

# Optimización de Red de Fibra Óptica (Degree-Constrained Minimum Spanning Tree)

Ernesto Abreu Peraza<sup>1</sup> and Eduardo Brito Labrada<sup>1</sup>

<sup>1</sup>Diseño y Análisis de Algoritmos - Universidad de La Habana

6 de enero de 2026

## Resumen

Este informe detalla el análisis y la solución al problema del Árbol de Expansión Mínimo con Grado Restringido (DCMST) para la infraestructura de red de la Universidad de La Habana. Se presenta la formalización matemática, la demostración de su complejidad NP-Hard y una comparativa experimental entre diferentes algoritmos implementados en C++.

# Índice

<b>1. Formalización del Problema</b>	<b>3</b>
1.1. Modelo Matemático . . . . .	3
1.2. Variantes del Problema . . . . .	3
1.2.1. DCMST en grafos completos . . . . .	3
1.2.2. DCMST con restricción de grado uniforme . . . . .	4
1.2.3. Degree-Constrained Spanning Tree (DCST) . . . . .	4
<b>2. Análisis de Complejidad Computacional</b>	<b>4</b>
2.1. Demostración de NP-Hardness . . . . .	4
2.2. Degree-Constrained Spanning Tree . . . . .	4
<b>3. Diseño de Soluciones Algorítmicas</b>	<b>4</b>
3.1. Enumeración con Máscara de Bits para el DCMST . . . . .	5
3.1.1. Descripción del Algoritmo . . . . .	5
3.1.2. Análisis de Correctitud . . . . .	5
3.1.3. Análisis de Complejidad . . . . .	6
3.2. Enumeración Lexicográfica para el DCMST . . . . .	6
3.2.1. Descripción del Algoritmo . . . . .	6
3.2.2. Análisis de Correctitud . . . . .	6
3.2.3. Análisis de Complejidad . . . . .	7
3.3. Enumeración con Código de Gray . . . . .	7
3.3.1. Marco Teórico: Código de Gray . . . . .	7
3.3.2. Descripción del Algoritmo Gray Iterativo . . . . .	8
3.3.3. Análisis de Correctitud del Algoritmo Gray Iterativo . . . . .	8
3.3.4. Análisis de Complejidad del Algoritmo Gray Iterativo . . . . .	8
3.3.5. Descripción del Algoritmo Gray Recursivo . . . . .	8
3.3.6. Análisis de Correctitud del Algoritmo Gray Recursivo . . . . .	9
<b>4. Implementación y Análisis Experimental</b>	<b>9</b>
<b>5. Conclusiones</b>	<b>9</b>
<b>A. Apéndice: Implementaciones en C++</b>	<b>10</b>
A.1. Solución 1: Fuerza Bruta con Máscara de Bits . . . . .	10
A.2. Solución 2: Generación Combinatoria Lexicográfica . . . . .	11
A.3. Solución 3: Código de Gray Iterativo . . . . .	12
A.4. Solución 4: Código de Gray Recursivo con Poda . . . . .	13

# 1. Formalización del Problema

Partiendo de la necesidad de interconectar los edificios de la universidad minimizando costos y respetando la capacidad de puertos de ETECSA, definimos el modelo matemático.

## 1.1. Modelo Matemático

Sea  $G = (V, E)$  un grafo conexo y no dirigido, donde:

- $V$  es el conjunto de edificios.
- $E$  es el conjunto de posibles conexiones de fibra.
- $w : E \rightarrow \mathbb{R}^+$  es una función de costo.
- $k : V \rightarrow \mathbb{N}$  es la capacidad de puertos por edificio.

El problema consiste en encontrar un subgrafo  $T = (V, E')$  tal que:

$$T^* = \arg \min_{T \in \mathcal{T}} \sum_{e \in E'} w(e) \quad (1)$$

Sujeto a:

1.  $T$  es un árbol de expansión de  $G$ .
2.  $\forall v \in V, \deg_T(v) \leq k(v)$ .

Este problema es conocido como *Degree-Constrained Minimum Spanning Tree* (DCMST).

## 1.2. Variantes del Problema

Analicemos algunas variantes, las cuales surgen a partir de diferentes supuestos sobre la estructura del grafo y las restricciones de grado.

