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

Ernesto Abreu Peraza and Eduardo Brito Labrada

Diseño y Análisis de Algoritmos - Universidad de La Habana

6 de enero de 2026

Resumen

Este informe detalla el análisis e implementa soluciones al conocido problema NP-Hard Degree Constrained Minimum Spanning Tree para la infraestructura de red de la Universidad de La Habana. Se presenta la formalización matemática, la demostración de su complejidad computacional y una comparativa experimental entre diferentes algoritmos.

Índice

1. Introducción

El diseño eficiente de redes de comunicación constituye un problema fundamental en múltiples dominios de la ingeniería y las ciencias de la computación, particularmente en el contexto de infraestructuras físicas como redes de fibra óptica. En estos escenarios, no solo resulta relevante minimizar el costo total de interconexión, sino también respetar restricciones técnicas impuestas por el equipamiento disponible, tales como el número máximo de conexiones que puede soportar cada nodo de la red.

El problema del Árbol de Expansión Mínimo con Restricciones de Grado (*Degree-Constrained Minimum Spanning Tree*, DCMST) surge de manera natural al modelar este tipo de situaciones. A diferencia del problema clásico del Árbol de Expansión Mínimo (MST), el DCMST impone cotas superiores al grado de cada vértice, lo cual incrementa significativamente su complejidad computacional. De hecho, el DCMST es un problema NP-Hard, lo que implica que, salvo que P = NP, no existen algoritmos que lo resuelvan de manera eficiente para grandes instancias.

Motivado por el problema definido en este documento, se realiza una formalización matemática rigurosa del problema, se analizan y se demuestra su complejidad computacional.

Adicionalmente, se diseñan e implementan varios algoritmos y se presenta un análisis experimental que permite comparar el comportamiento de las distintas estrategias implementadas.

2. 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.

2.1. 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.
- $w : E \to \mathbb{R}^+$ es una función de costo.
- $k : V \to \mathbb{N}$ es la capacidad de puertos por edificio.

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

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

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).

2.2. Variantes del Problema

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

2.2.1. 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,

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

2.2.2. 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.

2.2.3. 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.

3. Análisis de Complejidad Computacional

3.1. Demostración de NP-Hardness del problema DCMST

Para analizar la complejidad computacional del problema, demostramos que dicho problema es NP-Hard mediante una reducción polinomial desde el problema del Viajante de Comercio (*Traveling Salesman Problem*, TSP), uno de los problemas clásicos NP-Hard.

Teorema 3.1. *El problema Degree-Constrained Minimum Spanning Tree es NP-Hard.*

Demostración. Consideremos una instancia arbitraria del problema del Viajante. Dicha instancia está definida por un grafo completo no dirigido

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

junto con una función de costos $w : E \to \mathbb{R}^+$. El objetivo del TSP es encontrar un ciclo simple de costo mínimo que visite exactamente una vez cada vértice. Un ciclo simple que cubre todos los vértices es conocido como ciclo hamiltoniano.

A partir de esta instancia, construimos una instancia del problema DCMST de la siguiente manera:

- Obtenemos el mismo grafo $G = (V, E)$ y la misma función de costos w.
- Sea v un vértice cualquiera de G
- Dividimos v en dos vértices v_1 y v_2, tal que $k(v_1) = 1$ y $k(v_2) = 1$.
- Asignamos restricciones de grado $k(u) = 2$ para todo $u \in V \setminus \{v_1, v_2\}$.

Observemos que, bajo esta restricción, cualquier árbol de expansión válido T debe satisfacer

$$\deg_T(v) \leq 2 \quad \forall v \in V.$$

Un árbol de expansión con grado máximo 2 en todos los vértices es necesariamente un camino simple que cubre todos los vértices del grafo. En todo camino simple, los vértices extremos tienen grado 1 y los vértices intermedios tienen grado 2.

Si resolvemos esta instancia, obtenemos un árbol de expansión T^* de costo mínimo que cumple con las restricciones. Por lo que el árbol va a ser de la forma

$$v_1 - u_1 - u_2 - \ldots - u_{|V|-2} - v_2$$

, donde $u_i \in V \setminus \{v_1, v_2\}$.

