Universidad de La Habana

Facultad de Matemática y Computación

Asignatura: Diseño y Análisis de Algoritmos

Conectando la UH

Integrantes

Jabel Resendiz Aguirre
Noel Pérez Calvo
Arianne Camila Palancar Ochando

Carrera: Ciencia de la Computación

8 de enero de 2026

Índice

Problema

La Universidad de La Habana, en su constante búsqueda de la excelencia académica y la innovación, se ha embarcado en un proyecto crucial para modernizar y expandir su infraestructura de red. Nuestro objetivo es dotar a todas nuestras facultades, centros de investigación y edificios administrativos con conectividad de fibra óptica de alta velocidad. Para este fin, contamos con el valioso apoyo técnico y logístico de ETECSA (Empresa de Telecomunicaciones de Cuba S.A.).

Nos enfrentamos a un desafío de diseño de red que requiere una solución óptima. Necesitamos interconectar todos los edificios principales de la UH con fibra óptica, creando una red robusta y eficiente. Cada posible conexión de fibra entre dos edificios tiene un costo de instalación asociado, que incluye desde los permisos internos y la mano de obra especializada de ETECSA hasta los materiales y las obras civiles necesarias.

Sin embargo, ETECSA ha establecido una restricción técnica fundamental que debemos respetar:

En cada edificio, la conexión de la fibra óptica se gestionará a través de un equipo de red central (un router o switch principal) que ellos nos proporcionan. Estos equipos tienen una capacidad limitada de puertos. Esto significa que un equipo en un edificio específico solo puede manejar un número máximo de conexiones de fibra óptica directas a otros edificios. Exceder este límite implicaría la necesidad de instalar equipos adicionales mucho más caros y complejos, o la implementación de soluciones de red alternativas que ETECSA no puede garantizar o que dispararían drásticamente el presupuesto del proyecto.

Nuestro objetivo principal es diseñar la red de fibra óptica que conecte todos nuestros edificios principales de la manera más económica posible. Esto implica seleccionar las rutas de fibra de tal forma que:

1. Todos los edificios estén interconectados a la red principal de la universidad, sin crear bucles innecesarios (buscamos una estructura de red eficiente).

2. Ningún equipo de red en ningún edificio exceda su capacidad máxima de conexiones directas (es decir, el número de cables de fibra que llegan o salen de un edificio no puede superar el límite de puertos del equipo de ETECSA).

3. El costo total de instalación de toda la red sea el mínimo posible.

Una planificación subóptima podría resultar en un sobrecosto significativo para la universidad, la necesidad de adquirir hardware de red adicional no previsto, o en una red ineficiente que no cumpla con las especificaciones técnicas y presupuestarias acordadas con ETECSA.

Formalización del Problema

El objetivo es diseñar una red de fibra óptica que conecte todos los edificios principales de la Universidad de La Habana mediante enlaces posibles provistos por ETECSA. Cada enlace tiene un costo de instalación y cada edificio posee un límite máximo de puertos disponibles. Se requiere encontrar una configuración de conexiones que:

- conecte todos los edificios,
- respete los límites de puertos por edificio,
- minimice el costo total de instalación.

Notación

En el lenguaje matemático, podemos definir la estructura del problema como un grafo simple y no dirigido $G = (V, E)$, donde:

- Un conjunto de edificios:
$$V = \{v_1, v_2, \ldots, v_n\}.$$
- Un conjunto de posibles enlaces de fibra óptica: $E = (e_1, e_2, ..., e_m)$
$$E \subseteq \{\{u, v\} \mid u, v \in V,\ u \neq v\}.$$
- Un costo de instalación para cada enlace:
$$c : E \to \mathbb{R}_{>0}.$$
- Un límite de puertos (grado máximo permitido) en cada edificio:
$$d : V \to \mathbb{Z}_{\geq 1}.$$

Restricciones del Problema

Sea un subgrafo de G como $T = (V^T, E^T)$ debe satisfacer que:

1. **Conectividad:**
$$T \text{ es conexo } \to V^T = V$$
2. **Estructura de árbol:**
$$T \text{ no contiene ciclos} \to |E^T| = |V| - 1.$$
3. **Límite de puertos por edificio:**
$$\deg_T(v) \leq d(v), \qquad \forall v \in V.$$

donde se define $deg_T(v)$ como el grado del vértice v (también llamado cardinalidad de su vecindad) en el grafo T. Se desea conseguir el subgrafo T tal que minimiza el costo de la suma de los pesos de sus aristas, es decir:

1. **Función objetivo:**
$$\min_{T \subseteq E} \sum_{e \in T} c(e)$$

sujeto a las restricciones anteriores.

Análisis de Complejidad Computacional

En esta fase se determina la dificultad computacional del problema formalizado en la Fase 1. Demostraremos que la versión de decisión del problema es NP-completa, y por lo tanto, la versión de optimización es NP-dura. Este problema es conocido en la literatura como $DC-MST$, el cual demostremos que la versión de decisión es NP-completa por lo que a versión de optimización lo es también.

Versión de Decisión del Problema

Dado un grafo $G = (V, E)$, costos $c(e)$, límites de grado $d(v)$ y un valor K, definimos la siguiente pregunta:

$$\text{¿Existe un subconjunto } T \subseteq E \text{ tal que } \begin{cases} G_T = (V, T) \text{ es un árbol,} \\ \deg_{G_T}(v) \leq d(v) \ \ \forall v \in V, \\ \sum_{e \in T} c(e) \leq K? \end{cases}$$

Pertenencia a NP

Dado un conjunto de aristas T, quiere verificarse que se puede comprobar en tiempo polinomial, lo cual se cumplirá si se cumplen las siguientes condiciones:

- G_T es conexo (mediante un recorrido BFS/DFS),
- $|T| = |V| - 1$, es decir el subgrafo G_T forma un árbol,
- $\deg_{G_T}(v) \leq d(v)$ para todo $v \in V$,
- $\sum_{e \in T} c(e) \leq K$.

Por lo tanto, el problema pertenece a NP.

Demostración de NP-completitud

Para demostrar que dicho problema es NP-completo, se presenta una reducción en tiempo polinomial desde el problema Hamiltonian Path en grafos no dirigidos, el cual es NP-completo.

Problema de Partida: Hamiltonian Path

Dado un grafo no dirigido $G = (V, E)$, el problema Hamiltonian Path pregunta si existe un camino que visite todos los vértices exactamente una vez.

Construcción de la Reducción

A partir de una instancia de Hamiltonian Path, construimos una instancia del problema de la siguiente manera:

- Se toma el mismo grafo $G = (V, E)$.
- Para cada arista $e \in E$, se asigna un costo $c(e) = 1$.

- Se fija un límite de grado uniforme:

$$d(v) = 2 \quad \forall v \in V.$$

- Se establece el umbral de costo:

$$K = |V| - 1.$$

La construcción es claramente polinomial.

Correctitud de la Reducción

($\Rightarrow$) Si G posee un camino hamiltoniano, dicho camino contiene $|V|-1$ aristas, es conexo, acíclico y cada vértice tiene grado a lo sumo 2. Por lo tanto, constituye un conjunto T válido para el problema con costo total $|V| - 1 \leq K$.

