UNIVERSIDAD DE BUENOS AIRES

Facultad de Ciencias Exactas y Naturales

Departamento de Matemática

Tesis de Licenciatura

Optimización para la asignación de barrenderos y recolección de hojas en la ciudad de Trenque Lauquen

Valeria Di Tomaso

Director: Guillermo Durán

Codirector: Diego Delle Donne

Diciembre 2018

Resumen

En la presente tesis, estudiaremos una implementación para resolver el problema de asignación de rutas de los barrenderos en la ciudad de Trenque Lauquen (provincia de Buenos Aires).

La solución garantiza la compacidad y eficiencia de los recorridos; y determina puntos fijos para el depósito de los residuos generados por esta tarea. Por su parte, la recolección de estos montículos está a cargo de camiones destinados exclusivamente a ello, de los cuales se debe determinar su ruteo.

Anteriormente al presente trabajo, el municipio realizaba estas tareas mediante una programación manual que, a juicio de los responsables de las mismas, era ineficiente.

La estrategia de resolución aquí expuesta divide al problema en tres etapas distintas, cada una implementada sobre las tres zonas en las que se divide la ciudad.

La primera de ellas comienza con un algoritmo de reconocimiento de manzanas. A partir de esto, se generan segmentos conformados por conjuntos de manzanas contiguas, que son utilizados como input para resolver un problema de asignación, el cual fija qué operario barrerá cada una de las manzanas.

En la segunda etapa, se determina en qué orden se recorrerá cada una de las cuadras del segmento asignado al barrendero. Para ello, se lo trata como el Problema del Cartero Chino, y posteriormente, se determina dónde se fijan los puntos de depósito para los montículos de hojas, mediante un modelo de programación lineal entera desarrollado en este trabajo.

Por último, en la tercera etapa, se utiliza el algoritmo de Dijkstra para calcular las distancias de un montículo a otro, y con esta información se construye un grafo con nodos dados por todos los puntos de depósito. Finalmente, sobre ese grafo se utiliza el software Concorde para determinar el ruteo de los camiones de forma óptima.

Agradecimientos

A Willy y Diego por haberme guiado a lo largo de todo este trabajo, aportando sugerencias y ayudándome ante la presencia de dificultades.

A los jurados, Javier y Daniel, por haberse tomado el tiempo de leer este trabajo.

A la gente de la Municipalidad de Trenque Lauquen, especialmente a Marcelo Ferreyra y Adhemar Enrietti, por su buena predisposición para brindar todos los datos necesarios para resolver el problema, y también por la hospitalidad que tuvieron al recibirnos.

A mi familia, especialmente a mis papás, y amigos: Dulce, Carla, Anto y Nico, por todo el cariño y apoyo que me dieron durante estos años, quienes siempre me estimularon para que emprenda todos mis proyectos. Sin ellos, hubiera sido imposible que hoy llegue a este punto.

A mi novio, por todo su amor y paciencia infinita durante estos más de cinco años, por haber estado presente siempre, dándome aliento para que siga adelante en todos los aspectos de mi vida. Espero que estos cinco años se conviertan en muchos más. Y también gracias por tus aportes a este trabajo.

A mis compañeros de la facultad, con quienes compartí risas y muchas horas de cursada. Cami, Flor, Agus y Sofía, que estuvieron desde las primeras materias, Matías, Rocío, León, que se sumaron más adelante, Naza, y Lucho, con quienes estuve en Elavio, y Joaquín, Franco e Iván, con quienes compartí el último viaje a Bahía Blanca.

Índice general

1. Introducción **6**
1.1. Estructura de la tesis 6
1.2. Ciudad de Trenque Lauquen 7
1.3. Descripción del problema 7

2. Estado del arte **13**

3. Marco teórico **15**
3.1. Problemas de ruteo 15
3.1.1. Definiciones y resultados preliminares 15
3.1.2. Problema de ruteo: definición formal 17
3.1.3. Problema del cartero chino (CPP) 18
3.1.4. Problema del cartero rural (RPP) 36
3.2. Algoritmo de Jarvis March 40
3.3. Programación lineal 42
3.3.1. Branch and Bound 43
3.3.2. Branch and Cut 44
3.4. Fórmula de Gauss para el área de un polígono 45
3.5. Procedimiento para pasar de un multigrafo a grafo 45
3.6. Algoritmo de Dijkstra 46
3.7. BFS (Breadth-First Search) 50
3.8. Algoritmo GRASP (Greedy Randomized Adaptive Search Procedure) . . . 54
3.9. Problema del viajante de comercio (TSP) 55

4. Resolución del problema **58**
4.1. Estrategia inicial 58
4.1.1. Conjuntos 58
4.1.2. Parámetros 59
4.1.3. Variables 59
4.1.4. Restricciones 60
4.1.5. Función objetivo 61

4.2. Estrategia utilizada 62
4.2.1. Primera etapa: Asignación de cuadras a barrenderos 62
4.2.2. Segunda etapa: Recorrido de los barrenderos y asignación de montículos 73
4.2.3. Tercera etapa: Recorrido de los camiones 79

5. Implementación **80**
5.1. Procesamiento del mapa 80
5.1.1. Estructura de datos 82
5.1.2. Función distancia 83
5.2. Detalles importantes de la primera etapa 83
5.3. Detalles de la segunda etapa 86
5.4. Concorde 86
5.5. Detalles de la tercera etapa 87

6. Análisis y Resultados finales **89**
6.1. Etapa 1 89
6.2. Etapa 2 98
6.3. Etapa 3 100

7. Conclusiones y trabajo a futuro **103**

Capítulo 1

Introducción

En este capítulo se introducirá una breve descripción de la forma de organización de la tesis, junto con una primera aproximación al problema. Para eso se incluirán detalles importantes sobre la forma de trabajo utilizada en la ciudad de Trenque Lauquen previamente. Entre ellos, pueden encontrarse características de la ciudad, cantidad de operarios, zonas de barrido, dificultades, etc. Se fijará también cuál es el objetivo primordial que intentamos resolver en este trabajo.

1.1. Estructura de la tesis

Para una mayor comprensión, aquí se describe cómo está organizado el resto de la tesis, con una breve mención de los temas tratados en cada capítulo.

- **Capítulo 2: Estado del arte**: Brevemente se describirá la literatura existente para problemas que comparten ciertas similitudes con el presente.
- **Capítulo 3: Marco teórico**: Se verán definiciones y teoremas básicos sobre teoría de grafos, junto con la descripción y resolución de problemas de ruteo (CPP, RPP, TSP). También se verán algunos algoritmos conocidos que fueron utilizados en el presente trabajo: BFS, Dijkstra, Jarvis March, GRASP, Branch and Bound y Branch and Cut. La idea de este capítulo es poder dar todas las herramientas necesarias para comprender el trabajo desarrollado.
- **Capítulo 4: Resolución del problema**: Presentará una estrategia inicial y la que finalmente fue implementada. Se incluirán los diversos modelos de programación lineal entera y el pseudocódigo de cada uno de los algoritmos utilizados en el problema.
- **Capítulo 5: Implementación**: Se verán detalles concretos sobre la implementación: el software complementario utilizado y los obstáculos surgidos por características propias de la ciudad a la hora de desarrollar la solución.
- **Capítulo 6: Análisis y resultados finales**: Se presentarán los resultados, con una descripción de los parámetros elegidos en cada una de las etapas de resolución, y el análisis de las mejoras obtenidas.

- **Capítulo 7: Conclusiones y trabajo a futuro**: Describirá algunas ideas que podrían implementarse para explorar ciertos aspectos de la presente resolución.

1.2. Ciudad de Trenque Lauquen

Cabecera del partido homónimo de la Provincia de Buenos Aires, está ubicada a 445 km del oeste de la Ciudad Autónoma de Buenos Aires, y a 80 km de la Provincia de La Pampa. Se encuentra en la interesección entre las rutas nacionales N°5 (que conecta a C.A.B.A y Santa Rosa) y la N°33 (que une Rosario y Bahía Blanca). Según el Censo 2010, cuenta con un total de 33.442 habitantes, distribuidos en sus 400 hectáreas de superficie.

Una característica distintiva de la ciudad es que la mayoría de sus arterias cuentan con ramblas o boulevards. En total pueden contabilizarse 616 de ellos, constituyendo una ciudad que tiene una alta proporción de espacios verdes por habitante. Más adelante veremos que esta característica particular tuvo gran influencia sobre el trabajo, especialmente a la hora de generar un grafo a partir del mapa de la ciudad.

Figura 1.1: Ciudad de Trenque Lauquen

1.3. Descripción del problema

Con respecto a las tareas de limpieza, existen dos tipos de barrido: manual y mecánico. Este último método queda reservado para los cordones de ramblas, ya que al no haber vehículos estacionados, se puede circular con mayor facilidad que sobre los cordones de las aceras. En la actualidad se cuenta con una máquina barredora conducida por dos operarios, cuyo recorrido se puede seguir satelitalmente mediante el sistema “infotrak”.

En cuanto al barrido manual, la cantidad de calles a limpiar ronda en 1814. Esta tarea era realizada al momento del inicio de este estudio, por 84 empleados distribuidos en tres zonas. Cada una de ellas está supervisada por un capataz que determinaba, muchas veces a demanda, qué cuadras barrer. Con esta división, el número de operarios en la zona 1 era 21, quienes barrían un total de 629 cuadras; mientras que la zona 2 contaba con 35 operarios y 678 cuadras, y la zona 3, con 28 operarios y 507 cuadras.

Esta subdivisión, que ya se venía adoptando anteriormente por criterios geográficos, fue mantenida en el presente trabajo. Las zonas pueden apreciarse en la figura 1.2. Allí, marcadas con una flecha numerada, se encuentran las bases donde los operarios guardan las herramientas necesarias para realizar sus tareas (escobillones, carritos, etc), y deben fichar en el horario de entrada y salida. En la base 1, correspondiente al centro de monitoreo urbano, previamente al presente trabajo, fichaban 25 operarios, mientras que las bases 2,3 y 4 corresponden al Palacio Municipal, el espacio de Servicios Sanitarios Municipales, e Higiene Urbana respectivamente, mientras que la cantidad de operarios para cada una era 11, 10 y 38.

Con respecto a las horas de trabajo disponibles, cada barrendero realiza una jornada laboral, de lunes a viernes, de seis horas matinales, con la posibilidad de realizar un turno extra de tres horas por la tarde. Estos números pueden interpretarse como 504 horas de trabajo ordinarias realizadas cada día. En la práctica, sin embargo, sólo un 15 % de los operarios realizaba horas extra, y de las seis horas obligatorias, debido a la distancia desde las bases hasta los puntos de salida, sólo cinco eran horas de barrido efectivas. Considerando el tiempo extra, en promedio cada uno de los empleados tenía asignados teóricamente unos 24 cordones de cuadra por día. Notar que si se hacen las cuentas, el valor real debería ser de 21/22 cordones, pero dado que existe un porcentaje de ausentismo cercano al 20 % , la asignación teórica mencionada era un poco mayor.

Anteriormente al presente trabajo, la tarea de barrido se desarrollaba en un avance progresivo a lo largo de las cuadras, colocando las hojas sobre las esquinas preestablecidas. Posteriormente, pero durante el mismo rango horario, el servicio de recolección de montículos se encargaba de recolectarlos.

Dicha recolección era llevada a cabo por dos camiones por zona, cada uno con dos operarios distintos. Anteriormente, las esquinas preestablecidas se encontraban en todas las puntas de calle y/o rambla. Una vez finalizado el trabajo, todos los residuos del camión se llevan a un basural en las afueras de la ciudad, ubicado 7 kilómetros al noroeste.

Entre algunos de los inconvenientes que se presentaban antes del desarrollo del presente trabajo se puede mencionar:

- **Incumplimiento de los recorridos preestablecidos:** Muchas de las calles no se barrían con la frecuencia mínima esperada (cada tres días, o día por medio) por diversas causas. Esto sugería que faltaba personal, y por lo tanto era uno de los puntos que tuvimos que considerar.
- **Falta de un recorrido bien definido:** Debido al punto anterior, los recorridos se determinaban a demanda. Surgía entonces la necesidad de determinar recorridos preestablecidos, objetivo principal de la Municipalidad.

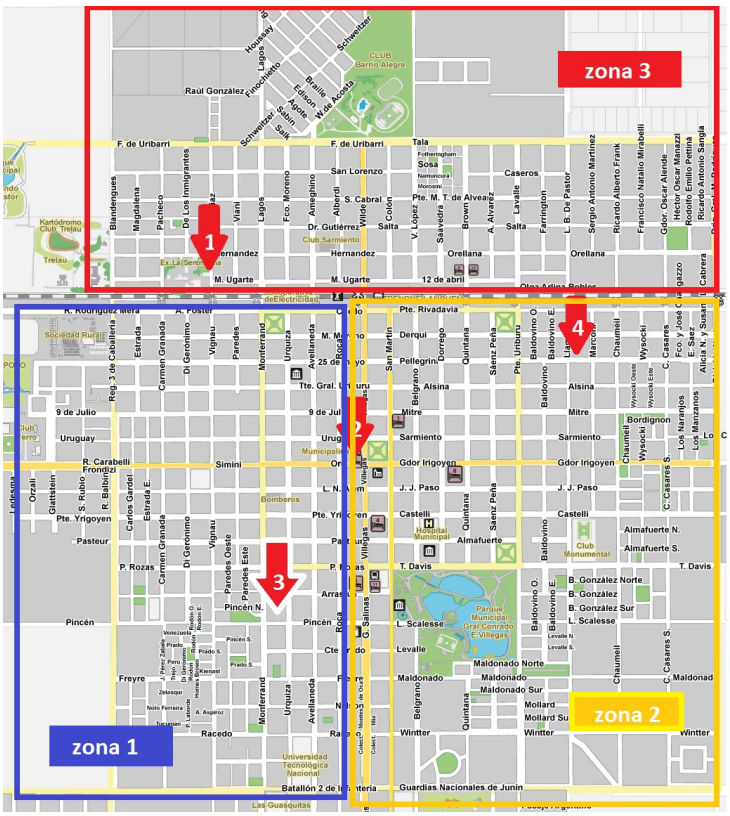


Figura 1.2: División de la tarea en tres zonas

- **Reducción en la eficacia:** En la práctica, y por diversas causas, la cantidad de cuadras barridas rondaba en promedio los 20 cordones, número inferior a los 24 cordones asignados.
- **Falta de coordinación entre los recolectores de residuos y los barrenderos:** Principalmente debido a la superposición de horarios.

Toda esta información fue brindada por la Municipalidad en un comienzo, pero a su vez, esto dio lugar a nuevas dudas que fueron surgiendo a medida que se planteaba una estrategia de resolución.

El primer interrogante fue si había avenidas o lugares específicos donde era necesario barrer más seguido. La respuesta fue que sí, y vino acompañada de un mapa con las frecuencias de cada una de las cuadras. En las figuras 1.3, 1.4, 1.5 se muestran las mismas.

Otro dato necesario fue determinar cuánto se tardaba en barrer cada calle, ya que la estimación inicial provista por la Municipalidad sólo determinaba que una cuadra insumía entre 10 y 15 minutos. Sin embargo, dado que el tiempo depende mucho del operario en cuestión, sumado al hecho que era demasiado complicado parametrizar cada una de las 1814 cuadras, finalmente se optó por estimarla según la longitud de la misma. Más adelante detallaremos cómo se realizó esta tarea.

Otras cuestiones que surgieron fueron la flexibilidad con respecto a los puntos dónde depositar los montículos, y la posibilidad de realizar un corrimiento en el horario de los camiones. En ambos casos, la Municipalidad se mostró conforme con realizar los cambios que consideráramos necesarios para optimizar la solución.

