División de Ciencias Básicas e Ingeniería

Un problema de barrido de calles

Tesis para obtener el grado de

Maestro en Optimización
por
Ing. Luis Francisco Hernández Sánchez

Asesores:
Dra. Laura Elena Chávez Lomelí
Departamento de Ciencias Básicas, UAM Azcapotzalco

Dr. Francisco Javier Zaragoza Martínez
Departamento de Sistemas, UAM Azcapotzalco

11 de febrero de 2015

Dedicatoria

A todos los que hicieron este trabajo posible.

Agradecimientos

Agradezco a todos los que hicieron este trabajo posible.

Resumen

El *problema de barrido de calles*, en inglés *Street Sweeping Problem* (SSP) es una variante del problema del cartero con viento, en inglés el Windy Postman Problem (WPP), en el cual se deben construir dos recorridos que pasen por cada calle por lo menos una vez en cada dirección, barriendo una dirección en el primer recorrido y la otra dirección en el segundo recorrido.

Se estudió el SSP junto con algunas variantes. Se presentó un modelo de programación entera del problema y se consideraron los poliedros definidos por la relajación de programación lineal del problema original. Se descrubrió algunas características de los poliedros asociados en el caso general. También se estudiaron los poliedros asociados a casos particulares de gráficas y sus características.

Posteriormente se diseñó e implementó un algoritmo de aproximación para el SSP con garantía $\frac{3}{2}\alpha + 1$ para cualquier gráfica, utilizando como subrutina algún algoritmo de aproximación para el WPP con garantía α. Después se presentó otro algoritmo de aproximación que mejora los algoritmos previos y da una garantía 2 para cualquier clase de gráfica con un tiempo polinomial de ejecución. Además se presentaron algoritmos exactos para algunas clases de gráficas. La complejidad computacional de este problema permanece abierta.

Contenido

Lista de Figuras

Capítulo 1
Introducción

El problema central de este trabajo es el *problema de barrido de calles*, en inglés el *Street Sweeping Problem* (SSP) que está en la categoría de problemas de cartero y por lo tanto en la de los problemas de ruteo. Para entender mejor el SSP se presentará con más claridad en qué consiste este tipo de problemas.

Por otra parte, se explicarán algunos conceptos de teoría de gráficas ya que el estudio de los problemas de cartero requiere este tipo de conocimiento para poder analizar los casos especiales o los algoritmos utilizados para resolverlo. Generalmente los recorridos son en ciudades que se pueden modelar como gráficas dirigidas o no dirigidas. Utilizando algunas propiedades de las gráficas se pueden encontrar algoritmos que resuelven los problemas para ciertos casos partículares de gráficas.

La técnica principal que se utiliza para resolver este tipo de problemas es la programación lineal o entera. En general se tiene al principio un modelo entero y de allí se obtiene su relajación de programación lineal para estudiar las características que tienen sus poliedros asociados. Con ellos se puede encontrar algunos casos especiales del problema que permitan resolver de manera exacta el problema en tiempo polinomial. Por el contrario, el estudio de estos poliedros permite reconocer también si el problema no se puede resolver de manera exacta utilizando únicamente la relajación de programación lineal.

1.1. Preliminares

1.1.1. Conceptos de gráficas

Una *gráfica no dirigida* G es una pareja (V, E) de conjuntos V de vértices y E de aristas. Hay una gráfica dirigida asociada D para cualquier gráfica G, la cuál está formada por un par (V, A) de conjuntos V de vértices y A de arcos. D se obtiene reemplazando cada arista $e \in E$ por dos arcos opuestos e^+, e^-. El número de arcos que entran a un vértice $v \in V$ se llama el *ingrado* y se denota como $deg^+(v)$. Al número de arcos que salen de un vértice $v \in V$ se llama el *exgrado* y se denota como $deg^+(v)$.

Dada una gráfica no dirigida $G = (V, E)$, n *paseo* es una secuencia ordenada $W = (v_0, e_1, v_1, ..., e_n, v_n)$ desde v_0 hasta v_n en $V \cup E$ tal que, para todo $1 \leq i \leq n$, e_i se puede recorrer desde v_{i-1} hasta v_i. Si $v_0 = v_n$, entonces W es un *paseo cerrado*. Si W es cerrado y utiliza todos los vértices de G, lo llamamos un *recorrido*. Si un recorrido atraviesa cada arista exactamente una vez, entonces se le llama *recorrido euleriano*. Cuando una gráfica G contiene un recorrido euleriano se llama *gráfica euleriana*. Un recorrido cerrado W se llama *recorrido de cartero* si recorre cada arista por lo menos una vez y a cualquier orientación de un recorrido de cartero se le llama *recorrido de cartero con viento*. Un *ciclo* es un paseo cerrado con $n \geq 3$, es decir, de longitud al menos 3.

Una gráfica no dirigida G es *conexa* si entre cada par de vértices distintos hay un paseo dentro de G. Una gráfica dirigida D es fuertemente conexa si es posible alcanzar cualquier nodo empezando en cualquier otro nodo de la gráfica sólo recorriendo los arcos en la dirección en la que apuntan.

Una gráfica G es un *árbol* cuando es una gráfica conexa sin ciclos.

Se dice que una gráfica es *plana exterior* cuando puede ser dibujada en el plano sin necesidad de cruces de aristas, de manera que todos los vértices tocan la región exterior del dibujo, es decir, que ningún vértice está totalmente rodeado de aristas.

Dada una gráfica no dirigida $G = (V, E)$ y un conjunto $S \subseteq V$, un *corte* $\delta_E(S)$ definido por S es un conjunto de aristas con un extremo en S y el otro extremo en $\overline{S} = V \setminus S$. Se llama *cardinalidad del corte* al número de aristas contenidas en él y se expresa como $|\delta_E(S)|$. Cuando un corte tiene cardinalidad impar se llama *corte impar*.

Dada una gráfica dirigida $D = (V, A)$ y un conjunto $S \subseteq V$. Definimos un *corte dirigido* $\delta_A(S)$ como el conjunto de arcos que tienen un extremo en S y el otro extremo en $\overline{S}$, con $\overline{S} = V \setminus S$. Adicionalmente, se llama *grado* del corte a la cardinalidad de un corte y definimos dos tipos para los cortes dirigidos. Se llama *exgrado* $d_A(S) = |\delta_A(S)|$, es decir, el número de arcos que comienzan en un S y apuntan hacia un vértice en $\overline{S}$. Se llama *ingrado* $d_A(\overline{S}) = |\delta_A(\overline{S})|$, es decir, el número de arcos que comienzan en un $\overline{S}$ y apuntan hacia un vértice en S.

1.1.2. Nociones básicas

Dado un entero positivo n y dos enteros a, b, se dice que a y b son congruentes módulo n si $a - b$ es divisible entre n. De manera equivalente decimos que a y b tienen una relación de congruencia módulo n si a y b tienen el mismo residuo al dividirlos entre n. Esta relación de congruencia se escribe de la siguiente manera: $a \equiv b \ (mod\ n)$.

1.1.3. Programación lineal

Un *problema de programación lineal* es un problema de optimización con restricciones en el cual se busca encontrar un conjunto de valores para las variables continuas $(x_1, x_2, ..., x_n)$ que maximice o minimice una función objetivo lineal z, al mismo tiempo

que satisface un conjunto de restricciones lineales (un sistema de ecuaciones o desigualdades lineales) [3]. La función objetivo define si una asignación particular de valores a las variables es óptima maximizando (o minimizando) el valor total de la función objetivo. Matemáticamente, un programa lineal (LP) se puede expresar de la siguiente manera:

$$\begin{aligned} \text{Maximizar } z &= \sum_j c_j x_j \\ \text{sujeto a:} & \\ \sum_j a_{ij} x_j &\leq b_i,\ i = 1, 2, ..., m \\ x_j &\geq 0,\ j = 1, 2, ..., n \end{aligned}$$

Usando notación de matrices el modelo se puede reescribir de la siguiente manera:

$$\begin{aligned} \text{Maximizar } z &= \mathbf{c}^T \mathbf{x} \\ \text{sujeto a:} & \\ \mathbf{Ax} &\leq \mathbf{b} \\ \mathbf{x} &\geq 0 \end{aligned}$$

En el modelo se tienen los siguientes elementos:

- m es el número de restricciones.
- n es el número de variables continuas.
- $\mathbf{x}$ es un vector columna que contiene n variables continuas, con cada entrada x_j.
- $\mathbf{c}^T$ es un vector renglón de n elementos c_j, que son valores constantes definidos para un problema.
- $\mathbf{A}$ es una matriz $m \times n$, con elementos constantes a_{ij}.
- $\mathbf{b}$ es un vector columna de m constantes, con elementos b_i.

El término *programación* se refiere en este contexto a *planear* actividades que consumen *recursos* o cumplen con requerimientos, tal como se expresan en las m restricciones; no se utiliza el otro significado relacionado con codificar programas de computadora. Los recursos pueden incluir materiales, equipamientos, edificios o plantas de producción, fuerza de trabajo, dinero, tecnología de la información, etc. En el mundo real estos recursos son limitados y se deben compartir entre varias actividades que compiten por ellos. Los requerimientos pueden ser impuestos implícitamente o explícitamente. El objetivo de un programa lineal es asignar los recursos compartidos en todas las actividades de manera óptima cumpliendo todos los requerimientos.

En caso de que no exista una asignación particular de valores a las variables que satisfagan todas las restricciones lineales al mismo tiempo, entonces se dice que el programa lineal es *infactible*.

Un *problema de programación entera* es un problema de programación lineal en el cual las variables están restringidas a valores enteros. Es decir, la programación entera agrega restricciones adicionales a la programación lineal. Este cambio, aparentemente sin importancia, permite cambiar el tipo de problemas que se pueden modelar. Sin embargo los modelos de programación entera se vuelven más difíciles de resolver. Incluso, modelos similares para un mismo problema pueden resultar en tiempos de cálculo completamente diferentes: una formulación puede llevar rápidamente a las soluciones óptimas, mientras que la otra puede tomar un tiempo excesivamente largo para resolver.

Por lo tanto, dado un programa entero (IP), hay siempre un programa lineal asociado llamado *relajación lineal (LR)*, que se forma descartando (relajando) las restricciones de integralidad. Dado que LR tiene menos restricciones que IP se concluye lo siguiente:

- Si IP es un problema de minimización, el valor objetivo óptimo de LR es menor o igual que el valor objetivo óptimo de IP.
- Si IP es un problema de maximización, el valor objetivo óptimo de LR es mayor o igual que el valor objetivo óptimo de IP.
- Si LR es infactible, entonces también IP lo es.
- Si todas las variables en una solución óptima de LR tienen valores enteros, entonces esa solución también es óptima para IP.

En conclusión, puede ser muy útil resolver la relajación lineal de un programa entero, ya que proporciona un límite en el valor óptimo de IP, e incluso, podría dar una solución óptima a IP.

1.1.4. Poliedros

Un poliedro en $\mathbb{R}^n$ es un conjunto del tipo $P = \{x \in \mathbb{R}^n : Ax \leq b\}$ para alguna matriz $A \in \mathbb{R}^{m \times n}$ y algún vector $b \in \mathbb{R}^m$.

Un punto $x \in P$ es llamado punto extremo de P si no es una combinación convexa de otros puntos en P.

Se dice que un poliedro P es entero cuando las coordenadas te todos sus pundos extremos son enteras.

Dados dos puntos a y b en $\mathbb{R}^N$. Un punto $z \in \mathbb{R}^N$ es combinación convexa de a y b si existe un escalar $\alpha \in [0,1]$, tal que $z = \alpha x + (1-\alpha)y$. En otras palabras z está en el segmento de recta que une a con b. Se puede generalizar a más de dos puntos diciendo que el punto y es una combinación convexa de $x^1, x^2, ..., x^M$ si existen unos valores escalares $\alpha_1, \alpha_2, ..., \alpha_M \geq 0$ tales que $y = \alpha_1 x^1 + \alpha_2 x^2 + ... + \alpha_M x^M$ y además $\alpha_1 + \alpha_2 + ... + \alpha_M = 1$.

Se llama *conjunto convexo* al conjunto de puntos $U \subseteq \mathbb{R}^N$ en el que la combinación convexa de cualquier par de puntos en U está en U. Todo el segmento de recta entre cada par arbitrario de puntos en U se mantiene dentro de U.

Dado un conjunto de puntos $x^1, ..., x^K \in \mathbb{R}^N$, su envolvente convexo es el conjunto convexo más pequeño que contiene a $x^1, ..., x^K \in \mathbb{R}^N$.

1.1.5. Algoritmos de aproximación

Un algoritmo α-aproximado para un problema de optimización es un algoritmo que se ejecuta en tiempo polinomial que para todas las instancias del problema produce una solución cuyo valor está dentro de un factor de α del valor de la solución óptima [14].

Para un algoritmo α-aproximado, llamaremos a α la *garantía de desempeño* del algoritmo. En la literatura, también se le nombra a α como *razón de aproximación* o *factor de aproximación* del algoritmo. Utilizarémos $\alpha > 1$ para problemas de minimización y $\alpha < 1$ para problemas de maximización. Entonces, un algoritmo $\frac{1}{2}$-aproximado para un problema de maximización es un algoritmo polinomial que siempre regresa una solución cuyo valor es por lo menos la mitad del valor óptimo.

1.1.6. Problemas de cartero

Los problemas de cartero tienen su origen en 1736, cuando Euler modeló el problema de los puentes de Königsberg en el cual había que encontrar un paseo en una gráfica que recorriera cada una de las aristas exactamente una vez [7]. Un paseo de ese tipo se le llama paseo euleriano. Al problema que consiste en decidir si una gráfica no dirigida tiene un paseo euleriano se le llama *paseo euleriano no dirigido*. Euler determinó las condiciones necesarias para que una gráfica no dirigida pueda tener un paseo euleriano [7]. Después, en 1873, Hierholzer determinó que esas condiciones también eran suficientes [9]. El problema se extendió también a las gráficas dirigidas y en 1936, Kőnig dio una caracterización de las gráficas dirigidas eulerianas [10].

1.1.7. Problema del cartero chino

En 1962, Guan extendió el problema formulado por Euler proponiendo que un cartero quiere recorrer todo un vecindario (gráfica), de forma que se siga un circuito cerrado (que empiece y termine en el mismo lugar), que cada calle (arco o arista) la recorra por lo menos una vez y que este circuito sea el más corto posible [8]. Este problema se le llama *problema del cartero chino* (CPP por sus siglas en inglés) y se aplica a gráficas dirigidas y no dirigidas. Hay una gran variedad de problemas relacionados con éste y se les llama *problemas de cartero*. Además, a un recorrido cerrado de la gráfica que pasa por cada una

de las aristas o arcos de una gráfica por lo menos una vez se le llama *recorrido de cartero*.

Para el caso de las gráficas no dirigidas, en inglés *Undirected Postman Problem* (UPP). El problema recibe como entrada una gráfica no dirigida conexa $G = (V, E)$ y un vector de costos racionales no negativos $c \in \mathbb{Q}_+^E$. La salida es un recorrido de cartero de costo mínimo UPP(G, c) para la gráfica G y los costos c. En 1965, Edmonds presentó un algoritmo polinomial para este problema basado en su algoritmo de acoplamiento perfecto [6]. Edmonds explica que dado un recorrido de cartero en G, cada arista e está en el paseo al menos una vez, pero tal vez está más de una vez. Entonces el número de veces que la arista e está en el recorrido se puede expresar como $1 + x_e$. Define como G' a la gráfica formada al agregar a G unas copias adicionales de la arista e, entonces G' tiene $1 + x_e$ copias de la arista e. Como consecuencia el recorrido de cartero se convierte en un paseo euleriano por la gráfica G' en la cual cada vértice es incidente con un número par de aristas. A partir de esto la formulación del problema queda de la siguiente manera:

$$\begin{aligned} \text{UPP}(G, c) \quad &: \quad \text{minimizar } \textstyle\sum c_e x_e \\ \text{s.a.} & \\ \textstyle\sum_e (1 + x_e) \quad &\equiv \quad 0 \ (mod\ 2) \\ x_e \quad &\geq \quad 0 \end{aligned}$$

Un caso especial es cuando la gráfica es euleriana, ya que la solución es un paseo euleriano por la gráfica.

En 1973, Edmonds y Johnson mostraron que la formulación de programación entera del problema de cartero en las gráficas no dirigidas es equivalente a su relajación de programación lineal [6]. Para el caso de las gráficas dirigidas se tiene el *problema de cartero dirigido*, en inglés el *Directed Postman Problem* (DPP). Edmonds y Karp encontraron también un algoritmo de tiempo polinomial [6]. En este caso el problema recibe como entrada una gráfica dirigida fuertemente conexa $D = (V, A)$ y un vector de costos racionales no negativos $c \in \mathbb{Q}_+^A$. Como salida da un recorrido de cartero DPP(D, c) a través de D recorriendo cada arco por lo menos una vez. Utilizando las restricciones propuestas por Kőnig [10] se puede formular este problema de la siguiente manera:

$$\begin{aligned} \text{DPP}(D, c) \quad &= \quad \text{minimizar } \textstyle\sum c_a x_a \\ \text{s.a} & \\ x(\delta^+(v)) - x(\delta^-(v)) \quad &= \quad 0, \forall v \in V \\ x_a \quad &\geq \quad 1, \forall a \in A, \text{ enteros} \end{aligned}$$

En esta formulación el valor de x_a representa el número de veces que se recorre el arco a. Además, $\delta(v)$ es el corte dirigido que separa v del resto de la gráfica, $\delta^-(v)$ es el conjunto de aristas de $\delta(v)$ que entran a v y $\delta^+(v)$ son las que salen de v.

1.1.8. Problema del cartero con viento

Una de las variantes del CPP no dirigido es el *problema del cartero con viento* (WPP por sus siglas en inglés), en el cual el costo de recorrer una calle (arista) depende de la dirección en que se recorra. Entonces se busca un recorrido de cartero de costo mínimo por toda la gráfica que pase por todas las aristas por lo menos una vez en cualquier dirección. El problema recibe como entrada una gráfica no dirigida conexa $G = (V, E)$ (siendo la gráfica $D = (V, E^+ \cup E^-)$ su gráfica dirigida asociada) y un vector de costos $c \in \mathbb{Q}_+^{E^+ \cup E^-}$ definido en los arcos de D. La salida del problema es el recorrido de cartero de longitud mínima WPP(G, c). Una formulación de programación entera hecha por Win [16, 15] es la siguiente:

$$\begin{array}{rcll} \text{WPP}(G,c) & : & \text{minimizar } \sum(c_{e^+}x_{e^+}) + (c_{e^-}x_{e^-}) & \\ \text{sujeto a} & & & \\ x(\delta_D^+(v)) - x(\delta_D^-(v)) & = & 0, & \forall v \in V \\ x_{e^+} + x_{e^-} & \geq & 1, & \forall e \in E \\ x_{e^+}, x_{e^-} & \geq & 0, & \forall e \in E \\ x_{e^+}, x_{e^-} & & \text{enteros}, & \forall e \in E \end{array}$$

El problema es NP duro y la versión de decisión de este problema es NP completa. Sin embargo Win [16, 15] demostró que se puede resolver en tiempo polinomial para las siguientes clases de gráficas: eulerianas no dirigidas, es decir las gráficas pares, las no dirigidas que tienen todo su poliedro con puntos enteros, los bosques no dirigidos, todas las de dos vértices. Además, otro caso en el que el problema tiene solución en tiempo polinomial es cuando las gráficas son serie-paralelo que fue demostrado por Zaragoza [17].