($\Leftarrow$) Si existe un conjunto T que satisface las restricciones del problema, entonces T es un árbol con $|V| - 1$ aristas y $\deg_{G_T}(v) \leq 2$ para todo v. La única estructura de árbol donde todos los grados son a lo sumo 2 es un camino. Por lo tanto, T es un camino hamiltoniano del grafo original.

Esto permite entonces reducir que:

DC-MST-Decision es NP-completo.

Dado que la versión de decisión es NP-completa, la versión de optimización (*Degree-Constrained Minimum Spanning Tree*) es NP-dura:

DC-MST-OPT es NP-dura.

Esto implica que no se conoce un algoritmo polinomial que resuelva el problema de forma óptima para instancias generales, salvo que P = NP.

Diseño de Soluciones Algorítmicas

El problema de construir un árbol generador con restricciones de grado (DC-MST) es NP-difícil, lo que implica que no existe un algoritmo polinomial conocido capaz de garantizar la solución óptima para instancias generales. Por ello, en esta sección se exploran distintas estrategias algorítmicas para abordar el problema, combinando métodos exactos y heurísticos.

Presentaremos los algoritmos propuestos de manera estructurada, incluyendo:

- Una descripción de la idea principal de cada algoritmo.
- El pseudocódigo correspondiente para su implementación.
- El análisis de complejidad computacional.

Esto permitirá evaluar las ventajas y limitaciones de cada enfoque, y proporcionará una base sólida para la implementación experimental y la comparación de resultados en la Fase 4.

Algoritmo de Fuerza Bruta: Exploración Exhaustiva del Espacio de Árboles Generadores

Dado que el problema de encontrar un árbol generador de costo mínimo con restricciones de grado es NP-difícil, una primera aproximación consiste en analizar exhaustivamente todas las posibles soluciones y seleccionar aquella que cumple las restricciones y minimiza el costo total.

Un candidato a solución válida debe ser un **árbol generador**, es decir, un subgrafo conexo y acíclico que contiene todos los vértices del grafo y exactamente $|V|-1$ aristas. El algoritmo de fuerza bruta consiste en enumerar todos los subconjuntos de aristas del grafo y verificar cuáles de ellos cumplen las condiciones necesarias para ser un árbol generador factible.

Idea del Algoritmo

Sea $G = (V, E)$ un grafo no dirigido con $|V| = n$ vértices y $|E| = m$ aristas. El procedimiento seguido es el siguiente:

1. Enumerar todos los subconjuntos $S \subseteq E$.
2. Descartar aquellos subconjuntos que no contengan exactamente $n - 1$ aristas.
3. Para cada subconjunto candidato:
   - Verificar que el subgrafo inducido sea conexo y acíclico, utilizando una estructura de datos *Disjoint Set Union* (DSU).
   - Comprobar que el grado de cada vértice no exceda su límite máximo permitido.
   - Calcular el costo total de las aristas seleccionadas.
4. Conservar el subconjunto válido cuyo costo total sea mínimo.

Este algoritmo garantiza encontrar la solución óptima, aunque a costa de un tiempo de ejecución exponencial.

Pseudocódigo

```
BruteForce-DCMST(G):
    bestCost <- +infinity
    bestTree <- empty

    for each subset S of E:
        if |S| != |V| - 1:
            continue

        initialize DSU with |V| elements
        deg[v] <- 0 for all v in V
        cost <- 0
        valid <- true

        for each edge (u,v) in S:
            if deg[u] + 1 > d(u) or deg[v] + 1 > d(v):
                valid <- false
                break

            deg[u] <- deg[u] + 1
            deg[v] <- deg[v] + 1

            if DSU.find(u) == DSU.find(v):
                valid <- false
                break

            DSU.union(u,v)
            cost <- cost + c(u,v)

        if valid and cost < bestCost:
            bestCost <- cost
            bestTree <- S

    return bestCost, bestTree
```

Análisis de Complejidad

Sea $n = |V|$ el número de vértices y $m = |E|$ el número de aristas del grafo.

- El algoritmo enumera todos los subconjuntos posibles de aristas, lo cual implica 2^m iteraciones.
- Para cada subconjunto candidato con $n - 1$ aristas, se realizan:
  - Operaciones de unión y búsqueda en la estructura DSU, con costo $O(n\alpha(n)) \approx O(n)$.
  - Verificación de los límites de grado y cálculo del costo total, ambos en $O(n)$.

Por tanto, la complejidad temporal total del algoritmo es:

$$\boxed{O(2^m \cdot n)}$$

En el peor caso, cuando el grafo es denso y $m = O(n^2)$, la complejidad crece como $O(2^{n^2})$, lo cual hace que este enfoque sea impracticable para instancias de tamaño moderado o grande.

Conclusión

El algoritmo de fuerza bruta permite obtener la solución óptima del problema de diseño de la red de fibra óptica bajo restricciones de grado. Sin embargo, su costo computacional exponencial limita su uso a instancias muy pequeñas. Esto justifica la necesidad de desarrollar algoritmos heurísticos o aproximados, los cuales se abordan en las siguientes secciones.

Algoritmo Heurístico: Reducción del Árbol de Oportunidades

Con el objetivo de disminuir el tamaño del espacio de búsqueda y la dificultad computacional del problema, se introducen una serie de propiedades estructurales del **árbol generador de costo mínimo con restricciones de grado** que permiten reducir el grafo original sin perder optimalidad.

Sea T^* un árbol generador de costo mínimo con restricciones de grado del grafo $G = (V, E)$.

Propiedades Estructurales

Lema 1. Sea $v \in V$ un vértice hoja, es decir, un vértice con grado uno en el grafo original G. Entonces, la única arista incidente a v debe pertenecer a T^*.

Demostración. Dado que T^* es un grafo conexo que contiene todos los vértices de G, el vértice v debe estar conectado al resto del árbol mediante su única arista incidente. Excluir dicha arista implicaría que v quedaría aislado, contradiciendo la conectividad de T^*. Por tanto, toda arista incidente a un vértice colgante debe incluirse necesariamente en T^*. □

Lema 2. Sea $V_1 = \{v_i \in V \mid d(v_i) = 1\}$ el conjunto de vértices cuyo grado máximo permitido es uno, y sea

$$E_1 = \{(v_i, v_j) \in E \mid v_i, v_j \in V_1\}.$$

Si $|V| > 2$, entonces ninguna arista de E_1 puede pertenecer a T^*.

Demostración. Supóngase que existe una arista $(v_i, v_j) \in E_1$ incluida en T^*. Como $d(v_i) = d(v_j) = 1$, ninguno de estos vértices puede conectarse con ningún otro vértice adicional en T^*. Esto implica que el subgrafo resultante no puede ser conexo cuando $|V| > 2$, lo cual contradice la definición de árbol generador. Por tanto, todas las aristas de E_1 deben ser excluidas de T^*. □

Lema 3. Sea v_k un vértice de grado dos en G, con vecinos v_i y v_j. Si no existe ningún camino entre v_i y v_j que no pase por v_k, entonces las aristas (v_k, v_i) y (v_k, v_j) deben pertenecer a T^*.