Lema 3.2. *El árbol de expansión T^* obtenido en la instancia del DCMST corresponde a un ciclo hamiltoniano de costo mínimo en la instancia original del TSP.*

Demostración. Dado el árbol T^*, notese que si unimos nuevamente v_1 y v_2 como un solo vértice v, obtenemos un ciclo hamiltoniano C^* en el grafo original G con costo igual al del árbol T^*. Supongamos que existe un ciclo hamiltoniano C en G con costo menor que el costo de T^*. Al dividir el vértice v en dos vértices v_1 y v_2, el ciclo C se transforma en un camino simple que cubre todos los vértices de la instancia del DCMST, obteniendose un árbol de expansión T' con costo menor que T^*, lo cual es una contradicción ya que T^* es el de costo mínimo.

Por reducción al absurdo, concluimos que C^* es un ciclo hamiltoniano de costo mínimo en la instancia original del TSP. □

Dado que todas las transformaciones realizadas son polinomiales en tiempo, tener una solución de DCMST en tiempo polinomial nos permite resolver cualquier instancia del TSP en tiempo polinomial. Por tanto DCMST es al menos tan difícil como TSP.

Por lo tanto, el problema DCMST es NP-Hard. □

3.2. NP-Completitud del problema Degree-Constrained Spanning Tree

Consideremos ahora la versión de decisión del problema, conocida como Degree-Constrained Spanning Tree (DCST), en la cual se pregunta si existe un árbol de expansión que satisfaga un conjunto dado de restricciones de grado, sin considerar una función de costo.

Teorema 3.3. *El problema Degree-Constrained Spanning Tree (DCST) es NP-Completo.*

Demostración. En primer lugar, observemos que DCST pertenece a la clase NP. En efecto, dada una solución candidata $T = (V, E')$, es posible verificar en tiempo polinomial que:

- T es conexo y acíclico,

- $|E'| = |V| - 1$,
- $\deg_T(v) \leq k(v)$ para todo $v \in V$.

A continuación, demostramos que DCST es NP-Hard mediante una reducción polinomial desde el problema *Hamiltonian Path*, el cual es NP-Completo.

Sea $G = (V, E)$ una instancia arbitraria del problema Hamiltonian Path. Construimos una instancia del problema DCST sobre el mismo grafo G, imponiendo una restricción de grado uniforme $k(v) = 2$ para todo vértice $v \in V$.

Bajo esta restricción, cualquier árbol de expansión factible debe tener grado a lo sumo 2 en cada vértice. Como consecuencia, dicho árbol solo puede ser un camino simple que cubra todos los vértices del grafo, es decir, un camino hamiltoniano.

Por tanto, existe un árbol de expansión que satisface las restricciones de grado si y solo si existe un camino hamiltoniano en el grafo original. La reducción es claramente polinomial.

Dado que DCST es NP-Hard y pertenece a NP, se concluye que el problema DCST es NP-Completo. □

4. Diseño de Soluciones Algorítmicas

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

4.1. 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.

4.1.1. Descripción del Algoritmo

Sea $G = (V, E)$ un grafo completo, no dirigido y ponderado, con $|V| = n$ vértices y $|E| = m = \dfrac{n(n-1)}{2}$ 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.

4.1.2. 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 4.1. *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 popcount(mask) $= n-1$, se enumeran exactamente todos los subconjuntos de E con $n-1$ aristas. □

Lema 4.2. *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. □

Lema 4.3. *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 4.2, esto es equivalente a ser un DCST factible. □

Teorema 4.4. *El algoritmo devuelve el costo mínimo entre todos los árboles generadores que cumplen la restricción de grado.*

Demostración. Por el Lema 4.1, el algoritmo considera todas las soluciones candidatas. Por el Lema 4.2, acepta exactamente las soluciones factibles. Finalmente, toma el mínimo costo entre ellas. Por lo tanto, el resultado es óptimo. □

4.1.3. 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: $O(2^m + \binom{m}{n-1} \cdot n)$.

4.2. 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).

4.2.1. 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.