Para el caso general del WPP hay dos algoritmos de aproximación diferentes presentados por Win [15, 16] que tienen una garantía de aproximación de 2. Posteriormente Raghavachari y Veerasamy presentaron un algoritmo de aproximación con garantía $\frac{3}{2}$ [12], siendo hasta ahora la mejor garantía conocida.

Capítulo 2
Problema de barrido de calles

2.1. Enunciado del problema

El problema central que vamos a estudiar es el *problema de barrido de calles*, en inglés *Street Sweeping Problem* (SSP), en donde se debe barrer cada calle de una ciudad con un vehículo barredor recorriendo la ruta más corta posible. Fue definido por primera vez en [2, 5]. Consideramos que cada calle tiene dos direcciones opuestas. Puede haber autos estacionados en los costados de las calles, por lo que no sería posible barrer en cualquier momento. Entonces, se propone utilizar señales de tránsito para estacionar todos los autos de cada calle en sólo alguno de sus sentidos mientras que la dirección opuesta se barre [2, 5]. Entonces, para poder barrer ambos sentidos de cada calle, se necesitan por lo menos periodos de dos días, para barrer un sentido de cada calle un día y barrer el sentido opuesto al otro día.

Modelamos este problema con una gráfica $G = (V, E)$ en donde cada arista representa una calle. Entonces, podemos transformar G en una gráfica dirigida $\vec{G} = (V, A)$ reemplazando cada arista $e \in E$ por dos arcos opuestos e^+, e^- que representan las dos direcciones de cada calle. Además, asociamos costos no negativos a cada arco $\vec{G}$, que representan la distancia o algún costo asociado con barrer una calle en cada dirección. El problema consiste en encontrar dos recorridos cerrados en $\vec{G}$ que minimicen el costo total de barrer un sentido de cada arista en el primer recorrido y la otra dirección de las aristas en el segundo recorrido.

2.2. Ejemplo

Comenzamos con la gráfica G en la figura 2.1 y la convertimos en la gráfica dirigida $\vec{G}$ de la misma figura 2.1.

Ahora, el objetivo es encontrar dos recorridos cerrados que pasen por cada una de las 6 aristas de G por lo menos una vez. Sin embargo, la versión dirigida de esos recorridos debe pasar al menos por la mitad de los arcos de $\vec{G}$ en el recorrido para el día 1 y al menos por la otra mitad en el día 2, al mismo tiempo que para cada par de arrcos asociados a

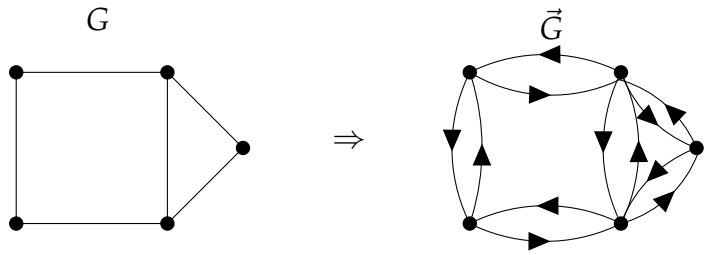


Figura 2.1: Transformación de una gráfica G en $\vec{G}$

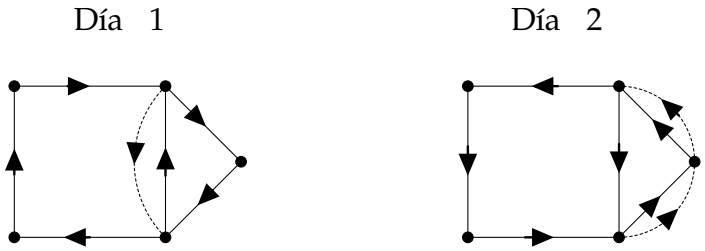


Figura 2.2: Ejemplo de solución en $\vec{G}$

una misma arista, uno se barre en el día 1 y el otro en el día 2.

Un ejemplo de solución sería el par de recorridos de cartero con viento ilustrados en la figura 2.2.

En este ejemplo, los arcos de línea punteada indican que se recorre un arco sin barrerlo. Los arcos dibujados con línea continua indican que ese arco se barre en ese recorrido.

Hay muchas posibles maneras en que se pueden elegir los arcos para barrerse en el día 1 o 2. Por lo menos hay $2^{|E|}$ posibilidades de formar los recorridos, siendo $|E|$ el número de aristas que tiene G, ya que para cada una de ellas hay dos posibilidades, barrerse en el día 1 o en el día 2. Por otra parte, una vez que se ha decidido cuáles son los arcos que se barrerán en cada día, hay una gran manera de unir los extremos de los arcos de forma que formen un recorrido cerrado por $\vec{G}$, lo mejor es unir los arcos elegidos con el menor costo posible. Lo anterior se puede calcular en tiempo polinomial usando algoritmos de flujo.

Para encontrar la manera más corta de formar los dos recorridos se procede a hacer una formulación de programación entera que permite minimizar el costo y cumplir con las restricciones del problema.

2.3. Formulación de programación entera

Para crear una formulación de programación entera para el SSP, definimos las variables siguientes:

- $x_{ij}^1 = 1$: indican si se barre el arco ij en el día 1.
- $x_{ij}^2 = 1$: indican si se barre el arco ij en el día 2.
- z_{ij}^1: cuántas veces recorremos sin barrer el arco ij en el día 1.
- z_{ij}^2: cuántas veces recorremos sin barrer el arco ij en el día 2.

Adicionalmente, el problema recibe como entrada la función de costos c_{ij} que indica el costo de recorrer de cualquier manera al arco ij.

Adicionalmente tenemos las siguientes observaciones:

- El costo de recorrer un arco es el mismo para ambos días.
- El costo del arco ij puede ser diferente al costo del arco ji.

Con las variables definidas obtenemos el siguiente programa entero (PE):

$$\text{mín} \sum_{ij \in A} c_{ij}(x_{ij}^1 + x_{ij}^2 + z_{ij}^1 + z_{ij}^2) \tag{2.1}$$

sujeto a:

$$\sum_{ij \in A} (x_{ij}^1 + z_{ij}^1) = \sum_{ji \in A} (x_{ji}^1 + z_{ji}^1), \forall i \in V \tag{2.2}$$

$$\sum_{ij \in A} (x_{ij}^2 + z_{ij}^2) = \sum_{ji \in A} (x_{ji}^2 + z_{ji}^2), \forall i \in V \tag{2.3}$$

$$x_{ij}^1 + x_{ij}^2 = 1, \forall ij \in A \tag{2.4}$$

$$x_{ij}^1 + x_{ji}^1 = 1, \forall ij \in A \tag{2.5}$$

$$x_{ij}^2 + x_{ji}^2 = 1, \forall ij \in A \tag{2.6}$$

$$x_{ij}^1, x_{ij}^2, z_{ij}^1, z_{ji}^2 \geq 0, \text{enteras}, \forall ij \in A \tag{2.7}$$

Las ecuaciones 2.2 y 2.3 son restricciones de conservación de flujo, para hacer que la solución sea factible como recorrido, manteniendo la secuencia de recorrido unida. La ecuación 2.4 forza que cada arco se barra en algún día. Finalmente, las ecuaciones 2.5 y 2.6 evitan que se barran ambos sentidos de una calle en el mismo día.

La relajación de programación lineal (LR) se obtiene eliminando la restricción de que las variables x_{ij}^1, x_{ij}^2, z_{ij}^1, z_{ij}^2 deban ser enteras.

2.3.1. Segunda formulación

A continuación proponemos una nueva formulación que reduce el número de variables necesarias para el modelo de programación entera del SSP.

Primero notamos que para cada arista de G definimos en el modelo anterior las variables: $x^1_{ij}, x^1_{ji}, x^2_{ij}, x^2_{ji}$. Estas variables se pueden reducir a sólo una, por ejemplo, la variable x^1_{ij}, la cual podemos renombrar al final como x_{ij}.

La restricción 2.5 del modelo anterior nos permite barrer sólo una dirección de una calle por día. Si para el arco ij, se cumple que $x^1_{ij} = 1$ entonces $x^1_{ji} = 0$. De manera similar, si nosotros no barremos el arco ij en el día 1, es decir, $x^1_{ij} = 0$ entonces $x^1_{ji} = 1$. Como estas dos variables x^1_{ij} y x^1_{ji} son complementarias $x^1_{ji} = 1 - x^1_{ij}$.

En segundo lugar, la restricción $x^1_{ij} + x^2_{ij}$ indica que podemos barrer un lado de una calle sólo una vez. Si barremos el arco ij, se cumple que $x^1_{ij} = 1$ entonces no podemos barrer esa calle en el mismo sentido durante el día 2, entonces $x^1_{ij} = 0$. Como las variables x^1_{ij} y x^2_{ij} son complementarias $x^2_{ij} = 1 - x^1_{ij}$.

De una manera similar, x^2_{ji} es complementaria a las variables x^2_{ij} y x^1_{ji}, entonces podemos asignar a x^2_{ji} el valor de x^1_{ij} directamente.

De esta manera, las cuatro variables se pueden reemplazar por x^1_{ij} y por el valor $1 - x^1_{ij}$, utilizando una sola variable para asignarles valor a las cuatro. Después podemos simplemente renombrar a la variable x^1_{ij} como x_{ij}.

Los valores quedan de la siguiente manera:

- $x^1_{ij} \to x_{ij}$
- $x^1_{ji} \to 1 - x_{ij}$
- $x^2_{ij} \to 1 - x_{ij}$
- $x^2_{ji} \to x_{ij}$

En particular, la variable x_{ij} que elegiríamos sería la siguiente. Dado que para cada arista $e \in E$ tenemos un par de arcos ij o ji. Después enumerando todos los nodos en V, elegiremos la variable x^1_{ij} cuando $i < j$.

Utilizando la reducción de variables explicada previamente el modelo se transforma en el siguiente:

$$\text{mín} \quad \sum_{ij \in A} c_{ij}(1 + z^1_{ij} + z^2_{ij}) \tag{2.8}$$

sujeto a:

$$\sum_{ij \in A, i<j} (x_{ij} + z^1_{ij}) + \sum_{ij \in A, i>j} (1 - x_{ji}) = \sum_{ij \in A, i>j} x_{ji} + \sum_{ij \in A, i<j} (1 - x_{ij}) + z^1_{ji}, \forall i \in V \tag{2.9}$$

$$\sum_{ij \in A, i>j} (x_{ji} + z^2_{ij}) + \sum_{ij \in A, i<j} (1 - x_{ij}) = \sum_{ij \in A, i<j} x_{ij} + \sum_{ij \in A, i>j} (1 - x_{ji}) + z^2_{ji}, \forall i \in V \tag{2.10}$$

$$x_{ij} \in \{0, 1\}, \text{enteras}, \forall ij \in A \tag{2.11}$$

$$z^1_{ij}, z^1_{ji}, z^2_{ji}, z^2_{ji} \geq 0, \text{enteras}, \forall ij \in A \tag{2.12}$$

Simplificando, se puede reescribir el modelo así:

$$\text{mín} \quad \sum_{ij \in A} c_{ij}(1 + z^1_{ij} + z^2_{ij}) \tag{2.13}$$

sujeto a:

$$\sum_{ij \in A, i<j} (2x_{ij} + z^1_{ij}) = \sum_{ij \in A, i>j} (2x_{ji} + z^1_{ji}), \forall i \in V \tag{2.14}$$

$$\sum_{ij \in A, i>j} (2x_{ji} + z^2_{ij}) = \sum_{ij \in A, i<j} (2x_{ij} + z^2_{ji}), \forall i \in V \tag{2.15}$$

$$x_{ij} \in \{0, 1\}, \text{enteras}, \forall ij \in A \tag{2.16}$$

$$z^1_{ij}, z^1_{ji}, z^2_{ji}, z^2_{ji} \geq 0, \text{enteras}, \forall ij \in A \tag{2.17}$$

En resumen, se puede reducir el número de variables por arista de 8 a 5, cuatro variables $z_{ij} : z^1_{ij}, z^1_{ji}, z^2_{ij}, z^2_{ji}$ y la variable x_{ij}.

2.4. Poliedros

Sea $\mathcal{P}(\vec{G})$ el poliedro definido por el envolvente convexo de las soluciones factibles del programa entero del SSP. Usando la relajación de programación lineal (LR) obtenida de PE, definimos el poliedro $\mathcal{Q}(\vec{G})$, el cual es el poliedro formado por todas las soluciones factibles de LR.

Un punto $x \in \mathcal{Q}(\vec{G})$ es llamado punto extremo de $\mathcal{Q}(\vec{G})$ si no es una combinación convexa de otros puntos en $\mathcal{Q}(\vec{G})$, en otras palabras, un punto extremo $x \in \mathcal{Q}(\vec{G})$ es una solución factible básica de LR. Por otra parte, un punto extremo $x \in \mathcal{P}(\vec{G})$ es una solución factible de PE.

2.5. Observaciones

Las soluciones para el SPP están representadas por cuatro vectores de incidencia x^1, x^2, z^1, z^2 teniendo cada uno una entrada diferente para cada arco en $\vec{G}$, es decir, cada vector agrupa $|A|$ variables.

En los problemas de cartero no dirigidos como el CPP se requiere encontrar una solución que puede expresarse como un vector x con cada entrada tomando el valor $\{0, 1\}$, indicando el número de veces extra que se necesita recorrer cada arista de la gráfica original G para completar el recorrido de cartero cerrado. El recorrido final pasa por cada arista una vez como mínimo, entonces basta con contabilizar las veces adicionales que se utilizan las aristas. Cuando se asigna el valor de 0 a una entrada particular ($x_e = 0$) esa arista sólo se recorrerá una vez en la solución final. Si se recorriera alguna de las aristas más de una vez, es decir $x_e \geq 2$, se puede disminuir x_e en dos unidades sin aumentar el costo de ese recorrido de cartero. En este caso lo que representan dos unidades de x_e es ir y venir por una arista no dirigida sacando ningún provecho de ello.

En nuestro caso con el SSP, la solución final consiste en recorridos de cartero dirigidos, uno para el día 1 y otro para el día 2. Si encontramos una solución que tiene alguna entrada con $z^1_a \geq 1$ y la entrada correspondiente a su arco opuesto $z^1_{a'} \geq 1$ se puede restar una unidad al par de arcos para obtener una solución factible con un costo total no mayor a la solución anterior. En un caso extremo en que ese par de arcos tengan $c_a = 0, c_{a'} = 0$ la solución conserva el mismo costo, de lo contrario nos aseguraríamos de obtener una solución con menor costo total. De manera similar se puede modificar el vector de incidencia z^2 para obtener soluciones con un costo total no mayor.

Tomando alguna solución S para el SPP. Si S tiene ciclos dirigidos en el recorrido para el día 1 o 2, en los cuales cada arco del ciclo dirigido está asociado a una variable de los vectores z^1 o z^2. Entonces, podemos obtener una nueva solución S' que tiene un costo no mayor al de S eliminando pasar por todos esos arcos del ciclo. Se ve claramente que recorrer ciclos extra no es de utilidad para obtener la solución de menor costo, ya que se pasaría por cada calle sin barrerla y volviendo al mismo lugar donde se empezó. Mejor se evitan ese tipo de ciclos y se continúa cuanto antes con el barrido de las calles.

Capítulo 3
Análisis de la relajación lineal del SSP

En este capítulo nos enfocamos en estudiar la relajación lineal de un modelo entero propuesto para el SSP. Ya que los programas lineales se pueden resolver con un tiempo de procesamiento menor y a partir de ellos es posible sacar conclusiones para el problema en general. Generalmente los programas enteros son mucho más complicados que los programas lineales, ya que muchas clases de programas enteros son conocidos como problemas NP completos. Tratar de resolver ese tipo de problemas toma un tiempo muy largo de procesamiento para encontrar la solución óptima, incluso puede tomar años para problemas de tamaño real.

3.1. Análisis experimental

Para estudiar el SSP, analizaremos los poliedros asociados a su relajación lineal (LR) y a su programa entero (PE). Para ello definiremos los siguientes poliedros.

Sea $\mathcal{P}(\vec{G})$ el poliedro definido por el envolvente convexo de las soluciones factibles de PE. Sea $\mathcal{Q}(\vec{G})$ el poliedro definido por las soluciones factibles de LR.

Recordemos que un punto $x \in \mathcal{Q}(\vec{G})$ es llamado punto extremo de $\mathcal{Q}(\vec{G})$ si no es una combinación convexa de puntos distintos a x en $\mathcal{Q}(\vec{G})$. Visto desde otro punto de vista, un punto extremo es una solución básica de LR. Por otra parte, un punto $x \in \mathcal{Q}(\vec{G})$ es llamado punto factible en $\mathcal{Q}(\vec{G})$ si corresponde a una solución factible de LR.

Se sabe que si cada punto extremo de $\mathcal{Q}(\vec{G})$ tuviera coordenadas enteras, entonces $\mathcal{P}(\vec{G}) = \mathcal{Q}(\vec{G})$. Consideramos que un *punto extremo es fraccionario* cuando una o más de sus coordenadas tienen un valor fraccionario. Por el contrario, consideramos que un punto extremo es *entero* cuando todas sus coordenadas tienen un valor entero.

Se dice que un poliedro es *entero* cuando todas las coordenadas de sus puntos extremos tienen valores enteros.

Se dice que un poliedro es *semientero* cuando todas las coordenadas de sus puntos extremos tienen como valor un múltiplo entero de $\frac{1}{2}$. Por ejemplo, $0, \frac{1}{2}, 1, \frac{3}{2}, 2, \frac{5}{2}, 3$,etc.

Estudiamos la relajación de programación lineal del SPP utilizando un software especializado llamado PORTA [4] que permite trabajar con los poliedros de los programas

lineales. Con este software verificamos si los poliedros analizados en la muestra tienen puntos extremos enteros o fraccionarios.

Utilizando PORTA podemos descubrir los puntos extremos de un modelo dando las desigualdades de las restricciones. Además de esto, se utilizará otro software llamado nauty [11] para generar una gran cantidad de gráficas diferentes para probar cada una de ellas y descubrir cuáles de todas ellas tienen características que permiten resolver más rápido el problema.

El proceso que se utilizó es el siguiente:

- Primero se generó una gran cantidad de gráficas para su estudio.
- Se creó el modelo de programación lineal del SSP para cada gráfica particular.
- A partir de ese modelo se generó un archivo en el formato de PORTA con las desigualdades del modelo y con una solución factible.
- Ese archivo se entregó a PORTA para que genere otro archivo con los puntos extremos del poliedro equivalente a las desigualdades.
- El archivo obtenido se analizó para verificar si todos los puntos obtenidos fueron enteros.
- Si todos los puntos eran enteros la gráfica se guardaba en la categoría de gráficas que permiten resolver su relajación lineal y obtener la solución entera óptima.
- En el caso contrario, es decir, cuando algunos de los puntos tuvieron coordenadas fraccionarias, se quitaron esos puntos fraccionarios y se ejecutó de nuevo PORTA con la instrucción inversa, para que recibiendo el conjunto de puntos extremos del poliedro, devolviera las desigualdades que generarían ese poliedro.
- Posteriormente se intentó descifrar las desigualdades nuevas para determinar sus características y saber cuáles son las restricciones necesarias para acotar el problema de forma que la relajación lineal dé soluciones enteras.

3.2. Ejemplos

A continuación se presentan dos ejemplos, para ilustrar el proceso descrito anteriormente.