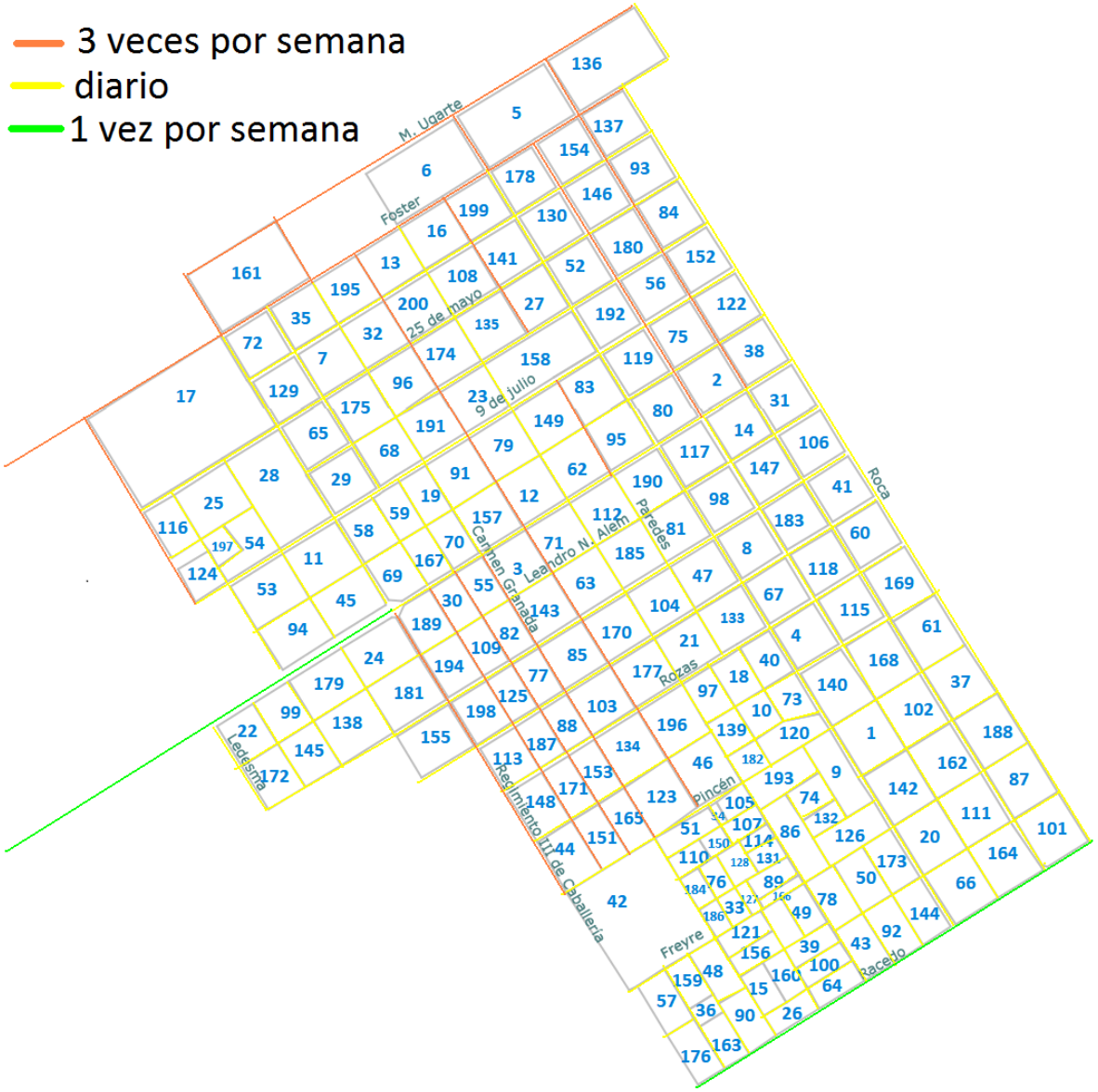


Figura 1.3: Categorización de las cuadras en la zona 1

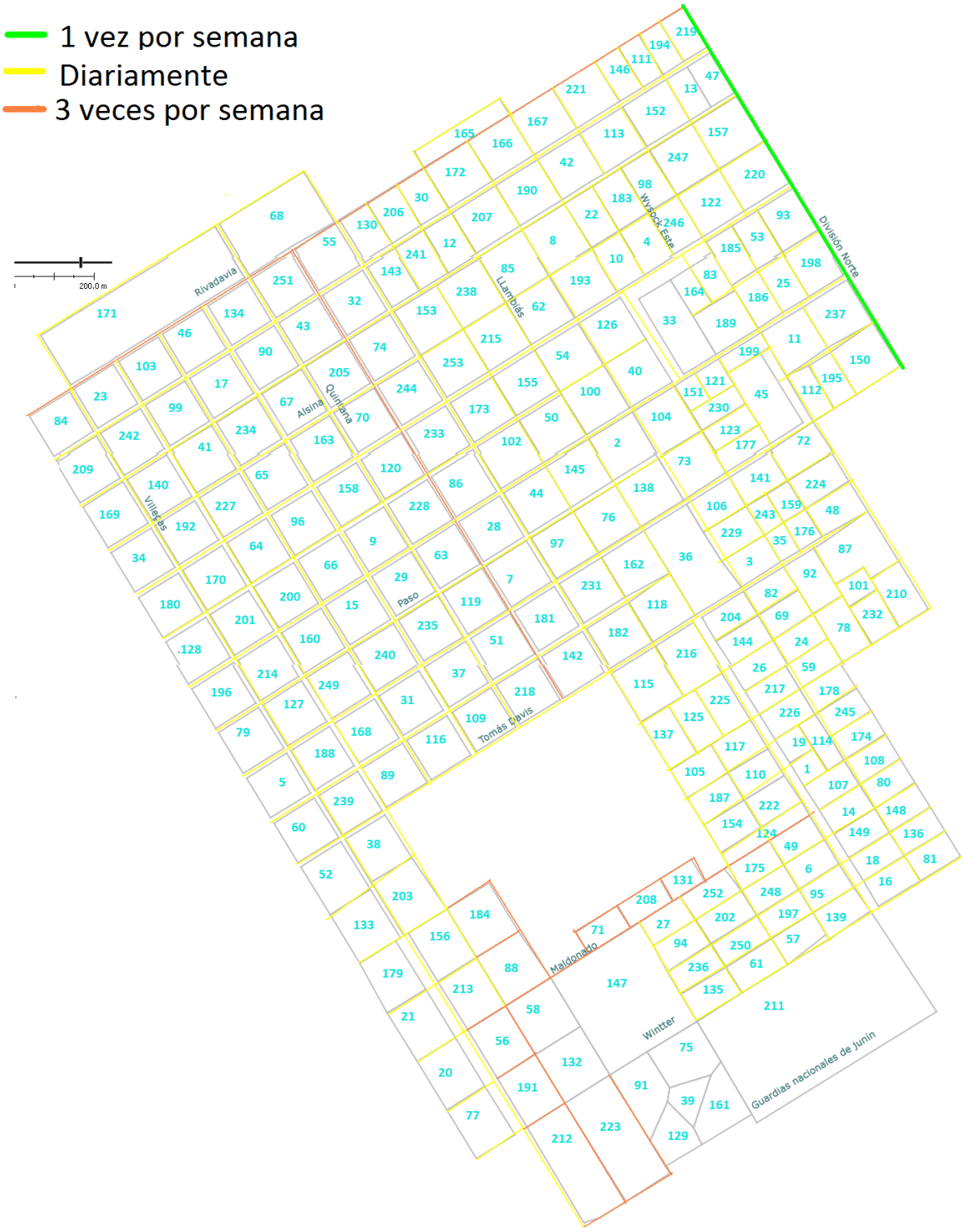


Figura 1.4: Categorización de las cuadras en la zona 2

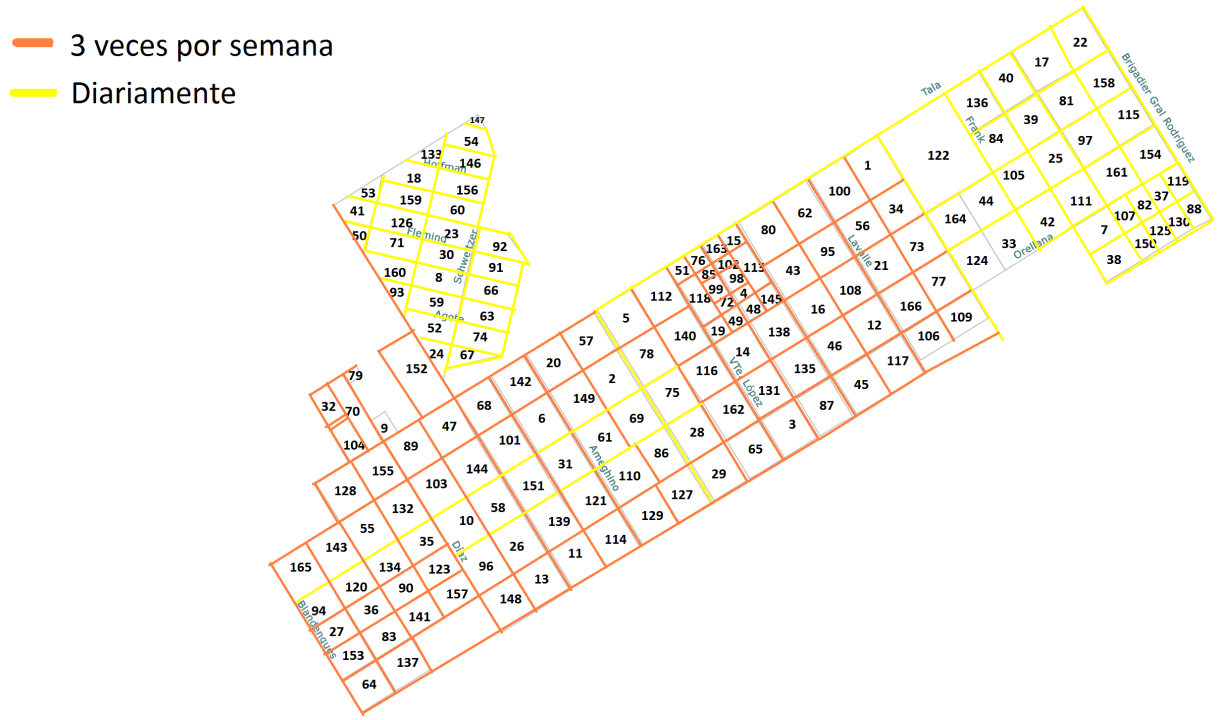


Figura 1.5: Categorización de las cuadras en la zona 3

Resumiendo, el objetivo del trabajo fue:

• Evaluar si la cantidad de operarios era la adecuada, generando para ellos recorridos a pie, que cumplan con una frecuencia mínima de barrido y tengan un cierto nivel de "elegancia".

• Fijar dónde depositar los residuos producidos por dicha tarea.

• Establecer el ruteo óptimo, a cargo de los camiones, para la recolección de esos montículos.

Capítulo 2

Estado del arte

En este capítulo presentaremos varios trabajos académicos realizados sobre el tema de recolección de residuos y barrido de calles. Algunos de ellos comparten ciertas similitudes con este problema, por lo constituyeron un gran aporte para la resolución.

En [**1**] se describe una implementación de recorridos de barredoras del área rural de Lancashire, Inglaterra. Allí cada distrito cuenta con un vehículo que comienza su recorrido desde un depósito. Una de las características del problema es la capacidad limitada de cada barredora, lo cual lo convierte en una versión NP-hard del problema del cartero chino. A diferencia de otros trabajos, al ser un área rural, tiene múltiples puntos para vaciar su contenido; las calles pueden considerarse de doble sentido porque el poco tráfico permite que las barredoras violen las reglas de tránsito, y no existen restricciones en cuanto a momentos de barrido disponibles. Esto último sí ocurre en algunas ciudades urbanas donde hay horarios de estacionamiento, que impiden que las barredoras realicen su trabajo en esas franjas temporales.

Para resolver el problema, lo descomponen en dos etapas: la primera determina la frecuencia de barrido (dependiente de cada cuadra) y la segunda determina dónde depositar la carga generada. El algoritmo también aprovecha el hecho de que las calles sean de doble sentido creando caminos que incluyan barrer ambas manos en un mismo día.

En [**2**] desarrollan un sistema para múltiples barredoras. El trabajo está centrado en New York y Washington D.C. A diferencia de [**1**], se enfocan en un área urbana con mayoría de calles de sentido único y múltiples restricciones de estacionamiento. Estas últimas son tratadas como ventanas de tiempo. Los resultados empíricos obtenidos en este trabajo, muestran que conviene más determinar un recorrido y luego dividirlo para cada barrendero, que realizar la inversa, ya que de esta manera se obtiene un ahorro mayor en cantidad de kilómetros recorridos.

En [**3**] se desarrollan técnicas de programación lineal entera para la optimización de la recolección de residuos reciclables en el municipio de Morón. Allí se resuelve el problema del cartero chino en un grafo mixto, pero con restricciones adicionales de tránsito, dado que dicha recolección se realiza con un camión. Dado la complejidad que representa el hecho de ser un grafo mixto, también se implementan algoritmos de aceleración de búsqueda, que unen los subciclos generados en la solución del modelo. Junto con esto, también se utiliza un algoritmo de búsqueda tabú que mejora la distribución del área de recolección

de cada uno de los camiones.

En [**4**] se estudian algoritmos heurísticos de zonificación para la recolección de residuos, también para camiones . Allí, además de fijar el área que recorre cada camión, a través de la identificación de ciclos, se determina cómo debe ser recorrida de manera tal que se respeten las reglas de tránsito y se minimice el desgaste del vehículo. Finalmente en [**5**] se analiza la optimización en la recolección de residuos en contenedores de la zona sur de la Ciudad de Buenos Aires. El problema se reduce al clásico TSP, y se utiliza el software Concorde para resolverlo. Aprovechando la aleatoridad de las soluciones que propone el software, se reduce el desgaste producido por los camiones, medido con el concepto físico de trabajo mecánico. Exceptuando la minimización del trabajo de los camiones, este último problema se asemeja bastante a la tercera etapa del nuestro.

Capítulo 3

Marco teórico

En este capítulo se brindarán numerosas definiciones (grafo, multigrafo, camino, circuito, ciclo, conexidad, camino y ciclo hamiltoniano, camino y ciclo Euleriano) y el teorema de caracterización de grafos eulerianos.

Con respecto a los problemas de ruteo, además de la definición formal, se analizarán en particular el CPP y el RPP en todas sus variantes, y las estrategias que se usan para resolverlo (algoritmo de Edmonds-Johnson, de blossom, Hierholzer, Fleury, van Aardenne-Ehrenfest y Brujin).

Más adelante, se describirán algunos algoritmos como el de Jarvis March para encontrar la cápsula convexa, BFS para búsqueda en grafos y Dijkstra para calcular el árbol de distancias mínimas, todos ellos utilizados en algún punto de este trabajo.

También se dará una breve introducción a la programación lineal y a los algoritmos GRASP, sumado a una explicación detallada sobre el TSP, con sus variantes y la formulación de PLE.

3.1. Problemas de ruteo

Antes de comenzar con la definición y el estudio de algunos problemas de ruteo, comenzaremos por dar algunos conceptos y resultados preliminares que serán necesarios para la comprensión del tema.

3.1.1. Definiciones y resultados preliminares

Definición 3.1.1. Un **grafo** G está dado por un par $(V(G), E(G))$, donde $V(G)$ es un conjunto finito de vértices y $E(G)$ es un conjunto de pares no ordenados de vértices de G, denominados aristas.

Definición 3.1.2. Un **multigrafo** es un grafo donde se permite que entre dos vértices haya más de una arista.

Definición 3.1.3. Un **camino** de un grafo G es una secuencia de vértices $P = v_1, v_2 \ldots, v_k$ distintos, tales que $(v_i, v_{i+1}) \in E(G)$, para $i = 1, \ldots, k-1$.

Definición 3.1.4. Un **tour** o **circuito** de un grafo G es una secuencia de vértices $C = v_1, v_2 \ldots, v_k$, no necesariamente distintos, tales que $v_1 = v_k$ y $(v_i, v_{i+1}) \in E(G)$, para $i = 1, \ldots, k-1$.

Definición 3.1.5. Un **ciclo** de un grafo G es un circuito con al menos tres vértices, todos distintos (con excepción del primero y el último).

Definición 3.1.6. Un grafo es **conexo** si para todo par de vértices existe un camino que los une.

Definición 3.1.7. Un **camino hamiltoniano** en un grafo G es un camino que recorre cada vértice una y sólo una vez.

Definición 3.1.8. Un **circuito hamiltoniano** en un grafo G es un circuito que recorre cada vértice una y sólo una vez.

Definición 3.1.9. Un **camino euleriano** en un grafo es un camino que recorre cada arista una y sólo una vez.

Definición 3.1.10. Un **circuito euleriano** en un grafo es un circuito que recorre cada arista una y sólo una vez.

Lema 3.1.1. Sea G un grafo en el que todos sus vértices tienen grado par. Entonces, para cada par de vértices adyacentes de G puede encontrarse un ciclo que contiene a la arista que forman ambos.

Demostración. Sean u y v dos vértices adyacentes de G, y sea γ un camino que comienza en u y continúa por la arista (u, v).

Cada vez que γ llega a un vértice w distinto de u, continuamos el camino por una arista que no esté en γ. Si $w = u$, terminamos el proceso. Como el grado de los vértices es par por hipótesis, cada vez que el camino γ pasa por un vértice, utiliza dos aristas con un extremo en el mismo.

Como el número de aristas y el de vértices es finito, el camino γ acaba por volver a u y γ es, según la construcción hecha, un ciclo.

□

Teorema 3.1.1. Sea G un grafo conexo y no dirigido. Son equivalentes:

1. G tiene un ciclo euleriano.
2. Todo vértice de G tiene grado par.
3. Las aristas de G pueden particionarse en circuitos.

Demostración. $1 \Rightarrow 2)$ Sea $v \in G$. Llamemos γ al ciclo de Euler.

Si v no es el primer vértice de γ, cada una de las veces que el ciclo pase por v, entrará y saldrá por dos aristas distintas a las de la vez anterior. Luego contribuirá con dos al grado de v.

Si v es el primer vértice, el ciclo γ contribuye con grado dos cada una de las visitas que se realicen a v, salvo la primera y la última que añade una cada vez.

$2 \Rightarrow 1$) Si $|V| = 1$ ó 2 es trivial. Supongamos que $|V| > 2$.

Sean u y v vértices adyacentes de G. Como G tiene todos sus vértices de grado par, el lema anterior asegura la existencia de un ciclo γ_1 que contiene a la arista (u, v). Sea $G' = (V, E')$ el subgrafo de G que se genera eliminando las aristas que están en γ_1, es decir $E' = E \setminus \{\text{aristas de } \gamma_1\}$.

G' tiene todos sus vértices de grado par (o cero), ya que en el ciclo γ_1 cada vértice habrá aportado dos aristas. Luego, si los vértices de G eran de grado par, los de G' seguirán siéndolo.

Si $E' = \emptyset$, entonces $\gamma = \gamma_1$ es el ciclo de Euler que se busca y la demostración está concluida.

