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

1 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.

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.

Conclusión

La existencia de un T válido para DC-MST-Decision es equivalente a la existencia de un camino hamiltoniano en G. Dado que la reducción es polinomial y Hamiltonian Path es NP-completo, se concluye:

DC-MST-Decision es NP-completo.

Consecuencia

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

DC-MST 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 expansivo 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 Expansivos

Dado que el problema de encontrar un árbol expansivo 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 expansivo**, 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 expansivo 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 expansivo mínimo con restricciones de grado** que permiten reducir el grafo original sin perder optimalidad.

Sea T^* un árbol expansivo 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 expansivo. 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 expansivo 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 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

El **Algoritmo Heurístico: Reducción del Árbol de Oportunidades** no garantiza encontrar una solución factible incluso cuando esta existe, debido a decisiones greedy irreversibles que pueden saturar restricciones de grado prematuramente. Además, las técnicas de edge exchange propuestas solo son aplicables cuando ya se ha construido un árbol generador, lo cual no siempre ocurre.

Input donde falla:

```
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
```