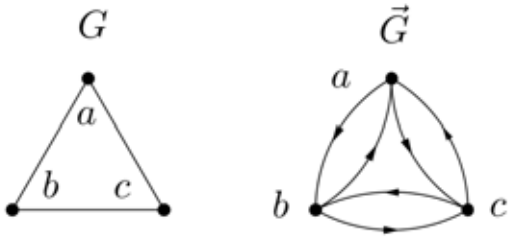


Figura 3.1: Gráfica G tríangulo y su $\vec{G}$ correspondiente.

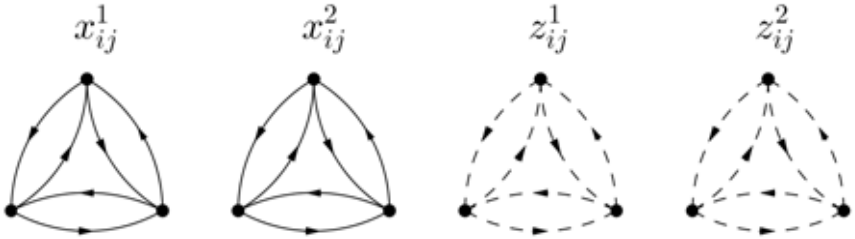


Figura 3.2: Variables para la gráfica triángulo.

3.2.1. Gráfica triángulo

En este caso las calles forman un triángulo. Primero obtenemos la gráfica $\vec{G}$ (fig. 3.1) y obtenemos las variables de limpieza y de recorrido que son las siguientes. Por cada arco de $\vec{G}$ se tiene una variable $x^1_{ij}, x^2_{ij}, z^1_{ij}, z^2_{ij}$ sumando $4|A|$ variables (fig. 3.2).

Las restricciones para el triángulo quedan de la siguiente manera:

$$
\begin{aligned}
x^1_{ab} + x^1_{ac} + z^1_{ab} + z^1_{ac} &= x^1_{ba} + x^1_{ca} + z^1_{ba} + z^1_{ca} \\
x^1_{ba} + x^1_{bc} + z^1_{ba} + z^1_{bc} &= x^1_{ab} + x^1_{cb} + z^1_{ab} + z^1_{cb} \\
x^1_{ca} + x^1_{cb} + z^1_{ca} + z^1_{cb} &= x^1_{ac} + x^1_{bc} + z^1_{ac} + z^1_{bc} \\
x^2_{ab} + x^2_{ac} + z^2_{ab} + z^2_{ac} &= x^2_{ba} + x^2_{ca} + z^2_{ba} + z^2_{ca} \\
x^2_{ba} + x^2_{bc} + z^2_{ba} + z^2_{bc} &= x^2_{ab} + x^2_{cb} + z^2_{ab} + z^2_{cb} \\
x^2_{ca} + x^2_{cb} + z^2_{ca} + z^2_{cb} &= x^2_{ac} + x^2_{bc} + z^2_{ac} + z^2_{bc}
\end{aligned}
$$

$$
\begin{aligned}
&x^1_{ab} + x^2_{ab} = 1, \quad x^1_{ba} + x^2_{ba} = 1, \quad x^1_{bc} + x^2_{bc} = 1, \quad x^1_{cb} + x^2_{cb} = 1 \\
&x^1_{ca} + x^2_{ca} = 1, \quad x^1_{ac} + x^2_{ac} = 1, \quad x^1_{ab} + x^1_{ba} = 1, \quad x^2_{ab} + x^2_{ba} = 1 \\
&x^1_{bc} + x^1_{cb} = 1, \quad x^2_{bc} + x^2_{cb} = 1, \quad x^1_{ca} + x^1_{ac} = 1, \quad x^2_{ca} + x^2_{ac} = 1
\end{aligned}
$$

$$
\begin{aligned}
x^1_{ij}, x^2_{ij}, z^1_{ij}, z^2_{ij} &\geq 0, \forall ij \in A \\
x^1_{ij}, x^2_{ij} &\leq 1, \forall ij \in A
\end{aligned}
$$

Se genera el archivo triangulo.ieq en el formato de PORTA para poder procesar las desigualdades. El archivo se entrega a PORTA y genera el archivo triangulo.ieq.poi. Este

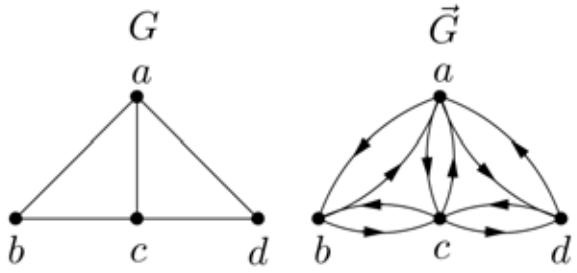


Figura 3.3: Gráfica G diamante y su $\vec{G}$ correspondiente.

archivo contiene los puntos extremos del poliedro. Cada uno de los renglones dice las coordenadas de un punto extremo del poliedro. En este caso cada uno de los puntos tiene coordenadas enteras, por lo tanto su relajación lineal tiene solución entera y se guarda esta gráfica junto con las demás que tienen esta propiedad.

3.2.2. Gráfica diamante

En este caso las calles forman un diamante (fig. 3.3). Cada arista del diamante representa una calle. Se genera el modelo de programación entera que corresponde a esta gráfica.

Las variables necesarias para esta gráfica son similares a las del ejemplo anterior; ocho para cada arista. Las restricciones del modelo lineal quedan de la siguiente manera:

$$
\begin{aligned}
x^1_{ab} + x^1_{ac} + x^1_{ad} + z^1_{ab} + z^1_{ac} + z^1_{ad} &= x^1_{ba} + x^1_{ca} + x^1_{da} + z^1_{ba} + z^1_{ca} + z^1_{da} \\
x^1_{ba} + x^1_{bc} + z^1_{ba} + z^1_{bc} &= x^1_{ab} + x^1_{cb} + z^1_{ab} + z^1_{cb} \\
x^1_{ca} + x^1_{cb} + x^1_{cd} + z^1_{ca} + z^1_{cb} + z^1_{cd} &= x^1_{ac} + x^1_{bc} + x^1_{dc} + z^1_{ac} + z^1_{bc} + z^1_{dc} \\
x^1_{da} + x^1_{dc} + z^1_{da} + z^1_{dc} &= x^1_{ad} + x^1_{cd} + z^1_{ad} + z^1_{cd} \\
x^2_{ab} + x^2_{ac} + x^2_{ad} + z^2_{ab} + z^2_{ac} + z^2_{ad} &= x^2_{ba} + x^2_{ca} + x^2_{da} + z^2_{ba} + z^2_{ca} + z^2_{da} \\
x^2_{ba} + x^2_{bc} + z^2_{ba} + z^2_{bc} &= x^2_{ab} + x^2_{cb} + z^2_{ab} + z^2_{cb} \\
x^2_{ca} + x^2_{cb} + x^2_{cd} + z^2_{ca} + z^2_{cb} + z^2_{cd} &= x^2_{ac} + x^2_{bc} + x^2_{dc} + z^2_{ac} + z^2_{bc} + z^2_{dc} \\
x^2_{da} + x^2_{dc} + z^2_{da} + z^2_{dc} &= x^2_{ad} + x^2_{cd} + z^2_{ad} + z^2_{cd} \\
x^1_{ab} + x^2_{ab} = 1, \quad x^1_{ba} + x^2_{ba} = 1, \quad x^1_{bc} + x^2_{bc} = 1, \quad x^1_{cb} + x^2_{cb} &= 1 \\
x^1_{ca} + x^2_{ca} = 1, \quad x^1_{ac} + x^2_{ac} = 1, \quad x^1_{da} + x^2_{da} = 1, \quad x^1_{ad} + x^2_{ad} &= 1 \\
x^1_{cd} + x^2_{cd} = 1, \quad x^1_{dc} + x^2_{dc} = 1, \quad x^1_{ab} + x^1_{ba} = 1, \quad x^2_{ab} + x^2_{ba} &= 1 \\
x^1_{bc} + x^1_{cb} = 1, \quad x^2_{bc} + x^2_{cb} = 1, \quad x^1_{ca} + x^1_{ac} = 1, \quad x^2_{ca} + x^2_{ac} &= 1 \\
x^1_{da} + x^1_{ad} = 1, \quad x^2_{da} + x^2_{ad} = 1, \quad x^1_{dc} + x^1_{cd} = 1, \quad x^2_{dc} + x^2_{cd} &= 1 \\
x^1_{ij}, x^2_{ij}, z^1_{ij}, z^2_{ij} &\geq 0, \forall ij \in A \\
x^1_{ij}, x^2_{ij} &\leq 1, \forall ij \in A
\end{aligned}
$$

Se genera el archivo diamante.ieg en el formato de PORTA para poder procesar las desigualdades. El archivo se entrega a PORTA y genera el archivo diamante.ieq.poi. Este

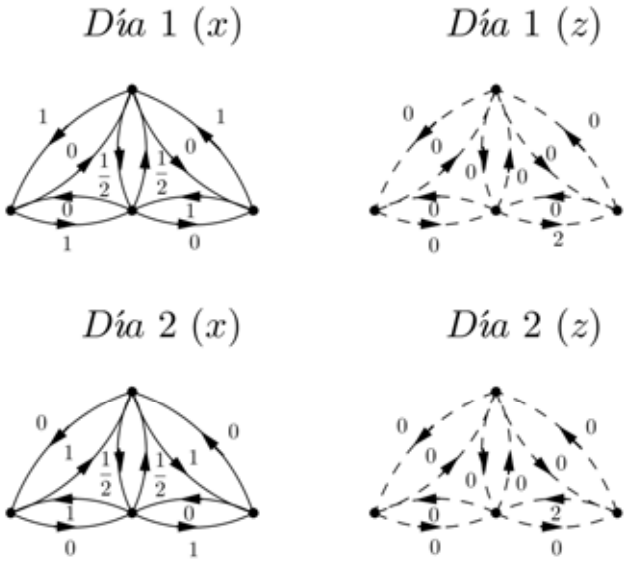


Figura 3.4: Ejemplo de punto extremo fraccionario.

archivo contiene los puntos extremos del poliedro y en este caso hay 582 puntos extremos fraccionarios. Un punto extremo fraccionario se ve como la siguiente solución:

Como hay puntos fraccionarios, es posible que la relajación lineal tenga una solución óptima que no es entera, debido a que tiene soluciones básicas factibles enteras y fraccionarias. Esta gráfica se guarda con las demás que no tienen esa propiedad.

Ahora se modifica el archivo con los puntos del poliedro dejando sólo los que tienen coordenadas enteras. Se envía el archivo modificado a PORTA para intentar obtener las restricciones que generarían el poliedro que se forma únicamente con los puntos enteros. En los casos en que PORTA logró calcular las nuevas desigualdades se estudiaron para tratar de caracterizarlas y aplicarlas a otras gráficas.

3.3. Programa elaborado

Para realizar el proceso de análisis de los poliedros de las gráficas con PORTA más rápido se elaboraron programas en C++ que automatizan varias tareas. Primero se elaboró un programa para generar un gran número de gráficas con el paquete de software *nauty* [11]. Posteriormente escribimos un programa que crea el archivo *.ieq* para un conjunto de gráficas. Cada archivo *.ieq* contiene todas las desigualdades del programa lineal correspondiente a una gráfica particular. Además, contiene una solución factible de ese programa lineal. Ese archivo se envía al paquete de software PORTA [4] para obtener como resultado un archivo *.poi* que contiene todos los puntos extremos del poliedro aso-

ciado al programa lineal de cada archivo *.ieq*. Finalmente el programa elaborado por nosotros revisa cada archivo *.poi* y revisa si el poliedro es entero o fraccionario y guarda registro de ello. Adicionalmente nuestro programa crea dos archivos con diagramas para cada gráfica en los que se muestra la solución factible enviada a PORTA y las variables asignadas a cada arista de la gráfica.

Después de este proceso, se intentó eliminar los puntos extremos de los poliedros fraccionarios en los archivos *.poi* para después enviar ese archivo de regreso a PORTA para obtener las desigualdades del programa lineal que correspondería a ese nuevo conjunto de restricciones. Desafortunadamente este proceso inverso tomó demasiado tiempo de cómputo y no se pudieron obtener resultados debido a la gran complejidad de este proceso en el SSP.

Por otra parte, se agregó un nuevo módulo al programa que agrega desigualdades de cortes impares a los modelos lineales de cada gráfica y vuelve a verificar si el poliedro resultante fue entero o fraccionario nuevamente.

El modelo de programación lineal que utiliza el programa es el modelo reducido de la sección 2.3.1, debido a que el modelo de la sección 2.3 que utiliza todas las variables sin sustitución hace que este proceso de análisis tarde demasiado tiempo computacional como para obtener los resultados de gráficas con más de 5 vértices.

En el apéndice A se muestra el código fuente de los programas mencionados.

3.4. Resultados

A continuación estudiaremos la estructura de $\mathcal{Q}(\vec{G})$ y demostraremos que la igualdad $\mathcal{P}(\vec{G}) = \mathcal{Q}(\vec{G})$ no siempre se cumple para el SSP. Se observó experimentalmente que hay casos en que puntos extremos de $\mathcal{Q}(\vec{G})$ tienen coordenadas con valores fraccionarios. Como sabemos que el poliedro $\mathcal{Q}(\vec{G})$ no siempre es entero, entonces investigamos si hay casos para los cuales $\mathcal{Q}(\vec{G})$ es entero.

De los resultados obtenidos proponemos la siguiente conjetura:

Conjetura 3.1. *$\mathcal{Q}(\vec{G})$ es semientero para cada gráfica G. En particular, los componentes asociados a variables x^1_{ij} y x^2_{ij} toman valores en $\{0, \frac{1}{2}, 1\}$ y los componentes asociados a variables z^1_{ij} y z^2_{ij} sólo toman valores enteros.*

Para la demostración sería necesario probar varias propiedades de los puntos extremos de los poliedros asociados a la relajación lineal.

Sea p un punto extremo de $\mathcal{Q}(\vec{G})$. Supongamos que p es fraccionario, entonces los componentes fraccionarios deben ser correspondientes a variables de barrido x^1_{ij}, x^2_{ij} o a variables de recorrido z^1_{ij}, z^2_{ij}, es decir, llamaremos arcos fraccionarios a esos arcos asociados a variables que toman valores fraccionarios en la solución p.

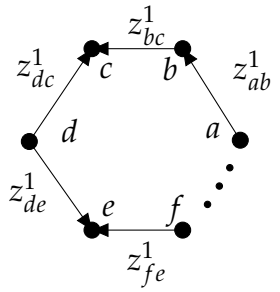


Figura 3.5: Ejemplo de ciclo C en la solución p.

Lema 3.1. *Cualquier punto extremo $p \in \mathcal{Q}(\vec{G})$ no tiene ciclos dirigidos formados únicamente con arcos fraccionarios correspondientes a variables z^1_{ij} o z^2_{ij}.*

Prueba. Con el objetivo de lograr una contradicción, supongamos que p es una solución factible correspondiente a un punto extremo de $\mathcal{Q}(\vec{G})$ y que además contiene un ciclo C formado únicamente con arcos fraccionarios asociados a variables de recorrido z^1_{ij} o z^2_{ij}.

La solución p está compuesta por dos recorridos disjuntos, uno para el día 1 y otro para el día 2. Podemos suponer sin pérdida de generalidad que C pertenece solamente a uno de los recorridos, es decir, que cada uno de los arcos que lo forman sólo pueden estar totalmente en el recorrido del día 1 o totalmente en el recorrido del día 2. Esto se sigue de las restricciones de conservación de flujo del modelo entero. En la figura 3.5 se puede ver un ejemplo de ciclo C formado únicamente por arcos fraccionarios correspondientes a variables z^1_{ij}.

Supongamos que el ciclo C se encuentra en el recorrido para el día 1. En este caso, podemos modificar el flujo enviado a través del ciclo siempre y cuando mantengamos las restricciones de conservación de flujo de PL. Adicionalmente, el cambio de flujo en C se puede lograr sin alterar el flujo en el recorrido del día 2, ya que los arcos correspondientes a variables z^1_{ij} no implican ningún flujo en los arcos z^2_{ij} como consecuencia de las restricciones del programa lineal.

Para modificar el flujo enviado por C, elegimos una orientación arbitraria de C y la llamamos positiva (fig. 3.6 *a*)). A la orientación opuesta de C la llamaremos negativa (fig. 3.6 *b*)). Decimos que un arco $a \in C$ es positivo si apunta en el mismo sentido que la orientación positiva. Por el contrario, decimos que un arco $a \in C$ es negativo si apunta en el mismo sentido que la orientación negativa. Nótese además, que los arcos que pertenecen al ciclo C no deben estar todos necesariamente con la misma orientación, ya que cuando los arcos apuntan en la orientación negativa se podría pensar que envían flujo negativo de la misma magnitud pero en la dirección opuesta. Entonces, definimos la siguiente partición de C como sigue:

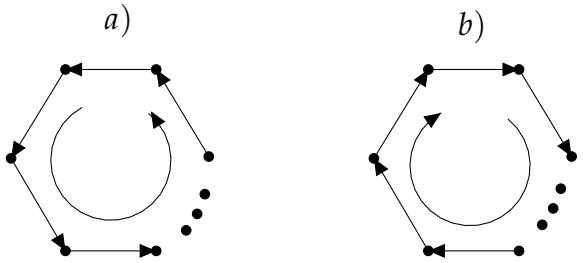


Figura 3.6: Orientaciones del ciclo C.

$$C^+ = \{a \in C \cap A : a \text{ es positivo}\},$$
$$C^- = \{a \in C \cap A : a \text{ es negativo}\}$$

Después, definimos como p_a, el valor que tiene el arco a en el punto extremo p. Con ello definimos lo siguiente:

$$\epsilon^+ = \text{mín } \{\lceil p_a \rceil - p_a, \forall a \in C^+\},$$
$$\epsilon^- = \text{mín } \{p_a - \lfloor p_a \rfloor, \forall a \in C^-\},$$
$$\epsilon = \text{mín } \{\epsilon^+, \epsilon^-\}$$

Dadas las restricciones de conservación de flujo podemos interpretar las variables como flujo factible en $\vec{G}$.

El valor ϵ^+ contempla cuánto flujo extra le falta a cada arco para enviar una cantidad de flujo entera y después toma el valor más pequeño de todos. El valor ϵ^- contempla cuánto flujo se puede quitar de cada arco de forma que cada arco envíe un flujo entero y después toma el valor más pequeño de todos ellos. Por consiguiente, la elección del valor de ϵ implica que $\epsilon > 0$ y es el valor que se elegirá para modificar el flujo de los arcos definiendo un nuevo punto factible p^1 de la forma siguiente:

$$p^1 = \begin{cases} p_a + \epsilon & \text{si } a \in C^+ \\ p_a - \epsilon & \text{si } a \in C^- \\ p_a & \text{si } a \notin C^- \text{ y } a \notin C^+ \end{cases}$$

Esto es equivalente a enviar ϵ unidades extra de flujo en la orientación positiva de C para formar un nuevo ciclo C^1. Se puede observar que $p^1 \in \mathcal{Q}(\vec{G})$, porque el balance de flujo en cada vértice $v \in C^1$ se mantiene en 0, es decir, la cantidad de flujo que entra y sale de cada vértice sigue siendo la misma durante el recorrido del día 1. El recorrido del día 2 se mantuvo intacto durante el proceso, entonces las restricciones para el recorrido del día 2 se siguen cumpliendo.

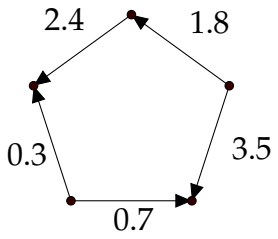


Figura 3.7: Ejemplo de ciclo C dentro del punto factible p.