Si $E' \neq \emptyset$, continúa el proceso. Elegimos un vértice cualquiera de γ_1. Si no está aislado en G', tomamos uno de sus adyacentes. Por el lema anterior, habrá un ciclo γ_1' que contenga la arista que forman ambos.

Si está aislado en G', entonces elegimos un vértice cualquiera que no esté aislado en G' (siempre existirá, ya que $E \neq \emptyset$). La primera arista del camino que une a ambos en G (recordar que el mismo es conexo) que no esté en γ_1 comenzará en un vértice de γ_1 no aislado en G'. Ahora basta con tomar este vértice y uno de sus adyacentes para hallar el ciclo γ_1'. De esta forma, unimos γ_1 con γ_1' de la siguiente forma: recorremos γ_1 hasta llegar al vértice que acabamos de elegir, seguimos a través de γ_1', y volvemos a γ_1. Así, obtendremos un nuevo ciclo γ_2 con más aristas que γ_1. Sea $E'' = E' \setminus \{\text{aristas de } \gamma_1'\}$.

Si $E'' = \emptyset$, entonces $\gamma = \gamma_2$ es el ciclo de Euler que buscamos y la demostración está concluida.

Si $E'' \neq \emptyset$ entonces se reitera el mismo proceso. Siguiendo así sucesivamente, como el número de aristas es finito y en cada una de las construcciones aumenta el número de aristas que tiene el ciclo construido, el proceso termina con la obtención de un ciclo de Euler.

$1 \Rightarrow 3$) Si $G = (V, E)$ contiene un tour euleriano T, entonces tenemos trivialmente esta descomposición.

$3 \Rightarrow 1$) Supongamos que tenemos una descomposición C en ciclos disjuntos $C_1, C_2, \ldots, C_m$, tal que $G = \cup C_i$. Por la definición de ciclo, en cada uno de ellos los nodos tienen grado par, y dado que son disjuntos, entonces el grado de cualquier nodo será la suma de su grado en cada uno de los ciclos C_i. Usando la implicación $1 \Leftrightarrow 2$ se concluye que el grafo tiene un tour euleriano.

□

3.1.2. Problema de ruteo: definición formal

Sea $G = (V, E)$ un grafo conexo. Para cada arista $(v_i, v_j) \in E$, se le asocia un costo no negativo c_{ij}. Se asume que si $(v_i, v_j) \notin E$, entonces $c_{ij} = \infty$. De esta manera, se considera la matriz $C = (c_{ij})$, que satisface la desigualdad triangular: $c_{ik} + c_{kj} \leq c_{ij}\ \forall i, j, k$. En

particular, si el grafo es no dirigido, esta matriz además es simétrica.

Un problema de ruteo consiste en encontrar un circuito de costo mínimo que incluya un subconjunto $Q \subseteq V$ de vértices requeridos y otro subconjunto $R \subseteq E$ de aristas requeridas, aunque la solución puede involucrar otros vértices o arcos si es necesario.

Dentro de esta categoría, se encuentran los problemas de ruteo de arcos, como el Problema del Cartero Chino, que detallaremos más adelante, y el Problema del Cartero Rural (RPP). Este último toma $Q = \emptyset$ y está inspirado en la entrega de correspondencia en zonas rurales: se tiene un conjunto R de calles de distintos pueblos, sobre las cuales hay que prestar servicio, y un conjunto $A \setminus R$ de rutas entre los pueblos, que no hace falta atravesar, pero que pueden ser recorridas para pasar de un poblado a otro.

3.1.3. Problema del cartero chino (CPP)

El problema fue introducido por el matemático chino Kwan Mei-Ko en 1960, y traducido al inglés dos años después [**6**]. Mei-ko trabajó durante algún tiempo en una oficina postal, donde se planteó por primera vez el problema que tenían los carteros para realizar su recorrido caminando la menor distancia posible.

Más formalmente, consiste en encontrar un tour de longitud mínima de manera tal que se recorran todas las aristas del grafo al menos una vez.

Como se vio en el teorema 3.1.1, si el grafo no es dirigido y es par (es decir todos sus vértices tienen grado par), entonces la solución es óptima, ya que se puede recorrer cada arista exactamente una vez. Este resultado fue demostrado por Euler en [**7**]. Si es dirigido, en cambio, alcanza con que el número de aristas que entran y salen en cada vértice sea par. Finalmente, si el grafo es mixto, cada vértice debe ser incidente a un número par de arcos (ya sean dirigidos o no). Más aún, para cada $S \subseteq V$, la diferencia entre la cantidad de arcos dirigidos de S a $V \setminus S$ y viceversa debe ser menor o igual al número de arcos no dirigidos que unen a ambos conjuntos. Esta condición suele llamarse "condición del conjunto balanceado".

Los algoritmos para resolver el CPP tienen dos etapas: primero hay que determinar un aumento de costo mínimo del grafo, es decir, un conjunto de arcos de costo mínimo que hacen al grafo euleriano, mientras que en la segunda etapa se calcula el tour.

Sea $1 + x_e$ la cantidad de veces que la arista e es recorrida, y sea G' el grafo de aumento, que contiene esa cantidad de copias de la arista. Luego el tour euleriano puede encontrarse sobre ese grafo.

Pero para eso es necesario determinar el valor de las variables x_e para que el tour sea óptimo. Esto es equivalente al siguiente problema:

$$\text{Minimizar} \sum_{e \in E} c_e x_e$$

sujeto a:

$$\sum_{e \in E_i} (1 + x_e) \equiv 0 \pmod 2 \ \forall i \in V$$

$\mathrm{x}_e \geq 0$

$\mathrm{x}_e \in \mathbb{Z}$

donde E_i es el conjunto de todas las aristas que inciden en el vértice i.

Si el grafo es completamente no dirigido o dirigido, la primera etapa puede resolverse fácilmente. En el primer caso, puede tratarse como un problema de matching (es decir, encontrar un subconjunto M de aristas tal que para todo vértice v del grafo, a lo sumo una arista de M incida sobre v). El algoritmo de Edmonds y Johnson [**8**] lo resuelve en tiempo polinomial. Por otro lado, si es dirigido, se lo resuelve como un problema de flujo mínimo (es decir, dado un grafo con costos, y una función de capacidad, obtener una función, o flujo, que a cada arista le asigne un valor que no supere la capacidad, y de manera tal que se minimice el costo del camino). Si el grafo es mixto, en cambio, la complejidad pasa a ser NP-hard. Las heurísticas para resolverlo consisten en encontrar un grafo de aumento de costo bajo, para satisfacer las condiciones de eulerianidad. Los mejores algoritmos, en este caso, utilizan la técnica de branch and cut para resolverlo.

Comencemos estudiando la estrategia de resolución sobre un grafo no dirigido.

CPP en un grafo no dirigido: Algoritmo de Edmonds-Johnson de matching

Para resolver la versión no dirigida, primero se lo lleva a una formulación de matching.

Comencemos definiendo la matriz $(a_n), i \in V, e \in E$ de incidencias del nodo, de la siguiente manera:

$$a_{ie} = \begin{cases} 1 & \text{si la arista e incide en el nodo i} \\ 0 & \text{si no} \end{cases}$$

Como ya dijimos, la idea del problema es encontrar enteros $x_e \geq 0, e \in E$, tal que minimice el costo del tour, y que se cumpla:

$$\sum_{e \in E} a_{ie}(1 + x_e) \equiv 0 \pmod 2$$

Esta congruencia también puede escribirse como :

$$\sum_{e \in E} a_{ie} x_e \equiv \sum_{e \in E} a_{ie} \pmod 2$$

El término de la derecha representa el grado del nodo n. Definimos entonces:

$$b_i \equiv \sum_{e \in E} a_{ie} \pmod 2$$

que representa la paridad del nodo.

Teniendo en cuenta esto, el problema puede reformularse como:

$$\text{Minimizar} \sum_{e \in E} c_e x_e$$

sujeto a:

$\sum_{e \in E_i} a_{ie} x_e - 2w_i = b_i \ \forall i \in V$

$\mathrm{x}_e \in \mathbb{N}_0, e \in E;$

$\mathrm{w}_i \in \mathbb{N}_0, i \in V;$

Se asume que los costos c_e no son negativos, de manera tal que si todos los b_n valen cero, entonces la solución óptima está dada por todas las x's y w's también valiendo cero. Esto es coherente con el teorema visto anteriormente, que dice que si todos los nodos tienen grado par, entonces el grafo tiene un tour euleriano.

También existe una formulación alternativa que es útil para los grafos ralos. Para eso definimos $E(S) = \{(v_i, v_j) : v_i \in S, v_j \in V \setminus S \text{ ó } v_i \in V \setminus S, v_j \in S\}$, con S subconjunto cualquiera de V. Luego el problema es:

$$\text{Minimizar} \sum_{(v_i, v_j) \in E} c_{ij} x_{ij}$$

sujeto a:

$\sum_{(v_i, v_j) \in E(S)} x_{ij} \geq 1$ ($S \subset V, S$ tiene un número impar de vértices con grado impar.)

$\mathrm{x}_{ij} \geq 0 \ ((v_i, v_j) \in E)$

$\mathrm{x}_{ij} \in \ \mathbb{Z} \ ((v_i, v_j) \in E)$

Acá x_{ij}, con $i < j$, es la cantidad de veces extra que se replica la arista (i, j). Además, la primera de las restricciones se denomina desigualdad de blossom.

Volviendo al problema, el mismo se resuelve mediante una adaptación del algoritmo de Edmonds y Johnson:

1. Separar los nodos de grado impar, que se se denotan $V' \subset V$.

2. Para cada par de nodos calcular el camino más corto entre ellos usando el algoritmo de Dijkstra o cualquier otro que calcule el camino más corto. De esta manera, construir el grafo completo $G' = (V', A')$ donde el costo de cada arista de A' es la distancia de dicho camino.

3. Resolver en G' el problema de matching perfecto de costo mínimo con el algoritmo de blossom, y denotar M a la solución óptima. Finalmente, agregar a G las aristas ficticias correspondientes a las aristas de M en G.

Una vez que se obtiene el grafo euleriano, resta determinar un ciclo euleriano en G'. Esto se realiza fácilmente mediante el algoritmo de Fleury o Hierholzer. Pero antes de llegar a ese punto, empecemos describiendo el algoritmo de blossom.

Algoritmo de blossom

La idea del algoritmo es ir mejorando un matching inicial vacío a través de caminos de aumento, de manera iterativa. A diferencia del matching en grafos bipartitos, la clave ahora es contraer todos los ciclos impares en un único vértice, y continuar la búsqueda en este grafo contraído. Comenzamos definiendo un *blossom* como un ciclo en G que consiste de $2k+1$ aristas de las cuales exactamente k pertecen al matching M, y están dispuestas de manera alternada con aristas que no son de matching.

Consideramos el grafo de superficie G_s, compuesto por nodos y pseudonodos. Estos últimos son aquellos nodos con grado impar en el grafo original G. Definimos un **pseudonodo deficiente** como aquel que no es incidente a ninguna arista de matching, es decir aquella cuyo valor asociado x_e es 1.

Por otro lado, consideramos el bosque F inducido por G_s. Inicialmente está compuesto por los nodos de G_s, excepto que los nodos impares originales ahora se reemplazan por pseudonodos deficientes con las mismas aristas incidentes. En general, F consta de árboles disjuntos, cada uno de ellos con un pseudonodo deficiente. Con respecto al resto de los nodos de F, estos serán pseudonodos y se alternarán entre nodos exteriores e interiores. En particular, los pseudonodos deficientes serán exteriores, como se verá en los siguientes ejemplos. En los gráficos cada pseudonodo está dado por un cuadrado, las aristas de matching, por líneas zigzagueantes, los interiores, por un signo -, y los exteriores por un signo +.

Cada nodo interiores es incidente a dos aristas, una de las cuales es de matching; mientras que los nodos exteriores son incidentes a un número cualquiera de aristas que no son de matching, pero sólo a una que sí lo es. Sin embargo, esto no ocurre con el pseudonodo deficiente (que es la raíz del árbol), el cual no es incidente a ninguna arista de matching.

Sea c_e y c'_e el costo y el costo revisado de la arista e, respectivamente. Este último comienza siendo igual al costo original. Por otro lado, a cada nodo n de G_s se le asocian tres números distintos: d_n^+, d_n^-, y_n. Inicialmente, $d_n^+ = +\infty$ e $y_n = 0$, salvo si n es un pseudonodo deficiente, en cuyo caso, $d_n^+ = 0$. Por su parte, d_n^- está dado por:

$$d_n^- = min\{c'_e : e \text{ es incidente a un nodo exterior } (\neq n) \text{ y } e \text{ toca a } n\}$$

Sea k_n la arista e donde se realiza el mínimo. Si la misma no existe, entonces $d_n = +\infty$ y $k_n = 0$.

Una vez definido esto, el algoritmo se divide en tres pasos:

Paso 1: Hallar el mínimo, sobre todos los nodos n en G_s, de:

$\alpha)$ d_n^-, si n no está en ningún árbol inducido.

$\beta)$ $\frac{1}{2}(d_n^+ + d_n^-)$, si n es un nodo exterior de algún árbol inducido.

$\gamma)$ $y_n + d_n^-$, si n es un nodo interior de algún árbol inducido.

Sea d^* el mínimo valor dado por el nodo i. Dependiendo de qué tipo de nodo sea este, es decir, si el mínimo se da en $\alpha), \beta)$ o $\gamma)$, pasar a $A), B)$ o $C)$ respectivamente.

Caso A) Se separa en casos dependiendo la naturaleza de i:

(1) Si i es un pseudonodo, entonces se convierte en un pseudonodo interior y se une a un árbol inducido a través de la arista k_i. Al mismo también se le agrega la arista de matching m. Sea j el otro nodo incidente a la arista m.

(1) a) Si j es un pseudonodo, entonces j será un nodo exterior. Cambiar d_j^+ por d^*, escanear el nodo j y volver al paso 1. Esto último significa mirar cada arista $e \in G_s$ que no es de matching (i.e. $x_e = 0$) y comparar d_n^- con $d_j^+ + c'_e$, donde n es el otro nodo de G_s incidente a e. Si $d_j^+ + c'_e < d_n^-$, entonces fijamos $k_n = e$ y $d_n^- = d_j^+ + c'_e$.

(1) b) Si j es un nodo original, entonces formamos un blossom B del nodo j y todas las aristas de matching, junto con los nodos incidentes al nodo j. A continuación, se contrae el blossom para formar un nuevo pseudonodo p. De esta manera se reemplaza G_s por G'_s, donde las aristas de este último, son las aristas del primero que no inciden en dos nodos de B, y los nodos son todos los de G_s, menos los de B, que son sustituidos por p. Al concentrar estos nodos allí, es necesario agregar aquellas aristas que tocaban exactamente un nodo de B, que ahora serán incidentes a p en G'_s.

De esta forma, el pseudonodo p se convierte en un nodo exterior, y fijamos $d_p^+ = d^*$, $d_p^- = +\infty$ y $y_p = 0$. Al igual que antes, escaneamos p, y finalmente volvemos al paso 1.

(2) Si i es un nodo original, fijamos $k = k_i$, y llamamos n al otro nodo incidente a la arista k. Formamos un blossom B del nodo n, la arista k, el nodo i y todas las aristas de matching que lo tocan, junto con todos los nodos incidentes a esas aristas.

Al igual que en (1) b), reemplazamos G_s contrayendo B a un pseudonodo p. El mismo es un nodo exterior, y establecemos $d_p^+ = d^*, y_p = 0$. Si n es un pseudonodo deficiente, entonces también lo será p. Para cada arista e de G (esté en G o no) que toque al nodo n, se resta $d^* - d_n^+$ del costo reducido c'_e. También cambiamos y_n por $y_n + d^+ - d_n^+$. A c_k, donde $k = k_i$ le otorgamos el valor de $d^* - d_n^+$. Finalmente, al igual que antes, se escanea p y se vuelve al paso 1.

Caso B) Sea j el otro nodo incidente a la arista k_i. Si i y j están en dos árboles distintos, ir al paso 2. De lo contrario, como i y j están en el mismo árbol, la arista k_i forma un circuito en dicho árbol.

Como ambos nodos son exteriores, hay un camino alternado a la raíz del árbol (es decir, alternando una arista de matching con otra que no lo es, o viceversa). Estos caminos se encuentran en algún nodo exterior m. Luego el circuito formado debe tener una cantidad impar de aristas.

La idea entonces, es formar un blossom B de los nodos y aristas en el circuito impar, y luego contraerlo en un pseudonodo p y reemplazar G_s como se había indicado en los otros pasos.

El pseudonodo p es exterior, y ahora $d_p^+ = d^*$ e $y_p = 0$. Si el pseudonodo m es deficiente, también lo será p. Ahora, para cada nodo n de B, es decir del circuito impar, y_n y c'_e se actualiza de la siguiente forma: $y_n = y_n + \Delta_n$, donde $\Delta_n = d^* - d_n^+$ para n nodo exterior, y $\Delta = -d^* + d_n^-$ para n nodo interno (del árbol que se tenía antes de achicar B). Además restamos Δ_n de cada costo reducido c'_e con e incidente al nodo n, y finalmente escaneamos p y volvemos al principio del paso 1.

Caso C) Este caso se alcanza con un pseudonodo interno i, para el cual $d^* = y_i + d_i^-$.

(1) Empezamos agregando y_i a cada arista e que toca el pseudonodo i, y cambiamos y_i por 0.