### 1.2.1. DCMST en grafos completos

Una variante relevante del DCMST es aquella en la que el grafo se asume completo. Formalmente, se considera un grafo no dirigido

$$G = (V, E),$$

donde

$$E = \{\{u, v\} \mid u, v \in V, u \neq v\}.$$

Este modelo se ajusta bien al problema original, dado que de no existir una conexión original entre dos vértices (porque sea imposible conectar dos edificios), puedes asignar un costo *infinito* a dicha arista, asegurando que no será seleccionada en la solución óptima. De ser seleccionada representaría que no se encontró solución factible.

Analizando la complejidad temporal, el carácter completo del grafo incrementa significativamente el número de aristas al peor caso,

$$|E| = \frac{|V|(|V| - 1)}{2} = O(|V|^2).$$

### 1.2.2. DCMST con restricción de grado uniforme

Otra variante ampliamente estudiada es aquella en la que el límite de grado es uniforme para todos los vértices del grafo. En este caso, se fija un valor entero constante  $K \geq 2$  tal que

$$\deg_T(v) \leq K \quad \forall v \in V,$$

donde  $\deg_T(v)$  denota el grado del vértice  $v$  en el árbol generador  $T$ .

Esta formulación es adecuada en el caso en que el equipamiento de red sea el mismo para todos los edificios.

Además si tomamos  $K = \min_{v \in V} k(v)$ , una solución optima de esta instancia es una solución factible, aunque no necesariamente óptima, para la instancia del problema original.

### 1.2.3. Degree-Constrained Spanning Tree (DCST)

Una variante adicional es el problema Degree-Constrained Spanning Tree. En esta versión, el objetivo no consiste en minimizar el peso total del árbol, sino únicamente en determinar la existencia de un árbol generador que satisfaga las restricciones estructurales.

Este problema puede ser formulado como un problema de decisión o factibilidad. A pesar de la ausencia de una función objetivo de optimización, la determinación de la existencia de una solución válida sigue siendo computacionalmente difícil en el caso general. Esta variante resulta de interés teórico y se emplea frecuentemente como base para el estudio de la complejidad del DCMST y el diseño de algoritmos exactos y aproximados.

En la siguiente sección, se analizará la complejidad computacional del DCMST y sus variantes. En la experimentación nos enfocaremos principalmente analizando DCMST en grafos completos con restricción de grado uniforme por los beneficios planteados anteriormente.

## 2. Análisis de Complejidad Computacional

### 2.1. Demostración de NP-Hardness

Para demostrar que el problema es NP-Hard, realizamos una reducción desde el problema *Hamiltonian Path*, un conocido problema NP-Completo.

**Teorema 2.1.** *El problema DCMST es NP-Hard.*

*Demostración.* Dada una instancia del problema *Hamiltonian Path*, construimos una instancia del problema DCMST donde cada vértice tiene grado máximo 2. Si existe un árbol de expansión que cumpla las restricciones de grado, entonces tenemos un camino de Hamilton del grafo original.

Si supiéramos resolver DCMST en tiempo polinomial, podríamos resolver *Hamiltonian Path* en tiempo polinomial. Por lo tanto, DCMST es NP-Hard.  $\square$

### 2.2. Degree-Constrained Spanning Tree

El problema de decisión Degree-Constrained Spanning Tree (DCST), en el cual queremos saber si existe un árbol de expansión que cumpla con las restricciones de grado, es NP-Completo. Ya que es fácil verificar que un árbol dado es de expansión de  $G$  y cumple con las restricciones de grado en tiempo polinomial.

## 3. Diseño de Soluciones Algorítmicas

Se desarrollaron e implementaron cuatro enfoques principales. El código fuente completo de cada implementación se encuentra en el Apéndice A.

### 3.1. Enumeración con Máscara de Bits para el DCMST

Esta es una solución exacta para el problema DCMST basada en **enumeración exhaustiva mediante máscara de bits**. Esta solución está diseñada para instancias de tamaño pequeño, sirviendo como algoritmo de referencia y verificación de metaheurísticas.

El algoritmo explora todos los subconjuntos posibles de aristas que contienen exactamente  $|V| - 1$  aristas y verifica cuáles inducen un árbol generador que cumple con la restricción de grado. Entre todas las soluciones factibles, selecciona la de costo mínimo. La implementación completa se encuentra en el Apéndice A.1.