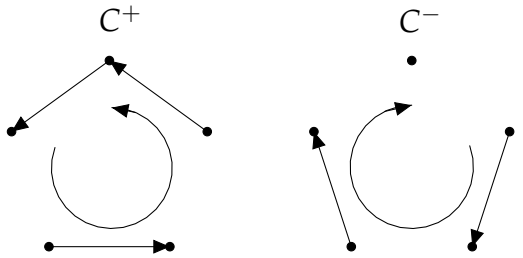


Figura 3.8: Orientaciones del ciclo C.

De la misma manera, podemos definir p^2 y un nuevo punto factible enviando las ϵ unidades de extra de flujo en la orientación negativa de C para formar el nuevo ciclo C^2:

$$p^2 = \begin{cases} p_a - \epsilon & \text{si } a \in C^+ \\ p_a + \epsilon & \text{si } a \in C^- \\ p_a & \text{si } a \notin C^- \text{ y } a \notin C^+ \end{cases}$$

Entonces, el punto p sería una combinación convexa de p^1 y p^2. En particular, $p = \frac{1}{2}(p^1 + p^2)$. El hecho de que p sea combinación convexa de otros puntos que están en $\mathcal{Q}(\vec{G})$ contradice que p sea punto extremo, por lo tanto, no pudo haber sido tal... □

Un ejemplo de la modificación de flujo de un ciclo C como fue definido para demostrar este lema es la siguiente:

Supongamos que dentro de p hay un ciclo C como en la figura 3.7.

Elegimos las orientaciones positivas y negativas como se muestra a continuación en la figura 3.8, para formar los conjuntos de aristas C^+ y C^-.

Calculamos los valores de ϵ^+ y ϵ^- utilizando los arcos positivos y negativos respectivamente:

$$\epsilon^+ = \text{mín}\,\{2-1.8, 3-2.4, 1-0.3, 1-0.7, 4-3.5\} = \text{mín}\,\{0.2, 0.6, 0.7, 0.3, 0.5\} = 0.2,$$
$$\epsilon^- = \text{mín}\,\{1.8-1, 2.4-2, 0.3-0, 0.7-0, 3.5-3\} = \text{mín}\,\{0.8, 0.4, 0.3, 0.7, 0.5\} = 0.3$$

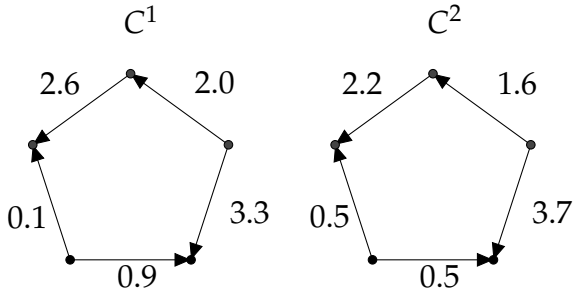


Figura 3.9: Nuevos ciclos C^1 y C^2 parte de p^1 y p^2 respectivamente.

$$\epsilon = \text{mín } \{0.2, 0.3\} = 0.2$$

Obteniendo el valor de ϵ calculamos unos nuevos ciclos para formar los nuevos puntos p^1 y p^2, como se muestra en la figura 3.9.

Continuando con la idea de la demostración de la conjetura 3.1, ahora podemos afirmar lo siguiente:

Lema 3.2. *Cualquier punto extremo $p \in \mathcal{Q}(\vec{G})$ no tiene ciclos dirigidos formados solamente por arcos fraccionarios correspondientes a variables x^1_{ij} o x^2_{ij}.*

Prueba. Supongamos que p es una solución factible que corresponde a un punto extremo y que además contiene un ciclo C formado únicamente por arcos fraccionarios de barrido asociados a variables x^1_{ij}. Entonces, este caso es diferente del lema anterior en que si un arco es fraccionario con una variable x^1_{ij}, entonces la variable x^1_{ji} también será fraccionaria y cualquier cambio a la variable x^1_{ij} inmediatamente refleja un cambio en la variable x^1_{ji}.

Adicionalmente, si un arco asociado a una variable x^1_{ij} es fraccionario, entonces el arco asociado a la variable x^2_{ij} también debe ser fraccionario, para poder cumplir las restricciones de PE. Además, cualquier cambio en la variable x^1_{ij} inmediatamente implica un cambio en el valor de la variable x^2_{ij}.

En resumen, un arco fraccionario asociado a una variable x^1_{ij} implica directamente que hay otros tres arcos fraccionarios correspondientes a las variables x^1_{ji}, x^2_{ij} y x^2_{ji}. Por todo lo anterior, si encontramos un conjunto de arcos fraccionarios que forman un ciclo C, podemos tomar los arcos complementarios que van en sentido opuesto y acompletar dos ciclos completos con orientaciones opuestas en el recorrido del día 1 (fig. 3.10. Además, se pueden formar otros dos ciclos fraccionarios en el recorrido del día 2 con las variables complementarias (fig. 3.10).

De nuevo tomamos los arcos que pertenecen al recorrido del día 1 y elegimos una orientación arbitraria del ciclo C y la llamamos positiva. La dirección opuesta la llama-

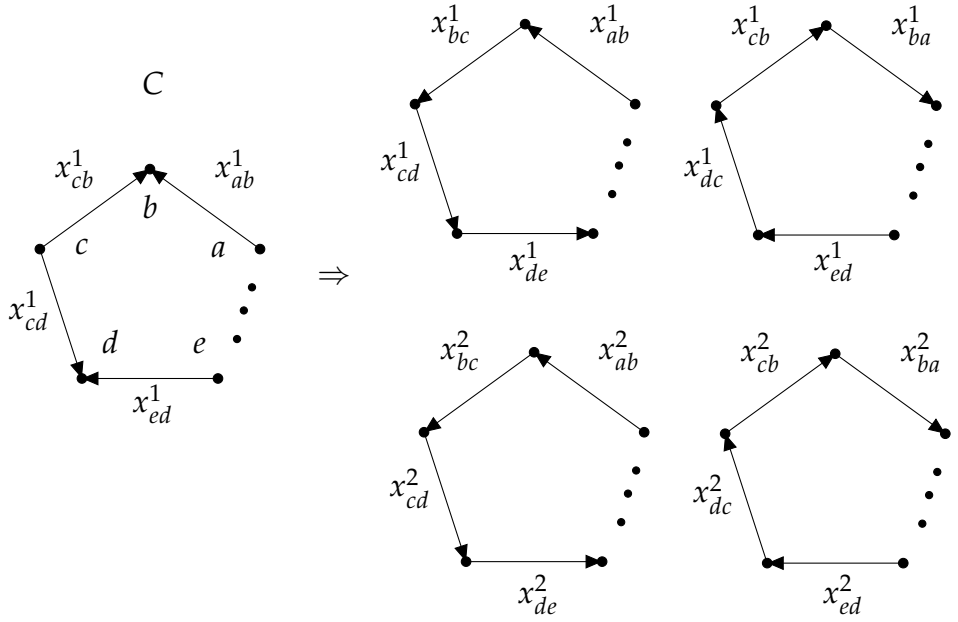


Figura 3.10: Ejemplo de ciclos fraccionarios a partir de un C de arcos x^1_{ij}.

mos negativa. Formamos el ciclo C^+ con todos los arcos fraccionarios que apuntan en la orientación positiva. De una manera similar, como cada arco fraccionario $x^1_{ij} \in C$ implica que hay otro arco fraccionario $x^1_{ji} \in C$ formamos otro ciclo C^- con todos los arcos fraccionarios que apuntan en la dirección negativa. Un ejemplo se muestra en la figura 3.11.

Nótese que, a diferencia del lema 3.1, ahora se pueden formar dos ciclos completos C^+ y C^-. En el caso anterior, los conjuntos C^+ y C^- no forman necesariamente dos ciclos, sino que los dos conjuntos de arcos juntos forman un solo ciclo.

A continuación definimos p^1_a como el valor que tiene el punto p asignado al arco a en el día 1. De manera similar llamaremos p^2_a al valor que toma el punto p en la coordenada correspondiente al arco a en el recorrido del día 2. Ahora, calculamos lo siguiente:

$$\begin{aligned} \epsilon^+ &= \text{mín } \{1 - p^1_a\}, \forall a \in C^+, \\ \epsilon^- &= \text{mín } \{p^1_a - 0\}, \forall a \in C^-, \\ \epsilon &= \text{mín } \{\epsilon^+, \epsilon^-\} \end{aligned}$$

Entonces al enviar ϵ unidades de flujo por C^+ simultáneamente restamos ϵ unidades de flujo de C^- en los arcos que pertenecen al recorrido del día 1. Para los arcos del día 2 hacemos lo opuesto, enviamos ϵ unidades de flujo por los arcos complementarios a los de C^- en el recorrido del día 2 y restamos ϵ unidades de flujo a los arcos complementarios a los arcos de C^+ en el día 2:

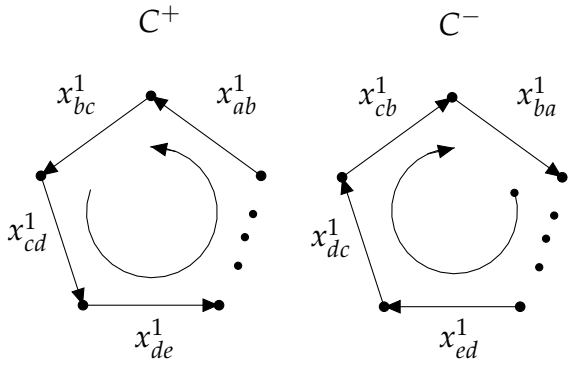


Figura 3.11: Ciclos C^+ y C^- de variables x^1_{ij} fraccionarias.

$$p_1 = \begin{cases} p^1_a + \epsilon & \text{si } a \in C^+ \\ p^1_a - \epsilon & \text{si } a \in C^- \\ p^2_a - \epsilon & \text{si } a \in C^+ \\ p^2_a + \epsilon & \text{si } a \in C^- \\ p^1_a & \text{si } a \notin C^- \text{ y } a \notin C^+ \\ p^2_a & \text{si } a \notin C^- \text{ y } a \notin C^+ \end{cases}$$

$$p_2 = \begin{cases} p^1_a - \epsilon & \text{si } a \in C^+ \\ p^1_a + \epsilon & \text{si } a \in C^- \\ p^2_a + \epsilon & \text{si } a \in C^+ \\ p^2_a - \epsilon & \text{si } a \in C^- \\ p^1_a & \text{si } a \notin C^- \text{ y } a \notin C^+ \\ p^2_a & \text{si } a \notin C^- \text{ y } a \notin C^+ \end{cases}$$

Una vez más el punto p es una combinación convexa de p^1 y p^2, siendo $p = \frac{1}{2}(p^1 + p^2)$. Esto presenta una contradicción con la elección de p, por lo tanto, p no pudo haber sido punto extremo si contenía ciclos fraccionarios formados úncamente con arcos asociados a variables x^1_{ij} o x^2_{ji}... $\square$

Cabe destacar, que en la demostración se eligió que el ciclo C estuviera formado por arcos fraccionarios pertenecientes al recorrido del día 1. Lo mismo aplicaría a arcos fraccionarios que pertenezcan al día 2. Las modificaciones realizadas al flujo de C para crear los nuevos puntos p^1 y p^2, mantienen el balance de flujos en todos los vértices durante los recorridos de ambos días.

3.5. Graficas con $\mathcal{Q}(\vec{G})$ entero

3.5.1. Gráficas eulerianas

Observando las pruebas realizadas a diferentes gráficas, encontramos que para cada gráfica euleriana que se probó $\mathcal{Q}(\vec{G})$ resultó ser entero. Incluso, no se encontró experimentalmente alguna gráfica no euleriana cuyo poliedro $\mathcal{Q}(\vec{G})$ fuera entero. Entonces se llegó la siguiente conjetura:

Conjetura 3.2. *$\mathcal{Q}(\vec{G})$ es entero si y sólo si la gráfica G es euleriana.*

3.5.2. Árboles

Podemos agregar desigualdades adicionales al programa entero original de la sección 2.3 para tratar de eliminar los puntos extremos fraccionarios y beneficiarnos de las ventajas de tener un poliedro entero para el modelo de programación lineal.

Para el caso de las gráficas que son árboles procedemos analizando primero las características de los poliedros asociados a esta clase de gráficas.

Teorema 3.1. *Si G es un árbol, el punto con $x^1_{ij} = x^2_{ij} = \frac{1}{2}\ \forall ij \in A$ y $z^1_{ij} = z^2_{ij} = 0\ \forall ij \in A$ es punto de $\mathcal{Q}(\vec{G})$.*

Demostración 1.

Por mera sustitución podemos revisar que tal punto cumple con las restricciones de conservación de flujo en cada vértice. Adicionalmente todas las restricciones de barrido se cumplen:

$x^1_{ij} + x^2_{ij} = 1$, $x^1_{ij} + x^1_{ji} = 1$, $x^2_{ij} + x^2_{ji} = 1$, $\forall ij \in A$

Entonces eliminamos todos los puntos extremos que contengan coordenadas fraccionarias de $\mathcal{Q}(\vec{G})$ agregando dos restricciones adicionales para cada arista e G:

$$z^1_{ij} + z^1_{ji} \geq 1, \qquad z^2_{ij} + z^2_{ji} \geq 1, \qquad \forall (i,j) \in E.$$

Definimos un nuevo poliedro $\mathcal{O}(\vec{G})$ asociado al nuevo programa lineal y concluimos lo siguiente:

Teorema 3.2. *Si G es un árbol, $\mathcal{P}(\vec{G}) = \mathcal{O}(\vec{G})$.*

Demostración 2.

En un árbol T cada arista es un *puente*. Esto significa que cada arista $(i,j) \in T$ es el único camino para ir desde el vértice i a j.

En una solución S, si tomamos un corte $i - j$ encontramos dos arcos opuestos. Para cada par de arcos asociados con la misma arista en $\vec{G}$, sólo uno de los arcos puede ser

barrido en el día 1. Para mantener la solución factible, debemos cruzar el mismo número de veces en cada dirección del corte. De lo contrario, el recorrido no regresaría del otro extremo de la gráfica. Entonces, forzamos que durante el recorrido del día 1 se recorra por lo menos un arco con las variables z^1_{ij} o z^1_{ji}. Entonces barremos un arco y recorremos otro, forzando que se cruce la arista puente en cada dirección una vez.

Nuevamente, dado que las variables z son enteras en cualquier punto extremo, forzamos que las variables de barrido se vuelvan enteras también evitando que valores fraccionarios de $\frac{1}{2}$ aparezcan.

Como el caso anterior es el único en el que se presentan coordenadas fraccionarias en $\mathcal{Q}(\vec{G})$, todos los puntos extremos resultantes en $\mathcal{O}(\vec{G})$ son enteros y concluimos que $\mathcal{P}(\vec{G}) = \mathcal{O}(\vec{G})$... $\square$

3.5.3. Cortes impares

Las desigualdades que se agregaron a los árboles se pueden generalizar como desigualdades de cortes impares. Primero consideramos cada corte impar de G con cada subconjunto $S \subseteq V$ y notamos que los recorridos del día 1 y 2 deben recorrer el mismo número de veces hacia cada dirección del corte, de lo contrario un recorrido se quedaría estancado de un lado del corte, en S o $\overline{S}$ y no se cerraría el circuito.

Entonces se agregan dos desigualdades adicionales a LP por cada corte impar en la gráfica G. Notemos que las variables de barrido del programa entero del SSP están asociadas a los arcos en $\vec{G}$, entonces para cada corte impar que haya en G agregaremos restricciones adicionales considerando los arcos del corte dirigido $\delta_A(S)$ en $\vec{G}$. Tomando un corte $\delta_E(S)$, restringimos que sólo se utlice una x^1_{ij} por cada arista en un corte impar, ya que sólo se barre un sentido de cada calle en un día particular. Entonces, sabemos que con las variables de barrido los recorridos para el día 1 y 2 atraviesan el corte $\delta_A(S)$ un número impar de veces, mientras que requerimos que se recorra un número par de veces, la mitad en una dirección y la otra de regreso. Entonces, por lo menos un arco asociado a variables z^1_{ij} y z^2_{ij} se debe utlizar para conseguir cruzar el corte un número par de veces.

Las desigualdes se ven de la siguiente manera para cada corte impar en G dado un conjunto de vértices $S \subseteq V$: $\sum_{(i,j)\in\delta_A(S)} z^1_{ij} \geq 1, \qquad \sum_{(i,j)\in\delta_A(S)} z^2_{ij} \geq 1$

Se probó agregar estas desigualdades de cortes impares a muchos ejemplos de gráficas y se llegó a esta conjetura:

Conjetura 3.3. *$\mathcal{O}(\vec{G})$ es entero si la gráfica G es plana exterior.*

Los indicios por los que se llegó a esto es que todos los ejemplos de gráficas planas exteriores obtivieron como resultado un poliedro entero, además de las gráficas eulerianas que ya obtenían un $\mathcal{Q}(\vec{G})$ entero sin agregar restricciones adicionales. Los árboles,

también están contenidos en la clase de gráficas planas exteriores. Fuera de esta clase de gráficas no hubo otros ejemplos que se encontraran que obtuvieran un $\mathcal{O}(\vec{G})$ entero.

Capítulo 4
Algoritmos exactos

En este capítulo se presentan algoritmos que resuelven de manera exacta el problema de barrido de calles. Sin embargo, para el caso general del problema el algoritmo con mejor tiempo de ejecución es exponencial.

Adicionalmente, a partir del estudio teórico y de la investigación experimental del SSP, encontramos algunas clases de gráficas para las cuales se puede resolver el problema en tiempo polinomial.

4.1. Algoritmo exhaustivo

Un algoritmo que resuelve el problema del SSP de manera exacta para cualquier tipo de gráfica G va de acuerdo a los siguientes razonamientos.

- Dado que hay que barrer un sentido de cada calle en el día 1 y el otro sentido en el día 2. Sólo hay dos posibilidades para cada una de las calles. Es decir, las variables de barrido de una arista que es adyacente a los vértices i, j toman los siguientes valores: $x^1_{ij} = 1$, $x^1_{ji} = 0$, $x^2_{ij} = 0$, $x^2_{ji} = 1$, de lo contrario toman los siguientes valores: $x^1_{ij} = 0$, $x^1_{ji} = 1$, $x^2_{ij} = 1$, $x^2_{ji} = 0$.
- Como hay exactamente dos posibilidades para cada arista, entonces podemos probar las $2^{|E|}$ combinaciones posibles.
- Para cada combinación, se puede encontrar una manera de conectar cada uno de los arcos elegidos para barrer en el recorrido del día 1 y hacer lo mismo para el día 2.
- Utilizando algoritmos de flujo podemos hallar una forma de cerrar los recorridos. Entonces, para cada combinación de barrido, resolvemos dos problemas de flujo utilizando alguno de los algoritmos conocidos de tiempo polinomial [1].

Además de poder implementar algoritmos para resolver el caso general del SSP, descubrimos que se puede resolver en tiempo polinomial para algunas clases de gráficas. Por ello, proponemos el siguiente teorema:

Teorema 4.1. *Hay un algoritmo exacto de tiempo lineal para el SPP cuando G es una gráfica euleriana o un árbol.*

La demostración es consecuencia directa de la aplicación de los algoritmos presentados a continuación. Para demostrar esto, presentamos a continuación los algoritmos para gráficas eulerianas y árboles.