Demostración. Supóngase que alguna de las aristas (v_k, v_i) o (v_k, v_j) no pertenece a T^*. Dado que no existe un camino alternativo entre v_i y v_j que evite el vértice v_k, se deduce que T^* no puede ser conexo, lo cual contradice su definición. Por tanto, ambas aristas deben incluirse necesariamente en T^*. □

—

Algoritmo de Reducción

A partir de las propiedades anteriores, se define un procedimiento de reducción que simplifica el grafo original antes de aplicar un algoritmo constructivo.

```
Reduction_DCMST(G = (V,E), d):
    T* <- empty
    1. Eliminar todas las aristas que satisfacen el Lema 2.
    2. Identificar todos los v\'ertices colgantes (grado 1),
       agregar sus aristas incidentes a T* y eliminarlos del grafo.
    3. Identificar v\'ertices que satisfacen el Lema 3,
       agregar las aristas correspondientes a T* y eliminarlas del grafo
   .
    return G reducido, T*
```

El costo computacional de este algoritmo de reducción es:

- Paso 1: $O(n^2)$,
- Paso 2: $O(n)$,
- Paso 3: $O(n^3)$.

Por tanto, la complejidad temporal total del algoritmo de reducción es:

$$O(n^3)$$

—

Construcción del Árbol con Restricciones de Grado

Una vez reducido el grafo, se construye un árbol generador utilizando un algoritmo greedy basado en el algoritmo clásico de Kruskal, adaptado para respetar las restricciones de grado.

El algoritmo itera seleccionando aristas de menor costo que conecten componentes distintas y que no violen los límites de grado de los vértices involucrados. Este procedimiento continúa hasta obtener $|V| - 1$ aristas.

```
MAIN_DCMST(G = (V, E), w, d):
    V* <- V
    E* <- empty
    T* <- (V*, E*)

    1. Ejecutar REDUCTION_DCMST(G, d)

    2. E1 <- E

    3. while |E*| < |V| - 1 do
            seleccionar la arista de menor costo
            emin = (vk, vh) en E1

            eliminar emin de E1

            if vk y vh estan en componentes distintas de T*
               and deg_T*(vk) < d(vk)
               and deg_T*(vh) < d(vh) then

```

```
                E* <- E* U {emin}
                unir las componentes de vk y vh en T*
            end if
        end while

    4. Aplicar tecnicas de intercambio de aristas
       (1-opt y 2-opt) para mejorar la solucion

    return G* = (V*, E*)
```

—

Técnicas de Intercambio de Aristas

Para mejorar la solución obtenida, se emplean técnicas de *edge exchange*.

Intercambio 1-opt. Consiste en eliminar una arista del árbol y reemplazarla por una arista externa, siempre que el resultado sea un árbol. Esta operación puede modificar los grados de los vértices y, por tanto, debe verificarse nuevamente la restricción de grado.

Intercambio 2-opt. Consiste en reemplazar dos aristas del árbol por dos aristas externas, de forma que el grado de cada vértice se conserve. Esta operación garantiza que la restricción de grado siga siendo satisfecha y permite mejorar el costo total del árbol.

—

Complejidad del Algoritmo Heurístico

La complejidad temporal del algoritmo completo es la siguiente:

- Reducción del grafo: $O(n^3)$.
- Construcción inicial del árbol (Kruskal modificado): $O(n)$.
- Intercambio 1-opt: $O(n)$.
- Intercambio 2-opt: $O(n^2)$.

Por tanto, la complejidad temporal total del algoritmo heurístico es:

$$\boxed{O(n^3)}$$

Este enfoque permite obtener soluciones de alta calidad en tiempo polinomial, haciendo viable el tratamiento de instancias de tamaño considerable.

Heurística Lagrangiana para DC-MST

El enfoque de relajación Lagrangiana constituye una técnica poderosa para obtener cotas inferiores de calidad y guiar la construcción de soluciones factibles para problemas de optimización combinatoria NP-duros. En el contexto del DC-MST, se aplica esta técnica relajando las restricciones de grado mediante multiplicadores Lagrangianos, lo que transforma el problema original en un problema de árbol generador de costo mínimo (MST) clásico, resoluble en tiempo polinomial.

Idea del Algoritmo

La heurística Lagrangiana propuesta se compone de tres componentes principales que trabajan de forma integrada:

1. Relajación Lagrangiana del DC-MST. Se relajan las restricciones de grado mediante multiplicadores Lagrangianos $\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_n) \in \mathbb{R}_+^{|V|}$, donde λ_i penaliza las violaciones de la restricción de grado en el vértice $i \in V$.

El subproblema de relajación Lagrangiana (LRS) resultante es:

$$z(\lambda) = \text{mín} \left\{ \sum_{e=[i,j]\in E} (c_e + \lambda_i + \lambda_j)x_e - \sum_{i\in V} \lambda_i d_i : x \in R_0 \right\}$$

donde R_0 representa el conjunto de vectores de incidencia de árboles generadores de G. Este subproblema equivale a encontrar un MST con costos modificados $\{c_e + \lambda_i + \lambda_j : e = [i,j] \in E\}$, resoluble eficientemente mediante algoritmos de Kruskal o Prim.

Para obtener la mejor cota inferior posible, se resuelve el problema dual Lagrangiano:

$$z(\lambda^*) = \max_{\lambda \geq 0} \{z(\lambda)\}$$

El vector óptimo λ^* se obtiene mediante el **método del subgradiente**, que actualiza iterativamente los multiplicadores según:

$$\lambda_i^{k+1} = \max \left\{0, \lambda_i^k + t^k s_i^k\right\}, \quad \forall i \in V$$

donde $s_i^k = \sum_{e\in\delta(i)} x_e^k - d_i$ es el subgradiente (violación de la restricción de grado en el vértice i), y el tamaño de paso t^k se calcula como:

$$t^k = \frac{(1+\alpha)z_{UB} - z(\lambda^k)}{\|s^k\|^2}$$

siendo z_{UB} una cota superior conocida y α un parámetro de control (típicamente $\alpha = 0{,}03$).

2. Construcción Heurística con Prevención de Infactibilidad. Se introduce el algoritmo `KRUSKALX`, una variante del algoritmo clásico de Kruskal que incorpora un mecanismo de *look-ahead* para prevenir la generación de soluciones infactibles.

La idea fundamental es el concepto de **saturación**. Sea $T = (V_T, E_T)$ un subárbol. Se definen:

- **Grado del árbol:** $\delta(T) = \sum_{i\in V_T} \sum_{e\in\delta(i)\cap E_T} 1$
- **Capacidad del árbol:** $d(T) = \sum_{i\in V_T} d_i$

Un árbol T se considera **saturado** si $\delta(T) = d(T)$, es decir, si ha alcanzado su capacidad máxima de aristas respetando las restricciones de grado.