#### 3.1.1. Descripción del Algoritmo

Sea  $G = (V, E)$  un grafo completo, no dirigido y ponderado, con  $|V| = n$  vértices y  $|E| = m = \frac{n(n-1)}{2}$  aristas. El algoritmo procede de la siguiente manera:

1. Se enumeran todas las aristas  $E$  y se indexan de 0 a  $m - 1$ .
2. Se recorre cada máscara de bits  $mask \in \{0, 1\}^m$ .
3. Solo se consideran las máscaras cuyo número de bits activos es exactamente  $n - 1$ .
4. Cada máscara define un subgrafo  $G_{mask}$  compuesto por las aristas seleccionadas.
5. Se verifica si:
  - El subgrafo es conexo (usando Depth-First Search).
  - El grado de cada vértice es a lo sumo  $k$ .
6. Si ambas condiciones se cumplen, el subgrafo es un árbol generador factible y se evalúa su costo.
7. Se devuelve el mínimo costo entre todas las soluciones factibles.

El uso de una máscara de bits permite representar subconjuntos de aristas de forma compacta y eficiente a nivel de implementación.

#### 3.1.2. Análisis de Correctitud

Demostraremos que el algoritmo es correcto, es decir, que devuelve exactamente el costo del árbol generador mínimo con restricción de grado.

**Lema 3.1.** *El algoritmo examina **todos** los subconjuntos de aristas de tamaño  $n - 1$ .*

*Demostración.* Cada máscara de bits  $m$  representa un subconjunto único de aristas. Al iterar sobre todas las máscaras con  $\text{popcount}(mask) = n - 1$ , se enumeran exactamente todos los subconjuntos de  $E$  con  $n - 1$  aristas.  $\square$

**Lema 3.2.** *Un subconjunto de aristas  $E' \subseteq E$  con  $|E'| = n - 1$  es un árbol generador factible si y solo si:*

1. *El grafo inducido es conexo.*
2. *Para todo vértice  $v$ ,  $\deg(v) \leq k$ .*

*Demostración.* Un grafo conexo con  $n - 1$  aristas es un árbol. La segunda condición garantiza la restricción de grado. Por lo tanto, ambas condiciones son necesarias y suficientes.  $\square$

**Lema 3.3.** *La función **check** devuelve verdadero si y solo si el subgrafo inducido por la máscara es un árbol generador factible.*

*Demostración.* Inicialmente, verifica que ningún vértice tenga grado mayor que  $k$ . Luego, ejecuta un DFS desde el vértice 0 y comprueba que todos los vértices son alcanzables, lo que implica conectividad. Por el Lema 3.2, esto es equivalente a ser un DCST factible.  $\square$

**Teorema 3.4.** *El algoritmo devuelve el costo mínimo entre todos los árboles generadores que cumplen la restricción de grado.*

*Demostración.* Por el Lema 3.1, el algoritmo considera todas las soluciones candidatas. Por el Lema 3.2, acepta exactamente las soluciones factibles. Finalmente, toma el mínimo costo entre ellas. Por lo tanto, el resultado es óptimo.  $\square$

### 3.1.3. Análisis de Complejidad

El número total de máscaras es  $2^m$ , pero las máscaras consideradas efectivamente son  $\binom{m}{n-1}$  y por cada una de ellas se hace lo siguiente:

- Construcción del subgrafo:  $O(n)$ .
- Verificación de grados:  $O(n)$ .
- DFS para conectividad:  $O(n)$ .

Por tanto, el costo por máscara es  $O(n)$  y la complejidad total es:  $O(2^m + \binom{m}{n-1} \cdot n)$ .

## 3.2. Enumeración Lexicográfica para el DCMST

El algoritmo se basa en la enumeración exhaustiva de subconjuntos de aristas mediante una representación binaria y generación lexicográfica de combinaciones. Su aplicabilidad es sobre todo para instancias de tamaño reducido y es aplicable como algoritmo de referencia para la validación de metaheurísticas. (Ver Apéndice A.2).

### 3.2.1. Descripción del Algoritmo

Sea  $G = (V, E)$  un grafo completo con  $|V| = n$  vértices y  $|E| = m = \frac{n(n-1)}{2}$  aristas. El algoritmo procede como sigue:

1. Se indexan todas las aristas de  $E$ .
2. Se construye una cadena binaria **state** de longitud  $m$  con exactamente  $n - 1$  bits activos.
3. Cada permutación lexicográfica de **state** representa un subconjunto distinto de  $n - 1$  aristas.
4. Para cada subconjunto:
  - Se construye el subgrafo inducido.
  - Se verifica la restricción de grado.
  - Se comprueba conectividad mediante DFS.
  - Si es factible, se evalúa su costo.
5. Se devuelve el mínimo costo encontrado.