4.1.1. Gráficas eulerianas

- Primero tomamos la gráfica G y encontramos un circuito euleriano utilizando alguno de los algoritmos conocidos para esto, tal como el algoritmo de Hierholzer [9], que nos permite encontrarlo en tiempo lineal.
- Elegimos arbitrariamente una dirección del circuito euleriano y lo usamos como secuencia para hacer el barrido del día 1. Cada vez que se recorre una arista en una dirección particular se marca para barrer con las variables x_{ij}^1. Como cada arista se recorre exactamente una vez, se cumple con la restricción de que cada calle se barra en exactamente un sentido durante el día 1.
- Para el recorrido del día 2 seguimos de nuevo la misma sequencia del circuito euleriano, pero en sentido opuesto al utilizado en el día 1.
- Hay que notar que al crear esta solución para el día 1 y 2, todas las variables z_{ij}^1 y z_{ij}^2 permanecen con un valor de 0, ya que no se recorre ninguna arista más de una vez durante el recorrido del día 1, ni durante el recorrido del día 2.
- El tiempo total de ejecución es polinomial. En particular, detectar si la gráfica G es euleriana toma tiempo $O(|V| + |E|)$, ya que hay que revisar que el grado de cada vértice sea par. El algoritmo utilizado para encontrar el circuito euleriano de Hierholzer toma tiempo lineal $O(|V| + |E|)$. Finalmente, asignar los valores a los vectores de adyacencia toma tiempo $O(|E|)$ ya que se asignan valores a 8 variables por cada arista de G. Como estas acciones se realizan en secuencia, una después de otra, entonces el tiempo total de ejecución permanece siendo lineal $O(|V| + |E|)$.

La solución encontrada por el algoritmo anterior, no sólo es factible, sino que además es óptima en todos los casos que G es euleriana. La razón es que, nunca recorre las calles más de una vez en el día 1 y una vez en el día 2. Las restricciones del problema fuerzan a que se recorran por lo menos una vez cada día. Entonces, en la solución se recorre cada calle el mínimo número de veces posible, sin recurrrir a tener costos extra en los recorridos durante ambos días. Por lo tanto, no hay mejores soluciones que ésta.

A continuación se muestra un ejemplo del proceso anterior. Primero comenzamos con una gráfica G, como se muestra en la figura 4.1. Posteriormente, revisamos el grado de cada vértice y vemos que los vértices tienen los siguientes valores:

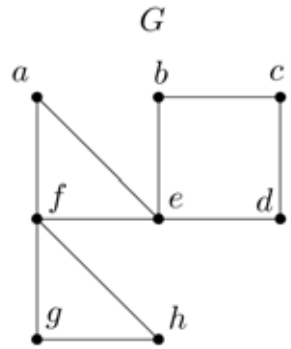


Figura 4.1: Ejemplo de gráfica euleriana G.

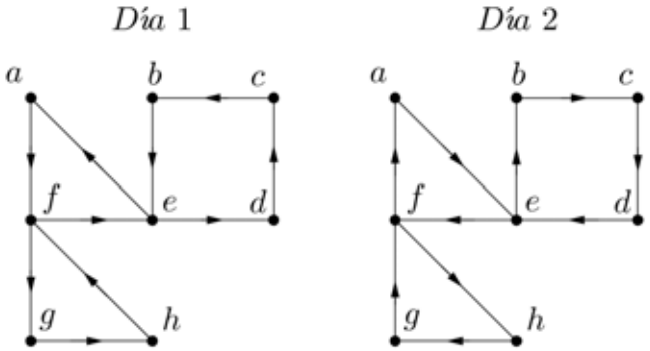


Figura 4.2: Ejemplos de recorridos.

$$a = 2, b = 2, c = 4, d = 2, e = 2, f = 4, g = 2, h = 2$$

Entonces como cada vértice tiene grado par, podemos encontrar un circuito euleriano como el siguiente utilizando el algoritmo de Hierholzer:

$$\{(a,f),(f,g),(g,h),(h,f),(f,e),(e,d),(d,c),(c,b),(b,e),(e,a)\}$$

Creamos el recorrido del día 1 siguiendo la misma secuencia del recorrido euleriano tal como se muestra en la figura 4.2. Dado que sólo se pasa una vez por cada arista durante el día 1, entonces se elige la dirección utilizada para barrer esa calle (arco). De manera similar marcamos los arcos que se barren en el día 2, pero utilizando el sentido opuesto de la secuencia del recorrido euleriano. En la solución no se recorre ningún arco sin barrer, por lo tanto no puede haber una solución más barata que esta.

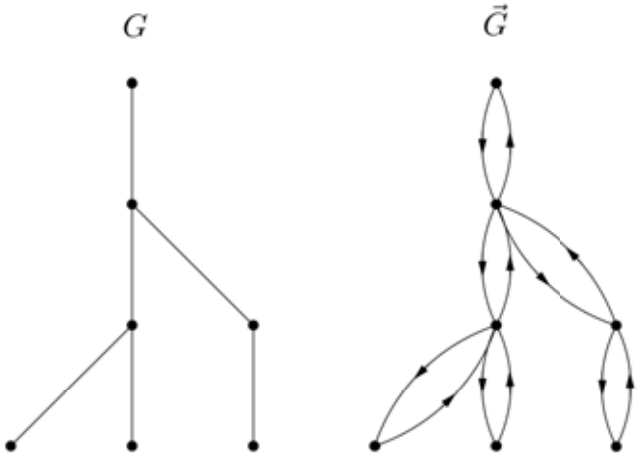


Figura 4.3: Ejemplo de árbol G y su $\vec{G}$ correspondiente.

4.1.2. Árboles

Para el caso de los árboles, encontramos una vez más un circuito euleriano, pero esta vez en la gráfica dirigida $\vec{G}$. Una vez más se utiliza el recorrido euleriano para crear los recorridos del día 1 y 2. Sin embargo, para recorrer cada arco de un árbol $\vec{G}$ se necesita pasar dos veces por cada arista de G, una vez en cada dirección. En una de las direcciones se barre (x_{ij}^1) y en la otra sólo se recorre (z_{ij}^1). Esto aplica para el día 1 y 2.

Para elegir cuál dirección de cada arista se barre en el día 1 y cuál sólo se recorre enumeramos todos los vértices: $1, 2, \ldots, |V|$. En el día 1 barremos cada arco (i, j) del circuito euleriano en el cual $i < j$ y recorremos sin barrer el resto de los arcos. En el día 2, seguimos la misma secuencia del día 1 en la misma dirección.

El tiempo de ejecución de este algoritmo es lineal también.

A continuación mostramos un ejemplo del algoritmo seguido para los árboles. Comenzamos un árbol G que se transforma en la gráfica $\vec{G}$ como en la figura 4.3. Posteriormente, numeramos todos los vértices como se muestra en la figura 4.4.

Encontramos un circuito euleriano en G tal como el siguiente:

$$\{(1,2),(2,3),(3,5),(5,3),(3,6),(6,3),(3,2),(2,4),(4,7),(7,4),(4,2),(2,1)\}$$

Creamos el recorrido de barrido para el día 1 y para el día 2 siguiendo la misma secuencia del circuito euleriano, pero intercambiando los sentidos de las calles que se barren en el día 1 y 2. Las calles (arcos) que se barren se muestran en la figura 4.5, indicando los arcos barridos con línea continua y los arcos que sólo se recorren sin barrer con líneas punteadas.

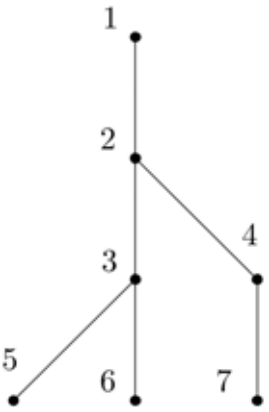


Figura 4.4: Ejemplo de numeración de vértices.

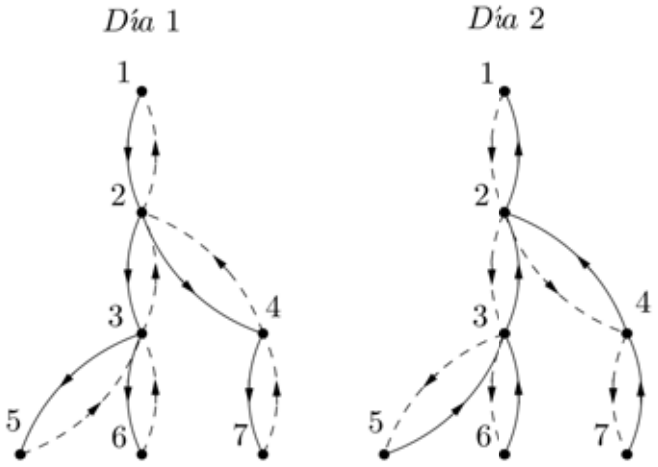


Figura 4.5: Ejemplo de arcos elegidos para barrer el día 1 y 2.

4.2. Gráficas con conservación de costo en ciclos

Win demostró en [16] que el WPP es soluble en tiempo polinomial para el caso de las gráficas que cumplen con la siguiente condición: Para cada ciclo C en las subgráficas de G, el costo de recorrer C en una dirección es igual al costo de recorrer C en la dirección opuesta.

Si es posible resolver el WPP en tiempo polinomial para este tipo de gráficas, entonces también se puede resolver el SPP en tiempo polinomial. Lo que se haría es crear el recorrido del día 1, es tomar directamente el recorrido empleado para el WPP, que es óptimo. Para el día 2, se utiliza el mismo recorrido pero en dirección opuesta, para barrer los sentidos opuestos de las calles. Como cada ciclo C cuesta lo mismo en ambas direcciones, entonces, la solución para el día 2 es óptima también para el WPP, y por lo tanto también para el SPP.

Capítulo 5
Algoritmos de aproximación

En este capítulo presentamos varios algoritmos de aproximación para el *SSP*. Algunos de ellos utilizan como subrutinas algoritmos de aproximación para el *WPP*. Obtuvimos diferentes garantías para diferentes tipos de gráficas, así como algoritmos para el caso general con cualquier gráfica.

Una idea instintiva sería usar el recorrido inverso.

5.1. Algoritmo de aproximación con garantía $\frac{3}{2}\alpha + 1$

La idea principal es encontrar un *recorrido de cartero con viento* para el día 1 utilizando un algoritmo de aproximación para el *WPP* para luego utilizar ese recorrido para crear la ruta del día 2. Este procedimiento resulta en un algoritmo de aproximación para el *SSP* con una garantía de $\beta = \frac{3}{2}\alpha + 1$, donde α es la garantía de aproximación del algoritmo para el *WPP* [1].

En cualquier algoritmo de aproximación del WPP tenemos la siguiente entrada y salida:

- Entrada: $G = (V, E)$, función de costos $c = (c_{ij}, c_{ji})$.
- Salida: Un recorrido de cartero con viento T' representado por un vector de incidencia x.

El vector x se puede transformar en una multigráfica dirigida $D = (V, A)$, en la que cada ocurrencia de una variable $x_{ij} \in x$ se transforma en x_{ij} arcos $(i, j) \in A$. La gráfica resultante D es euleriana dirigida, ya que en cada vértice $deg^+(v) = deg^-(v)$ para representar la circulación sobre D.

Por otro lado, una solución para el *SSP* se puede pensar como la unión de dos recorridos de cartero dirigidos T_1 y T_2, donde T_1 es el recorrido para el día 1 y T_2 es el recorrido para el día 2. La mitad de los arcos se barren en T_1 y la otra mitad en T_2. En el *SSP* los recorridos T_1 y T_2 son representados por cuatro vectores de incidencia x^1, x^2, z^1, z^2.

[1] Los resultados de esta sección fueron publicados en [13].

Encontrar T_1 se puede pensar como resolver un *WPP* en la misma gráfica G, porque queremos barrer cada calle (arista) en sólo una dirección durante el día 1. La condición es que se haga un recorrido que pase por cada arista por lo menos una vez en cualquier dirección, entonces se puede utilizar una solución para el *WPP* llamada T'. En T_1 podemos marcar para barrer cada arista en la dirección en que cada arista (i, j) fue recorrida en T'. En caso de que ambos sentidos de la arista (i, j) sean recorridos en T', se elige arbitrariamente una dirección (i, j) ó (j, i).

Para el día 2, intentamos seguir el mismo recorrido que en el día 1, pero barriendo los arcos opuestos. T_1 está compuesto de arcos correspondientes a variables que forman una secuencia como la siguiente $(v_0, e_1, v_1, ..., e_n, v_n)$ con $v_0 = v_n$. Cada arco $e_k \in T_1$ corresponde ya sea al vector x^1 o al vector z^1, en otras palabras, cada vez que el tour atraviesa un arco, se hace barriendo o no. En T_2 se debe marcar para barrer cualquier arco no barrido en el día 1. Entonces construimos T_2 a partir de T_1 como se muestra en la figura 5.1.

Si un arco e_k corresponde a un x^1_{ij} en la dirección en que se barrió, entonces en T_2 atravesamos e_k sin barrer en la misma dirección en x^2_{ij}. Después se marca para barrer el arco x^2_{ji} correspondiente al sentido opuesto en que la arista e_k se barrió en el día 1. Finalmente se vuelve a recorrer sin barrer la arista e_k en la misma dirección en que se barrió en T_1. En resumen, cada arco de barrido $x^1_{ij} \in T_1$ se reemplaza por tres arcos en T_2, para poder seguir un recorrido similar al de T_1, pero barriendo los sentidos opuestos de las aristas.

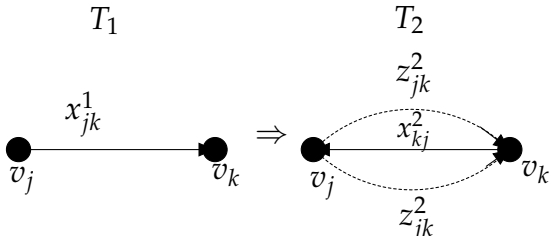


Figura 5.1: Cada arco de barrido en T_1 se reemplaza por tres arcos en T_2.

Si una arista e_k se recorrió de forma que corresponde a un arco z^1 entonces todas las ocurrencias de ese arco z^1 se copian directamente al vector de incidencia z^2.

El algoritmo de aproximación propuesto para el *SSP* es el siguiente:

1. Construir una instancia auxiliar I del *WPP* dándole como entrada la gráfica original G y la misma función de costos $c = (c_{ij}, c_{ji})$.

2. Encontrar una solución aproximada T' para la instancia I con un algoritmo de aproximación para el *WPP* que tenga garantía α.

3. Utilizar el recorrido T' como recorrido T_1 para el día 1 de la solución del SSP.

4. Enumerar todos los vértices: 1,2, ..., $|v|$.

5. Crear los vectores de incidencia x^1 y z^1. Para cada arista $(i,j) \in E$:

   - Si $x_{ij} > 0$ y $x_{ji} > 0$ entonces
     - Si $i < j$ entonces $x^1_{ij} = 1$, $x^1_{ji} = 0$, $z^1_{ij} = x_{ij} - 1$, $z^1_{ji} = x_{ji}$.
     - De lo contrario, si $i > j$ entonces $x^1_{ij} = 0$, $x^1_{ji} = 1$, $z^1_{ij} = x_{ij}$, $z^1_{ji} = x_{ji} - 1$.
   - Si $x_{ij} > 0$ y $x_{ji} = 0$ entonces $x^1_{ij} = 1$, $x^1_{ji} = 0$, $z^1_{ij} = x_{ij} - 1$, $z^1_{ji} = 0$.
   - Si $x_{ij} = 0$ y $x_{ji} > 0$ entonces $x^1_{ij} = 0$, $x^1_{ji} = 1$, $z^1_{ij} = 0$, $z^1_{ji} = x_{ij} - 1$.

6. Crear el vector de incidencia x^2 marcando todos los arcos faltantes para que sean barridos durante el día 2. Para cada arista $(i,j) \in E$:

   - Si $x^1_{ij} = 1$ entonces $x^2_{ij} = 0$ y $x^2_{ji} = 1$.
   - De lo contrario, si $x^1_{ij} = 0$ entonces $x^2_{ij} = 1$ y $x^2_{ji} = 0$.

7. Crear el vector de incidencia z^2. Para cada arista $(i,j) \in E$:

   - Hacer el valor de z^2_{ij} igual a z^1_{ij}.
   - Hacer el valor de z^2_{ji} igual a z^1_{ji}.
   - Si $x^1_{ij} = 1$ entonces agregar dos unidades a la variable z^2_{ij}.
   - Si $x^1_{ji} = 1$ entonces agregar dos unidades a la variable z^2_{ji}.

8. Una vez que tenemos los cuatro vectores de incidencia obtenemos dos multigráficas dirigidas $D_1 = (V, A_1)$ y $D_2 = (V, A_2)$ las cuales resultan ser eulerianas. Por cada componente x^1_{ij}, agregamos un arco desde i a j al conjunto A_1 por cada unidad de x^1_{ij}. Procedemos de una manera similar con z^1_{ij} y A_1.

9. Aplicar el mismo procedimiento a x^2_{ij}, z^2_{ij} y A_2.

10. Calculamos un circuito euleriano dirigido en D_1 y otro sobre D_2, para obtener la secuencia con la que se visitan todos los arcos en S.

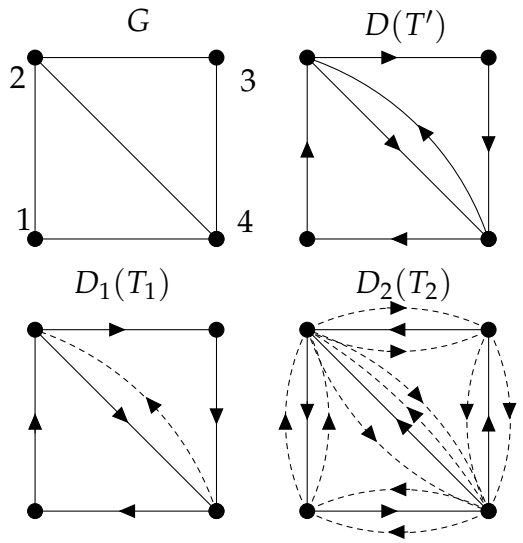


Figura 5.2: Una gráfica G con la solución de ejemplo formada por D_1 y D_2.

5.1.1. Ejemplo

Dada la gráfica G en la figura 5.2 calculamos una solución aproximada para el WPP resultando en un vector de incidencia $x = \{1, 0, 1, 1, 1, 0, 1, 0, 1, 0\}$ el cual indica el número de veces que se recorren las aristas en cada dirección. En este ejemplo el orden de los arcos en el vector x es $A = \{(1, 2), (1, 4), (2, 3), (2, 4), (3, 4), (2, 1), (4, 1), (3, 2), (4, 2), (4, 3)\}$.

Con x obtenemos la gráfica D en la cual podemos calcular un circuito euleriano para obtener la secuencia en la cual las aristas deben ser recorridas en la solución T'. Utilizando el vector x obtenemos los vectores de incidencia para el día 1: $x^1 = \{1, 0, 1, 1, 1, 0, 1, 0, 0, 0\}$ y $z^1 = \{0, 0, 0, 0, 0, 0, 0, 0, 1, 0\}$. Con x^1 y z^1 obtenemos D_1, entonces calculamos un circuito euleriano para definir la secuencia para el día 1. Entonces los vectores de incidencia para el día 2: x^2 y z^2 se calculan utilizando los vectores del día 1. En este caso $x^2 = \{0, 1, 0, 0, 0, 1, 0, 1, 1, 1\}$ y $z^2 = \{2, 0, 2, 2, 2, 0, 2, 0, 1, 0\}$.