4.2.2. Análisis de Correctitud

Lema 4.5. *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. □

Lema 4.6. *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. □

Lema 4.7. *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 4.6, esto es equivalente a ser un DCST factible. □

Teorema 4.8. *El algoritmo devuelve el costo mínimo entre todos los árboles generadores que satisfacen la restricción de grado.*

Demostración. Por el Lema 4.5, todas las soluciones candidatas son evaluadas. Por el Lema 4.6, se aceptan exactamente las factibles. El algoritmo selecciona el mínimo costo entre ellas, lo que implica optimalidad. □

4.2.3. 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 $O(\binom{m}{n-1} \cdot n)$.

4.3. 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.

4.3.1. 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:

$$G(n) = n \oplus (n \check{} 1)$$

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:

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

4.3.2. 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.

4.3.3. Análisis de Correctitud del Algoritmo Gray Iterativo

Lema 4.9. *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. □

Lema 4.10. *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. □

Teorema 4.11. *El algoritmo Gray iterativo devuelve un árbol generador mínimo que satisface la restricción de grado.*

Demostración. Por el Lema 4.9 y por el Lema 4.10 el algoritmo evalúa exhaustivamente todas las soluciones factibles y selecciona aquella de menor costo, la solución devuelta es óptima. □

4.3.4. 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: $O(2^m + \binom{m}{n-1} \cdot n)$.

4.3.5. Descripción del Algoritmo Gray Recursivo

El algoritmo Gray recursivo implementa directamente la relación:

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

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.

4.3.6. Análisis de Correctitud del Algoritmo Gray Recursivo

Lema 4.12. *El algoritmo genera exactamente todas las combinaciones de $n-1$ aristas sin repetición.*

Demostración. Procedemos por inducción fuerte sobre N, la longitud de la cadena binaria (número total de aristas disponibles).

Caso base: Para $N = 0$, $G(0,0) = \{\epsilon\}$, que representa la única combinación de 0 aristas. Para $K > 0$, $G(0,K) = \emptyset$, correctamente generando ninguna combinación.

Hipótesis inductiva: Supongamos que para todo $n' < N$, $G(n',K)$ genera exhaustiva y sin repeticiones todas las combinaciones binarias de longitud n' con exactamente K bits activos.

Paso inductivo: Para $G(N,K)$, consideremos cualquier combinación válida s de longitud N con K bits 1. Examinamos su primer bit:

- Si $s_1 = 0$, entonces el sufijo $s_{2..N}$ es una combinación válida de longitud $N-1$ con K bits 1. Por hipótesis, aparece en $G(N'1,K)$, luego $0s_{2..N}$ aparece en $0G(N-1,K)$.
- Si $s_1 = 1$, entonces $s_{2..N}$ tiene $K-1$ bits 1. Por hipótesis, aparece en $G(N-1,K-1)$, luego $1s_{2..N}$ aparece en $1G(N-1,K-1)^R$.

La unión disjunta $\cup$ de ambos conjuntos cubre todas las posibilidades sin superposición, pues difieren en su primer bit.

Por tanto, $G(N,K)$ genera exactamente todas las combinaciones buscadas, sin duplicados. □

Lema 4.13. *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. □

Teorema 4.14. *El algoritmo Gray recursivo produce una solución óptima del DCMST.*

Demostración. Por el Lema 4.12 y por el Lema 4.13 el algoritmo explora exhaustivamente el espacio de soluciones factibles y conserva el mínimo costo encontrado, garantizando optimalidad. □

5. Análisis Experimental

5.1. Implementación

Con el objetivo de evaluar los algoritmos propuestos para la resolución del problema DCMST, se implementaron todas las soluciones descritas en la sección anterior utilizando el lenguaje de programación `C++`. La elección de este lenguaje se debe a su eficiencia, amplia disponibilidad de estructuras de datos de bajo nivel y la experiencia de los autores en el mismo, lo cual hace mas sencilla la implementación de algoritmos.

5.2. Intancias de Prueba

Las instancias de prueba utilizadas corresponden a grafos completos de distintos tamaños representados en matriz de adyacencia ponderada.

Cada instancia tiene el siguiente formato:

- La primera línea contiene dos enteros: n (número de vértices) y k (restricción de grado para todo vértice).
- Las siguientes n líneas contienen n enteros cada una, representando la matriz de costos entre los vértices.
- Se garantiza que la matriz es simétrica y que los valores en la diagonal son cero.

