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.

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

La idea intuitiva del algoritmo presentado 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, se utiliza 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, el algoritmo utiliza una tabla para registrar información sobre los subárboles que se pueden formar a partir de ese nodo. En particular, se guarda:

   - Cuántos subárboles conectados se pueden formar con un cierto número de nodos.
   - El costo asociado a formar esos subárboles, que se mide en términos del número de aristas utilizadas.

   Este cálculo se hace en el nodo actual, basándose en los subárboles que ya se han calculado en sus nodos hijos. A medida que se procesan los hijos, el algoritmo combina los subárboles pequeños que se pueden formar en los hijos para crear subárboles más grandes en el nodo actual.

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.

4. Durante el proceso de combinación de subárboles, el algoritmo guarda un registro de las aristas que han sido utilizadas para formar los subárboles. Esto es importante porque al final se necesita saber qué aristas se utilizan para conectar los subárboles.

5. El objetivo final es encontrar un conjunto de subárboles en el que al menos uno tenga exactamente K nodos, minimizando la cantidad de aristas que conectan esos subárboles entre sí. Para esto, el algoritmo compara diferentes combinaciones de subárboles y elige la que resulta en el menor número de aristas

6. 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 6

4 Demostración de correctitud

4.1 Subestructura óptima

Vamos a demostrar que el problema tiene la propiedad de **subestructura óptima** por contradiccion.

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, estamos asumiendo 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 $T_{v_1}, T_{v_2}, \ldots, T_{v_m}$.

   Dado que asumimos que la solución para todo el árbol T no está compuesta de soluciones óptimas para sus subproblemas, esto significa que hay una manera de resolver los subproblemas $T_{v_1}, T_{v_2}, \ldots, T_{v_m}$ que es no óptima, pero que cuando se combina, conduce a una solución óptima para T.

   Esto implica que:

   (a) Podemos encontrar una partición no óptima de los subárboles $T_{v_1}, T_{v_2}, \ldots, T_{v_m}$ (es decir, soluciones que no minimizan la cantidad de aristas que conectan diferentes estados en cada subárbol).

   (b) Pero de alguna manera, la combinación de estas particiones no óptimas aún da como resultado una partición óptima general para todo el árbol T, donde se minimiza la cantidad de aristas que conectan diferentes estados.

3. Contradiccion: 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, se minimiza la cantidad de aristas entre estados).

   Por lo tanto, la suposición de que el problema no tiene la propiedad de subestructura óptima lleva a la conclusión de que una solución no óptima para los subproblemas de alguna manera produciría una solución óptima para todo el problema. Esto es una clara contradicción.

Puesto que nuestra suposición conduce 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 de todo el problema está compuesta de hecho por soluciones óptimas de sus subproblemas.

Por lo tanto, la propiedad de subestructura óptima se cumple para este problema y la programación dinámica se puede aplicar de manera efectiva para resolverlo.

4.2 Subproblemas superpuestos

Recordemos 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 particionado 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 el caso de nuestro problema, los subproblemas superpuestos surgen porque:

1. Cálculos repetitivos de subárboles: La estructura de árbol conduce a cálculos repetitivos de subárboles en diferentes llamadas recursivas. Por ejemplo, considere 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:

   - Si 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 (por ejemplo, cómo particionar un subárbol con x nodos) se puede resolver varias veces en diferentes llamadas recursivas.

2. Asignaciones repetitivas de recuento de nodos: la naturaleza recursiva del problema implica calcular repetidamente el costo mínimo de particionar un subárbol con números específicos de nodos. Dado que los subproblemas implican determinar cómo dividir cada subárbol en función de diferentes valores de x (la cantidad de nodos en el estado), los mismos valores de x se usan repetidamente para el mismo subárbol.

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.

Por lo tanto, durante el proceso recursivo, los mismos subproblemas se resuelven varias veces porque diferentes partes del árbol llevan a que el mismo subproblema se encuentre repetidamente.

4.2.2 Conclusión

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 (es decir, dividir un subárbol con una cierta cantidad de nodos) 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 proporcionada 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 k 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.
   - 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.
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 fusionarlo con su padre. Esto garantiza que, al fusionar dos subárboles, los valores de DP utilizados en la recurrencia 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 nxn, 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.

6 Código

```
const long long N = 1000 + 77 , inf = 1e18 + 77 , MOD = 1e9 + 7;
const long double eps = 1e-11;
using namespace std;

int binpow(int a , int b){
        if(!b) return 1;
        int val = binpow(a , b / 2);
        if(b % 2 == 0) return val * val % MOD;
        else return val * val * a % MOD;
}

int n , k , ans;

vector <int> g[N], dpr[N][N], nwr[N], answ;
int dp[N][N];
int rebro[N][N];

void dfs(int v , int p = 0){
        dp[v][1] = 0;
        for(int to : g[v]) if(to != p){
                dfs( to, v );
                for(int i = n; i >= 1; i--){
                        if(dp[v][i] > 500) pofik;
                        for(int j = 1; j <= n; j++){
                                if(dp[v][i + j] > dp[to][j] + dp[v][i]){
                                        dp[v][i + j] = dp[to][j] + dp[v][i];
                                        dpr[v][i + j] = dpr[to][j];
                                        for(auto ab : dpr[v][i]) dpr[v][i + j].pb(ab);
                                }
                        }
                        dp[v][i]++;
                        dpr[v][i].pb(rebro[v][to]);
                }
        }
        if(dp[v][k] + (p != 0) < ans){
                ans = dp[v][k] + (p != 0);
                answ = dpr[v][k];
                if(p) answ.pb( rebro[p][v] );
        }

}

void solve(){
    cin >> n >> k;
    for(int i = 1; i < n; i++){
            int v , u;
            cin >> v >> u;
            g[v].pb(u);
            g[u].pb(v);
            rebro[v][u] = rebro[u][v] = i;
    }
    for(int i = 0; i <= n; i++){
            for(int j = 0; j <= k; j++){
```

```
                    dp[i][j] = inf;
            }
    }
    ans = inf;
    dfs(1);
    cout << answ.sz << '\n';
    for(int it : answ){
            cout << it << ' ';
    }
}

signed main(){
    solve();
}
```