5.1.2. Garantía de aproximación $\frac{3}{2}\alpha + 1$

Habiendo presentado el algoritmo de aproximación podemos probar lo siguiente:

Teorema 5.1. *Sea α la garantía de un algoritmo de aproximación para el WPP. Hay un algoritmo de aproximación para el SPP con garantía $\beta = \frac{3}{2}\alpha + 1$.*

Prueba: Sea S una solución factible para el SSP, con costo $c(S)$, obtenida con el algoritmo de aproximación. Sea S^* una solución óptima con costo $c(S^*)$. Sean $c(T_1)$ y $c(T_2)$ los costos de recorrido para el día 1 y 2, respectivamente. Entonces $c(S) = c(T_1) + c(T_2)$.

También, sea $c(x^1)$ y $c(x^2)$ la suma de todos los costos de los arcos que son barridos el día 1 y 2, respectivamente. Sea W una solución α-aproximada para el WPP en G con costo $c(W)$. Sea W^* la solución óptima para el WPP en G con costo total $c(W^*)$. Por consiguiente, $c(W^*) \geq \frac{c(W)}{\alpha}$.

Tenemos que el costo total $c(S) \leq 3c(W) + c(x^2)$. El primer término, $3c(W)$ es porque S recorre cada arco en W a lo más tres veces, una vez como parte de T_1 y como máximo dos veces más como parte de T_2, para hacer que T_2 se mantenga en la secuencia de aristas de T_1. El segundo término $c(x^2)$ es porque se suma el costo de barrer los arcos faltantes en el día 2. El costo de barrer la mitad de los arcos en T_1 se contabiliza en el término $3c(W)$ ya que se sigue una vez $c(W)$ barriendo la mitad de todos los arcos de $\vec{G}$. Por otra parte $c(S^*) \geq 2c(W^*)$, porque en el mejor de los casos se necesita por lo menos dos veces el costo de la solución óptima del WPP, para completar los recorridos del día 1 y 2. Entonces se concluye que $c(S^*) \geq \frac{2}{\alpha}c(W)$, esto es, $\frac{\alpha}{2}c(S^*) \geq c(W)$.

Posteriormente, $c(x^2) \leq c(S^*)$ debido a que el costo total sea igual al costo total de barrer los arcos elegidos para el día 2. Finalmente $c(S) \leq 3c(W) + c(x^2) \leq \frac{3}{2}\alpha c(S^*) + c(S^*) = (\frac{3}{2}\alpha + 1)c(S^*)$.

Teorema 5.2. *Hay un algoritmo de aproximación de garantía $\beta = 3.25$ para el SSP con cualquier gráfica.*

Demostración: Utilizando el algoritmo de aproximación con garantía $\frac{3}{2}$ para el WPP obtenemos $\beta = \frac{3}{2}\alpha + 1 = \frac{3}{2}(\frac{3}{2}) + 1 = 3.25$.

5.1.3. Gráficas serie-paralelo

Una gráfica G es *serie-paralelo* si G no tiene a K_4 como menor. De acuerdo con [17] el WPP puede ser resuelto en tiempo polinomial para las gráficas serie-paralelo.

Teorema 5.3. *Hay un algoritmo de aproximación con garantía $\beta = 2.5$ para el SPP en gráficas serie-paralelo.*

Demostración: Usando un algoritmo exacto para el WPP tenemos que $\alpha = 1$, entonces tenemos que $\beta = \frac{3}{2}\alpha + 1 = \frac{3}{2}(1) + 1 = 2.5$.

5.2. Algoritmo de aproximación con garantía 2

Como sabemos, en el caso general del SSP todas las calles tienen ambos sentidos. A partir de eso, definimos la gráfica $\vec{G}$, reemplazando cada arista $e \in E$ por dos arcos opuestos e^+ y e^-. Para cualquier vértice $v \in V$, cada vez que se crea un arco que entra a v, también se crea un arco que sale de v. Como cada uno de los vértices cumple que $deg^+(v) = deg^-(v)$, entonces $\vec{G}$ es una gráfica dirigida euleriana. Utilizando este conocimiento podemos definir el siguiente algoritmo:

- Obtener un circuito euleriano dirigido T_e en $\vec{G}$, en el cual cada arco se utiliza exactamente una vez.
- Enumerar todos los vértices: 1, 2,...$|v|$.
- Crear el recorrido para el día 1 (T_1) haciendo lo siguiente para cada par de arcos e^+, e^- que están asociados a una misma arista $e \in E$:
  - Elegir el arco e^+ o e^- que apunta desde el vértice i al j tal que $i < j$ en la numeración dada anteriormente.
  - Asignar el valor de 1 a la variable correspondiente x^1_{ij}.
  - Asignar el valor de 0 a la variable correspondiente x^1_{ji} del arco opuesto.
  - Asignar el valor de 0 a z^1_{ij}.
  - Asignar el valor de 1 a z^1_{ji}.
- Crear el recorrido para el día 2 (T_2) haciendo lo siguiente para cada par de arcos e^+, e^- que están asociados a una misma arista $e \in E$:
  - Elegir el arco e^+ o e^- que apunta desde el vértice i al vértice j tal que $i > j$ en la numeración:
  - Asignar el valor de 1 a x^2_{ij}.
  - Asignar el valor de 0 a x^2_{ji}.
  - Asignar el valor de 0 a z^2_{ij}.
  - Asignar el valor de 1 a z^2_{ji}.

La solución completa $S = T_1 + T_2$, sigue en ambos días la secuencia dada por T_e utilizando cada arco $a \in A$ cada día, pero barriendo la mitad de los arcos en el día 1 y la otra mitad en el día 2.

5.2.1. Ejemplo

Primero tomamos la gráfica $G = (V, E)$ y la transformamos en la gráfica dirigida $\vec{G} = (V, A)$ que se muestran en la figura 5.3. Posteriormente numeramos los vértices de la gráfica como se muestra en la figura 5.4. Después encontramos un circuito euleriano dirigido en $\vec{G}$ tal como el siguiente:

$$\{(2,5),(5,4),(4,1),(1,2),(2,1),(1,4),(4,5),(5,2),(2,3),(3,4),(4,3),(3,2)\}$$

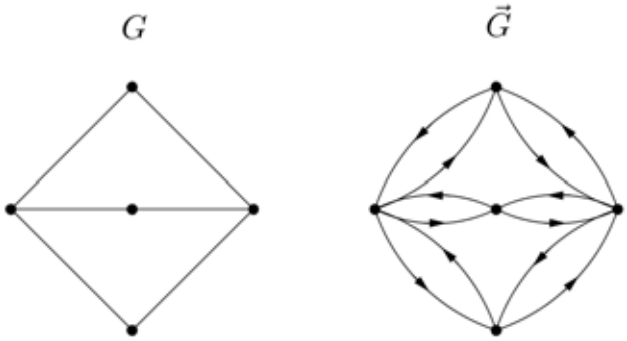


Figura 5.3: Ejemplo de gráfica G y su transformación $\vec{G}$.

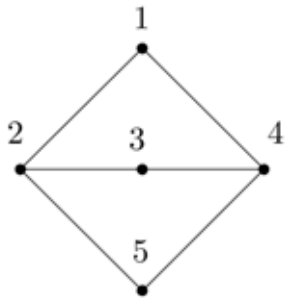


Figura 5.4: Ejemplo de numeración de vértices en G.

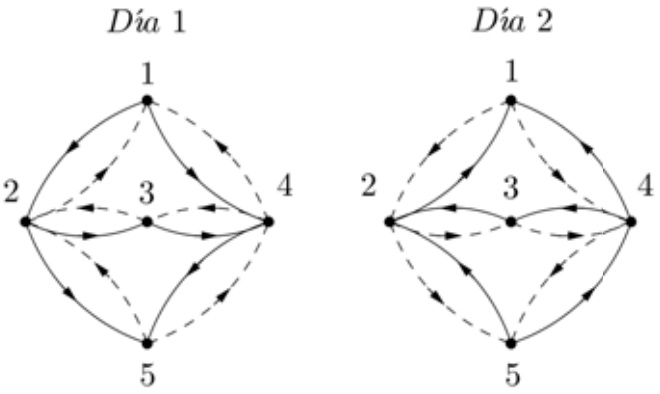


Figura 5.5: Ejemplo de barrido.

Marcamos para barrer el día 1 los arcos (i, j) que pertenecen al circuito euleriano y que cumplen que $i < j$, el resto se marcan para recorrer sin barrer. Para el día 2 se marcan para barrer los arcos opuestos. Los arcos de los días 1 y 2 se muestran en la figura 5.5.

La solución aproximada vale $\sum 2c_{ij}$, $\forall (i, j) \in A$. Como la solución óptima es igual o mayor a $\sum c_{ij}$, $\forall (i, j) \in A$, entonces nuestra solución es a lo más 2 veces el costo de la óptima.

5.2.2. Garantía 2

Sabemos que $c(S^*) \geq c(A)$. Recordamos que el costo de la solución $c(S) = c(T_1) + c(T_2)$. Utilizando este algoritmo de aproximación recorremos cada arco durante el día 1 y cada arco durante el día 2, para que sea posible hacer un recorrido euleriano. Entonces, $c(T_1) = c(A)$ y $c(T_2) = c(A)$. Por lo tanto $c(S) = 2c(A)$ y $\frac{c(S)}{2} = c(A)$.

Sustituyendo en $c(S*) \geq c(A)$ obtenemos $c(S*) \geq \frac{c(S)}{2}$. Escrito de otra manera $2c(S*) \geq c(S)$, demostrando que el algoritmo da una garantía de a lo más 2 veces el óptimo.

Capítulo 6
Conclusiones y trabajo futuro

En este trabajo, analizamos el problema de limpieza periódica con un enfoque de programación lineal y entera estudiando sus poliedros asociados. Con base en lo anterior diseñamos algoritmos exactos y de aproximación, avanzando mucho en los resultados que se habían obtenido para el *SSP*.

Para estudiar la relajación lineal del *SSP* se elaboró un programa que analiza la integralidad de los poliedros asociados a gráficas con las siguientes características:

- Recibe como entrada una lista de gráficas.
- Genera el programa lineal asociado a cada gráfica.
- Envía el modelo a PORTA para obtener el poliedro asociado $\mathcal{Q}(G)$.
- Revisa los puntos extremos para encontrar puntos fraccionarios.
- Separa en dos categorías los poliedros enteros y los fraccionarios.

Además elaboramos un programa para generar y agregar desigualdades adicionales al modelo para probar la integralidad con planos de corte adicionales, tal como las desigualdades de cortes impares.

Explorando la estructura de los puntos extremos de $\mathcal{Q}(G)$ encontramos lo siguiente:

- Demostramos que los puntos extremos del poliedro asociado a los árboles sólo tienen coordenadas enteras o $\frac{1}{2}$.
- Propusimos desigualdades adicionales que eliminan todos los puntos extremos fraccionarios para los árboles.

Adicionalmente conjeturamos que:

- Para toda gráfica G, cada punto extremo x del poliedro $\mathcal{Q}(\vec{G})$ tiene coordenadas cuyos valores son $\frac{1}{2}$ o un entero no negativo.
- Si G es euleriana entonces $\mathcal{Q}(\vec{G})$ es entero.

En cuanto a algoritmos logramos concretar lo siguiente:

- Se demostró que hay un algoritmo de aproximación con garantía $\frac{3}{2}\alpha + 1$ para el caso general del SSP con α la garantía de un algoritmo de approximación para el WPP.
- Se diseñó e implementó un algoritmo de aproximación con garantía 3.25 para el caso general del SSP, siendo el primer algoritmo de aproximación con garantía presentado para el *SSP*.
- Se diseñó e implementó un algoritmo de aproximación con garantía 2.5 para el caso de las gráficas eulerianas, árboles y serie-paralelo.
- Posteriormente se diseñó e implementó un algoritmo de aproximación factor 2 para el caso general del SSP.
- Diseñamos e implementamos un algoritmo exponencial para resolver el SSP en el caso general basandose en algoritmos de flujo.
- Diseñamos e implementamos un algoritmo polinomial para resolver el SSP para los siguientes tipos de gráficas:
  - Eulerianas.
  - Árboles.
  - Las que cada ciclo cuesta lo mismo en ambas direcciones.

Como posible trabajo futuro queda abierta la posibilidad de trabajar en los siguientes aspectos:

- Encontrar la complejidad computacional del *SSP*.
- Encontrar otros casos en los que el *SSP* es resoluble en tiempo polinomial.
- Verificar si los algoritmos de aproximación propuestos tienen una garantía más estricta.
- Encontrar otros algoritmos de aproximación con una mejor garantía.
- Debido a la manera en que se representan las soluciones del SSP, se podrían crear algoritmos genéticos para resolver el SSP. Por ejemplo, es posible hacer un algoritmo como el siguiente:
  - Se toman como cromosomas de los individuos a los vectores x_{ij} de las variables en el modelo de programación lineal reducido.
  - Generamos aleatoriamente la población inicial.

- Para evaluar la aptitud de los individuos que se tienen en la población actual, podemos resolver un problema de flujo de costo mínimo para encontrar la solución óptima de hacer el recorrido para un vector de variables x_{ij}. Esto toma tiempo polinomial.
- Se selecciona a los cromosomas con mejor aptitud y se cruzan para crear una nueva generación.

Como se ve, la representación de soluciones con el x_{ij} facilita la implementación de los algoritmos genéticos, debido a que cada entrada del vector tiene valores 0, 1. Y cada asignación de valores al vector x_{ij} tiene un valor óptimo de recorrido que se puede usar como aptitud.

Apéndice A
Programas

A.1. Generador de gráficas

Este programa genera un archivo de texto con una gráfica en cada renglón siguiendo el formato g6. Se utiliza como generador de gráficas al paquete de software *nauty* [11].

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>
#include <sstream>

using namespace std;

namespace patch{
 template < typename T > std::string to_string( const T& n){
  std::ostringstream stm;
  stm << n;
  return stm.str();
 }
}

int main(){

 string comando = "";

 for(int I = 3; I <= 9; I++){
  comando = "./geng -c " + patch::to_string(I) + " >> listaNueva.txt";
  system(comando.c_str());
 }

 return 0;
}
```

A.2. Clasificador

Los siguientes archivos forman el programa utilizado para clasificar las gráficas en enteras y no enteras. Primero toman una lista de gráficas y se generan sus desigualdades correspondientes a su programa lineal. Posteriormente se envían a PORTA [4] para obtener el poliedro asociado a las restricciones. Finalmente se analiza si son enteros o fraccionarios y se guardan las gráficas en la carpeta adecuada.

Adicionalmente agregan desigualdades de cortes impares para verificar si el poliedro se vuelve entero.

A.2.1. Archivo: clasificador.cpp

```
#include <iostream>
#include <cmath>
#include <cstdlib>
#include <sys/types.h>
#include <unistd.h>

#include "graficas.h"
#include "diagrama.h"
#include "proglin.h"

using namespace std;

bool oddcuts;
int graf_inicio, graf_final;

void leerConf();

int main(int argc, char *argv[]){

 pid_t child_pid;
 ofstream fraccionarias("graf_pol_frac.txt");
 ofstream enteras("graf_pol_entero.txt");

 leerConf();

  for(int nG = 1; nG < argc; nG++){

   for(int A = 0; A < 50; A++)
    cout << "*";
   cout << endl << endl;

   int V;
   vector< list<int> > G;
   vector< pair<int,int> > E;
```

```
 string nombre = argv[nG];
 map< pair<int,int>,int > varsX1, varsX2, varsZ1, varsZ2;

 leerGraficaRegular(nombre, &V, E, G);
 crearVariablesXZ( E, varsX1, varsX2, varsZ1, varsZ2);
 crearArchivoIeq(nombre, V, E, G, varsX1, varsX2, varsZ1, varsZ2,
    oddcuts);
 crearDibujos(nombre, V, E, varsX1, varsX2, varsZ1, varsZ2);
 string transformar = "traf -v graficas/" + nombre + ".ieq";
 system(transformar.c_str());
 revisarIntegralidad(nombre, V, E, fraccionarias, enteras, oddcuts);
}

ifstream arch_graficas("listaNueva.txt");
string g6;
unsigned long long num = 0;

while(arch_graficas >> g6){
 num++;
 if(num < graf_inicio)
  continue;
 if(num > graf_final )
  break;
 if(num%1000 == 0)
  sleep(1200);
 if(num == 5000)
  break;

 for(int A = 0; A < 50; A++)
  cout << "*";
 cout << endl << endl;

 int V;
 vector< list<int> > G;
 vector< pair<int,int> > E;
 string nombre = "geng" + patch::to_string(num);
 map< pair<int,int>,int > varsX1, varsX2, varsZ1, varsZ2;

 traducirG6(g6, &V, E, G);
 crearVariablesXZ( E, varsX1, varsX2, varsZ1, varsZ2);
 crearArchivoIeq(nombre, V, E, G, varsX1, varsX2, varsZ1, varsZ2,
    oddcuts);
 crearDibujos(nombre, V, E, varsX1, varsX2, varsZ1, varsZ2);
 string transformar = "traf -v graficas/" + nombre + (string)(oddcuts?"
    _oddcuts":"") + ".ieq";

 child_pid = fork();
 if(child_pid == 0){
  int retVal = system(transformar.c_str());
```

```
    cout << "Return value " << retVal << endl;
    if(retVal != 0)
     return 0;
    revisarIntegralidad(nombre, V, E, fraccionarias, enteras, oddcuts);
    break;
   }
  }
  fraccionarias.close();
  enteras.close();

 return 0;
}

void leerConf(){
 ifstream arch_conf("conf.txt");
 string valor;
 arch_conf >> valor >> oddcuts;
 arch_conf >> valor >> graf_inicio >> graf_final;
 return;
}
```

A.2.2. Archivo: graficas.cpp

```
#include "graficas.h"

void leerGraficaRegular(string nombre, int *V, vector< pair<int,int> > &E,
    vector< list<int> > &G){

 int d, h;

 //Abre el archivo de la gráfica
 ifstream grafica(("graficas/" + nombre + ".txt").c_str());

 grafica >> *V;
 G.resize(*V+1);
 // Leo cada arista (d,h) en E
 while(grafica >> d >> h){
  //A cada arista le asigno dos variables x y dos variables z
  if(d < h){  // El menor obtiene primero su variable
   E.push_back(make_pair(d,h));
  }
  else{
   E.push_back(make_pair(h,d));
  }

  //Agrego esta arista a las listas de adyacencia de los nodos que son
     vecinos en G
  G[d].push_back(h); //Agrego h como vecino de d
  G[h].push_back(d); //Agrego d como vecino de h
```

```
 }

 return;
}

void traducirG6(string grafica_g6, int *V, vector< pair<int,int> > &E,
    vector< list<int> > &G){

 // Función para obtener las aristas en E y para representar G como listas
      de adyacencia

 //Obtengo el número de vértices

 // Si el primer caracter no es 126 entonces: 0 <= n <= 62
 if(grafica_g6[0] != 126){
  *V = grafica_g6[0]-63;
  grafica_g6.erase(0,1); // Elimino los caracteres que corresponden al nú
      mero de vértices.
 }
 // Si sólo el primer caracter es 126 entonces: 63 <= n <= 258047
 else if(grafica_g6[1] != 126){

  // Resto 63 a cada byte 1,2,3
  for(int K = 0; K <= 3; K++){
   grafica_g6[K] -= 63;
  }