5.3. Experimentación

Para cada instancia se evaluó el costo mínimo encontrado, así como el tiempo de ejecución de cada algoritmo, permitiendo comparar su desempeño relativo.

El propósito principal de este análisis experimental no es demostrar escalabilidad, sino:

- Validar la correctitud de las implementaciones.
- Comparar el impacto de distintas estrategias sobre el tiempo de ejecución.

5.4. Resultados

Resultados de ejecución de soluciones

Archivo de prueba	Ejecutable	Salida	Tiempo (s)
sample0.txt	bitmask	8.000000000000	0.072
sample0.txt	comb	8.000000000000	0.0716
sample0.txt	gray_iterative	8.000000000000	0.0478
sample0.txt	gray_recursive	8.000000000000	0.0694

Figura 1: Resultados experimentales: comparación de costos y tiempos entre los algoritmos implementados.

6. Diseño de metaheurísticas para resolver el DCMST

6.1. Método Primal Aleatorizado (RPM)

El problema del Árbol de Expansión Mínima con Restricción de Grado (DCMST) busca encontrar un subgrafo T de un grafo ponderado $G = (V, E)$ tal que T sea un árbol que cubra todos los vértices en V,

minimice la suma de los pesos de las aristas y satisfaga la restricción de que ningún vértice tenga un grado mayor a d.

Matemáticamente, buscamos minimizar:

$$Z(x) = \sum_{e \in E} w_e x_e \tag{2}$$

Sujeto a:

$$\sum_{e \in \delta(v)} x_e \leq d, \quad \forall v \in V \tag{3}$$

donde $x_e \in \{0, 1\}$ indica si la arista e está en el árbol, y $\delta(v)$ es el conjunto de aristas incidentes en v. A diferencia del MST clásico, el DCMST es un problema NP-hard para $2 \leq d < |V| - 1$.

La solución implementada utiliza el *Randomised Primal Method* (RPM). Este enfoque híbrido combina la eficiencia constructiva del algoritmo de Prim con la capacidad de exploración de un Algoritmo Genético (AG).

6.1.1. Codificación del Cromosoma

En lugar de codificar aristas directamente (lo cual generaría muchas soluciones infactibles), el RPM utiliza una codificación indirecta. Un cromosoma es una matriz C de dimensiones $n \times (d-1)$, donde n es el número de nodos.

El alelo $C[i][k]$ representa un índice de preferencia. Específicamente, indica que cuando el algoritmo de construcción está considerando el nodo i y este tiene actualmente un grado k, debe seleccionar la $C[i][k]$-ésima mejor arista disponible incidente a i.

6.1.2. Decodificación (Algoritmo de Prim Modificado)

El decodificador construye el árbol iterativamente:

1. Se inicia con un vértice arbitrario en el árbol S.
2. En cada paso, se identifican todas las aristas válidas que conectan un nodo $u \in S$ con un nodo $v \notin S$, siempre que $grado(u) < d$.
3. Para cada nodo $u \in S$, en lugar de elegir obligatoriamente la arista de menor peso (como en Prim clásico), se consulta el cromosoma. Se selecciona la arista en la posición $C[u][grado\_actual(u)]$ de su lista de adyacencia ordenada.
4. De este subconjunto de aristas candidatas seleccionadas por el cromosoma, se elige la de menor peso global para añadirla al árbol.

6.1.3. Operadores Genéticos e Inicialización

- **Inicialización Sesgada:** Para asegurar que la búsqueda comience en regiones prometedoras, los alelos no se generan uniformemente. Se utiliza una distribución geométrica (o exponencial negativa discreta) para favorecer valores bajos $(0, 1)$. Esto significa que el algoritmo tiende a comportarse como el algoritmo de Prim (codicioso) con pequeñas perturbaciones estocásticas.
- **Cruce:** Se implementa un cruce uniforme (*Uniform Crossover*), donde cada gen del hijo se toma de uno de los padres con probabilidad 0.5.
- **Mutación:** Se aplica una mutación puntual con baja probabilidad, reasignando el valor del gen utilizando la misma distribución geométrica sesgada de la inicialización.