(2) Expandimos el pseudonodo i, es decir, recuperamos el blossom B que se había contraído para formar a i, y también las aristas que no estaban en B, pero que eran incidentes a dos nodos de B. Todos los nodos de B pueden llegar a ser pseudonodos, pero ninguno de ellos está expandido. A diferencia de antes, ahora definimos G_1 tal que $G_s = G_1 \setminus B$, y reemplazamos G_s por G_1.

(3) Algunas aristas de matching y otras que no lo son puede que tengan que intercambiarse para que alguna arista de matching sea incidente a cada uno de los pseudonodos de B. Este cambio se hace igual que en el paso 3 cuando los pseudonodos se expanden, con la diferencia que acá no se lo hace con todos los pseudonodos de B, sino que solamente con i.

(4) En este punto, el árbol incluye tanto de B como es posible. Esta parte de C), al igual que A) se divide en distintos casos. Sin embargo, en todos ellos hay una arista que no es de matching, que apunta hacia la raíz del pseudonodo i y una que sí lo es, ubicada desde la raíz que toca a i. Ambas aristas inciden en nodos exteriores.

(4) a) El blossom puede consistir en un circuito impar como en la imagen 3.1. En este caso, siempre hay un camino alternado desde la arista que no es de matching a la que sí lo es. El mismo se vuelve una parte del árbol con los nodos alternados entre interiores y exteriores, d_n^+ para nodos exteriores n igual a d_i^-, y d_n^- para interiores también igual a d_i^-. Estos nuevos nodos exteriores también tienen que ser escaneados.

El resto de B, es decir la parte del circuito que no está en el camino alternado, está formado por aristas de matching que tocan dos pseudonodos. Para cada uno de ellos, se define:

$$d_n^- = \min \{d_i^* + c'_e\text{: } e \text{ es incidente a un nodo exterior } i \text{ y a } n\}$$

Sea k_n la arista e en donde se alcanza el mínimo. Si la misma no existe, $d_n^- = \infty$ y $k_n = 0$. Como puede observarse, esta parte es similar a lo que se hizo antes del paso 1. A continuación, debemos volver al paso 1.

(4) b) El segundo tipo de blossom B tiene un nodo original y todas aristas de matching como en 3.2. En este caso, convertimos los dos nodos exteriores, junto con los de B y sus aristas, en un nuevo nodo exterior, como se hizo en A) 2). Para cada uno de los dos nodos

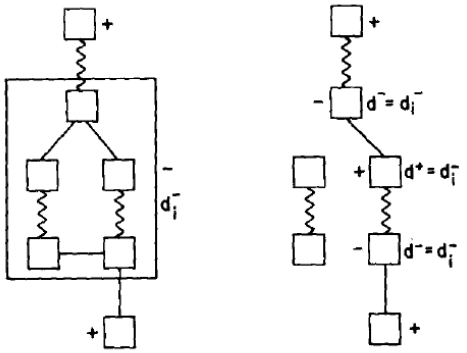


Figura 3.1: Blossom 4) a)

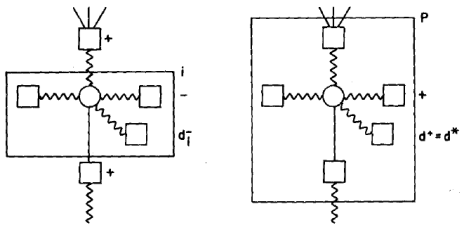


Figura 3.2: Blossom 4) b)

exteriores, se le resta $d^* - d_n$ a c'_e a cada arista incidente E y se le agrega $d^* - d_n$ a y_n. El nuevo pseudonodo p debe ser exterior, con $d_p^+ = d^*$. A continuación, escaneamos el nodo p y volvemos al principio del paso 1.

(4) c) El tercer tipo de blossom B tiene un nodo original y una arista que no es de matching como en 3.3. En este caso, formamos un nuevo pseudonodo del nodo externo que se dirige hacia la raíz, junto con la arista del nodo original, y las de matching incidentes. El otro nodo j de B se convierte en un nodo interior con $d_j^- = d^*$. El resto de este paso es exactamente el mismo que en A) 2).

Paso 2: Se alcanza cuando $d^* = \frac{1}{2}(d_j^+ + d_i^-)$ para algún nodo exterior i, con arista k_i incidente a un nodo j que está en otro árbol inducido.

A) Hay un camino de aumento que incluye a k_i y al bosque inducido, es decir, un camino alternado con aristas de matching y no matching, de manera tal que los extremos del camino son pseudonodos deficientes. Esta situación se da en general en todos los árboles

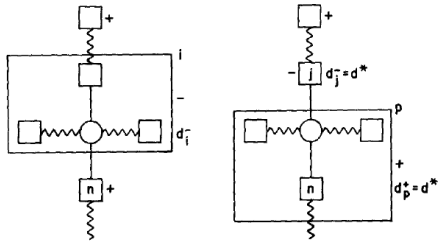


Figura 3.3: Blossom 4) c)

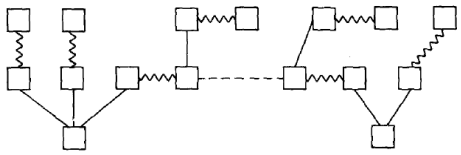


Figura 3.4: Camino de aumento

inducidos: siempre hay un camino alternado entre nodos exteriores y la raíz. En 3.4 puede observarse con más claridad. Se aumenta el camino al intercambiar las aristas de matching y no matching como indica la línea en la imagen. Después de aumentar, los pseudonodos deficientes dejan de serlo. Entonces, se borran los dos árboles del bosque.

(B) Para cada nodo $n \in G_s$, si tanto d_n^+ como d_n^- son mayores o iguales que d^*, dejamos y_n como está. Si ese no es el caso, entonces n está en un árbol inducido. Luego, si n es un nodo exterior, entonces $d_n^+ < d^+$.

Reemplazamos y_n por $y_n + (d^* - d_n^+)$ y le restamos $d^* - d_n^+$ a c'_e, donde e es toda arista incidente al nodo n. Si este es interior, entonces $d_n^- < d^*$. Luego reemplazamos y_n por $y_n - (d^* - d_n^-)$, y agregamos d* $- d_n-$ a c'_e para toda arista que incide en n.

Si no hay más pseudonodos deficientes, ir al paso 3. De lo contrario, para cada nodo n, calcular:

$$d_n^- = \{\min_e d_i^+ + c'_e \ e \text{ incide en un nodo exterior } (\neq n) \text{ y que incide en } n\}$$

Sea k_n la arista e en donde se alcanza el mínimo. Si la misma no existe, entonces fijamos $d_n^- = +\infty$ y $k_n = 0$. Volver al paso 1.

Paso 3: Ahora recuperamos una solución óptima especificando cuál x_e debe valer 1. Para las aristas e en G_s, fijamos $x_e = 1$ si es una arista de matching. Resta entonces expandir los pseudonodos de G_s para recuperar los blossoms y determinar qué aristas del mismo deberían tener $x_e = 1$. Existen tres tipos de blossom:

(A) Un blossom puede consistir de un ciclo impar como en la figura 3.5. Cuando se formó, el nodo 1 era o bien deficiente, o incidente a una arista de matching que tocaba a un único nodo del blossom. Aunque cualquier nodo puede ser incidente a dicha arista, sólo uno lo será. Para cada nodo, hay un camino alternado, que empieza por una arista de matching, hasta llegar al nodo 1. Por lo tanto, podemos cambiar las aristas de matching de manera tal que cada nodo toque exactamente una de ellas. De esta manera fijamos, para cada una de las aristas de matching del blossom, $x_e = 1$. Ahora los pseudonodos del blossom pueden ser expandidos.

(B) La segunda forma del blossom es un único nodo original junto con algún número de aristas de matching que lo alcanzan, como ocurre en 3.6. Cuando se formó, el nodo original era, o bien un nodo impar incidente a una arista que no era de matching, o incidente a una que sólo tocaba ese nodo del blossom. Si la arista de matching que toca el pseudonodo debe tocar al nodo original, entonces dejamos que las aristas del blossom sigan siendo de matching. Sin embargo, si la arista de matching incide en un pseudonodo como en 3.6, entonces cambiamos la arista de blossom que toca el pseudonodo a una arista que no es de matching como en 3.7. Ahora los pseudonodos del blossom pueden ser expandidos.

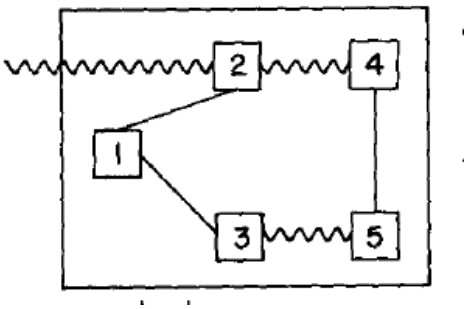


Figura 3.5: Ejemplo de Blossom

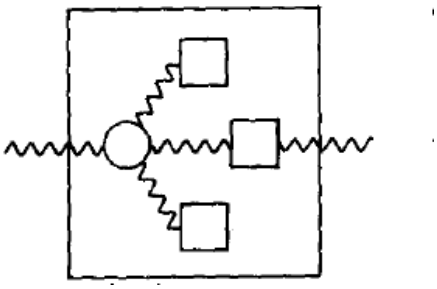


Figura 3.6: Blossom contraído con un único nodo original

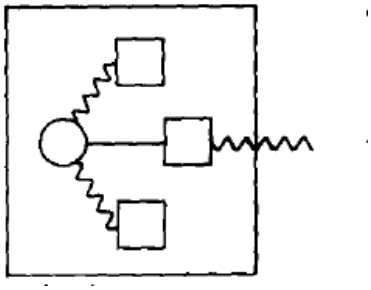


Figura 3.7: Cambios realizados en el paso 3 B)

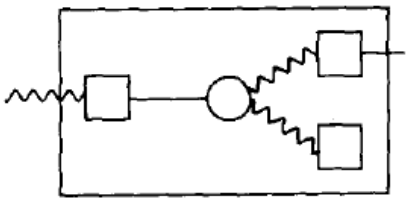


Figura 3.8: Blossom contraído cuyo nodo original toca a una arista que no es de matching

(C) La tercera forma de blossom se muestra en 3.8. El nodo original toca a una arista que no es de matching, y algún número de aristas que sí lo son. El pseudonodo que no es incidente a la arista de matching del blossom, era o bien deficiente, o incidente a una arista de matching que tocaba sólo un nodo del blossom al momento de formarse. Sin embargo, cualquier nodo ahora puede ser incidente a una arista de matching que toca el blossom. Si el nodo original era incidente a él, entonces cambiamos la arista que no era de matching del blossom a una que sí lo es. Si otro pseudonodo es tocado por ella, como se muestra en 3.8, entonces cambiamos las aristas de matching y no matching como se muestra en 3.9.

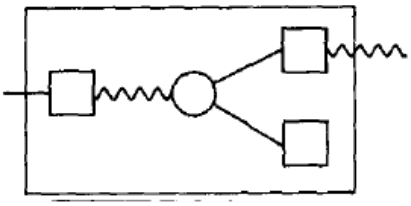


Figura 3.9: Cambios realizados en el paso 3 C)

De esta manera finaliza el algoritmo.

Algoritmo de Fleury

Veamos ahora cómo obtener el ciclo euleriano. Definimos primero un **puente** como una arista que al ser removida desconecta el grafo G.

El algoritmo consta de dos pasos:

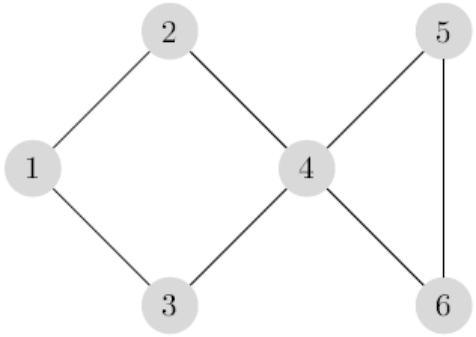
1. Empezar con un vértice v_i arbitrario, atravesar una arista (v_i, v_j) que no es un puente, y borrarla.

2. Fijar $v_i = v_j$ y repetir el paso 1 empezando por la otra extremidad de la arista borrada, o parar si todas fueron eliminadas.

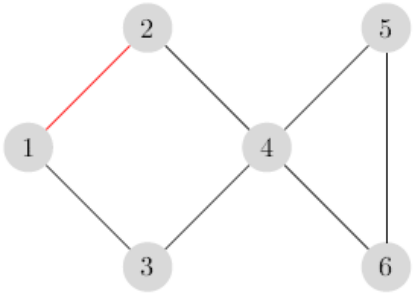
A pesar de ser aparentemente simple, puede insumir mucho tiempo porque resulta difícil determinar a cada paso si una arista es un puente o no. Esto hace que la complejidad sea cuadrática en la cantidad de aristas. A diferencia de otros algoritmos, como por ejemplo,

el de Hierholzer, que requiere que el grafo sea par, el algoritmo de Fleury, también sirve para el caso en que el grafo sólo tenga un camino euleriano. Sin embargo, hay que tener en cuenta que el algoritmo debe iniciarse en un nodo de grado impar.

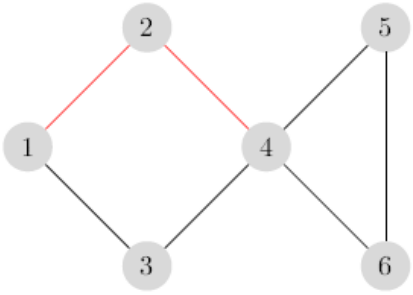
A continuación, veremos un ejemplo de aplicación en el siguiente grafo:



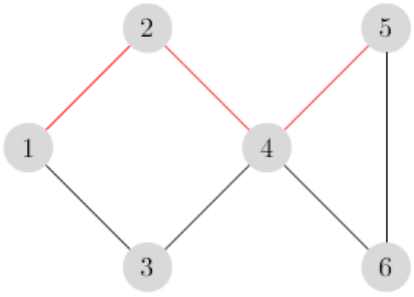
Comenzamos eligiendo un nodo inicial v_0. En este caso el grafo es par, por lo que es indistinto tomar cualquiera. Arbitrariamente, tomamos entonces el nodo 1. Para mayor entendimiento, en vez de borrar la arista del grafo, las iremos pintando de rojo a medida que son eliminadas. Como puede observarse, en este caso se puede comenzar eliminando la arista $(1, 2)$ o $(1, 3)$, ya que al hacerlo, el grafo sigue siendo conexo. Arbitrariamente elegimos la arista $(1, 2)$. Y tomamos $v_1 = 2$.



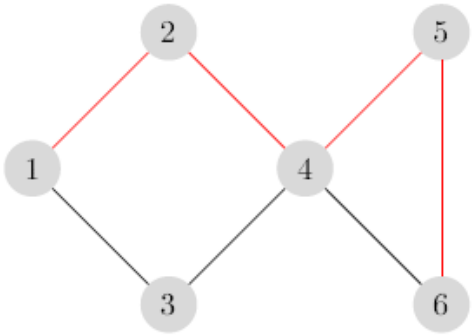
Como quedan aristas por eliminar, definimos $v_0 = 2$, y repetimos el proceso. Ahora sólo hay una arista incidente al nodo 2, por lo que eliminamos la arista $(2, 4)$, y tomamos $v_1 = 4$.

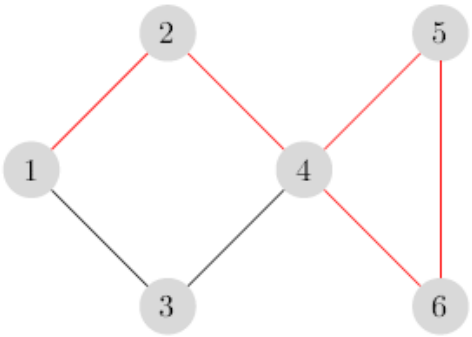


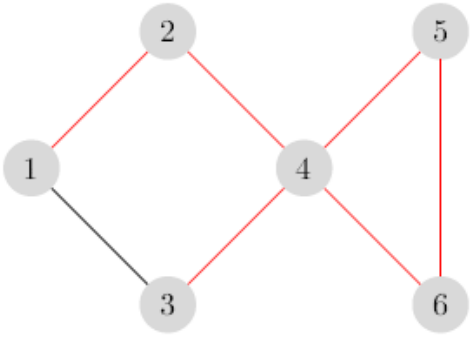
A continuación, tomamos $v_0 = 4$. Ahora tenemos tres aristas que tienen como nodo de origen al 4, pero hay que tener especial cuidado que al borrar la arista $(4, 3)$, el grafo deja de ser conexo salvo nodos aislados, por lo que la misma no puede ser elegida.

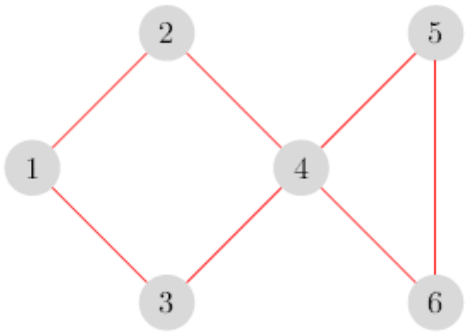


A partir de este momento, el algoritmo se convierte en trivial porque sólo hay una única arista candidata a ser eliminada en cada iteración. Por este motivo, sólo representaremos gráficamente los pasos realizados.