La generación de subconjuntos se realiza mediante la función `next_permutation`, lo que garantiza que cada combinación se visita exactamente una vez.

### 3.2.2. Análisis de Correctitud

**Lema 3.5.** *El algoritmo enumera todos los subconjuntos de aristas de tamaño  $n - 1$ .*

*Demostración.* La cadena binaria inicial contiene exactamente  $n - 1$  unos y  $m - (n - 1)$  ceros. El uso de `next_permutation` genera todas las permutaciones distintas de dicha cadena, que corresponden biyectivamente a los subconjuntos de  $E$  con  $n - 1$  elementos.  $\square$

**Lema 3.6.** *Un subconjunto  $E' \subseteq E$  con  $|E'| = n - 1$  es un árbol generador factible si y solo si:*

1. El subgrafo inducido es conexo.
2. Para todo  $v \in V$ ,  $\deg(v) \leq k$ .

*Demostración.* Un grafo conexo con  $n - 1$  aristas es un árbol. La segunda condición garantiza la restricción de grado. Ambas son necesarias y suficientes.  $\square$

**Lema 3.7.** La función *check* devuelve verdadero si y solo si el subgrafo inducido por la máscara es un árbol generador factible.

*Demostración.* Inicialmente, verifica que ningún vértice tenga grado mayor que  $k$ . Luego, ejecuta un DFS desde el vértice 0 y comprueba que todos los vértices son alcanzables, lo que implica conectividad. Por el Lema 3.6, esto es equivalente a ser un DCST factible.  $\square$

**Teorema 3.8.** El algoritmo devuelve el costo mínimo entre todos los árboles generadores que satisfacen la restricción de grado.

*Demostración.* Por el Lema 3.5, todas las soluciones candidatas son evaluadas. Por el Lema 3.6, se aceptan exactamente las factibles. El algoritmo selecciona el mínimo costo entre ellas, lo que implica optimalidad.  $\square$

### 3.2.3. Análisis de Complejidad

El número de combinaciones evaluadas es:  $\binom{m}{n-1}$  y para cada combinación:

- Construcción del subgrafo:  $O(n)$ .
- Verificación de grados:  $O(n)$ .
- DFS de conectividad:  $O(n)$ .

Por lo tanto, la complejidad es  $O(\binom{m}{n-1} \cdot n)$ .

## 3.3. Enumeración con Código de Gray

A continuación se presenta dos algoritmos exactos para el DCMST basados en la enumeración exhaustiva de subconjuntos de aristas utilizando Código de Gray. El enfoque garantiza que subconjuntos consecutivos difieran en exactamente una arista, lo que permite una generación sistemática y eficiente del espacio de soluciones. Se propondrá una solución recursiva con podas.

### 3.3.1. Marco Teórico: Código de Gray

El código de Gray constituye una secuencia de representaciones binarias tal que dos códigos consecutivos difieren exactamente en un solo bit. Formalmente, el código Gray de un entero  $n$  se define como:

$$G(n) = n \oplus (n >> 1)$$

Una técnica para generar todas las combinaciones de tamaño  $K$  consiste en generar la secuencia completa de códigos Gray para los enteros desde 0 hasta  $2^N - 1$  y filtrar únicamente aquellas máscaras que contienen exactamente  $K$  bits activos. Un resultado fundamental es que, en la secuencia filtrada, dos combinaciones adyacentes (consideradas en sentido cíclico) difieren exactamente en dos bits, lo que equivale a eliminar un elemento y añadir otro.

Esta propiedad puede demostrarse por inducción utilizando la construcción recursiva del código Gray:

$$G(N) = 0G(N-1) \cup 1G(N-1)^R$$

### 3.3.2. Descripción del Algoritmo Gray Iterativo

Sea  $G = (V, E)$  un grafo completo con  $|V| = n$  vértices y  $|E| = m = \frac{n(n-1)}{2}$  aristas. El algoritmo procede como sigue:

El algoritmo genera todas las máscaras de bits correspondientes a los códigos Gray de longitud  $m$ . Para cada máscara se verifica si contiene exactamente  $n - 1$  bits activos, lo que corresponde a un subconjunto candidato a árbol generador. (Ver Apéndice A.3).