6.2. Detalles de la Implementación

La implementación se ha realizado en C++ enfocándose en la eficiencia de las estructuras de datos.

6.2.1. Estructuras de Datos

La clase `Graph` utiliza listas de adyacencia. Un aspecto crítico para la eficiencia del RPM es que las listas de adyacencia se **pre-ordenan por peso** al inicio del programa.

Esto permite que el decodificador acceda a la k-ésima mejor arista en tiempo constante (o lineal respecto al índice k, que suele ser pequeño) sin necesidad de reordenar en cada iteración.

6.2.2. Manejo de Soluciones Inválidas

Dado que las restricciones de grado pueden, en ciertas topologías, llevar a callejones sin salida donde no es posible conectar más nodos sin violar restricciones, el decodificador incluye un mecanismo de seguridad. Si el árbol no logra conectar los n nodos, la solución se marca como `valid = false` y se le asigna un costo infinito, siendo descartada por el proceso de selección del AG.

7. 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

```cpp
#include <bits/stdc++.h>
using namespace std;

#define sz(x) (int)((x).size())

int main() {
    cin.tie(0)->sync_with_stdio(0);

    auto check = [&](int V, int K, const vector<vector<int>> &adj) {
        for (int i = 0; i < V; i++) {
            if (sz(adj[i]) > K) return false;
        }

        vector<bool> vis(V); int cntCC = 0;
        auto dfs = [&](auto &&self, int u) -> void {
            cntCC++; vis[u] = true;
            for (auto &v: adj[u]) {
                if (!vis[v]) self(self, v);
            }
        };

        dfs(dfs, 0);

        return cntCC == V;
    };

    int V, K; cin >> V >> K;

    vector<tuple<int, int, long double>> E;

    vector mat(V + 1, vector(V + 1, 0.0L));
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            string val; cin >> val;
            mat[i][j] = stold(val);

            if (i < j) E.push_back({i, j, mat[i][j]});
        }
    }

    vector<vector<int>> adj(V);

    long double res = numeric_limits<long double>::max();
    for (int mask = 0; mask < (1 << sz(E)); mask++) {
        if (__builtin_popcount(mask) != V - 1) continue;

        long double cost = 0;
        for (int e = 0; auto &[u, v, w]: E) {
            if (mask >> e & 1) {
                adj[u].push_back(v);
                adj[v].push_back(u);
                cost += w;
            }
            e++;
        }

        if (check(V, K, adj))
            res = min(res, cost);

        for (int i = 0; i < V; i++)
            adj[i].clear();
```

```
    }

    cout << setprecision(12) << fixed << res << "\n";
}
```

A.2. Solución 2: Generación Combinatoria Lexicográfica

```
#include <bits/stdc++.h>
using namespace std;

#define sz(x) (int)((x).size())

int main() {
    cin.tie(0)->sync_with_stdio(0);

    auto check = [&](int V, int K, const vector<vector<int>> &adj) {
        for (int i = 0; i < V; i++) {
            if (sz(adj[i]) > K) return false;
        }

        vector<bool> vis(V); int cntCC = 0;
        auto dfs = [&](auto &&self, int u) -> void {
            cntCC++; vis[u] = true;
            for (auto &v: adj[u]) {
                if (!vis[v]) self(self, v);
            }
        };

        dfs(dfs, 0);

        return cntCC == V;
    };

    int V, K; cin >> V >> K;

    vector<tuple<int, int, long double>> E;

    vector mat(V + 1, vector(V + 1, 0.0L));
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            string val; cin >> val;
            mat[i][j] = stold(val);

            if (i < j) E.push_back({i, j, mat[i][j]});
        }
    }

    long double res = numeric_limits<long double>::max();

    vector<vector<int>> adj(V);

    string state = string(sz(E) - (V - 1), '0') + string(V - 1, '1');
    do {
        long double cost = 0;
        for (int e = 0; auto &[u, v, w]: E) {
            if (state[e] == '1') {
                adj[u].push_back(v);
                adj[v].push_back(u);
                cost += w;
            }
            e++;
        }

        if (check(V, K, adj))
            res = min(res, cost);

        for (int i = 0; i < V; i++)
            adj[i].clear();
```

```
    } while (next_permutation(begin(state), end(state)));

    cout << setprecision(12) << fixed << res << "\n";
}
```