Finalmente, el tour euleriano serán las aristas en el mismo orden que las eliminamos:
$C = \{(1,2),(2,4),(4,5),(5,6),(6,4),(4,3),(3,1)\}$

Algoritmo de Hierholzer

El algoritmo de Fleury recién visto, tiene un inconveniente a la hora de determinar si al eliminar una arista al grafo, el mismo sigue siendo conexo salvo nodos aislados.

El algoritmo de Hierholzer, que tiene complejidad lineal en la cantidad de aristas, evita este problema y es más sencillo de programar. Consta de tres pasos:

1. Comienza tomando un nodo de inicio v_0, y construye un ciclo C_1, atravesando las aristas adyacentes al mismo tiempo que se van eliminando, de manera tal que el ciclo termine (obviamente en v_0). Si en este paso se logra conseguir un ciclo euleriano que incluya todas las aristas, se toma ese ciclo. De lo contrario, se continúa al paso siguiente.

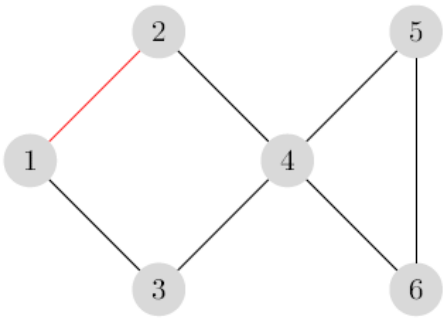
2. Se toma como nodo inicial cualquier v que esté en el ciclo C_1, y que todavía tenga aristas incidentes sin eliminar. De esta forma se obtiene el ciclo C_2.

3. A continuación, se fusionan ambos ciclos en uno solo. Este pasa a denotarse C_1. La forma más simple de fusionarlo, es tomar dos aristas $a_1, a_2 \in C_1$, ambas incidentes al nodo v, y luego colocar el ciclo C_2 entre dichas aristas.

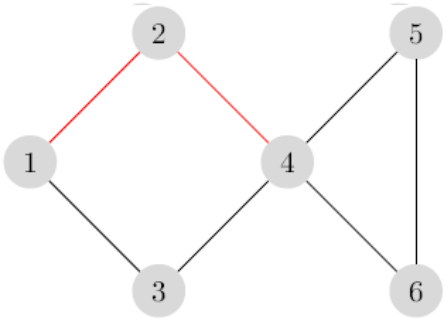
4. Una vez llegado a este paso, si todas las aristas ya están eliminadas, entonces el algoritmo está finalizado . Caso contrario, se vuelve al paso 2

A continuación, veremos un ejemplo sencillo del algoritmo con el mismo grafo utilizado en Fleury.

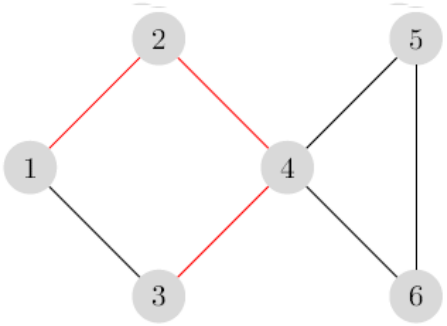
Comenzamos tomando $v_0 = 1$. Tenemos dos aristas incidentes en el nodo v_0: $(1,2)$ y $(1,3)$. Cualquiera de las dos puede elegirse para ser eliminada. En este caso, elegimos $(1,2)$. En el siguiente gráfico, la arista que acabamos de borrar aparece en color rojo.



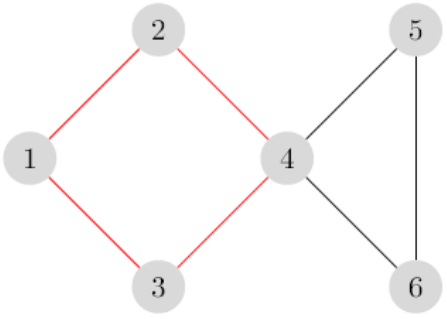
Volvemos a repetir el procedimiento, pero ahora con el nodo 2, y borrando la arista $(2, 4)$.



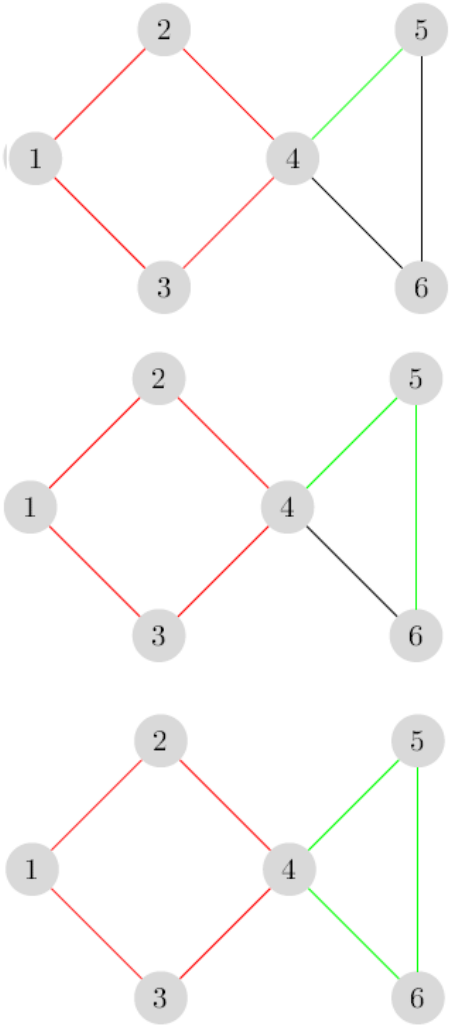
Ahora tenemos tres aristas adyacentes al nodo 4 que aún no fueron eliminadas: $(4, 3)$, $(4, 5)$ y $(4, 6)$. Si eliminàramos cualquiera de las últimas dos, también podría seguirse con el algoritmo y hacer un tour euleriano. Pero aquí elegiremos la arista $(4, 3)$:



En el paso siguiente, al eliminar la arista $(3, 1)$, ya tendremos un ciclo formado:



Ahora, entonces, ya estamos en el paso 2 del algoritmo, por lo que debemos elegir un nodo del ciclo con alguna arista incidente que no haya sido eliminada todavía. Vemos que la única opción posible es el nodo 4. Análogamente hacemos el mismo proceso que en el paso anterior, eliminando las aristas de a una:



Ahora tenemos dos ciclos: $C_1 = \{(1,2),(2,4),(4,3),(3,1)\}$ construido inicialmente, y el que acabamos de construir: $C_2 = \{(4,5),(5,6),(6,4)\}$. Luego, es necesario fusionar ambos ciclos, y actualizar C_1 con dicha fusión: $\{(1,2)(2,4),(4,5),(5,6),(6,4),(4,3),(3,1)\}$

Como todas las aristas están eliminadas, entonces el algoritmo ya está finalizado.

CPP en un grafo dirigido: Problema de costo mínimo

Como se dijo anteriormente, si el grafo es dirigido, para determinar el grafo G' se resuelve un problema de flujo mínimo. A diferencia de los grafos no dirigidos, ahora el hecho de que G sea conexo no es suficiente para garantizar una solución; sino que hay que pedir que sea fuertemente conexo. Esto es equivalente al hecho de que entre cualquier par de vértices exista un camino dirigido.

Sea I el conjunto de vértices v_i tales que el $d_{in}(v_i) = d_{out}(v_i)+s_i$, y J el conjunto de vértices tales $d_{in}(v_i) + d_i = d_{out}(v_i)$. Es decir, s_i y d_i pueden interpretarse como el suministro y la demanda, respectivamente. Por otro lado, sea c_{ij} la longitud del camino mínimo de v_i a v_j. Luego, el problema se puede plantear de la siguiente forma:

Minimizar $\sum_{v_i \in I} \sum_{v_j \in J} c_{ij} x_{ij}$

sujeto a:

$$\sum_{v_j \in J} x_{ij} = s_i \ (v_i \in I)$$

$$\sum_{v_i \in I} x_{ij} = d_j \ (v_j \in J)$$

$$\mathrm{x}_{ij} \geq 0 \ (v_i \in I, v_j \in J)$$

Al igual que antes, los valores óptimos x_{ij} representan la cantidad de veces extra que se atraviesa un arco. Una vez resuelto esto, se puede determinar el circuito euleriano adaptando el algoritmo de Fleury para grafos no dirigidos. Alternativamente, se puede aplicar el siguiente algoritmo de van Aardenne-Ehrenfest y Bruijn.

Algoritmo de van Aardenne-Ehrenfest y Bruijn

El mismo consta de tres pasos:

1. Construir el árbol generador dirigido (por ejemplo con Dijkstra) con un vértice v_r como raíz.
2. Etiquetar todas las aristas como sigue: comenzar por aquellas que salen de v_r, ordenándolas de manera arbitraria. Luego hacer lo mismo con los vértices consecutivos (segundo nivel), y así sucesivamente hasta llegar al último, pero siempre de manera tal que la última arista etiquetada de cada nivel sea la que se usa en el árbol.
3. Obtener un tour euleriano a partir de la arista etiquetada con menor orden. Cada vez que se entra a un vértice, se sale de él con una arista aún no recorrida que tiene el orden más chico. De esta manera se itera y finalmente cuando todas las aristas fueron recorridas, se obtiene el tour de Euler buscado.

CPP en un grafo mixto

Para esta sección, consideraremos $G = (V, A \cup E)$, donde A es el conjunto de arcos (aristas dirigidas) y E el conjunto de aristas (sin orientación). Agregamos también dos nuevas definiciones.

Definición 3.1.11. Un grafo es **simétrico** si para cada vértice el número de arcos que entran es igual al número de arcos que salen.

Definición 3.1.12. Un grafo es **balanceado** si se cumplen las condiciones del conjunto balanceado, es decir, si para cada subconjunto $S \subseteq V$, la diferencia entre la cantidad de

arcos dirigidos de S a $V \setminus S$, y viceversa, es menor o igual al número de arcos no dirigidos que unen a ambos conjuntos.

Notar que si un grafo conexo mixto es par y simétrico, entonces también es balanceado, pero que la condición de simetría no es necesaria para que un grafo sea euleriano.

En este caso, para resolver el problema sobre un grafo mixto, hay que dividirlo en tres fases:

1. Asignar direcciones a algunas de las aristas para convertir a G en un grafo simétrico.
2. Asignar direcciones al resto de las aristas.
3. Determinar el tour de G.

Para la primera fase se puede aplicar el algoritmo propuesto por Ford y Fulkerson que veremos a continuación.

Procedimiento de Ford y Fulkerson para transformar un grafo mixto a simétrico

La idea de este algoritmo comienza reemplazando cada arista del grafo por un par de arcos con orientación opuesta entre sí. De esta manera obtenemos $G' = (V, A')$. Luego, para cada arco de $A' \cap A$, se asigna una cota inferior 1, mientras que para el resto de los arcos de A' la misma vale 0. Además para ambos conjuntos se designa una cota superior de 1. Estos límites se imponen para resolverlo como un problema de flujo.

En el paso siguiente, se usa un algorimo de flujo, como por ejemplo el de Edmonds y Karp, para determinar un camino factible en G'. Llamemos x_{ij} al flujo en el arco (v_i, v_j). Finalmente, se orientan algunas de las aristas de la siguiente forma: si $(v_i, v_j) \in E$ y $x_{ij} = 0$, entonces definimos la orientación v_i a v_j.

Una vez que el grafo es simétrico, pasamos a orientar el resto de las aristas con el siguiente algoritmo.

Procedimiento para orientar totalmente un grafo simétrico

Este algoritmo consta de los siguientes pasos:

1. Si todas las aristas están dirigidas, entonces listo.
2. Sea v un vértice con al menos una arista incidente no dirigida (v, w). Definamos $v_1 := v$ y $v_2 := w$.
3. Orientar (v_1, v_2) de v_1 a v_2. Si $v_2 = v$, ir al paso 1.
4. Fijar $v_1 := v_2$ e identificar una arista (v_1, v_2) incidente a v_1. Ir al paso 3.

De esta manera finaliza el procedimiento, y a partir del grafo simétrico, se puede usar el algoritmo de van Aardenne-Ehrenfest y de Bruijn, ya descripto para el caso de grafos dirigidos, para hallar el tour euleriano.

Con respecto al algoritmo de flujo máximo, este puede aplicarse a un grafo par sin necesidad de saber a priori si es euleriano. Si este algoritmo falla, entonces significa que el grafo no lo es, y por lo tanto hay que replicar las aristas hasta que lo sea. Cabe destacar que no en todos los grafos se puede realizar un aumento de costo mínimo (como ocurre en grafos que no son fuertemente conexos), pero para los casos en los que sí se puede, existen modelos de programación lineal entera que pueden utilizarse para esta tarea.

Uno de ellos, por ejemplo, fue propuesto por Christofides (1984). Comencemos definiendo las variables y parámetros que se emplearán en la formulación. Sea $A_k^+ = \{(v_i, v_j) \in A : v_i = v_k\}$, $A_k^- = \{(v_i, v_j) \in A : v_j = v_k)\}$, y V_k el conjunto de todos los vértices que se conectan a v_k a través de alguna arista. Por otro lado, sea x_{ij} la cantidad de veces extra que se recorre el arco (v_i, v_j) en la solución óptima, y y_{ij} el número total de veces que la arista (v_i, v_j) se recorre de v_i a v_j. Finalmente, sea p_k una constante binaria que vale 1 si y sólo si el grado del vértice v_k es par, y z_k una variable entera. Ahora se está en condiciones de determinar la formulación del problema:

Minimizar $\sum_{(v_i,v_j)\in A} c_{ij}(1 + x_{ij}) + \sum_{(v_i,v_j)\in E} c_{ij}(y_{ij} + y_{ji})$

sujeto a:

(1) $\sum_{(v_i,v_j)\in A_k^+} (1 + x_{ij}) - \sum_{(v_i,v_j)\in A_k^-} (1 + x_{ij}) + \sum_{v_j\in V_k} y_{kj} - \sum_{v_j\in V_k} y_{jk} = 0 \ (v_k \in V)$

(2) $\sum_{(v_i,v_j)\in A_k^+} x_{ij} + \sum_{(v_i,v_j)\in A_k^-} x_{ij} + \sum_{v_j\in V_k} (y_{kj} + y_{jk} - 1) = 2z_k + p_k \ (v_k \in V)$

(3) $\mathrm{y}_{ij} + y_{ji} \geq 1 \ ((v_i, v_j) \in E)$

(4) $\mathrm{z}_k, x_{ij}, y_{ij}, y_{ji} \geq 0, \ \in \mathbb{Z}$

Este problema se resuelve con un algoritmo de enumeración, en el que se computan dos cotas inferiores en cada uno de los nodos del árbol. La primera cota se obtiene relajando la primera restricción de manera lagrangiana, y resolviendo el matching de mínimo costo. La segunda, en cambio, se obtiene relajando la restricción (3) y resolviendo el problema de flujo de mínimo costo.

Dado que el problema es NP-hard, existen heurísticas que permiten resolverlo en tiempo polinomial. Entre ellas, puede mencionarse una propuesta por Frederickson, con complejidad $O(max\{|V|^3, |A|(max\{|A|, |E|\})^2\})$:

1. Usando un algoritmo para encontrar el flujo de costo mínimo, convertir el grafo en uno simétrico, y denominarlo $G' = (V, A' \cup E')$
2. Identificar el conjunto de vértices impares en $G'' = (V, E')$, y luego encontrar todos los caminos mínimos entre pares de vértices sobre el grafo $\hat{G} = (V, E)$. Realizar

un matching de costo mínimo de todos los vértices de $\hat{G}$ usando las distancias de cada camino mínimo. Finalmente agregar las aristas usadas en el matching a E' y encontrar el tour sobre A' y E'.

Sin embargo, las heurísticas aplicadas al CPP mixto, a diferencia de las aplicadas a otros problemas como el TSP, no son tan eficientes. Por ejemplo para instancias con $7 \leq |V| \leq 50, 3 \leq |A| \leq 85$ y $4 \leq |E| \leq 39$, los resultados en promedio están a un 3 % del valor óptimo, pero en el peor de los casos se encuentra a un 17 % del mismo.

3.1.4. Problema del cartero rural (RPP)

En la práctica, la mayoría de los casos no requiere pasar por todos los arcos, por lo que suelen modelarse como un Problema del Cartero Rural en lugar del Cartero Chino. Como ya se mencionó antes, se tiene el conjunto $R \subset E$ de aristas por las cuales se debe pasar.

A diferencia del problema del cartero chino, el RPP es NP-hard, tanto en su versión dirigida, no dirigida como mixta.

La estrategia para resolverlo es determinar primero un grafo de aumento para hacerlo par y luego hallar un camino euleriano. Dada su complejidad, para resolver la primera etapa suelen utilizarse heurísticas, que por lo general involucran algoritmos de matching o de creación de un árbol generador mínimo.

RPP no dirigido

Si el grafo es conexo, el problema puede resolverse obteniendo caminos mínimos entre vértices de grado impar. Los mismos pueden tener aristas que están en $E \setminus R$. Luego de esto, puede tratarse al igual que el problema del cartero chino. En general, el RPP se puede resolver en un grafo modificado $G' = (V', E')$, con $V' = \{v_i \in V : (v_i, v_j) \in R$ para algún $v_j \in V\}$, y E' obtenido de la siguiente forma: primero se agrega a E una arista (v_i, v_j) para cada $v_i, v_j \in V'$ cuyo costo c_{ij} es igual a la longitud del camino mínimo entre esos vértices; luego se borran las aristas $(i, j) \in E \setminus R$ para las cuales $c_{ij} = c_{ik} + c_{kj}$ para algún k, así como también una de las aristas entre c_{ij} y c_{ji}, si ambas tienen el mismo costo.