Para cada subconjunto válido, se construye el grafo inducido y se comprueba:

- Que el grado de cada vértice no exceda  $k$ ;
- Que el grafo sea conexo, mediante un DFS.

El costo mínimo entre todas las configuraciones factibles es devuelto como solución.

### 3.3.3. Análisis de Correctitud del Algoritmo Gray Iterativo

**Lema 3.9.** *El algoritmo enumera exhaustivamente todos los subconjuntos de aristas de tamaño  $n - 1$ .*

*Demostración.* La secuencia de códigos Gray recorre todas las máscaras binarias de longitud  $m$ . El filtrado por número de bits activos igual a  $n - 1$  garantiza que se consideran exactamente todos los subconjuntos de ese tamaño.  $\square$

**Lema 3.10.** *Todo subconjunto aceptado por el algoritmo que pasa las verificaciones corresponde a un árbol generador factible.*

*Demostración.* Un subconjunto con  $n - 1$  aristas que induce un grafo conexo es, por definición, un árbol generador. La verificación adicional del grado asegura el cumplimiento de la restricción  $\deg(v) \leq k$  para todo vértice  $v$ .  $\square$

**Teorema 3.11.** *El algoritmo Gray iterativo devuelve un árbol generador mínimo que satisface la restricción de grado.*

*Demostración.* Por el Lema 3.9 y por el Lema 3.10 el algoritmo evalúa exhaustivamente todas las soluciones factibles y selecciona aquella de menor costo, la solución devuelta es óptima.  $\square$

### 3.3.4. Análisis de Complejidad del Algoritmo Gray Iterativo

El algoritmo recorre  $2^m$  máscaras, y analiza aquellas con exactamente  $n - 1$  aristas. Para cada una, la verificación de conectividad y grados requiere  $O(n)$  tiempo. Por lo tanto, la complejidad temporal es:  $O(2^m + \binom{m}{n-1} \cdot n)$ .

### 3.3.5. Descripción del Algoritmo Gray Recursivo

El algoritmo Gray recursivo implementa directamente la relación:

$$G(N, K) = 0G(N - 1, K) \cup 1G(N - 1, K - 1)^R$$

construyendo únicamente combinaciones válidas de  $k = n - 1$  aristas. (Ver Apéndice A.4).

Durante la recursión, se mantiene de forma incremental:

- el costo acumulado;
- la estructura de adyacencia;
- las restricciones de grado.

Las ramas que violan la restricción de grado se podan anticipadamente.

### 3.3.6. Análisis de Correctitud del Algoritmo Gray Recursivo

**Lema 3.12.** *El algoritmo genera exactamente todas las combinaciones de  $n - 1$  aristas sin repetición.*

*Demostración.* La recurrencia  $G(N, K)$  genera todas las combinaciones binarias de longitud  $N$  con exactamente  $K$  bits activos, sin duplicados, por construcción inductiva.  $\square$

**Lema 3.13.** *Toda solución considerada por el algoritmo cumple la restricción de grado.*

*Demostración.* Antes de continuar la recursión, el algoritmo verifica que el grado de los vértices involucrados no exceda  $K$ . Las ramas inválidas son descartadas.  $\square$

**Teorema 3.14.** *El algoritmo Gray recursivo produce una solución óptima del DCMST.*

*Demostración.* Por el Lema 3.12 y por el Lema 3.13 el algoritmo explora exhaustivamente el espacio de soluciones factibles y conserva el mínimo costo encontrado, garantizando optimalidad.  $\square$

## 4. Implementación y Análisis Experimental

Se implementaron todos los algoritmos descritos en C++. Los códigos se encuentran detallados en el Apéndice A.

## 5. Conclusiones

En este trabajo se detalló el análisis y solución del problema DCMST. Se demostró su complejidad NP-Hard y se presentaron cuatro variantes algorítmicas exactas para su resolución.

## A. Apéndice: Implementaciones en C++

A continuación se presentan las implementaciones completas utilizadas en el estudio.