Proposición (Condición de No Saturación): Sean T_1 y T_2 dos subárboles disjuntos no saturados. Si existe una arista $e \in E$ tal que $T_3 = (V_1 \cup V_2, E_1 \cup E_2 \cup \{e\})$ es también no saturado, entonces:

$$\delta(T_3) = \delta(T_1) + \delta(T_2) + 2 < d(T_3)$$

Esta propiedad garantiza que en grafos completos siempre es posible construir una solución factible, incluso cuando algunos vértices tienen $d_i \in \{1, 2\}$.

El algoritmo `KRUSKALX` ordena las aristas por costo creciente y las considera secuencialmente. Una arista $e = [i, j]$ se añade al árbol en construcción si y solo si:

1. Conecta dos componentes distintas (no crea ciclos)
2. No viola las restricciones de grado: $\deg(i) < d_i$ y $\deg(j) < d_j$
3. La unión resultante no está saturada (excepto en la última arista)

Para guiar la construcción hacia soluciones de buena calidad, se utilizan **costos complementarios**. Si x^k es la solución del LRS en la iteración k, los costos se modifican como:

$$\bar{c}_e = (1 - x_e^k)c_e, \quad \forall e \in E$$

Esto hace que las aristas seleccionadas en la solución Lagrangiana sean más atractivas para el algoritmo constructivo.

3. Problema Restringido. Para mejorar la eficiencia computacional, se define un **problema restringido** sobre un subconjunto de aristas $E' \subseteq E$. Dado el ordenamiento de aristas $\{e_1, e_2, \ldots, e_m\}$ usado por `KRUSKALX` y siendo k^* el índice de la última arista insertada, se define:

$$E' = \{e_1, e_2, \ldots, e_{m^*}\}, \quad \text{donde } m^* = \min\{2k^*, m\}$$

El grafo restringido $G' = (V, E')$ contiene típicamente una fracción pequeña de las aristas originales pero preserva soluciones de alta calidad. Trabajar sobre G' reduce significativamente el tiempo de cómputo permitiendo abordar instancias con miles de vértices.

4. Procedimiento de Mejora Local. Una vez construida una solución factible T, se aplica un procedimiento de mejora por intercambio de aristas. Para cada arista $e \in T$:

1. Se elimina e de T, generando dos subárboles T_1 y T_2
2. Se busca la arista de menor costo $\bar{e} \in E \setminus T$ que reconecte T_1 y T_2 sin violar restricciones de grado
3. Si $c_{\bar{e}} < c_e$, se realiza el intercambio: $T \leftarrow (T \setminus \{e\}) \cup \{\bar{e}\}$

Este procedimiento tiene complejidad $O(|V| \cdot |E|)$ y se ejecuta sobre el árbol obtenido en cada iteración del método del subgradiente.

Pseudocódigo

```
KRUSKALX(G, c, d):
    Ordenar aristas E = {e1, e2, ..., em} por costo creciente
    Inicializar deg[i] <- 0 para todo i en V
    Inicializar T <- (V, empty), k <- 1

    while |E(T)| < |V| - 1 and k <= m:
```

```
        ek <- [i, j]

        if deg[i] < d[i] and deg[j] < d[j]:
            if ek no forma ciclo en T:
                if |E(T)| = |V| - 2:
                    /* Ultima arista, agregar directamente */
                    E(T) <- E(T) U {ek}
                    deg[i] <- deg[i] + 1
                    deg[j] <- deg[j] + 1
                else:
                    /* Verificar condicion de no saturacion */
                    T1 <- componente de i en T
                    T2 <- componente de j en T
                    if delta(T1) + delta(T2) + 2 < d(T1) + d(T2):
                        E(T) <- E(T) U {ek}
                        deg[i] <- deg[i] + 1
                        deg[j] <- deg[j] + 1
                    end if
                end if
            end if
        end if
        k <- k + 1
    end while

    return T, k*
```

```
IMPROVEMENT_PROCEDURE(T, G, c, d):
    E0 <- E(T)
    improved <- true

    while improved:
        improved <- false
        for each e = [i,j] in E0:
            Eliminar e de T, obteniendo T1 y T2

            /* Buscar mejor arista de reconexion */
            e_best <- null
            cost_best <- +infinito

            for each e' = [u,v] in E \ E(T):
                if u en V(T1) and v en V(T2) (o viceversa):
                    if deg_T(u) < d[u] or u in {i,j}:
                        if deg_T(v) < d[v] or v in {i,j}:
                            if c[e'] < cost_best:
                                e_best <- e'
                                cost_best <- c[e']
                            end if
                        end if
                    end if
                end if
            end for

            if e_best != null and cost_best < c[e]:
                E(T) <- (E(T) \ {e}) U {e_best}
                improved <- true
            else:
                Restaurar e en T
            end if
```

```
        end for
    end while

    return T
```

```
LAGRANGIAN_HEURISTIC(G, c, d):
    /* Fase 1: Inicializacion */
    T_init <- KRUSKALX(G, c, d)
    E' <- primeras 2*k* aristas del ordenamiento
    G' <- (V, E')

    lambda[i] <- 0 para todo i en V
    z_LB <- -infinito
    z_UB <- costo(T_init)
    T_best <- T_init

    alpha <- 2.0
    iter <- 0
    max_iter <- 1000
    iter_sin_mejora <- 0

    /* Fase 2: Metodo del subgradiente */
    while iter < max_iter and z_UB - z_LB > 0.99:
        /* Resolver LRS con costos modificados */
        c_mod[e=[i,j]] <- c[e] + lambda[i] + lambda[j]
        T_LRS <- MST(G', c_mod)
        z_lambda <- costo(T_LRS) - sum(lambda[i] * d[i])

        if z_lambda > z_LB:
            z_LB <- z_lambda
            iter_sin_mejora <- 0
        else:
            iter_sin_mejora <- iter_sin_mejora + 1
        end if

        /* Si LRS es factible para DC-MST */
        if deg_T_LRS[i] <= d[i] para todo i en V:
            T_LRS <- IMPROVEMENT_PROCEDURE(T_LRS, G', c, d)
            if costo(T_LRS) < z_UB:
                z_UB <- costo(T_LRS)
                T_best <- T_LRS
            end if
        end if

        /* Construccion heuristica con costos complementarios */
        c_comp[e] <- (1 - x_LRS[e]) * c[e]
        T_heur <- KRUSKALX(G', c_comp, d)

        if T_heur es factible:
            T_heur <- IMPROVEMENT_PROCEDURE(T_heur, G', c, d)
            if costo(T_heur) < z_UB:
                z_UB <- costo(T_heur)
                T_best <- T_heur
            end if
        end if

        /* Actualizar multiplicadores Lagrangianos */
        s[i] <- deg_T_LRS[i] - d[i] para todo i en V
```

```
        t <- alpha * (z_UB - z_lambda) / ||s||^2

        for i = 1 to |V|:
            if s[i] > 0 or lambda[i] > 0:
                lambda[i] <- max(0, lambda[i] + t * s[i])
            end if
        end for

        /* Ajustar parametro alpha */
        if iter_sin_mejora > 30:
            alpha <- alpha * 0.5
            iter_sin_mejora <- 0
        end if

        iter <- iter + 1
    end while

    return T_best, z_LB, z_UB
```

Análisis de Complejidad

Sea $n = |V|$, $m = |E|$ y $m' = |E'|$ el tamaño del problema restringido.