En la figura 3.10, pueden observarse los grafos correspondientes a G y G'. Allí, las aristas de R están representadas con mayor grosor, mientras que los números indican los costos. En G', el conjunto R induce p componentes conexas $G_1, \ldots G_p$, con sus respectivos conjuntos de vértices $V_1, \ldots V_p$, que conforman una partición de V'.

Si los costos satisfacen la desigualdad triangular, se tiene una heurística con complejidad $O(n^{3/2})$, la cual fue creada por Frederickson [**9**], pero que se basa en la desarrollada en 1976 por Christofides [**10**] para resolver el TSP simétrico. A continuación detallaremos la misma:

1. **Paso 1: Árbol generador mínimo** Generar un árbol T conectando $G_1 \ldots G_p$, y sea $l(T)$ su longitud. Fijar $l(R)$ la suma de las longitudes de todas las aristas de R y sea z^* el valor óptimo de la solución del RPP, entonces se ve fácilmente que $l(T) + l(R) \leq z^*$

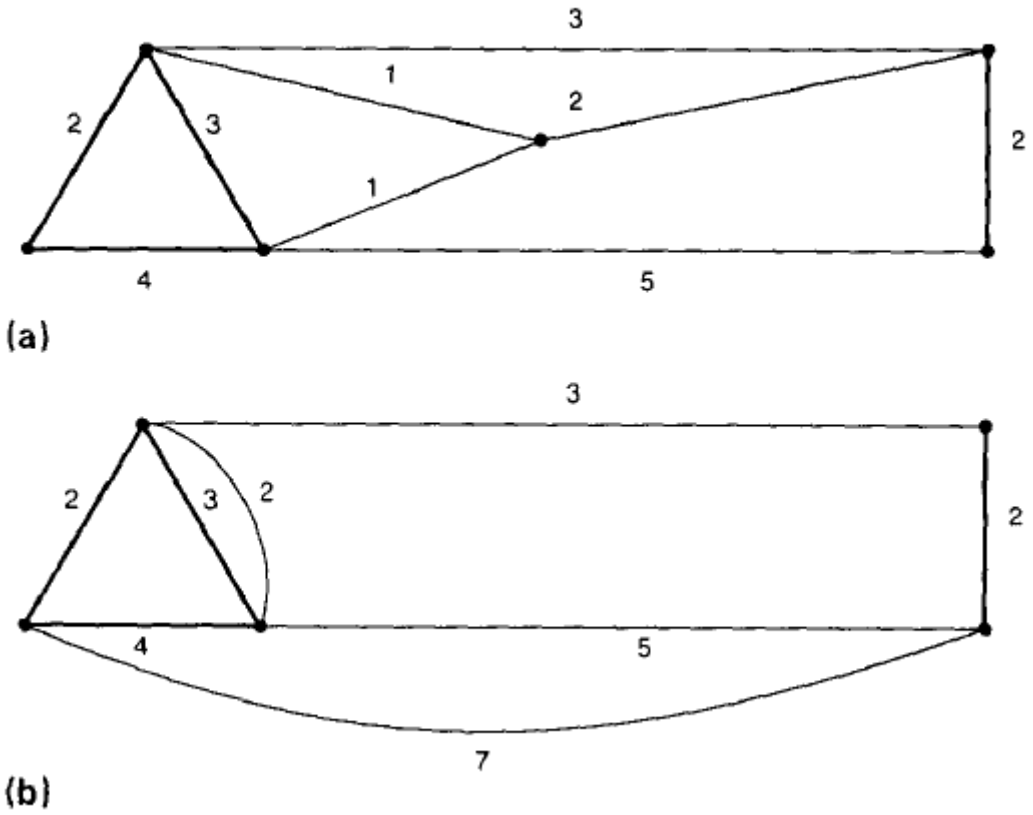


Figura 3.10: Antes y después de generar el grafo de aumento

2. **Paso 2: Matching de mínimo costo** Determinar un matching con todos los vértices de grado impar del grafo inducido por $R \cup T$. Notemos M al conjunto de aristas incluidas en el matching óptimo, y l(M) su longitud total.

3. **Paso 3: Ciclo Euleriano** Una solución para el RPP está dada por un ciclo euleriano en el grafo formado por $R \cup T \cup M$. Si los costos satisfacen la desigualdad triangular, se puede probar, mediante el mismo argumento que en Christofides (1976), que $l(M) \leq z^*/2$. Por lo tanto, la longitud $l(T) + l(R) + l(M)$ del ciclo euleriano no excede $\frac{3z^*}{2}$.

Además, existen dos formulaciones de programación lineal entera que se propusieron para resolver el problema. La primera de ellas fue ideada por Christofides et. al. Allí, las variables x_{ij} se definen de la siguiente forma: si $(v_i, v_j) \in R$, entonces la variable representa la cantidad de veces que esa arista es replicada en la solución óptima. En cambio, si $(v_i, v_j) \in A' \setminus R$, entonces representa la cantidad de veces que la arista es atravesada.

Formulación 1

Minimizar $\sum_{(v_i,v_j)\in R} c_{ij}(1 + x_{ij}) + \sum_{(v_i,v_j)\in'} c_{ij}x_{ij}$

sujeto a:

(1)El grado de cada vértice debe ser par.
$\sum_{(v_i,v_j)\in R,j>1} (1+x_{ij}) + \sum_{(v_i,v_j)\in Rj<i} (1+x_{ji}) + \sum_{(v_i,v_j)\in A',j>i} x_ij + \sum_{(v_i,v_j)\in A'\setminus R,j<i} x_{ji} \equiv 0 \pmod 2$ $(v_i \in V')$

(2) En el ciclo óptimo todas las componentes conexas deben estar conectadas.
$\sum_{v_i\in S, v_l\in \bar{S}} x_{ij} \geq 1 \left(S = \bigcup_{k\in P}, V_k, \bar{S} = \left(\bigcup_{k=1}^{p} V_k \right) \setminus S, P \subset \{1, \ldots, p\} \right)$

(3)Naturaleza de las variables.
$x_{ij} \in \mathbb{N}_0$ $(v_i \in I, v_j \in A')$

También se puede considerar la restricción $(1)'$ obtenida reemplazando la parte derecha de la igualdad por $2z_i$ con $z_i \in \mathbb{N}_0, v_i \in V'$.

Formulación 2

Fue propuesta por Corberán y Sanchis (1991). Para eso, se define $A_i = \{(v_i, v_j) \in A'\}$ como el conjunto de aristas incidentes a $v_i \in G'$. Un vértice se denomina R-par (o R-impar

respectivamente), si es incidente a una cantidad par (respectivamente, impar) de aristas de R. Las variables x_{ij} se definen igual que antes. Luego, la formulación queda:
Minimizar $\sum_{(v_i,v_j)\in R} c_{ij}(1+x_{ij}) + \sum_{(v_i,v_j)\in A'\setminus R} c_{ij}x_{ij}$

sujeto a

(1) $\sum_{(v_i,v_j)\in A_i} (x_{ij}) \equiv 0 \pmod 2$ ($v_i \in V'$ v_i es R-impar)

(2) $\sum_{(v_i,v_j)\in A_i} (x_{ij}) \equiv 1 \pmod 2$ ($v_i \in V'$ v_i es R-impar)

(3) $\sum_{v_i\in S, v_l\in\bar{S}, o v_i\in\bar{S}, v_j\in S} x_{ij} \geq 2 \left(S = \bigcup_{k\in P}, V_k, \bar{S} = \left(\bigcup_{k=1}^{p} V_k\right) \setminus S, P \subset \{1,\dots,\}\right)$

(4) $\mathrm{x}_{ij} \in \mathbb{N}_0$ $(v_i, v_j) \in A'$

RPP dirigido

El problema se reduce a uno del cartero chino dirigido cuando $\bar{G} = (V, R)$ es conexo. En el caso general, el problema se resuelve en el grafo modificado $G' = (V', E')$ construido como en el caso no dirigido, salvo que los arcos (v_i, v_j) que se agregan ahora sí son dirigidos y con el peso correspondiente a un camino mínimo.
Nuevamente Christofides propuso una heurística para resolverlo:

1. **Bosque generador mínimo** Construir el bosque con raíz aleatoria, y conectar $G_1 \dots G_p$ definidos como antes. $\bar{G}$ es el grafo resultante.
2. **Problema de transporte** Al igual que en el problema del cartero chino, obtener un grafo euleriano de $\bar{G}$ agregando arcos con el menor costo posible, de manera que la cantidad de vértices que entran y salen de cada vértice sea la misma.
3. **Circuito euleriano** Determinar este circuito en el grafo de aumento.

Este procedimiento puede repetirse usando como raíz un vértice distinto por vez.
También existe una formulación exacta propuesta nuevamente por Christofides. Sea:

$$b_{ij} = \begin{cases} 1 & \text{si } (v_i, v_j) \in R, v_i, v_j \in V' \\ 0 & \text{si no} \end{cases}$$

$$\bar{b}_{ij} = \begin{cases} 1 & \text{si } (v_i, v_j) \in A' \setminus R, v_i, v_j \in V' \\ 0 & \text{si no} \end{cases}$$

Luego el problema puede formularse con respecto a un conjunto particular de vértices $\bar{V} \in \{V_1, \dots V_p\}$:
Minimizar $\sum_{(v_i,v_j)\in R}(1+x_{ij}) + \sum_{(v_i,v_j)\in A'} x_{ij}$

sujeto a:

(1) El grado in y el grado out de cada vértice debe ser el mismo.
$$\sum_{j,(v_i,v_j)\in R}(1+x_{ij})b_{ij} + \sum_{j,(v_i,v_j)\in A'\setminus R} x_{ij}\bar{b}_{ij} = \sum_{j,(v_i,v_j)\in R}(1+x_{ij})b_{ij} + \sum_{j,(v_i,v_j)\in A'\setminus R} x_{ij} \ (v_i \in V')$$

(2) La solución forma un árbol dirigido con los nodos de la componente conexa inducida por $\bar{V}$, para todas las componentes conexas $G_1, \dots, G_p$.
$$\sum_{v_i\in S, v_l\in\bar{S} \text{ ó } v_i\in\bar{S}, v_j\in S} x_{ij} \geq 1 \ \left(S = \bigcup_{k\in P} V_k, \bar{S} = \left(\bigcup_{k=1}^{p} V_k\right) \setminus S, P \subset \{1, \dots, \}, \bar{V} \subset S\right)$$

(4) $\mathrm{x}_{ij} \in \mathbb{N}_0 \ (v_i, v_j) \in A'$

Este modelo se resuelve mediante un procedimiento de Branch and Bound, pero incorporando la restricción (1) en la función objetivo de manera lagrangiana. Es decir, sumando este término multiplicado por una constante.

3.2. Algoritmo de Jarvis March

Empecemos dando una definición:

Definición 3.2.1. La **envoltura o cápsula convexa** de un conjunto de puntos S está dado por: $e(s) = \{\sum_{i\in J} \lambda_i x^i \forall \{x^i\}_{i\in J} \subseteq S, \text{ J finito } ; \sum_{i\in J} \lambda_i = 1, \lambda_i \geq 0, i \in J\}$ Es decir, es el menor conjunto convexo que contiene a todos los puntos de S.

El algoritmo de Jarvis March sirve para hallar la cápsula convexa de un conjunto de puntos de $\mathbb{R}^2$. Tiene complejidad temporal $O(nh)$ donde $n = |S|$ y h es la cantidad de vértices de la cápsula convexa.
La idea detrás del algoritmo es tomar el punto que se encuentra más a la izquierda, y realizar una especie de barrido de los vértices, girando a la izquierda hasta que ya no sea posible, y tomar ese último vértice para volver a iterar el procedimiento, hasta que finalmente se cierre el polígono. En 3.11 se encuentra una imagen que grafica esta idea, mientras que en 1 se observa el pseudocódigo del algoritmo.

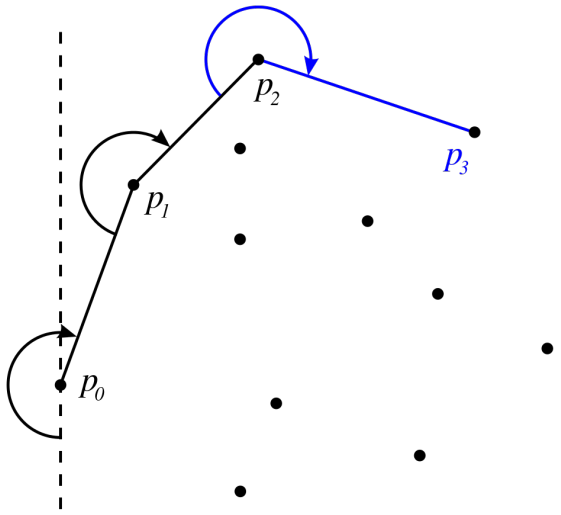


Figura 3.11: Idea del algoritmo de Jarvis March

Algoritmo 1 Algoritmo de Jarvis March

Entrada: S= {Conjunto de puntos}

```
1: puntoEnCapsula ← extremoIzquierdo(S)
2: i ← 0
3: mientras no se cierre el polígono de la cápsula convexa hacer
4:     P[i] ← puntoEnCapsula
5:     puntoFinal ← S[0]
6:     para j= 1, . . . , |S| hacer
7:         si puntoFinal == puntoEnCapsula o S[j]== CCW(S[j],P[i],puntoFinal) entonces
8:             puntoFinal ← S[j]
9:         fin si
10:    fin para
11:    i ← i + 1
12:    puntoEnCapsula ← puntoFinal
13: fin mientras
       devolver P
```

La función *extremoIzquierdo(S)* se encarga de encontrar el punto que se encuentra más a la izquierda entre todos los puntos de S, ya que el mismo va a estar en la cápsula convexa de S; mientras que CCW(s,p,f) devuelve Verdadero si s se encuentra girando a la izquierda del segmento que va desde p hasta f. Más adelante, veremos que este algoritmo fue usado para calcular una valuación en uno de los modelos de programación lineal entera.

3.3. Programación lineal

Un problema de programación lineal es de la forma:

$$\text{Maximizar: } z=\sum_{j=1}^{n} c_j x_j$$

$$\text{Sujeto a: } \sum_{j=1}^{n} a_{ij} x_j \leq b_i (i = 1, 2, \ldots, m)$$

$$x_j \geq 0 (j = 1, 2, \ldots, n)$$

Aquí, a_{ij}, b_i y c_j se denominan parámetros, mientras que x_j son las variables cuyo valor habrá que determinar. Por otro lado c_j se denomina costo de la variable x_j, mientras que $x^\star$ y $z^\star$ se utilizan para denotar al punto y valor óptimo, respectivamente.

Esta forma se denomina notación estándar en [**11**], aunque la misma suele depender de cada autor. En algunos casos, se lo toma como un problema de minimización, lo cual es equivalente a multiplicar todos los costos c_j por -1, y una vez resuelto, volver a multiplicar el valor de la función objetivo por -1.

Muchas veces este problema también se escribe en forma matricial:

$$\text{Maximizar: } c^T x$$

$$\text{Sujeto a: } Ax = b$$

$$x \geq 0 \ (j = 1, 2, \ldots, n)$$

En este caso $x \in \mathbb{R}^n$, $c \in \mathbb{R}^n$ y $b \in \mathbb{R}^m$ y $A \in \mathbb{R}^n \times \mathbb{R}^m$. Notar que aquí, la primera restricción es de igualdad, y no de desigualdad como en la forma estándar. Esto no supone ningún problema, ya que se puede llevar a la forma matricial agregando variables, llamadas de holgura, y reescribiendo las restricciones de desigualdad $a_{i1}x_1 + \ldots + a_{in}x_n \leq b_i$ como:

$$a_{i1x1} + \ldots + a_{in}x_n + x_{n+i} = b_i.$$

En cambio, si la restricción original es de la forma $a_{i1}x_1 + \ldots + a_{in}x_n \geq b_i$, la escribimos como :

$$a_{i1x1} + \ldots + a_{in}x_n - x_{n+i} = b_i.$$

En todos los casos, las variables de holgura toman valores positivos. Las mismas no tienen ninguna influencia en el valor de la función objetivo, por lo que si bien aumentan el tamaño del problema, brindan una solución equivalente fácil de reconstruir.

Otro aspecto a tener en cuenta es cuando la variable x_j toma valores negativos, o tiene la flexibilidad de ser irrestricta (es decir, puede tomar valores tanto positivos como negativos). En el primero de los casos, la restricción se reemplaza por la variable -x_k (que será

mayor que cero), y en el segundo se agregan dos variables nuevas y se reemplaza x_k por $x'_k - x''_k$.

Volviendo al problema, más allá de las distintas notaciones, lo más importante es captar la esencia: maximizar o minimizar una función lineal en sus variables, cuyas restricciones también lo sean.

En el caso de un problema de programación lineal entera se agrega la restricción de que todas o algunas de las variables x_j además sean enteras.

Aunque los dos problemas parecieran ser bastante similares, la metodología utilizada para resolver ambos no es la misma. Más aún, la complejidad es muy distinta entre uno y otro. Mientras que un problema de programación lineal (PL) con variables continuas puede resolverse en tiempo polinomial, el problema de programación lineal entera (PLE) es NP-hard.