### A.1. Solución 1: Fuerza Bruta con Máscara de Bits

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 #define sz(x) ((int)((x).size()))
5
6 int main() {
7     cin.tie(0) -> sync_with_stdio(0);
8
9     auto check = [&](int V, int K, const vector<vector<int>> &adj) {
10         for (int i = 0; i < V; i++) {
11             if (sz(adj[i]) > K) return false;
12         }
13
14         vector<bool> vis(V); int cntCC = 0;
15         auto dfs = [&](auto &&self, int u) -> void {
16             cntCC++; vis[u] = true;
17             for (auto &v: adj[u]) {
18                 if (!vis[v]) self(self, v);
19             }
20         };
21
22         dfs(dfs, 0);
23
24         return cntCC == V;
25     };
26
27     int V, K; cin >> V >> K;
28
29     vector<tuple<int, int, long double>> E;
30
31     vector mat(V + 1, vector(V + 1, 0.0L));
32     for (int i = 0; i < V; i++) {
33         for (int j = 0; j < V; j++) {
34             string val; cin >> val;
35             mat[i][j] = stold(val);
36
37             if (i < j) E.push_back({i, j, mat[i][j]});
38         }
39     }
40
41     vector<vector<int>> adj(V);
42
43     long double res = numeric_limits<long double>::max();
44     for (int mask = 0; mask < (1 << sz(E)); mask++) {
45         if (__builtin_popcount(mask) != V - 1) continue;
46
47         long double cost = 0;
48         for (int e = 0; auto &[u, v, w]: E) {
49             if (mask >> e & 1) {
50                 adj[u].push_back(v);
51                 adj[v].push_back(u);
52                 cost += w;
53             }
54             e++;
55         }
56
57         if (check(V, K, adj))
58             res = min(res, cost);
59
60         for (int i = 0; i < V; i++)
61             adj[i].clear();
62     }
```

```

62 }
63
64 cout << setprecision(12) << fixed << res << "\n";
65 }

```

## A.2. Solución 2: Generación Combinatoria Lexicográfica

```

1  #include <bits/stdc++.h>
2  using namespace std;
3
4  #define sz(x) ((int)((x).size()))
5
6  int main() {
7      cin.tie(0) -> sync_with_stdio(0);
8
9      auto check = [&](int V, int K, const vector<vector<int>> &adj) {
10         for (int i = 0; i < V; i++) {
11             if (sz(adj[i]) > K) return false;
12         }
13
14         vector<bool> vis(V); int cntCC = 0;
15         auto dfs = [&](auto &&self, int u) -> void {
16             cntCC++; vis[u] = true;
17             for (auto &v: adj[u]) {
18                 if (!vis[v]) self(self, v);
19             }
20         };
21
22         dfs(dfs, 0);
23
24         return cntCC == V;
25     };
26
27     int V, K; cin >> V >> K;
28
29     vector<tuple<int, int, long double>> E;
30
31     vector mat(V + 1, vector(V + 1, 0.0L));
32     for (int i = 0; i < V; i++) {
33         for (int j = 0; j < V; j++) {
34             string val; cin >> val;
35             mat[i][j] = stold(val);
36
37             if (i < j) E.push_back({i, j, mat[i][j]});
38         }
39     }
40
41     long double res = numeric_limits<long double>::max();
42
43     vector<vector<int>> adj(V);
44
45     string state = string(sz(E) - (V - 1), '0') + string(V - 1, '1');
46     do {
47         long double cost = 0;
48         for (int e = 0; auto &[u, v, w]: E) {
49             if (state[e] == '1') {
50                 adj[u].push_back(v);
51                 adj[v].push_back(u);
52                 cost += w;
53             }
54             e++;
55         }
56
57         if (check(V, K, adj))
58             res = min(res, cost);
59
60         for (int i = 0; i < V; i++)
61             adj[i].clear();

```

```

62     } while (next_permutation(begin(state), end(state)));
63
64     cout << setprecision(12) << fixed << res << "\n";
65 }

```

### A.3. Solución 3: Código de Gray Iterativo