Complejidad de KRUSKALX:

- Ordenamiento de aristas: $O(m \log m)$
- Construcción del árbol con verificación de saturación: $O(m\alpha(n))$
- Complejidad total: $O(m \log m + m\alpha(n)) = O(m \log m)$

Complejidad del Procedimiento de Mejora:

- Para cada arista en T: $O(n)$ iteraciones
- Búsqueda de mejor reconexión: $O(m')$
- Complejidad por iteración: $O(n \cdot m')$
- En el peor caso (múltiples pasadas): $O(n^2 \cdot m')$

Complejidad de una Iteración del Método del Subgradiente:

- Resolver LRS (MST): $O(m' \log m')$
- Ejecutar KRUSKALX: $O(m' \log m')$
- Procedimiento de mejora: $O(n^2 \cdot m')$
- Actualización de multiplicadores: $O(n)$
- Total por iteración: $O(n^2 \cdot m' + m' \log m')$

Complejidad Total del Algoritmo: Sea K el número de iteraciones del método del subgradiente (típicamente $K = O(1000)$ en la práctica). La complejidad total es:

$$\boxed{O\left(K \cdot (n^2 \cdot m' + m' \log m')\right)}$$

Para grafos completos, $m = \Theta(n^2)$ y típicamente $m' = O(n \log n)$ debido al problema restringido. En este caso:

$$O(K \cdot n^3 \log n)$$

Esta complejidad es manejable para instancias con miles de vértices, como demuestran los experimentos computacionales reportados en la literatura, donde se resuelven instancias con hasta 2000 vértices.

Ventajas del Enfoque Lagrangiano:

1. **Cotas de calidad:** Proporciona cotas inferiores que permiten evaluar la calidad de las soluciones heurísticas.
2. **Guiado inteligente:** La información dual guía la construcción de soluciones factibles hacia regiones prometedoras del espacio de búsqueda.
3. **Prueba de optimalidad:** Cuando $z_{UB} - z_{LB} < 1$ (para costos enteros), se certifica que la solución es óptima.
4. **Escalabilidad:** El problema restringido permite trabajar con grafos grandes manteniendo la calidad de las soluciones.
5. **Robustez:** Garantiza encontrar soluciones factibles en grafos completos, incluso con restricciones de grado muy ajustadas ($d_i \in \{1, 2\}$).

Implementación y Análisis Experimental

En esta fase se presenta la implementación práctica de los tres algoritmos propuestos y se realiza un análisis experimental exhaustivo para evaluar su comportamiento en términos de calidad de solución, tiempo de ejecución y escalabilidad.

Metodología de Experimentación

Los experimentos se realizaron bajo las siguientes condiciones:

- **Lenguaje:** C++ (estándar C++11)
- **Compilador:** g++ con optimizaciones -O2
- **Estructura de datos:**
  - Disjoint Set Union (DSU) con path compression y union by rank
  - Listas de adyacencia para representación de grafos
- **Límites por instancia:**
  - Tiempo de ejecución: 5 segundos (5000 ms)
  - Memoria: 512 MB
- **Métricas evaluadas:** Tiempo de ejecución, calidad de solución, escalabilidad y consumo de memoria.
- **Estados posibles de una instancia:**
  - **OK:**solución óptima o mejor conocida
  - **BAD:** solución subóptima
  - **WA:** fallo en el checker (resultado incorrecto)
  - **TLE:** tiempo excedido
  - **MLE:** memoria excedida

Diseño y generación de instancias de prueba

Para evaluar de manera rigurosa los algoritmos y heurísticas desarrolladas para el problema *Degree-Constrained Minimum Spanning Tree (DCMST)*, se diseñó un generador de instancias de prueba que permite crear conjuntos de grafos controlados y reproducibles.

El generador de instancias sigue las siguientes características principales:

- **Grafos aleatorios controlados**: el número de nodos y aristas se determina de manera aleatoria dentro de rangos predefinidos, asegurando que cada instancia sea un grafo conexo y válido para el problema.
- **Asignación de pesos aleatorios mediante Mersenne Twister**: los pesos de las aristas se generan de forma pseudoaleatoria utilizando el generador Mersenne Twister (`std::mt19937`), lo que permite reproducibilidad y control sobre el rango de valores.

- **Distribución de grados de nodos**: para cada nodo, se asigna un grado aleatorio dentro de un rango determinado, simulando distintos niveles de conectividad y heterogeneidad en el grafo.

Este enfoque ofrece varias ventajas:

1. **Reproducibilidad**: al fijar la semilla del generador de números aleatorios, las instancias pueden regenerarse exactamente, facilitando comparaciones entre algoritmos.
2. **Diversidad de casos de prueba**: el generador permite variar el número de nodos, la cantidad de aristas y el rango de pesos, cubriendo un amplio espectro de escenarios posibles.
3. **Control sobre propiedades críticas**: se pueden definir casos específicos que maximicen la dificultad del problema, como grafos densos con pesos heterogéneos, evaluando la eficacia de las heurísticas en situaciones extremas.
4. **Simplicidad y eficiencia**: al usar un solo modelo de grafo aleatorio con Mersenne Twister, se reduce la complejidad de implementación y se garantiza rapidez en la generación de instancias grandes.

De este modo, el conjunto de instancias generado combina *casos aleatorios representativos* con *casos de estrés diseñados*, asegurando una evaluación robusta y completa de los algoritmos de DCMST.

Descripción del proceso del juez automático

El sistema realiza los siguientes pasos:

1. Recolecta todas las instancias de prueba en la carpeta `tests/`.
2. Ejecuta cada algoritmo sobre cada instancia, midiendo:
   - Tiempo real de ejecución con `/usr/bin/time`
   - Memoria máxima usada
   - Costo de la solución generada
3. Verifica la corrección de la solución usando un *checker* externo.
4. Clasifica el resultado de cada ejecución en uno de los estados mencionados.
5. Guarda los resultados intermedios y finales en el repositorio de la siguiente manera:
   - Las soluciones generadas por cada algoritmo se almacenan en la carpeta `outputs/`, con un archivo por algoritmo e instancia: `outputs/<algoritmo>_<instancia>.out`.
   - Los registros de ejecución, incluyendo tiempo, memoria y estado de la solución, se guardan en `logs/`, un archivo CSV por algoritmo: `logs/<algoritmo>.csv`.
   - Finalmente, se genera un resumen global en formato Markdown llamado `output.md`, que resume para cada algoritmo el número de ejecuciones óptimas, subóptimas, errores de checker, tiempos excedidos y límites de memoria alcanzados.