6.1 Explicación

Este código aborda el problema de dividir Berland en estados con conexiones interestatales mínimas, al tiempo que garantiza que al menos un estado tenga exactamente ‘k‘ ciudades. A continuación, se muestra un desglose de cómo funciona:

6.1.1 Constantes y bibliotecas:

- ‘N‘: Número máximo de ciudades (1077).
- ‘inf‘: Representa el infinito para los cálculos (valor suficientemente grande).
- ‘MOD‘: Módulo para los cálculos (evita el desbordamiento).
- ‘eps‘: Valor de épsilon para comparaciones de punto flotante.
- ‘T‘: Número de casos de prueba (comentados, suponiendo un solo caso).

6.1.2 Funciones de utilidad:

- ‘binpow(a, b)‘: Calcula la exponenciación modular de manera eficiente.

6.1.3 Variables globales:

- ‘n‘: Número de ciudades en Berland.
- ‘k‘: Número objetivo de ciudades en al menos un estado.
- ‘ans‘: Almacena el número mínimo de conexiones interestatales encontradas hasta el momento.
- ‘g[N]‘: Lista de adyacencia que representa la red de carreteras (ciudades conectadas por carreteras).
- ‘dpr[N][N]‘: Almacena información adicional para soluciones óptimas (explicado más adelante).
- ‘answ‘: Almacena la lista de carreteras que forman las conexiones interestatales mínimas.
- ‘dp[N][N]‘: Almacena el número mínimo de conexiones intraestatales necesarias para un estado con ‘i‘ ciudades (i filas) considerando estados con ‘j‘ ciudades encontrados previamente (j columnas).
- ‘rebro[N][N]‘: Almacena el índice de la carretera que conecta cada par de ciudades para referencia posterior.

6.1.4 Función ‘dfs‘ (Búsqueda en profundidad):

1. Toma un vértice (‘v‘) y su padre (‘p‘, opcional) como argumentos.
2. Inicializa ‘dp[v][1]‘ (conexiones mínimas para un estado con 1 ciudad) a 0.
3. Itera a través de todos los vecinos (‘to‘) de ‘v‘ excepto el padre.
4. Llama recursivamente a ‘dfs‘ para cada vecino (‘to‘).
5. Itera a través de todos los tamaños de estado posibles (‘i = n‘ hasta 1).
6. Comprueba si el mínimo actual para un estado con ‘i + j‘ ciudades (‘dp[v][i + j]‘) es mayor que la suma de las conexiones mínimas para los estados con ‘j‘ ciudades (‘dp[to][j]‘) y ‘i‘ ciudades (‘dp[v][i]‘).
   (a) Si es así, actualiza ‘dp[v][i + j]‘ con la suma y almacena la combinación correspondiente de los estados encontrados previamente en ‘dpr[v][i + j]‘.
   (b) Esta matriz ‘dpr‘ ayuda a rastrear la solución óptima al mantener información sobre las carreteras utilizadas para lograr ese recuento mínimo de conexiones.
7. Incrementa ‘dp[v][i]‘ (número de conexiones necesarias para un estado con ‘i‘ ciudades considerando ‘v‘).
8. Agrega el índice de la carretera que conecta ‘v‘ y su vecina (‘to‘) a ‘dpr[v][i]‘.
9. Comprueba si las conexiones mínimas para un estado con ‘k‘ ciudades a partir de ‘v‘ (‘dp[v][k]‘) más una conexión al padre (‘(p != 0)‘) es menor que la mejor solución actual (‘ans‘).
   (a) Si es así, actualiza ‘ans‘ con el nuevo mínimo y ‘answ‘ con la lista correspondiente de carreteras que forman las conexiones interestatales mínimas utilizando la información almacenada en ‘dpr[v][k]‘.
   (b) Si el padre existe (‘p != 0‘), agrega la carretera que conecta ‘v‘ y su padre a ‘answ‘.

6.1.5 ‘solve‘ Función:

1. Lee la cantidad de ciudades (‘n‘) y el tamaño del estado de destino (‘k‘) de la entrada.
2. Lee las conexiones entre ciudades y las almacena en la lista de adyacencia ‘g‘ y en la matriz ‘rebro‘ para indexar las carreteras.
3. Inicializa ‘dp‘ con infinito para todas las combinaciones.
4. Establece ‘ans‘ en infinito para la comparación inicial.
5. Llama a ‘dfs(1)‘ para iniciar la búsqueda en profundidad desde la primera ciudad (‘1‘).
6. Imprime el tamaño de la lista ‘answ‘ (número de conexiones entre estados).
7. Imprime cada índice de carretera en la lista ‘answ‘, que representa las conexiones mínimas entre estados.

En resumen, el código utiliza programación dinámica con una búsqueda en profundidad para explorar todas las particiones de estado posibles. Realiza un seguimiento de la cantidad mínima de conexiones necesarias para diferentes tamaños de estado y las combinaciones de carreteras correspondientes utilizando las matrices ‘dp‘ y ‘dpr‘