```

1  #include <bits/stdc++.h>
2  using namespace std;
3
4  #define sz(x) ((int)((x).size()))
5
6  int gray_code (int n) {
7      return n ^ (n >> 1);
8  }
9
10 int count_bits (int n) {
11     int res = 0;
12     for (; n; n >>= 1)
13         res += n & 1;
14     return res;
15 }
16
17 int main() {
18     cin.tie(0)->sync_with_stdio(0);
19
20     auto check = [&](int V, int K, const vector<vector<int>>> &adj) {
21         for (int i = 0; i < V; i++) {
22             if (sz(adj[i]) > K) return false;
23         }
24
25         vector<bool> vis(V); int cntCC = 0;
26         auto dfs = [&](auto &&self, int u) -> void {
27             cntCC++; vis[u] = true;
28             for (auto &v: adj[u]) {
29                 if (!vis[v]) self(self, v);
30             }
31         };
32
33         dfs(dfs, 0);
34
35         return cntCC == V;
36     };
37
38     int V, K; cin >> V >> K;
39
40     vector<tuple<int, int, long double>> E;
41
42     vector mat(V + 1, vector(V + 1, 0.0L));
43     for (int i = 0; i < V; i++) {
44         for (int j = 0; j < V; j++) {
45             string val; cin >> val;
46             mat[i][j] = stold(val);
47
48             if (i < j) E.push_back({i, j, mat[i][j]});
49         }
50     }
51
52     long double res = numeric_limits<long double>::max();
53
54     vector<vector<int>> adj(V);
55     for (int i = 0; i < (1 << sz(E)); i++) {
56         int cur = gray_code(i);
57         if (count_bits(cur) == V - 1) {
58             long double cost = 0;
59             for (int j = 0; auto [u, v, w]: E) {
60                 if (cur & (1 << j)) {
61                     adj[u].push_back(v);

```

```

62         adj[v].push_back(u);
63         cost += w;
64     }
65     j++;
66 }
67
68     if (check(V, K, adj))
69         res = min(res, cost);
70
71     for (int j = 0; j < V; j++)
72         adj[j].clear();
73 }
74 }
75
76     cout << setprecision(12) << fixed << res << "\n";
77 }

```

#### A.4. Solución 4: Código de Gray Recursivo con Poda

```

1  #include <bits/stdc++.h>
2  using namespace std;
3
4  #define sz(x) ((int)((x).size()))
5
6  int main() {
7      cin.tie(0)->sync_with_stdio(0);
8
9      auto check = [&](int V, const vector<vector<int>> &adj) {
10         vector<bool> vis(V); int cntCC = 0;
11         auto dfs = [&](auto &&self, int u) -> void {
12             cntCC++; vis[u] = true;
13             for (auto &v: adj[u]) {
14                 if (!vis[v]) self(self, v);
15             }
16         };
17
18         dfs(dfs, 0);
19
20         return cntCC == V;
21     };
22
23     int V, K; cin >> V >> K;
24
25     vector<tuple<int, int, long double>> E;
26
27     vector mat(V + 1, vector(V + 1, 0.0L));
28     for (int i = 0; i < V; i++) {
29         for (int j = 0; j < V; j++) {
30             string val; cin >> val;
31             mat[i][j] = stold(val);
32
33             if (i < j) E.push_back({i, j, mat[i][j]});
34         }
35     }
36
37     long double res = numeric_limits<long double>::max();
38
39     long double cost = 0;
40     vector<vector<int>> adj(V);
41     auto solve = [&](auto &&self, int n, int k, int idx, bool rev) -> void {
42         if (k > n || k < 0) return;
43
44         if (n == 0) {
45             if (check(V, adj))
46                 res = min(res, cost);
47             return;
48         }
49

```

```

50     auto &[u, v, w] = E[idx];
51
52     {
53         if (rev) {
54             adj[u].push_back(v);
55             adj[v].push_back(u);
56             cost += w;
57         }
58
59         if (sz(adj[u]) <= K && sz(adj[v]) <= K)
60             self(self, n - 1, k - rev, idx + 1, false);
61
62         if (rev) {
63             adj[u].pop_back();
64             adj[v].pop_back();
65             cost -= w;
66         }
67     }
68
69     {
70         if (!rev) {
71             adj[u].push_back(v);
72             adj[v].push_back(u);
73             cost += w;
74         }
75
76         if (sz(adj[u]) <= K && sz(adj[v]) <= K)
77             self(self, n - 1, k - !rev, idx + 1, true);
78
79         if (!rev) {
80             adj[u].pop_back();
81             adj[v].pop_back();
82             cost -= w;
83         }
84     }
85 };
86
87 solve(solve, sz(E), V - 1, 0, false);
88
89 cout << setprecision(12) << fixed << res << "\n";
90 }

```

## Referencias

- [1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed., Chapter 34). MIT Press.