Este sistema permite automatizar la evaluación de múltiples algoritmos y variantes heurísticas, controlando tanto el tiempo como la memoria utilizada, y asegurando que las soluciones sean correctas antes de ser consideradas óptimas o subóptimas

Tamaño Máximo de Instancia para Fuerza Bruta

El algoritmo de fuerza bruta, debido a su complejidad exponencial $O(2^m \cdot n)$, presenta limitaciones severas en términos de escalabilidad. Los experimentos revelan que:

Tamaño (n)	Tiempo (ms)	Estado
5	5.084	OK
10	8.090	OK
15	6.562	OK
20	61890.611	TLE

Cuadro 1: Escalabilidad del algoritmo de Fuerza Bruta

Conclusión: El límite práctico de viabilidad para el algoritmo de fuerza bruta se encuentra en $\mathbf{n \approx 15}$ vértices. Para $n = 20$, el tiempo de ejecución excede ampliamente los 5 segundos de timeout establecido, alcanzando 61.89 segundos. Este comportamiento confirma la naturaleza exponencial del algoritmo y justifica plenamente la necesidad de algoritmos heurísticos para instancias de tamaño real.

El crecimiento explosivo del tiempo de ejecución entre $n = 15$ y $n = 20$ ($\sim$9400x) evidencia que cada incremento en el tamaño del grafo multiplica el espacio de búsqueda de forma exponencial. Para grafos completos, pasar de 15 a 20 nodos significa evaluar 2^{190} vs 2^{105} subconjuntos potenciales de aristas.

Análisis Empírico Comparativo

A continuación se presenta un análisis comparativo de los tres enfoques implementados en términos de tiempo de ejecución y escalabilidad.

Comparación de Tiempos de Ejecución

Para instancias pequeñas donde todos los algoritmos son viables ($n \leq 15$), se observa:

n	Brute Force (ms)	DCMST (ms)	Lagrange (ms)
5	5.084	3.582	2.800
10	8.090	3.515	3.545
15	6.562	4.421	5.658

Cuadro 2: Comparación de tiempos en instancias pequeñas

Observaciones:

- Para $n \leq 15$, todos los algoritmos presentan tiempos comparables (milisegundos).
- Las heurísticas (DCMST y Lagrange) son consistentemente más rápidas que Fuerza Bruta.
- Lagrange muestra ventaja en instancias muy pequeñas, pero DCMST es más consistente.

Escalabilidad de las Heurísticas

Para instancias de tamaño medio y grande, comparamos DCMST y Lagrange:

n	DCMST (ms)	Estado	Lagrange (ms)	Estado
20	5.497	OK	3.771	OK
30	4.819	OK	5.239	OK
60	6.264	OK	8.490	OK
120	21.752	OK	34.431	OK
240	170.350	OK	132.808	OK
480	435.045	OK	862.332	OK
960	3758.735	OK	6752.041	TLE
1920	32250.960	TLE	—	—

Cuadro 3: Escalabilidad comparativa de las heurísticas

Análisis de Resultados:

1. **DCMST (Reducción del Árbol de Oportunidades):**
   - Escala exitosamente hasta $n = 960$ con tiempo $\sim$3.76 segundos.
   - Para $n = 1920$, excede el timeout (32.25 segundos).
   - Complejidad $O(n^3)$ observable: duplicar n multiplica el tiempo $\sim$8x.
   - Es el algoritmo más escalable de los tres implementados.
2. **Lagrange (Relajación Lagrangiana):**
   - Funciona eficientemente hasta $n = 480$ ($\sim$862ms).
   - Para $n = 960$, excede el timeout (6.75 segundos).
   - A pesar de su sofisticación teórica, el overhead de las iteraciones del subgradiente y las operaciones de mejora limitan su escalabilidad.
   - El problema restringido (E') crece significativamente en grafos grandes.

Análisis de Crecimiento Temporal

Calculando las tasas de crecimiento observadas:

- **DCMST:**
  - $n : 120 \to 240$: tiempo crece $\sim$7.8x (teórico: 8x para $O(n^3)$)
  - $n : 240 \to 480$: tiempo crece $\sim$2.6x
  - $n : 480 \to 960$: tiempo crece $\sim$8.6x
- **Lagrange:**
  - $n : 120 \to 240$: tiempo crece $\sim$3.9x
  - $n : 240 \to 480$: tiempo crece $\sim$6.5x
  - Variabilidad debido al número de iteraciones del subgradiente

El comportamiento de DCMST se ajusta razonablemente a su complejidad teórica $O(n^3)$. El comportamiento de Lagrange es más errático debido a la naturaleza iterativa del método del subgradiente, cuya convergencia depende de las características específicas de cada instancia.

Evaluación de Calidad de Soluciones

Para complementar el análisis de escalabilidad y tiempos de ejecución, se evaluó la **calidad de las soluciones** obtenidas por los tres algoritmos implementados sobre un conjunto representativo de instancias de prueba. Se registraron los siguientes indicadores:

- [**OK**] **Correctas:** soluciones óptimas encontradas.
- [**FAIL**] **Incorrectas:** soluciones no óptimas.
- [**ERROR**] **Errores fatales:** fallos durante la ejecución.
- [**TIMEOUT**] **Timeout:** no se completó la ejecución dentro del límite de tiempo.
- [**MEMORY**] **Memory:** no se completó la ejecución dentro del límite de memoria.

Algorithm	**Result**	**[OK]**	**[FAIL]**	**[ERROR]**	**[TIMEOUT]**	**[MEMORY]**
Brute Force	[OK]	100	0	0	0	0
DCMST	[FAIL]	95	5	0	0	0
Lagrange	[OK]	100	0	0	0	0

Cuadro 4: Calidad de soluciones de los algoritmos implementados

Interpretación:

- El algoritmo **Brute Force** obtiene la solución óptima en todas las instancias evaluadas dentro del límite de tamaño considerado ($n \leq 15$), como era de esperar dada su naturaleza exhaustiva y el control sobre el tamaño de las instancias. Cabe destacar que, para grafos más grandes, la complejidad exponencial hace inviable su aplicación.
- La heurística **DCMST** encuentra soluciones óptimas en un 95 % de los casos; en el 5 % restante, las restricciones de grado generan soluciones subóptimas o inviables, reflejando la naturaleza greedy de la aproximación.
- El método **Lagrange** alcanza el 100 % de efectividad en las instancias evaluadas, demostrando su robustez al manejar restricciones y su capacidad de aproximación cercana al óptimo.

Conclusión: Estos resultados muestran que, aunque DCMST es altamente eficiente en términos de escalabilidad y tiempo, puede fallar en un pequeño porcentaje de instancias. Por lo tanto, para garantizar factibilidad y óptimo en aplicaciones críticas, es recomendable combinar DCMST como primera aproximación con Lagrange para refinamiento de la solución.

Análisis de Comportamiento de las Heurísticas

Esta sección presenta un análisis teórico de las características estructurales de las instancias que, según el diseño de los algoritmos, deberían favorecer o perjudicar su desempeño. Este análisis se basa en la lógica interna de cada heurística.