Para el primer caso, se usan métodos como Simplex. Este algoritmo se va moviendo a través de los distintos vértices del conjunto de soluciones factibles $\{x \in \mathbb{R}n : Ax = b, x \geq 0\}$, de manera tal que la función objetivo aumente, siempre verificando si se cumplen las condiciones KKT de optimalidad, o que el problema no sea acotado. Esto último quiere decir que el conjunto de soluciones factibles no es acotado en la dirección de crecimiento de la función objetivo (la del gradiente, si estamos maximizando, o la dirección contraria si estamos minimizando).

Por su parte, para PLE se usan distintos métodos inteligentes de enumeración de soluciones, como Branch and Bound o Branch and Cut, que veremos a continuación.

3.3.1. Branch and Bound

Básicamente se encarga de ir ramificando en subproblemas llamados relajaciones lineales, y resolviendo hasta encontrar el óptimo, siempre guardando una cota de la mejor solución obtenida hasta el momento. De esta manera, se hace una enumeración "inteligente", evitando pasar por todas las soluciones factibles, ya que al ramificar se descartan soluciones que sabemos que no pueden ser óptimas.

En cuanto a las relajaciones, las mismas están dadas por el mismo problema pero permitiendo que las variables tomen valores continuos. Por lo general la solución de la relajación no es entera, pero sirve como cota inferior (en el caso de minimización) de la solución entera óptima. Esto se debe a que el conjunto de soluciones factibles de la relajación lineal contiene al de soluciones enteras.

A diferencia del algoritmo de branch and cut que veremos a continuación, cuando ramificamos o dividimos el problema en dos, las restricciones que se agregan a cada uno de ellos son mutualmente exclusivas (generan conjuntos de soluciones factibles disjuntos) y exhaustivas (considerando ambos, se tiene todo el conjunto de soluciones enteras factibles del nodo padre).

La idea para ramificar consiste en elegir una de las variables no enteras x_i y considerar las restricciones extra: $x_i \leq \lfloor x_i \rfloor$ y $x_i \geq \lceil x_i \rceil + 1$. La elección de esta variable puede realizarse de manera heurística (dándole prioridad a aquella que más contribuye a la mejora de la función objetivo, es decir aquella con c_j más chico), o bien utilizando alguna técnica que

estime el grado de mejora de la función objetivo que se tendrá luego de ramificar.

Por otro lado, la ramificación puede ser a lo ancho o en profundidad. Esta última es la que garantiza más rápidamente una cota para la función objetivo.

Veamos el pseudocódigo de este algoritmo. Para eso, consideremos L={Lista de problemas candidatos a ser ramificados}, y llamemos incumbente $\overline{z}$ a la mejor solución entera obtenida hasta el momento. Luego:

1. Tomar $L = \{(P_0)\}$ y $\overline{z} = +\infty$, donde P_0 es el problema de minimización original, pero quitando la restricción de que las variables sean enteras.

2. Elegir un problema (es decir, un nodo) de la lista L. Sea (P) el problema seleccionado. Si $L = \emptyset$, terminar con las siguientes conclusiones:

   • Si $\overline{z} = +\infty$, entonces el problema original de programación lineal entera es infactible.

   • En caso contrario, la solución óptima es $(\overline{x}, \overline{z})$

3. Resolver el problema (P). Si el mismo es infactible, eliminarlo de L, es decir, podar la rama que nace de (P), y volver a (1).

4. Si (P) es factible, sean z' y x' el valor y la solución óptima. Si $z' \leq \overline{z}$, eliminar (P) de L y volver a (1).

5. Si $z' < \overline{z}$ y x' tiene todas sus variables enteras, entonces actualizar los siguientes valores:

   •$\overline{z} \leftarrow z'$

   •$\overline{x} \leftarrow x'$

   Luego, eliminar (P) de L e ir a (1).

6. Caso contrario, tomar x'_k que no sea entera. Ramificar entonces (P) creando dos subproblemas: (P^+) y (P^-), al primero agregándole la restricción $x_k \geq \lceil x'_k \rceil + 1$, y al segundo $x_k \leq \lfloor x'_k \rfloor$.

   Por último actualizar $L \leftarrow L \cup \{P^-, P^+\}$ e ir al paso (1).

3.3.2. Branch and Cut

Utilizado para resolver problemas de PLE, este método de optimización combinatoria consiste en utilizar el algoritmo de Branch and Bound, pero agregando planos de corte para restringir las relajaciones lineales. Es decir, una vez obtenida la solución de la relajación lineal, si alguna de las variables de la misma no es entera, se agrega una restricción extra que cumplen todos los puntos enteros factibles, pero que es violada por la solución fraccionaria.

Hay diversos métodos para elegir cómo realizar la ramificación:

- **Ramificación por pseudo-costo**: Consiste en guardar para cada variable x_i el cambio en la función objetivo cuando la misma ya fue seleccionada para ramificar. A partir de esto se selecciona aquella que tuvo mayor influencia en aumentar la función objetivo (asumiendo que el problema es de maximización).
- **Ramificación fuerte**: Consiste en testear previamente, entre una lista de candidatos, cuál será la variable que mejora en mayor medida la función objetivo, antes de ramificar realmente.
- **Ramificación fuerte completa**: Similar a la anterior, pero ahora testeando entre todas las variables.

Si los cortes se usan solamente para la relajación lineal inicial, entonces el algoritmo se llama cut and branch. Más adelante veremos que al utilizar el software Concorde, en la elección de los parámetros usados para resolver el problema del TSP se seleccionó este algoritmo.

3.4. Fórmula de Gauss para el área de un polígono

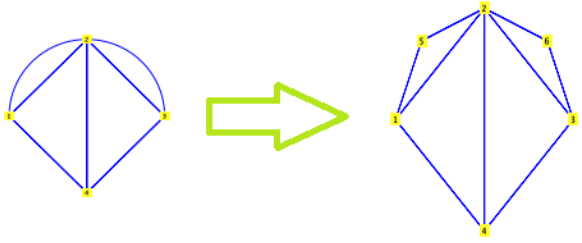
La fórmula surge de numerar los vértices $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ en sentido antihorario y dividir al polígono en triángulos que se forman al trazar los segmentos que unen los vértices con el baricentro del polígono. A partir de esto, se calcula el área de cada uno de ellos, usando el hecho de que el determinante permite calcular el área de un paralelogramo, y por ende, dividiendo a la mitad, se puede obtener el área de un triángulo. En [**12**] se puede ver una demostración más detallada. De este modo se llega a la siguiente forma:

$$\text{Área}=\frac{1}{2}\left(\begin{vmatrix} x_1 & x_2 \\ y_1 & y_2 \end{vmatrix} + \begin{vmatrix} x_2 & x_3 \\ y_2 & y_3 \end{vmatrix} + \ldots \begin{vmatrix} x_n & x_1 \\ y_n & y_1 \end{vmatrix}\right)$$

Esta fórmula se utiliza, junto con el algoritmo de Jarvis March, en una de las valuaciones del modelo de programación lineal entera de la primera etapa.

3.5. Procedimiento para pasar de un multigrafo a grafo

Este procedimiento es bastante sencillo. La idea es agregar un vértice entre los dos vértices que forman una arista con multiplicidad mayor a 1. El siguiente grafo muestra un ejemplo de conversión de multigrafo a grafo:



Este método se utiliza durante la segunda etapa en aquellas aristas que son replicadas al resolver el Problema del Cartero Chino.

3.6. Algoritmo de Dijkstra

Este algoritmo se utiliza para obtener el camino más corto desde un vértice de origen hacia el resto de los vértices que son alcanzables. Se aplica sobre grafos dirigidos (aunque si no lo es, fácilmente se puede transformarlo a uno que sí lo sea, al replicar cada arista por dos y determinando en cada una de ellas orientaciones opuestas entre sí); y el concepto de por qué funciona se basa en el hecho de que todo subcamino de un camino óptimo también es óptimo (esto se puede ver fácilmente por el absurdo, pensando que si un subcamino no fuera óptimo, entonces podría reemplazarlo en el camino total por el subcamino óptimo, y entonces, obtendría un camino total con longitud menor que el óptimo). La idea detrás de Dijkstra consiste en ir explorando todos los caminos más cortos que parten del vértice de origen, y que llevan a los demás vértices del grafo. El algoritmo sólo funciona cuando no se tienen aristas de costo negativo, ya que como veremos, al elegir el nodo con distancia menor, pueden quedar excluidos de la búsqueda nodos que en iteraciones siguientes bajarían el costo total del camino al pasar por una arista con costo negativo.

Dependiendo de la forma de implementación, el algoritmo tiene complejidad $O(|V|^2)$ (sin utilizar cola de prioridad), o $O(|E| + |V|log|V|) = O(|E|log|V|)$ (utilizando montículo de prioridad, como por ejemplo el de Fibonacci).

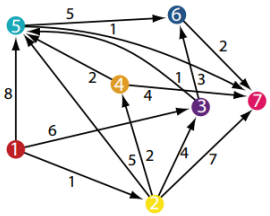
En 2 puede observarse el pseudocódigo del algoritmo, considerando que el nodo 1 es el vértice de origen (fácilmente puede cambiarse para que el mismo sea distinto):

Algoritmo 2 Algoritmo Dijkstra

Entrada: G=(V,E), $V = \{1, \ldots, , n\}$, costo de las aristas: $c : E \to \mathbb{R}^+$

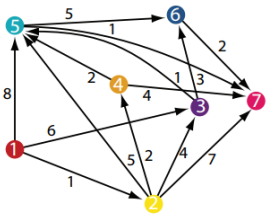
```
1: cantidadCC ← cantidadCC +1
2: S ← conjunto vacío
3: distancias ← vector de longitud n
4: padres ← vector de longitud n
5: distancias[1] ← 0
6: padres[1] ← 0
7: para 2 ≤ n hacer
8:     distancias[i] ← ∞
9:     padres[i] ← None
10: fin para
11: mientras S ≠ V hacer
12:     j ← elegirMinimoNoAgregado(S)
13:     S ← agregarNodoAlConjuntoS(j)
14:     para cada i en sucesores(j) que no esté en S hacer
15:         si distancias[i] > distancias[j] + c(j, i) entonces
16:             distancias[i] ← distancias[j] + c(j,i)
17:         fin si
18:     fin para
19: fin mientras
        devolver padres
```

Para mayor comprensión, veremos un ejemplo de aplicación del algoritmo:



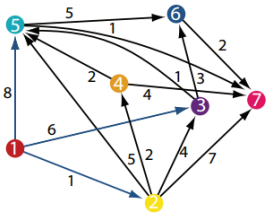
$S = \emptyset$

distancias$= [0, \infty, \infty, \infty, \infty, \infty, \infty]$
padres $= [1, 0, 0, 0, 0, 0, 0]$



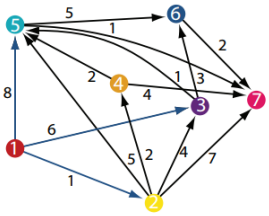
$S = \{1\}$

distancias$= [0, \infty, \infty, \infty, \infty, \infty, \infty]$
padres $= [1, 0, 0, 0, 0, 0, 0]$



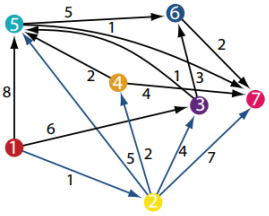
$S = \{1\}$

distancias$= [0, 1, 6, \infty, 8, \infty, \infty]$
padres$= [1, 1, 1, 0, 1, 0, 0]$



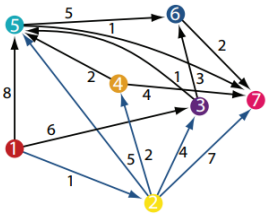
$S = \{1, 2\}$

distancias$= [0, 1, 6, \infty, 8, \infty, \infty]$
padres $= [1, 1, 1, 0, 1, 0, 0]$



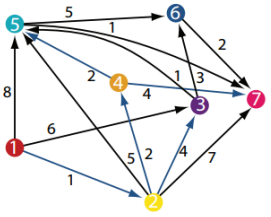
$S = \{1, 2\}$

distancias $= [0, 1, 5, 3, 6, \infty, 8]$
padres $= [1, 1, 2, 2, 2, 0, 2]$



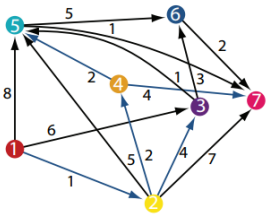
$S = \{1, 2, 4\}$

distancias $= [0, 1, 5, 3, 6, \infty, 8]$
padres $= [1, 1, 2, 2, 2, 0, 2]$



$S = \{1, 2, 4\}$

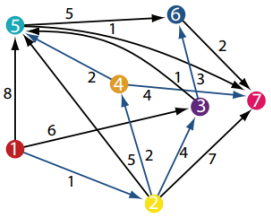
distancias$= [0, 1, 5, 3, 5, \infty, 7]$
padres $= [1, 1, 2, 2, 4, 0, 4]$



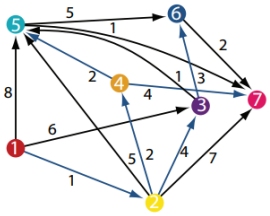
$S = \{1, 2, 4, 3\}$

distancias $= [0, 1, 5, 3, 5, \infty, 7]$
padres $= [1, 1, 2, 2, 4, 0, 4]$

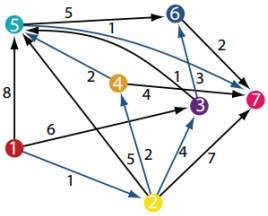
Figura 3.12: Ejemplo de Dijkstra - Parte 1



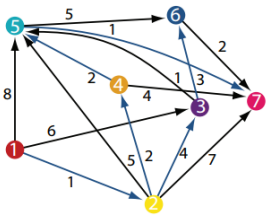
$S = \{1, 2, 4, 3\}$
distancias $= [0, 1, 5, 3, 5, 8, 7]$
padres $= [1, 1, 2, 2, 4, 3, 4]$



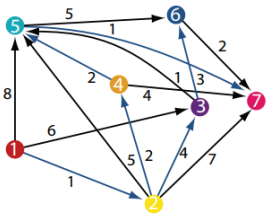
$S = \{1, 2, 4, 3, 5\}$
distancias $= [0, 1, 5, 3, 5, 8, 7]$
padres $= [1, 1, 2, 2, 4, 3, 4]$



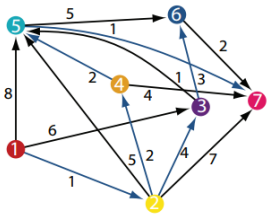
$S = \{1, 2, 4, 3, 5\}$
distancias $= [0, 1, 5, 3, 5, 8, 6]$
padres $= [1, 1, 2, 2, 4, 3, 5]$



$S = \{1, 2, 4, 3, 5, 7\}$
distancias $= [0, 1, 5, 3, 5, 8, 6]$
padres $= [1, 1, 2, 2, 4, 3, 5]$



$S = \{1, 2, 4, 3, 5, 7\}$
distancias $= [0, 1, 5, 3, 5, 8, 6]$
padres $= [1, 1, 2, 2, 4, 3, 5]$



$S = V$
distancias $= [0, 1, 5, 3, 5, 8, 6]$
padres $= [1, 1, 2, 2, 4, 3, 5]$

Figura 3.13: Ejemplo de Dijkstra - Parte 2

3.7. BFS (Breadth-First Search)

Este algoritmo de búsqueda a lo ancho se utiliza para recorrer los elementos en un grafo. La idea del algoritmo es comenzar por un vértice distinguido, luego recorrer sus vecinos, a continuación los vecinos de los vecinos, y así sucesivamente. Si el grafo no es conexo, es decir, si existe al menos un par de vértices sin un camino que los una, el algoritmo se reinicia desde un vértice que no fue alcanzado. Es por este motivo, que con una simple modificación, BFS sirve para identificar cuántas componentes conexas tiene un grafo. Además, permite conocer cuál es la distancia (medida en cantidad de aristas) al vértice distinguido, y reconocer grafos bipartitos.

A continuación veremos el pseudocódigo del algoritmo con la modificación que permite contar las componentes conexas.

Para implementar BFS, es necesario usar la estructura de datos **cola** o queue, donde el primero en llegar es el primero en salir. Este TAD tiene tres operaciones básicas que son:

1. **encolar**: Coloca el objeto al final de la cola
2. **primero**: Permite ver el primer objeto de la cola
3. **desencolar**: Elimina de la cola el primer objeto.

Algoritmo 3 Algoritmo BFS

```
Entrada: G=(V,E)
 1: ord ← 0
 2: C ← [ ]
 3: cantidadCC ← 0
 4: mientras queden vértices sin visitar hacer
 5:     cantidadCC ← cantidadCC +1
 6:     v ← elegirVérticeNoVisitado()
 7:     marcar(v)
 8:     encolar(C,v)
 9:     mientras C no sea vacía hacer
10:         w ← primero(C)
11:         ord ← ord +1
12:         desencolar(C)
13:         para cada vecino z de w hacer
14:             si z no está visitado entonces
15:                 marcar(z)
16:                 encolar(C,z)
17:             fin si
18:         fin para
19:     fin mientras
20: fin mientras
        devolver cantidadCC
```

En 3.14, 3.15 y 3.16 se encuentra un ejemplo del algoritmo.