  // Convertir a decimales los caracteres 1,2,3
  char c;
  int pos = 0;
  *V = 0;
  for(int K = 3; K >= 1; K--){ //Para cada dígito
   c = grafica_g6[K];
   for (int i = 0; i < 6; ++i) { //Recorro los 6 bits
    if((c >> i) & 1){
     *V |= 1 << pos;   //Prendo el bit en la posición
    }
    /*else{
     *V &= ~(1 << pos);  //Apago el bit en la posición
    }*/
    pos++;
   }
  }
  cout << endl;
  grafica_g6.erase(0,4); // Elimino los caracteres que corresponden al nú
      mero de vértices.
 }

 cout << "Tiene " << *V << " vértices." << endl;
```

```
 cout << grafica_g6 << endl;

 // Resto 63 a cada caracter restante
 for(int pos = 0; pos < grafica_g6.length(); pos++){
  grafica_g6[pos] -= 63;
  cout << (int)grafica_g6[pos] << " ";
 }
 cout << endl;

 int bit = 0;
 G.resize(*V+1);

 //Recorro cada una de las aristas
 for(int C = 1; C < *V; C++){
  for(int R = 0; R < C; R++){

   //printf("(%d,%d) ", R, C);

   //cout << ((grafica_g6[bit/6] & (1 << (5-(bit%6))))?"1":"0") << "\t";
   if((grafica_g6[bit/6] & (1 << (5-(bit%6))))){
    printf("(%d,%d) ", R+1, C+1);
    E.push_back(make_pair(R+1,C+1));
    G[R+1].push_back(C+1);
    G[C+1].push_back(R+1);
   }
   bit++;
  }
 }
 cout << endl;
 return;
}
```

A.2.3. Archivo: proglin.cpp

```
#include "proglin.h"
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

void crearVariablesXZ( vector< pair<int,int> > &E, map< pair<int,int>,int
   > &varsX1, map< pair<int,int>,int > &varsX2, map< pair<int,int>,int > &
   varsZ1, map< pair<int,int>,int > &varsZ2){

 int cont = 1;
 for(int e = 0; e < E.size(); e++){
  varsX1.insert(make_pair( make_pair(E[e].first, E[e].second), cont));
     cont++;
  varsX1.insert(make_pair( make_pair(E[e].second, E[e].first), cont));
```

```
      cont++;
   varsZ1.insert(make_pair( make_pair(E[e].first, E[e].second), cont));
      cont++;
   varsZ1.insert(make_pair( make_pair(E[e].second, E[e].first), cont));
      cont++;
   varsX2.insert(make_pair( make_pair(E[e].first, E[e].second), cont));
      cont++;
   varsX2.insert(make_pair( make_pair(E[e].second, E[e].first), cont));
      cont++;
   varsZ2.insert(make_pair( make_pair(E[e].first, E[e].second), cont));
      cont++;
   varsZ2.insert(make_pair( make_pair(E[e].second, E[e].first), cont));
      cont++;
 }

 return;
}

void crearArchivoIeq(string nombre, int V, vector< pair<int,int> > &E,
    vector< list<int> > &G, map< pair<int,int>,int > &varsX1, map< pair<int
    ,int>,int > &varsX2, map< pair<int,int>,int > &varsZ1, map< pair<int,
    int>,int > &varsZ2, bool oddcuts){

 ofstream ieq(("graficas/" + nombre + (string)(oddcuts?"_oddcuts":"") + ".
     ieq").c_str());
 map< pair<int,int>,int >::iterator it_vars;
 list<int>::iterator it_vecinos;

 // Hay 8 variables por cada arista, entonces esa es la dimension del
     archivo .ieq para PORTA
 ieq << "DIM = " << varsX1.size() + varsX2.size() + varsZ1.size() + varsZ2
     .size() << "\n\n";   // Como en el archivo de la grafica no dice
     cuantas aristas hay en G, entonces las cuento contando directamente el
      numero de variables creadas que debe ser 8 |E|.

 //Encontrar solucion factible: Es la que resulta del algoritmo de
     aproximacion factor 2.
  // Para cada arista de G se recorren sus dos sentidos. El sentido del
      menor al mayor se barre el dia 1 y el otro sentido se recorre. Al dia
       2 es lo opuesto.

 ieq << "VALID\n";
 // Las variables del dia 1
 for(it_vars = varsX1.begin(); it_vars != varsX1.end(); it_vars++, it_vars
     ++){
  if(it_vars->first.first > it_vars->first.second){ //Tomo un para de
      variables, reviso solo una para ver si van del pequeno al grande
   ieq << "1 0 0 1 0 1 1 0 "; //Imprimo el valor de x y z para el primer
       dia para esta arista
```

```
 }
 else{
  ieq << "0 1 1 0 1 0 0 1 "; //Imprimo el valor de x y z para el primer
      dia para esta arista
 }
}
ieq << endl << endl;

//Crear desigualdades
ieq << "INEQUALITIES_SECTION\n\n";
for(int I = 1; I <= varsX1.size() + varsX2.size() + varsZ1.size() +
    varsZ2.size(); I++) // Imprimo la restriccion de que cada variable sea
     positiva
 ieq << "x" << I << " >= 0\n";
for(it_vars = varsX1.begin(); it_vars != varsX1.end(); it_vars++){ //
    Imprimo la restriccion de que todas las x sean menores a 1
 ieq << "x" << it_vars->second << " <= 1\n";
}
for(auto var : varsX2){
 ieq << "x" << var.second << " <= 1\n";
}

//Restricciones de balance de flujo para el dia 1
for(int I = 1; I <= V; I++){ //En cada vertice: hago que el balance sea
    correcto
 //Todo lo que sale
 for(it_vecinos = G[I].begin(); it_vecinos != G[I].end(); it_vecinos++){
     //Recorro la lista de adyacencia del nodo actual I hacia cada vecino
     *it_vecinos
  ieq << "+x" << varsX1[make_pair(I, *it_vecinos)]; // Obtengo el numero
      de variable para ese arco con varsX1, varsZ1
  ieq << "+x" << varsZ1[make_pair(I, *it_vecinos)];
 }
 // Igual a todo lo que entra
 for(it_vecinos = G[I].begin(); it_vecinos != G[I].end(); it_vecinos++){
  ieq << "-x" << varsX1[make_pair(*it_vecinos,I)];
  ieq << "-x" << varsZ1[make_pair(*it_vecinos,I)];
 }
 ieq << " >= 0" << endl;
 //Todo lo que sale
 for(it_vecinos = G[I].begin(); it_vecinos != G[I].end(); it_vecinos++){
     //Recorro la lista de adyacencia del nodo actual I hacia cada vecino
     *it_vecinos
  ieq << "+x" << varsX1[make_pair(I, *it_vecinos)]; // Obtengo el numero
      de variable para ese arco con varsX1, varsZ1
  ieq << "+x" << varsZ1[make_pair(I, *it_vecinos)];
 }
 // Igual a todo lo que entra
 for(it_vecinos = G[I].begin(); it_vecinos != G[I].end(); it_vecinos++){
```

```
   ieq << "-x" << varsX1[make_pair(*it_vecinos,I)];
   ieq << "-x" << varsZ1[make_pair(*it_vecinos,I)];
  }
  ieq << " <= 0" << endl;
 }
 ieq << endl;

 //Restricciones de balance de flujo para el dia 2
 for(int I = 1; I <= V; I++){ //En cada vertice: hago que el balance sea
     correcto
  //Todo lo que sale
  for(it_vecinos = G[I].begin(); it_vecinos != G[I].end(); it_vecinos++){
     //Recorro la lista de adyacencia del nodo actual I hacia cada vecino
     *it_vecinos
   ieq << "+x" << varsX2[make_pair(I, *it_vecinos)]; // Obtengo el numero
      de variable para ese arco con varsX1, varsZ1
   ieq << "+x" << varsZ2[make_pair(I, *it_vecinos)];
  }
  // Igual a todo lo que entra
  for(it_vecinos = G[I].begin(); it_vecinos != G[I].end(); it_vecinos++){
   ieq << "-x" << varsX2[make_pair(*it_vecinos,I)];
   ieq << "-x" << varsZ2[make_pair(*it_vecinos,I)];
  }
  ieq << " >= 0" << endl;
  //Todo lo que sale
  for(it_vecinos = G[I].begin(); it_vecinos != G[I].end(); it_vecinos++){
     //Recorro la lista de adyacencia del nodo actual I hacia cada vecino
     *it_vecinos
   ieq << "+x" << varsX2[make_pair(I, *it_vecinos)]; // Obtengo el numero
      de variable para ese arco con varsX1, varsZ1
   ieq << "+x" << varsZ2[make_pair(I, *it_vecinos)];
  }
  // Igual a todo lo que entra
  for(it_vecinos = G[I].begin(); it_vecinos != G[I].end(); it_vecinos++){
   ieq << "-x" << varsX2[make_pair(*it_vecinos,I)];
   ieq << "-x" << varsZ2[make_pair(*it_vecinos,I)];
  }
  ieq << " <= 0" << endl;
 }
 ieq << endl;


 //Restricciones de que se limpia un arco el dia 1   //Restricciones de
     que se limpia un arco el dia 2
 for(int e = 0; e < E.size(); e++){
  ieq << "x" << varsX1[make_pair(E[e].first, E[e].second)] << " + x" <<
     varsX1[make_pair(E[e].second, E[e].first)] << " >= 1\n";
  ieq << "x" << varsX1[make_pair(E[e].first, E[e].second)] << " + x" <<
     varsX1[make_pair(E[e].second, E[e].first)] << " <= 1\n";
```

```
 ieq << "x" << varsX2[make_pair(E[e].first, E[e].second)] << " + x" <<
    varsX2[make_pair(E[e].second, E[e].first)] << " >= 1\n";
 ieq << "x" << varsX2[make_pair(E[e].first, E[e].second)] << " + x" <<
    varsX2[make_pair(E[e].second, E[e].first)] << " <= 1\n";
}
ieq << endl << endl;
// Restricciones de que cada sentido se limpia algun dia
for(auto e:E){
 ieq << "x" << varsX1[e] << " +x" << varsX2[e] << " >= 1\n";
 ieq << "x" << varsX1[e] << " +x" << varsX2[e] << " <= 1\n";
 ieq << "x" << varsX1[make_pair(e.second, e.first)] << " +x" << varsX2[
    make_pair(e.second, e.first)] << " >= 1\n";
 ieq << "x" << varsX1[make_pair(e.second, e.first)] << " +x" << varsX2[
    make_pair(e.second, e.first)] << " <= 1\n";
}
ieq << endl << endl;
//Agrego las restricciones de cortes impares
if(oddcuts){
 vector<bool> bs(V, 0);
 cout << "Generando los cortes impares...." << endl;
 //Genero todos los posibles subconjuntos desde tamano 1 hasta tamano |V|
 for(int T = 1; T <= V/2; T++){
  cout << "Conjuntos de tamano: " << T << endl;
  bs[0] = 1;
  sort(bs.begin(),bs.begin()+V);
  do{
   for (auto nodo : bs){
    cout << nodo << " ";
   }
   cout << endl;
   //Cuento el numero de aristas en este corte
   vector< pair<int, int> > E_corte;
   for(auto e : E){
    if(bs[e.first-1] != bs[e.second-1]){
     E_corte.push_back(e);
    }
   }
   if(E_corte.size()%2 == 1){
    cout << "************* Tengo un corte impar!!!!!!" << endl;
    //Agrego la restriccion correspondiente a este subconjunto
    for(auto e : E_corte){
     //cout << "La arista " << e.first << " - " << e.second << " esta en
        el corte." << endl;
     // Todas las aristas que van del conjunto 0 al conjunto 1 son
        positivas
     //Dia 1
     ieq << "+x" << varsZ1[e] << "+x" << varsZ1[make_pair(e.second, e.
        first)];
    }
```

```
      ieq << " >= 1\n";
      for(auto e : E_corte){
       // Todas las aristas que van del conjunto 0 al conjunto 1 son
          positivas
       //Dia 2
       ieq << "+x" << varsZ2[e] << "+x" << varsZ2[make_pair(e.second, e.
          first)];
      }
      ieq << " >= 1\n";
     }
    }while(next_permutation(bs.begin(), bs.begin()+V));
   }
  }

  ieq << "END" << endl;

  return;
}


void revisarIntegralidad(string nombre, int V, vector< pair<int,int> > &E,
     ofstream &fraccionarias, ofstream &enteras, bool oddcuts){

  ifstream poi(("graficas/" + nombre + (string)(oddcuts?"_oddcuts":"") + ".
     ieq.poi").c_str());

  string str;
  char caracter;
  int fraccionario = 0;

  do{
   poi >> str;
  }while(str != "CONV_SECTION");

  do{
   poi >> caracter;
   if(caracter == '/'){
    fraccionario = 1;
    break;
   }
  }while(caracter != 'E');

  ofstream arch_graf;
  if(fraccionario){
   fraccionarias << nombre << endl;
   arch_graf.open(("fraccionarias/" + nombre + (string)(oddcuts?"_oddcuts":
      "") + ".txt").c_str());
  }
  else{
```

```
  enteras << nombre << endl;
  arch_graf.open(("enteras/" + nombre + (string)(oddcuts?"_oddcuts":"") +
      ".txt").c_str());
 }
 arch_graf << V << endl;
 for (auto e : E){
  arch_graf << e.first << " " << e.second << endl;
 }
 poi.close();
 return;
}
```

A.2.4. Archivo: diagrama.cpp

```
#include "diagrama.h"

void crearDibujos(string nombre, int V, vector< pair<int,int> > &E, map<
    pair<int,int>,int > &varsX1, map< pair<int,int>,int > &varsX2, map<
    pair<int,int>,int > &varsZ1, map< pair<int,int>,int > &varsZ2){

 ofstream dibujo_variables(("graficas/" + nombre + "_variables.gz").c_str
     ());
 ofstream dibujo_simple(("graficas/" + nombre + "_simple.gz").c_str());
 map< pair<int,int>,int >::iterator it_vars;
 vector< pair<int,int> >::iterator it_aristas;
 list<int>::iterator it_vecinos;

 /********** Gráfica simple *************/
 // Imprime los nodos
 dibujo_simple << "graph G {\n\n";
 for(int I = 1; I <= V; I++){
  dibujo_simple << "\"" << I << "\"";
  if(I+1 <= V){
   dibujo_simple << ",";
  }
 }
 dibujo_simple << "\n\n";

 // Imprime las aristas y arcos entre los nodos
 for(it_aristas = E.begin(); it_aristas != E.end(); it_aristas++){
  dibujo_simple << "\"" << it_aristas->first << "\" -- \"" << it_aristas->
      second << "\";\n";
 }

 dibujo_simple << "\n\n}" << endl;

 /********** Gráfica con variables *************/
 // Imprime los nodos
 dibujo_variables << "digraph G {\n\n";
```

```
 // Imprime las aristas y arcos entre los nodos
 for(it_vars = varsX1.begin(); it_vars != varsX1.end(); it_vars++){
  dibujo_variables << "\"px" << it_vars->first.first << "\" -> \"px" <<
     it_vars->first.second << "\" [label = \"x" << it_vars->second << "\"
     ];\n";
 }
 for(it_vars = varsX2.begin(); it_vars != varsX2.end(); it_vars++){
  dibujo_variables << "\"sx" << it_vars->first.first << "\" -> \"sx" <<
     it_vars->first.second << "\" [label = \"x" << it_vars->second << "\"
     ];\n";
 }
 for(it_vars = varsZ1.begin(); it_vars != varsZ1.end(); it_vars++){
  dibujo_variables << "\"pz" << it_vars->first.first << "\" -> \"pz" <<
     it_vars->first.second << "\" [color=\"darkgreen\" style=\"solid\"
     label = \"z" << it_vars->second << "\"];\n";
 }
 for(it_vars = varsZ2.begin(); it_vars != varsZ2.end(); it_vars++){
  dibujo_variables << "\"sz" << it_vars->first.first << "\" -> \"sz" <<
     it_vars->first.second << "\" [color=\"blue\" style=\"solid\" label =
     \"z" << it_vars->second << "\"];\n";
 }

 // Imprime la solución factible
 for(int v = 0; v < E.size(); v++){
  dibujo_variables << "\"up" << E[v].first << "\" -> \"up" << E[v].second
     << "\" [style=\"solid\" color=\"darkgreen\" label = \"x" << varsX1[
     make_pair(E[v].first, E[v].second)] << "\" ];\n";
  dibujo_variables << "\"up" << E[v].second << "\" -> \"up" << E[v].first
     << "\" [style=\"solid\" color=\"orange\" label = \"z" << varsZ1[
     make_pair(E[v].second, E[v].first)] << "\" ];\n";
  dibujo_variables << "\"us" << E[v].second << "\" -> \"us" << E[v].first
     << "\" [style=\"solid\" color=\"darkgreen\" label = \"x" << varsX2[
     make_pair(E[v].second, E[v].first)] << "\" ];\n";
  dibujo_variables << "\"us" << E[v].first << "\" -> \"us" << E[v].second
     << "\" [style=\"solid\" color=\"orange\" label = \"z" << varsZ2[
     make_pair(E[v].first, E[v].second)] << "\" ];\n";
 }
 dibujo_variables << "\n\n}" << endl;

 return;
}
```

A.3. Implementación de un algoritmo de aproximación

```
#include <iostream>
#include <vector>
```

```
#include <string>
#include <map>
#include <list>
#include <fstream>
#include <cstdlib>
#include <queue>

using namespace std;

void leerGraficaRegular(string nombre, int *V, vector< pair<int,int> > &E,
     vector< list<int> > &G){

 int d, h;

 //Abre el archivo de la grafica
 ifstream grafica(("graficas/" + nombre + ".txt").c_str());

 grafica >> *V;
 G.resize(*V+1);
 // Leo cada arista (d,h) en E
 while(grafica >> d >> h){
  //A cada arista le asigno dos variables x y dos variables z
  if(d < h){  // El menor obtiene primero su variable
   E.push_back(make_pair(d,h));
  }
  else{
   E.push_back(make_pair(h,d));
  }

  //Agrego esta arista a las listas de adyacencia de los nodos que son
     vecinos en G
  G[d].push_back(h); //Agrego h como vecino de d
  G[h].push_back(d); //Agrego d como vecino de h
 }

 return;
}

void traducirG6(string grafica_g6, int *V, vector< pair<int,int> > &E,
    vector< list<int> > &G){

 // Funcion para obtener las aristas en E y para representar G como listas
     de adyacencia

 //Obtengo el numero de vertices

 // Si el primer caracter no es 126 entonces: 0 <= n <= 62
 if(grafica_g6[0] != 126){
  *V = grafica_g6[0]-63;
```

```
 grafica_g6.erase(0,1); // Elimino los caracteres que corresponden al
     numero de vertices.
}
// Si solo el primer caracter es 126 entonces: 63 <= n <= 258047
else if(grafica_g6[1] != 126){

 // Resto 63 a cada byte 1,2,3
 for(int K = 0; K <= 3; K++){
  grafica_g6[K] -= 63;
 }

 // Convertir a decimales los caracteres 1,2,3
 char c;
 int pos = 0;
 *V = 0;
 for(int K = 3; K >= 1; K--){ //Para cada digito
  c = grafica_g6[K];
  for (int i = 0; i < 6; ++i) { //Recorro los 6 bits
   if((c >> i) & 1){
    *V |= 1 << pos;   //Prendo el bit en la posicion
   }
   /*else{
    *V &= ~(1 << pos);  //Apago el bit en la posicion
   }*/
   pos++;
  }
 }
 cout << endl;
 grafica_g6.erase(0,4); // Elimino los caracteres que corresponden al
     numero de vertices.
}

cout << "Tiene " << *V << " vertices." << endl;
cout << grafica_g6 << endl;