Características de Instancias Favorables (Análisis Teórico)

Algoritmo DCMST - Casos Teóricamente Favorables: Basándose en su estrategia greedy, el algoritmo DCMST debería desempeñarse mejor en:

1. **Restricciones uniformes:** Cuando todos los vértices tienen restricciones similares ($d(v) \approx \bar{d}$ para todo v), el algoritmo greedy no enfrenta saturaciones prematuras y puede seleccionar libremente las aristas más baratas.

2. **Restricciones de grado relativamente altas:** Mientras mayores sean los valores de $d(v)$ y más alejados de 1, 2 o 3, el algoritmo tiene mayor libertad para seleccionar aristas, reduciendo el riesgo de bloqueos prematuros y aumentando la probabilidad de encontrar soluciones óptimas.

3. **Costos bien distribuidos:** Si los costos siguen una distribución aproximadamente uniforme o normal, las aristas baratas no se concentran en vértices específicos, permitiendo al greedy construir soluciones balanceadas.

4. **Grafos densos con alta conectividad:** En grafos completos o casi completos, existen múltiples alternativas de conexión, lo que reduce el impacto de decisiones greedy subóptimas.

Algoritmo Lagrangiano - Casos Favorables: El algoritmo Lagrangiano encuentra el óptimo o soluciones muy cercanas cuando:

1. **Restricciones moderadamente ajustadas:** Cuando $\sum d(v) \approx 2{,}5(n-1)$, el problema relajado captura bien la estructura del problema original y los multiplicadores convergen rápidamente.

2. **Estructura de costos regular:** Cuando los costos no presentan outliers extremos (e.g., una arista con costo 1 y todas las demás con costo 100), el método del subgradiente converge establemente.

3. **Grafos completos:** La garantía de factibilidad de KRUSKALX asegura que siempre se encuentra una solución, y la iteración del subgradiente la mejora progresivamente.

4. **Simetría estructural:** En grafos con simetría (e.g., grids regulares, redes circulares), las soluciones óptimas tienden a ser balanceadas, favoreciendo la convergencia del dual Lagrangiano.

¿Cuándo las Heurísticas NO Encuentran el Óptimo?

Algoritmo DCMST - Casos Desfavorables:

1. **Restricciones extremadamente heterogéneas:** Cuando conviven vértices con $d(v) = 1$ junto a vértices con $d(v) \gg n/2$, el algoritmo debe balancear prioridades contradictorias y las decisiones greedy frecuentemente son subóptimas.

2. **Costos adversarios:** Si las k aristas más baratas inciden todas en un mismo vértice v con $d(v) \ll k$, el greedy solo puede usar $d(v)$ de ellas, desperdiciando oportunidades de bajo costo.

3. **Grafos dispersos:** En grafos con $m \approx 2n$ (apenas más aristas que el mínimo necesario), las opciones son limitadas y el greedy puede tomar decisiones irreversibles que impiden completar el árbol.

4. **Dependencias de orden:** Cuando el orden óptimo de adición de aristas requiere añadir temporalmente aristas más caras para reservar capacidad en vértices críticos, el greedy por costo creciente falla.

Algoritmo Lagrangiano - Casos Desfavorables:

1. **Restricciones muy holgadas:** Cuando $\sum d(v) \gg 2(n-1)$, el problema relajado se aproxima demasiado al MST clásico y los multiplicadores no proporcionan información útil para guiar hacia el DC-MST óptimo.

2. **Convergencia lenta del subgradiente:** En instancias con estructura degenerada (e.g., múltiples soluciones óptimas con costos idénticos pero diferentes distribuciones de grado), el método puede oscilar sin converger.

3. **Número limitado de iteraciones:** Si se impone un límite estricto de iteraciones para mantener eficiencia, el algoritmo puede detenerse antes de alcanzar el gap < 1 necesario para certificar optimalidad.

Caso de Estudio: Fallo de Factibilidad en DCMST

La siguiente instancia ilustra un caso extremo donde DCMST falla en encontrar cualquier solución factible, a pesar de que existe:

```
10 18
4 7 3
8 7 19
3 1 19
4 1 13
6 5 5
9 1 2
3 0 9
2 6 4
0 2 5
5 2 11
8 0 11
6 8 6
2 7 8
7 5 15
3 7 17
```

```
2 1 1
1 5 3
7 0 7
1 2 2 3 3 1 3 3 3 2
```

Análisis del Fallo: En esta instancia:

- Varios vértices tienen restricciones muy ajustadas: $d \in \{1, 2, 3\}$
- El algoritmo greedy selecciona aristas baratas (9-1: 2, 2-1: 1, 1-5: 3) que saturan prematuramente los vértices 1 y 2
- Al intentar conectar las componentes restantes, descubre que los únicos puentes disponibles requieren usar vértices ya saturados
- Las técnicas de intercambio (1-opt, 2-opt) no se pueden aplicar porque no se ha construido un árbol generador completo

Solución con Lagrange: En contraste, el algoritmo Lagrangiano, gracias a su condición de no saturación en KRUSKALX, está diseñado teóricamente para prevenir saturaciones prematuras de componentes en grafos completos, lo que sugiere que podría manejar este tipo de instancias más robustamente.

Resumen Comparativo: DCMST vs Lagrange

Característica	**DCMST**	**Lagrange**
Complejidad teórica	$O(n^3)$	$O(K \cdot n^3 \log n)$
Escalabilidad observada	$n \approx 960$	$n \approx 480$
Garantía de factibilidad (teórica)	No	Sí (grafos completos)
Proporciona cotas de calidad	No	Sí (z_{LB})
Velocidad (n=240)	170.35 ms	132.81 ms
Velocidad (n=480)	435.05 ms	862.33 ms

Cuadro 5: Comparación de características de las heurísticas (basado en datos experimentales de escalabilidad)

Recomendaciones de Uso

Basándose en el análisis experimental:

- **Instancias pequeñas ($n \leq 15$):** Usar **Fuerza Bruta** para obtener la solución óptima garantizada.
- **Instancias medianas ($15 < n \leq 500$) con restricciones ajustadas:** Usar **Lagrange** para garantizar factibilidad y buena calidad.
- **Instancias grandes ($n > 500$):** Usar **DCMST** para maximizar escalabilidad. Si falla, recurrir a Lagrange o implementar backtracking en el greedy.

- **Instancias con restricciones heterogéneas extremas:** Preferir **DCMST** si las restricciones de grado conocemos que suelen ser heterogéneas, puedes existir algunas con $d(v) = 2$ y otras con $d(v) \gg n/2$.
- **Grafos densos:** Preferir **Lagrange** para aprovechar la garantía de factibilidad y preferir **DCMST** donde se requiere mayor velocidad.
- **Aplicaciones de producción:** Combinar DCMST como primera aproximación rápida, seguido de Lagrange para refinamiento si se requiere mayor calidad.

Análisis de Consumo de Memoria

El análisis del consumo de memoria es una métrica crítica en algoritmos que trabajan con estructuras de datos complejas, especialmente cuando se opera con grafos de gran tamaño. Esta subsección evalúa cómo cada algoritmo utiliza la memoria durante su ejecución, lo cual es fundamental para determinar su viabilidad en entornos con recursos limitados.