A.3. Solución 3: Código de Gray Iterativo

```
#include <bits/stdc++.h>
using namespace std;

#define sz(x) (int)((x).size())

int gray_code (int n) {
    return n ^ (n >> 1);
}

int count_bits (int n) {
    int res = 0;
    for (; n; n >>= 1)
        res += n & 1;
    return res;
}

int main() {
    cin.tie(0)->sync_with_stdio(0);

    auto check = [&](int V, int K, const vector<vector<int>> &adj) {
        for (int i = 0; i < V; i++) {
            if (sz(adj[i]) > K) return false;
        }

        vector<bool> vis(V); int cntCC = 0;
        auto dfs = [&](auto &&self, int u) -> void {
            cntCC++; vis[u] = true;
            for (auto &v: adj[u]) {
                if (!vis[v]) self(self, v);
            }
        };

        dfs(dfs, 0);

        return cntCC == V;
    };

    int V, K; cin >> V >> K;

    vector<tuple<int, int, long double>> E;

    vector mat(V + 1, vector(V + 1, 0.0L));
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            string val; cin >> val;
            mat[i][j] = stold(val);

            if (i < j) E.push_back({i, j, mat[i][j]});
        }
    }

    long double res = numeric_limits<long double>::max();

    vector<vector<int>> adj(V);
    for (int i = 0; i < (1 << sz(E)); i++) {
        int cur = gray_code(i);
        if (count_bits(cur) == V - 1) {
            long double cost = 0;
            for (int j = 0; auto [u, v, w]: E) {
                if (cur & (1 << j)) {
                    adj[u].push_back(v);
```

```
                    adj[v].push_back(u);
                    cost += w;
                }
                j++;
            }

            if (check(V, K, adj))
                res = min(res, cost);

            for (int j = 0; j < V; j++)
                adj[j].clear();
        }
    }

    cout << setprecision(12) << fixed << res << "\n";
}
```

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

```
#include <bits/stdc++.h>
using namespace std;

#define sz(x) (int)((x).size())

int main() {
    cin.tie(0)->sync_with_stdio(0);

    auto check = [&](int V, const vector<vector<int>> &adj) {
        vector<bool> vis(V); int cntCC = 0;
        auto dfs = [&](auto &&self, int u) -> void {
            cntCC++; vis[u] = true;
            for (auto &v: adj[u]) {
                if (!vis[v]) self(self, v);
            }
        };

        dfs(dfs, 0);

        return cntCC == V;
    };

    int V, K; cin >> V >> K;

    vector<tuple<int, int, long double>> E;

    vector mat(V + 1, vector(V + 1, 0.0L));
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            string val; cin >> val;
            mat[i][j] = stold(val);

            if (i < j) E.push_back({i, j, mat[i][j]});
        }
    }

    long double res = numeric_limits<long double>::max();

    long double cost = 0;
    vector<vector<int>> adj(V);
    auto solve = [&](auto &&self, int n, int k, int idx, bool rev) -> void {
        if (k > n || k < 0) return;

        if (n == 0) {
            if (check(V, adj))
                res = min(res, cost);
            return;
        }

```

```
        auto &[u, v, w] = E[idx];

        {
            if (rev) {
                adj[u].push_back(v);
                adj[v].push_back(u);
                cost += w;
            }

            if (sz(adj[u]) <= K && sz(adj[v]) <= K)
                self(self, n - 1, k - rev, idx + 1, false);

            if (rev) {
                adj[u].pop_back();
                adj[v].pop_back();
                cost -= w;
            }
        }

        {
            if (!rev) {
                adj[u].push_back(v);
                adj[v].push_back(u);
                cost += w;
            }

            if (sz(adj[u]) <= K && sz(adj[v]) <= K)
                self(self, n - 1, k - !rev, idx + 1, true);

            if (!rev) {
                adj[u].pop_back();
                adj[v].pop_back();
                cost -= w;
            }
        }
    };

    solve(solve, sz(E), V - 1, 0, false);

    cout << setprecision(12) << fixed << res << "\n";
}
```

Referencias

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