// Resto 63 a cada caracter restante
for(int pos = 0; pos < grafica_g6.length(); pos++){
 grafica_g6[pos] -= 63;
 cout << (int)grafica_g6[pos] << " ";
}
cout << endl;

int bit = 0;
G.resize(*V+1);

//Recorro cada una de las aristas
for(int C = 1; C < *V; C++){
 for(int R = 0; R < C; R++){
```

```
   //printf("(%d,%d) ", R, C);

   //cout << ((grafica_g6[bit/6] & (1 << (5-(bit%6))))?"1":"0") << "\t";
   if((grafica_g6[bit/6] & (1 << (5-(bit%6))))){
    printf("(%d,%d) ", R+1, C+1);
    E.push_back(make_pair(R+1,C+1));
    G[R+1].push_back(C+1);
    G[C+1].push_back(R+1);
   }

   bit++;
  }
 }
 cout << endl;
 return;
}

void encontrarCircuitoEuleriano(int *V, vector< pair<int,int> > &A, map<
    pair<int,int>, pair<int,int> > &next_arc){
 vector< list< pair<int,int> > > untraversed;   // untraversed(vertice)
    Por cada vertice tiene una lista de arcos que no se han recorrido
 queue<int> Q;            // Q lista de vertices que todavia tienen arcos
    adyacentes sin recorrer
                // next(arco) es un apuntador para cada arco que dice cual
                   es el siguiente arco
 map< int, pair<int,int> > in;        //Guarda un arco recorrido que entre
    a cada vertice
 vector< pair<int,int> >::iterator it_A;
 int ciclo_listo = 0;
 int nodo_inicial;           //nodo inicial de un ciclo
 pair<int,int> arco_inicial;         //arco inicial de un ciclo
 pair<int,int> arco_actual, arco_siguiente, arco_temp;

 // Marco todos los arcos como no recorridos y el siguiente de cada arco
    como el mismo
 untraversed.resize(*V +1);
 for(it_A = A.begin(); it_A != A.end(); it_A++){ //Para cada arco
  untraversed[it_A->first].push_back(*it_A); // Agrego los arcos al nodo
     correspondiente
  next_arc[*it_A] = *it_A;      // Marco el siguiente de cada arco como el
     mismo
 }

 Q.push(1);
 //Mientras haya nodos con arcos sin recorrer, continuo formando ciclos
 while(!Q.empty()){

  while(untraversed[Q.front()].empty()){
   Q.pop();
```

```
 if(Q.empty())
  break;
}
if(Q.empty())
 break;

// Tomo el noto del principio de la cola y hago un ciclo desde el

nodo_inicial = Q.front();         // Tomo el siguiente nodo con arcos sin
    recorrer
arco_inicial = untraversed[nodo_inicial].front();  //Recorro el primer
   arco de la lista de ese nodo
untraversed[nodo_inicial].pop_front();      // Marco ese arco como
   recorrido
if(untraversed[nodo_inicial].empty())
 Q.pop();
arco_actual = arco_inicial;

//Mientras el ciclo no se cierre, tomo un arco nuevo
//printf("Uso el arco: (%d, %d)\n", arco_actual.first, arco_actual.
   second);
while(arco_actual.second != nodo_inicial){
 arco_siguiente = untraversed[arco_actual.second].back(); //Encuentro un
     nuevo arco
 next_arc[arco_actual] = arco_siguiente;       //Marco ese arco como el
    siguiente del otro arco
 untraversed[arco_actual.second].pop_back();      //Marco el nuevo arco
    como recorrido
 if(in.find(arco_actual.second) == in.end())
  in[arco_actual.second] = arco_actual;
 if(!untraversed[arco_actual.second].empty())     //Si el nodo que quedo
    en medio todavia le quedaron otros arcos sin recorrer, entonces lo
    meto a Q
  Q.push(arco_actual.second);
 arco_actual = arco_siguiente;
 //printf("Uso el arco: (%d, %d)\n", arco_actual.first, arco_actual.
    second);
}
next_arc[arco_actual] = arco_inicial;
if(in.find(nodo_inicial) == in.end())
 in[nodo_inicial] = arco_actual;

//Si hace falta pego el nuevo ciclo
if(ciclo_listo > 0){
 //Tomo el arco inicial de este ciclo
 arco_temp = next_arc[in[nodo_inicial]];  //Guardo el arco siguiente del
     arco que entra al nodo inicial del ciclo
 //printf("El arco siguiente de (%d,%d) es (%d,%d)\n", in[nodo_inicial].
    first, in[nodo_inicial].second, arco_temp.first, arco_temp.second);
```

```
    next_arc[in[nodo_inicial]] = arco_inicial;
    //printf("Reemplazo como nuevo siguiente de (%d,%d) con (%d,%d)\n", in[
        nodo_inicial].first, in[nodo_inicial].second, arco_inicial.first,
        arco_inicial.second );
    next_arc[arco_actual] = arco_temp;
    //printf("Conecto el final de este ciclo con la continuacion del otro
        ciclo: (%d,%d) con (%d,%d)\n", arco_actual.first, arco_actual.second
        , arco_temp.first, arco_temp.second);
   }
   ciclo_listo++;
  }

  //Imprime la secuencia

  cout << "\nUn circuito euleriano es:" << endl;
  arco_inicial = *A.begin();
  arco_actual = arco_inicial;
  do{
   printf("(%d,%d) ---> (%d,%d)\n", arco_actual.first, arco_actual.second,
       next_arc[arco_actual].first, next_arc[arco_actual].second);
   arco_actual = next_arc[arco_actual];
  }while(arco_actual != arco_inicial);

  return;
}

int main(){

  freopen("input.txt", "r", stdin);
  freopen("output.txt", "w", stdout);

  string G_string, format;
  int V;        // Numero de vertices de la grafica
  vector< list<int> > G;    // La estructura para guardar G: como listas de
      adyacencia de los nodos
  vector< pair<int,int> > E;   // La estructura para guardar todas las
     aristas
  vector< pair<int,int> > A;   // La estructura para guardar todos los
     arcos
  map< pair<int,int>, pair<int,int> > next_arc; // Secuencia del circuito
     euleriano
  pair<int,int> arco_inicial, arco_actual;
  vector< pair<int,int> >::iterator it_E;
  vector< pair<int,int> >::iterator it_A;

  map< pair<int,int>,int > varsX1; // Creo las estructuras para guardar los
      vectores de incidencia
  map< pair<int,int>,int > varsX2; // Cada entrada es: <arista, ocurrencias
     >
```

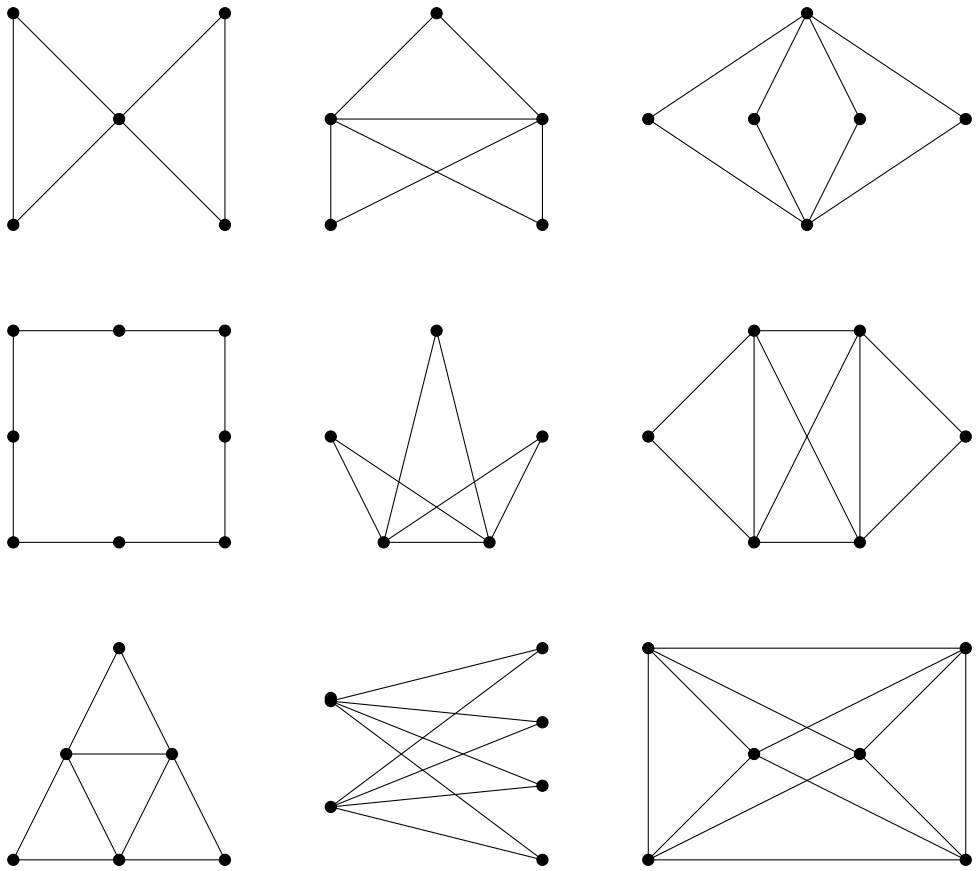
```
map< pair<int,int>,int > varsZ1;
map< pair<int,int>,int > varsZ2;

//Read an undirected graph G from standard input
cin >> format;
if(format == "g6"){
 cin >> G_string;
 //Translate G to regular format
 traducirG6(G_string, &V, E, G);
}
else if(format == "regular"){
 cin >> G_string;
 leerGraficaRegular(G_string, &V, E, G);
}

//Get the graph D (directed version of G)
for(it_E = E.begin(); it_E != E.end(); it_E++){
 A.push_back(make_pair(it_E->first, it_E->second));
 A.push_back(make_pair(it_E->second, it_E->first));
}

//Get a directed eulerian tour over D
encontrarCircuitoEuleriano(&V, A, next_arc);

arco_inicial = *A.begin();
arco_actual = arco_inicial;
do{
 if(arco_actual.first < arco_actual.second){
  varsX1[arco_actual] = 1;   //Choose arcs to be swept on day 1
  varsZ1[arco_actual] = 0;
  varsX2[arco_actual] = 0;
  varsZ2[arco_actual] = 1;   //Mark the arcs to be traversed on day 2
 }
 else{
  varsX1[arco_actual] = 0;
  varsZ1[arco_actual] = 1;   //Mark the arcs to be traversed on day 1
  varsX2[arco_actual] = 1;   //Mark the rest of the arcs to be swept on
      day 2
  varsZ2[arco_actual] = 0;
 }
 arco_actual = next_arc[arco_actual];
}while(arco_actual != arco_inicial);

//Output the incidence vectors x1, x2, z1 and z2
cout << "\nLos vectores de incidencia son: " << endl << endl;
printf("     \t x1\t x2\t z1\t z2\n");
for(it_A = A.begin(); it_A != A.end(); it_A++)
 printf("\{ %d,%d \}  %d \t %d \t %d \t %d \n", it_A->first, it_A->second
     , varsX1[*it_A], varsX2[*it_A], varsZ1[*it_A], varsZ2[*it_A]);
```

```
 cout << endl;


 //Output the sequence of tours T1 and T2
 cout << "El circuito T1 es: \n\n";
 arco_inicial = *A.begin();
 arco_actual = arco_inicial;
 do{
  if(arco_actual.first < arco_actual.second){
   printf("x1\{%d,%d\}", arco_actual.first, arco_actual.second);
  }
  else{
   printf("z1\{%d,%d\}", arco_actual.first, arco_actual.second);
  }
  arco_actual = next_arc[arco_actual];
  if(arco_actual != arco_inicial)
   printf(", ");
 }while(arco_actual != arco_inicial);

 cout << "\n\nEl circuito T2 es: \n\n";
 arco_inicial = *A.begin();
 arco_actual = arco_inicial;
 do{
  if(arco_actual.first < arco_actual.second){
   printf("z2\{%d,%d\}", arco_actual.first, arco_actual.second);
  }
  else{
   printf("x2\{%d,%d\}", arco_actual.first, arco_actual.second);
  }
  arco_actual = next_arc[arco_actual];
  if(arco_actual != arco_inicial)
   printf(", ");
 }while(arco_actual != arco_inicial);

 return 0;
}
```

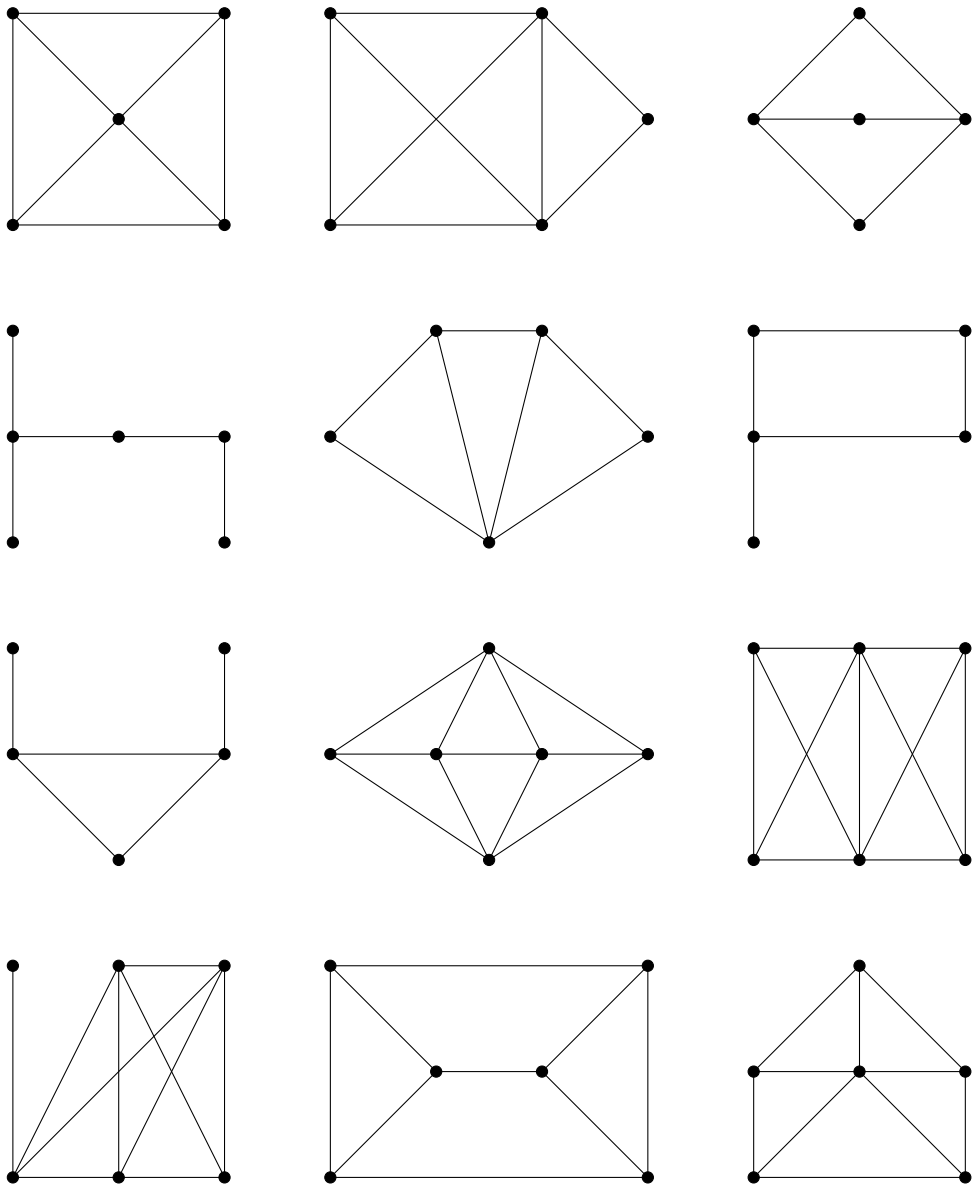
Apéndice B

Ejemplos de gráficas

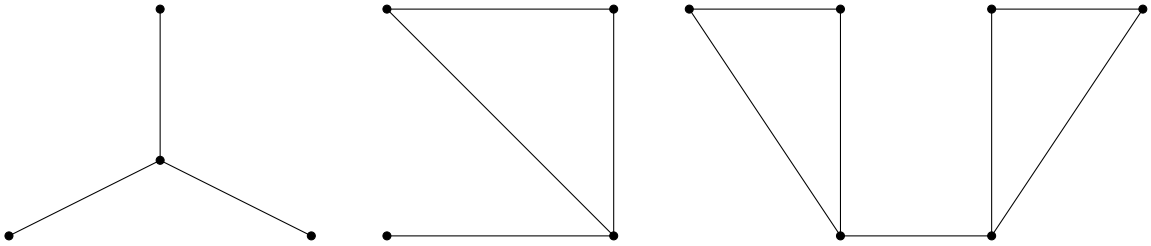
B.1. Ejemplos de gráficas con poliedro entero



B.2. Ejemplos de gráficas con poliedro fraccionario



B.3. Ejemplos de gráficas con poliedro entero después de agregar desigualdades de puentes



Bibliografía

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. Network flows: theory, algorithms, and applications. 1993.

[2] C. Cerrone, B. Dussault, B. Golden, and E. Wasil. Multi–period street scheduling and sweeping. *International Journal of Metaheuristics*, 3(1):21–58, 2014.

[3] D.-S. Chen, R. G. Batson, and Y. Dang. *Applied integer programming: modeling and solution*. John Wiley & Sons, 2011.

[4] T. Christof and A. Löbel. PORTA: POlyhedron Representation Transformation Algorithm, version 1.4.1. *Konrad-Zuse-Zentrum für Informationstechnik Berlin*, 2009.

[5] B. Dussault. *Modeling and Solving Arc Routing Problems in Street Sweeping and Snow Plowing*. PhD thesis, University of Maryland, 2012.

[6] J. Edmonds and E. L. Johnson. Matching, euler tours and the chinese postman. *Mathematical programming*, 5(1):88–124, 1973.

[7] L. Euler. Solutio problematis ad geometriam situs pertinentis. *Commentarii academiae scientiarum Petropolitanae*, 8:128–140, 1741.

[8] M. G. Guan. Graphic programming using odd or even points. *Chinese Math*, 1(273-277):110, 1962.

[9] C. Hierholzer and C. Wiener. Über die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. *Mathematische Annalen*, 6(1):30–32, 1873.

[10] D. Kőnig. Theorie der endlichen und unendlichen Graphen, akad. *Verlagsgesellschaft, Leipzig*, 1936.

[11] B. D. McKay. Practical graph isomorphism. *Congresus Numerantium*, 30:45–87, 1981.

[12] B. Raghavachari and J. Veerasamy. Approximation algorithms for the asymmetric postman problem. In *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, pages 734–741. Society for Industrial and Applied Mathematics, 1999.

[13] L. Sanchez, L. Lomeli, and F. Martinez. Approximation algorithms for the street sweeping problem. In *Electrical Engineering, Computing Science and Automatic Control (CCE), 2014 11th International Conference on*, pages 1–4. IEEE, 2014.

[14] D. P. Williamson and D. B. Shmoys. *The design of approximation algorithms*. Cambridge University Press, 2011.

[15] Z. Win. *Contributions to routing problems*. PhD thesis, Universität Augsburg, 1987.

[16] Z. Win. On the windy postman problem on eulerian graphs. *Mathematical Programming*, 44(1-3):97–112, 1989.

[17] F. J. Zaragoza Martínez. Series–parallel graphs are windy postman perfect. *Discrete Mathematics*, 308(8):1366–1374, 2008.