(T_p, k)$ para algún nodo padre p de u. Esto podría requerir nuevamente resolver $OPT(T_u, x_3)$, donde x_3 es una cierta cantidad de nodos asignados al subárbol con raíz en u. Si $x_3 = k$, entonces estamos repitiendo el mismo cálculo que ya se hizo para $OPT(T_u, k)$.

De manera similar, al particionar T_p, es posible que necesitemos resolver $OPT(T_{v_1}, x_1)$ y $OPT(T_{v_2}, x_2)$ nuevamente, lo que lleva a una mayor repetición.

4.2.2 Conclusión

Para resumir, la propiedad de superposición de subproblemas se cumple en este problema porque:

- Los subárboles con raíces en diferentes nodos del árbol hacen que los mismos subproblemas recursivos se resuelvan varias veces.
- Las diferentes llamadas recursivas implican la división de nodos entre hijos de maneras que dan como resultado que el mismo subproblema se resuelva repetidamente.

  La superposición de subproblemas ocurre porque el mismo subárbol puede ser parte de diferentes particiones de un subárbol más grande

4.3 Correctitud de la recurrencia de dp

La solución propuesta utiliza la siguiente recurrencia DP:

```
if(dp[v][i + j] > dp[child][j] + dp[v][i]){
            dp[v][i + j] = dp[child][j] + dp[v][i];
```

donde:
$dp[v][i]$ representa el número mínimo de aristas necesarias para dividir el subárbol con raíz en el nodo v en estados, uno de los cuales contiene i ciudades. child es un nodo hijo de v.

1. **Caso Base:** El caso base es cuando un subárbol consta de un solo nodo, que no requiere aristas para formarse:
$$dp[v][1] = 0$$
Esto es correcto porque si el subárbol con raíz en el nodo v tiene un solo nodo, no se necesitan aristas para conectarlo con otros nodos (ya que es un subárbol de un solo nodo).

2. **Transición en DP** Al procesar el nodo v, el algoritmo considera todas las posibles divisiones de los tamaños de los subárboles entre el nodo actual v y sus hijos.
   - Queremos actualizar el costo mínimo para obtener un estado con $i+j$ nodos, donde estamos tomando i nodos del subárbol de v y j nodos del subárbol de un hijo child.
   - La recurrencia dp[v][i+j]=min(dp[v][i+j],dp[child][j]+dp[v][i]) garantiza que la solución del subárbol más grande con raíz en v se construya a partir de soluciones óptimas para subproblemas más pequeños.
   - Esto captura correctamente la subestructura óptima del problema: si la solución para los subárboles con raíz en los hijos es óptima, la combinación de estos subárboles producirá una solución óptima para el subárbol con raíz en v.

3. **Minimización:** La recurrencia garantiza que siempre elijamos la cantidad mínima de aristas para formar un subárbol de tamaño $i+j$ en el nodo v. Si hay múltiples formas de formar un subárbol de tamaño $i+j$, la recurrencia toma la que requiere menos aristas, ya que solo actualiza su valor si el nuevo es menor que el que ya está calculado.

4. **Optimalidad:** Dado que el DFS explora el árbol de abajo hacia arriba (comenzando por las hojas y avanzando hasta la raíz), cada subárbol se calcula por completo antes de combinarlo con su padre. Esto garantiza que, al fusionar dos subárboles, los valores guardados en la tabla DP ya sean óptimos para los tamaños dados i y j. Por lo tanto, los valores de DP finales representan la cantidad mínima global de aristas necesarias para formar subárboles de diferentes tamaños en cada nodo.

5 Complejidad

5.1 Temporal

La complejidad temporal de la solución es $O(n^3)$.

1. DFS: La búsqueda en profundidad (DFS) itera sobre todos los vértices del gráfico, lo que lleva $O(n)$ tiempo.
2. Actualizaciones de la tabla DP: Para cada vértice 'v' y cada tamaño de estado posible 'i', el código itera sobre todos los tamaños de estado 'j' encontrados previamente. Esto da como resultado bucles anidados, cada uno de los cuales lleva O(n) tiempo. Por lo tanto, el tiempo total para las actualizaciones de la tabla DP es $O(n^3)$.
3. Otras operaciones: Las operaciones restantes, como inicializar la tabla DP, verificar las conexiones mínimas y actualizar 'answ', tienen una complejidad de tiempo de $O(n^2)$ o menos.

Dado que el término $O(n^3)$ de las actualizaciones de la tabla DP domina la complejidad de tiempo general, la complejidad de tiempo final de la solución es $O(n^3)$.

5.2 Espacial

La complejidad espacial de la solución dada es: $O(n^2)$.

1. Los array 'dp[N][N]','dpr[N][N]' y 'rebro[N][N]' tienen dimensiones $n \times n$, por lo que tienen complejidad espacial de $O(n^2)$.
2. Otras estructuras de datos: El array 'g[N]' tiene dimensión n por lo que su complejidad espacial es $O(n)$.

Dado que el término $O(n^2)$ de la tabla DP domina la complejidad espacial, esta es la complejidad total del problema.