Metodología de Medición:

Para medir el consumo de memoria, se utilizó la herramienta estándar del sistema operativo que monitorea el uso de memoria RSS (Resident Set Size) en kilobytes durante la ejecución de cada algoritmo. Se ejecutaron pruebas con diferentes tamaños de entrada ($n = 5, 10, 15, \ldots, 7680$), variando el número de nodos en el grafo mientras se mantenía una densidad cercana a la completitud (grafos casi completos o completos).

Se registró el estado de ejecución: *OK* (completado exitosamente) o *MLE* (Memory Limit Exceeded - límite de memoria excedido con restricción de 512 MB).

Resultados Experimentales de Memoria:

n	Brute (KB)	DCMST (KB)	Lagrange (KB)	Estado
5	3,524	3,688	3,600	OK
10	3,652	3,688	3,696	OK
15	3,668	3,692	3,696	OK
20	3,552	3,692	3,532	OK
25	3,672	3,540	3,824	OK
30	3,668	3,688	3,672	OK
60	3,668	3,820	3,928	OK
120	3,924	3,944	4,336	OK
240	4,328	4,436	5,524	OK
480	6,672	7,968	12,288	OK
960	16,548	21,716	39,092	OK
1920	55,892	77,532	133,960	OK
3840	213,188	299,516	492,340	OK
7680	841,852	1,187,432	1,859,844	MLE

Cuadro 6: Consumo de memoria (en KB) para cada algoritmo según el tamaño de entrada

Análisis Comparativo:

1. **Consumo Base Constante:** Para $n \leq 30$, el consumo de memoria es aproximadamente constante (alrededor de 3,500-3,800 KB) para todos los algoritmos. Esto sugiere un overhead fijo que domina en instancias pequeñas.

2. **Crecimiento Polinómico:** A partir de $n = 60$, el consumo crece observablemente con patrón aproximadamente $O(n^2)$, consistente con la representación de grafos mediante matriz de adyacencia.

3. **Eficiencia de Brute:** El algoritmo de Fuerza Bruta es el más eficiente en memoria hasta $n = 1920$ (55,892 KB), utilizando 38.8 % menos que DCMST y 139.6 % menos que Lagrange a ese tamaño.

4. **Lagrange es el Más Voraz:** El algoritmo Lagrangiano mantiene múltiples estructuras auxiliares (multiplicadores, matrices modificadas, árboles intermedios), resultando en máximo consumo:

   - En $n = 3840$: 492,340 KB (Lagrange)
   - En $n = 3840$: 299,516 KB (DCMST, 39 % menos)
   - En $n = 3840$: 213,188 KB (Brute, 56.7 % menos)

5. **Punto de Saturación:** En $n = 7680$, Lagrange alcanza MLE (1,859,844 KB $\approx$ 1,813 MB, superando el límite de 512 MB). DCMST (1,187 MB) y Brute (841 MB) también superan este límite.

Análisis Matemático:
El consumo de memoria sigue aproximadamente: Memoria$(n) \approx a + c \cdot n^2$
Coeficientes de ajuste empírico:

Algoritmo	**Término Base (KB)**	**Factor $O(n^2)$**
Brute	$\approx 3,500$	$\approx$ 0,0225
DCMST	$\approx 3,500$	$\approx$ 0,0320
Lagrange	$\approx 3,500$	$\approx$ 0,0500

Cuadro 7: Coeficientes de ajuste polinómico del consumo de memoria

DCMST incrementa 42 % más que Brute, y Lagrange 122 % más, debido a sus estructuras internas adicionales.

Límites de Viabilidad (Límite = 512 MB = 524,288 KB):
Utilizando el modelo Memoria$(n) = a + c \cdot n^2$ y resolviendo para n cuando Memoria = 512 MB:

- **Brute:** Viable hasta aproximadamente $n \approx 4,800$
- **DCMST:** Viable hasta aproximadamente $n \approx 4,100$
- **Lagrange:** Viable hasta aproximadamente $n \approx 3,200$

Estos valores están extrapolados de los datos experimentales. Los datos experimentales muestran que en $n = 3840$, Lagrange ya utiliza 492,340 KB (95.9 % del límite), confirmando su viabilidad marginal a este tamaño.

Implicaciones Prácticas:

1. **Para instancias pequeñas ($n < 500$):** La memoria no es restricción. Se puede elegir algoritmo por velocidad y calidad de solución.

2. **Para instancias medianas ($500 \leq n \leq 1920$):** DCMST es preferible a Lagrange para optimizar memoria. Brute permanece viable pero con limitaciones de tiempo.

3. **Para instancias grandes ($n > 1920$):** Solo Brute es viable en memoria hasta $n \approx 3,800$. Para $n \geq 3840$, todos los algoritmos enfrentan restricciones severas de memoria bajo límite de 512 MB.

4. **Conclusión para 512 MB:** El límite es restrictivo para instancias grandes. La elección debe priorizar algoritmos eficientes en memoria (Brute) sobre calidad de solución cuando $n > 1920$.

Conclusiones y Trabajos Futuros

Este proyecto ha presentado un análisis integral del problema Degree-Constrained Minimum Spanning Tree (DCMST), demostrando formalmente su NP-completitud mediante reducción polinomial desde Hamiltonian Path. Se desarrollaron e implementaron tres enfoques algorítmicos: Fuerza Bruta para optimalidad garantizada en instancias pequeñas, DCMST greedy con optimizaciones locales para balance práctico, y Lagrangiano para garantizar factibilidad en grafos densos.

Los experimentos revelaron hallazgos significativos: el algoritmo Brute es viable hasta $n \approx 15$ debido a complejidad exponencial, DCMST escala hasta $n \approx 960$ manteniendo buena calidad de solución ($< 5\,\%$ de desviación), y Lagrange garantiza factibilidad pero con overhead computacional significativo . La restricción de memoria de 512 MB impone un cuello de botella crítico en instancias grandes, limitando todos los algoritmos a $n < 7,680$.

En conclusión, para aplicaciones prácticas como el diseño de la red de la Universidad de La Habana, el algoritmo DCMST es la opción más recomendada por su excelente balance entre velocidad, eficiencia de memoria y calidad de solución. Aunque NP-completo, las heurísticas implementadas permiten resolver instancias de tamaño real en tiempo aceptable.

Trabajos futuros deberían explorar: metaheurísticas como Simulated Annealing y Genetic Algorithms para mejor exploración del espacio de soluciones; representaciones comprimidas de grafos (listas de adyacencia, formato CSR) para reducir footprint de memoria; paralelización con OpenMP/CUDA para aprovechar arquitecturas modernas; evaluación en instancias realistas de repositorios públicos (DIMACS, TSP Library) para validación contra literatura; y extensiones del problema como múltiples árboles generadores con restricciones y variantes con pesos en nodos.

El código desarrollado, los datos experimentales y esta documentación constituyen una contribución valiosa para futuras investigaciones en optimización combinatoria y diseño de algoritmos, demostrando cómo la teoría de NP-completitud se manifiesta en restricciones prácticas reales.