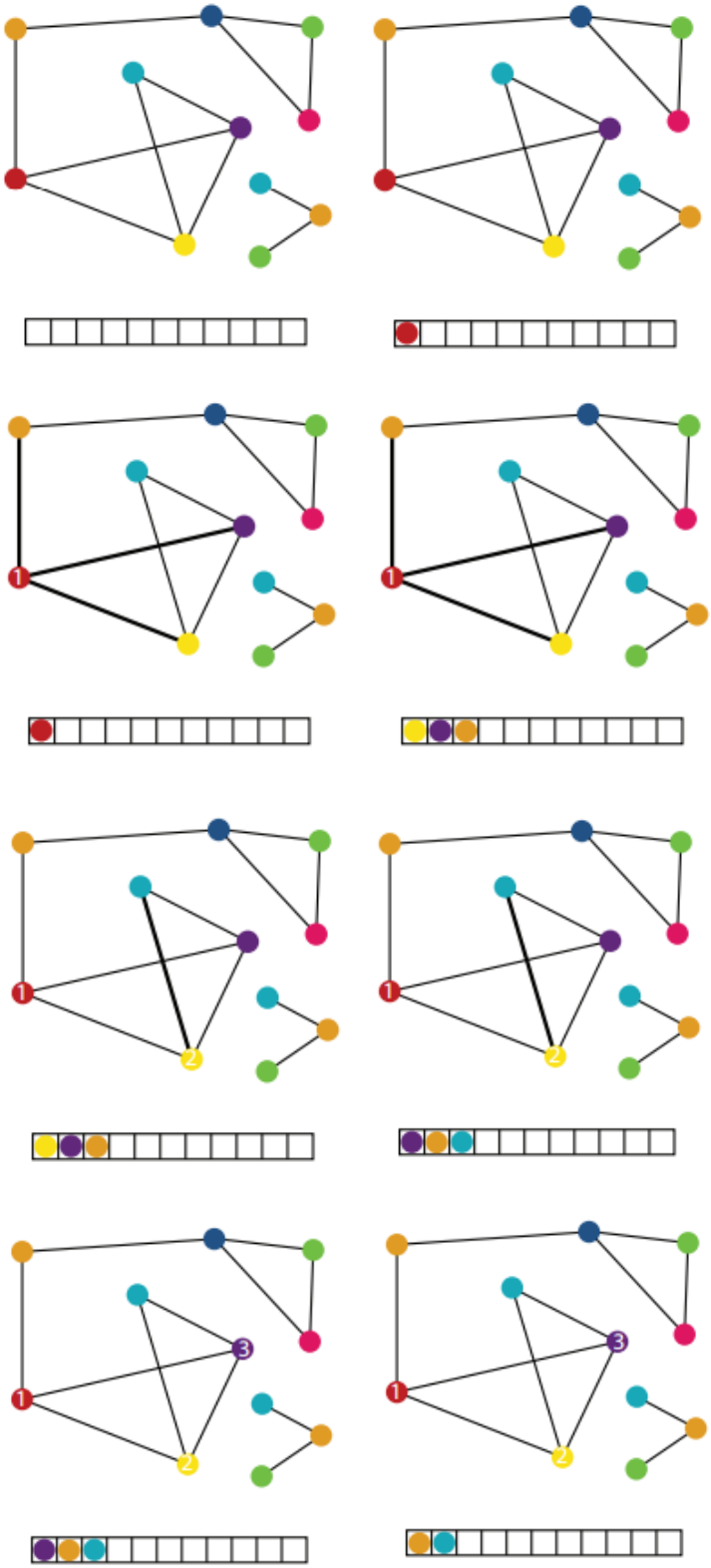


Figura 3.14: Ejemplo de BFS - Parte 1

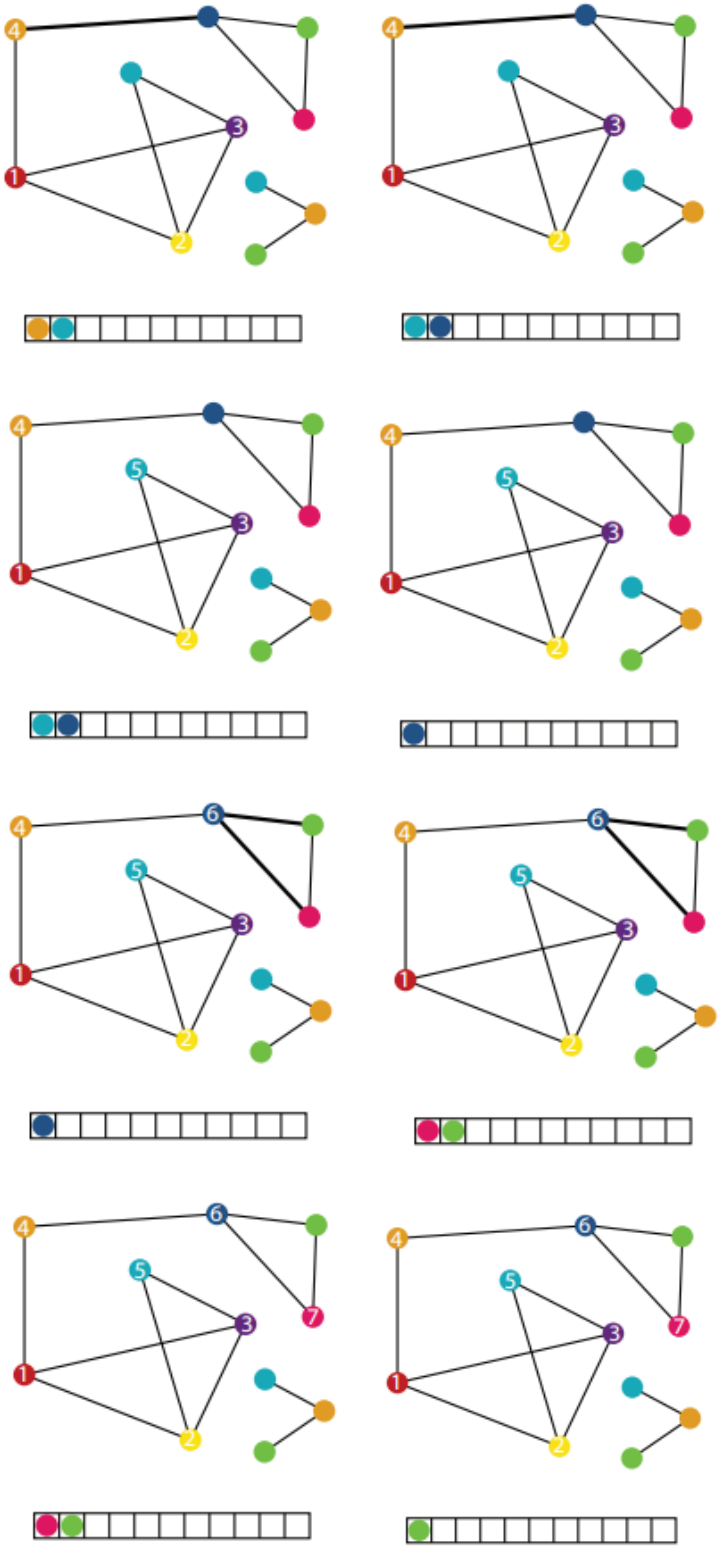


Figura 3.15: Ejemplo de BFS - Parte 2

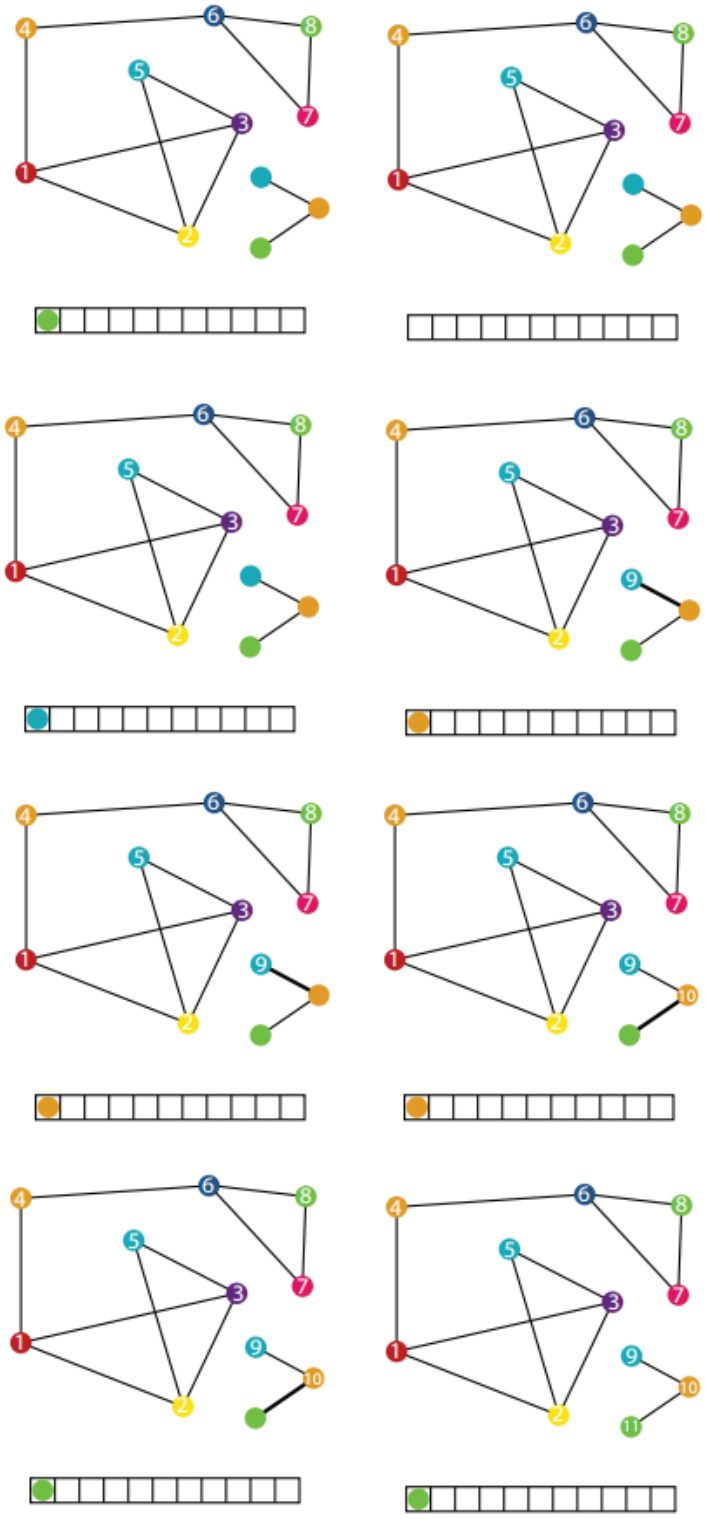


Figura 3.16: Ejemplo de BFS - Parte 3

3.8. Algoritmo GRASP (Greedy Randomized Adaptive Search Procedure)

Esta metaheurística, que permite resolver problemas de optimización combinatoria, surgió en 1989 de la mano de Feo y Resende. Desde ese momento, se han escrito numerosos trabajos sobre el tema. En [**13**], en los capítulos 4 y 5 puede encontrarse un análisis más profundo, junto con aplicaciones a problemas concretos.

La idea de GRASP surge de aplicar múltiples veces un procedimiento de búsqueda local, sobre una solución inicial generada por un algoritmo semi-greedy. En cada una de estas iteraciones, vamos guardando el mejor óptimo local obtenido hasta el momento.

Veamos con más detalle en qué consisten las dos etapas del algoritmo:

- **Fase Greedy Randomized Adaptive:** Aquí se construye una solución. En cada iteración, se tiene una lista de candidatos conformada por todos los elementos que se pueden incorporar a la solución parcial en construcción, sin eliminar la factibilidad de la misma. Para seleccionar el siguiente elemento a agregar, se utiliza una función greedy. Por lo general, la misma representa el aumento incremental en el costo de la función una vez incorporado dicho elemento. En particular, la evaluación de cada uno de estos elementos lleva a la creación de una lista de candidatos restringidos (RCL), formada por aquellos que provocan los menores aumentos en el costo. El elemento que finalmente se incorpora en la solución, se elige al azar entre esa lista RCL (constituyendo el aspecto probabilístico de GRASP), y a continuación la lista de candidatos se actualiza, pero ahora con los costos reevaluados (aspecto adaptativo de la heurística).

- **Búsqueda local:** Comienza con una solución factible y visita otra solución vecina (que podría ser infactible), hasta que se encuentra una solución factible que no se puede mejorar (óptimo local). El concepto de vecindad en general involucra una solución ligeramente modificada, y depende del problema que se esté considerando.

A continuación puede observarse el pseudocódigo del algoritmo GRASP. Aquí se asume que estamos ante un problema de minimización y la función objetivo está dada por f.

Algoritmo 4 Algoritmo GRASP

```
1: f* ← ∞
2: mientras no se satisfaga el criterio de parada hacer
3:     S ← semi-greedy()
4:     si S no es factible entonces
5:         S ← Arreglar(S)
6:     fin si
7:     S ← búsquedaLocal(S)
8:     si f(S) < f* entonces
9:         S* ← S
10:        f* ← f(S)
11:    fin si
12: fin mientras
       devolver S *
```

En general, la flexibilidad que permite un algoritmo GRASP es muy grande, y sirve para resolver todo tipo de problemas. En el presente trabajo, se utilizó en una estrategia inicial para abordar la primera etapa, aunque finalmente, por motivos que veremos más adelante, no se terminó implementando en la solución final.

3.9. Problema del viajante de comercio (TSP)

Este problema es probablemente el más conocido entre los de ruteo.

Dadas n ciudades, consiste en encontrar un recorrido (es decir, una permutación de las ciudades) tal que cada una de ellas se visite exactamente una vez, siempre comenzando y terminando en la ciudad de origen, y de manera tal que se minimice la distancia. Es decir, la solución consiste en encontrar la permutación que minimice la distancia entre $\frac{(n-1)!}{2}$ posibilidades. Esto significa que por ejemplo, para una instancia de 10 ciudades se tienen 181.440 recorridos, mientras que para el caso de 50 ciudades el número asciende a $3,041 \times e^{62}$.

El TSP pertenece a la rama de optimización combinatoria, y es NP-hard (la demostración es bastante extensa, y se realiza haciendo una reducción al problema del ciclo hamiltoniano, puede consultarse en [**14**]).

En 1954 Dantzig, Fulkerson y Johnson resolvieron un caso de 49 ciudades del TSP, entendiendo "resolver" como probar que la solución era la mejor entre un conjunto de 60 decillones alternativas posibles. A medida que fue avanzando el tiempo, se resolvieron instancias con un número mayor de ciudades.

En 2001, por ejemplo, Applegate, Bixby, Chvátal y Cook (autores del software Concorde que se usa en este trabajo) lograron resolverlo para 15.112 ciudades de Alemania. En esta tarea se usó una red de 110 computadoras de la universidad de Rice y Princeton, con un tiempo total de cómputo de 22.6 años de una PC de 500 MHz. La longitud total de esta solución tenía 66.000 km (más de una vuelta y media alrededor de la Tierra), lo que da una idea de la magnitud del problema.

Más adelante, en 2005, se obtuvo un nuevo récord con 33.810 ciudades, resuelto por Cook, Espinoza y Goycoolea, mientras que un año después Cook obtuvo la solución óptima para 85.900 ciudades, récord conocido hasta el momento.

Volviendo a la descripción del problema, existe una **versión expandida**, en donde en lugar de un solo viajante de comercio se tienen m (mTSP). Aquí los m individuos deben visitar n ciudades, de manera que cada una de ellas sea recorrida exactamente una sola vez. Al igual que antes, todos los viajeros salen de una misma ciudad, y deben volver allí al final del recorrido.

También existe otra versión con **ventanas de tiempo** (TSPTW), donde se le agrega la restricción de que algunas de las ciudades sólo pueden visitarse en ciertas franjas temporales. Es por eso que la simetría que tiene el TSP, en donde cualquier permutación de las ciudades es una solución factible, se rompe, lo cual hace que sea más complicado de resolver.

La versión **asimétrica** (ATSP), que será la que utilizamos en este trabajo en la tercera etapa, se caracteriza porque la distancia de una ciudad i hasta una ciudad j no necesariamente es la misma que de j a i. Existe una forma sencilla de convertir un ATSP en una versión simétrica. El método propuesto en [**15**] consiste en duplicar la cantidad de nodos (ciudades), creando vértices ficticios. Sea (C_{ij}), con $i, j \in V$ y $V = \{1 \dots, n\}$. Tomemos $\overline{C}$ igual a C pero con los elementos de la diagonal $C_{ii} = -M$, donde M es un número lo suficientemente grande, y sea $U = (u_{ij})$ con $u_{ij} = \infty$ para $i, j \in V$. Para trasformar la instancia asimétrica del TSP, se define la siguiente matriz:

$$\tilde{C} = \begin{pmatrix} U & \overline{C}' \\ \overline{C}' & U \end{pmatrix}$$

Como vemos, $\tilde{C} \in \mathbb{R}^{2n \times 2n}$. Dado que los elementos de la diagonal de $\overline{C}$ son negativos, por cómo construimos la matriz $\tilde{C}$, el costo de la arista de un nodo con su correspondiente nodo ficticio también lo es. Luego una solución óptima para $\tilde{C}$ debería aprovechar este hecho, y por lo tanto, va a estar dada por la forma:

$i_1 \to (i_1 + n) \to i_2 \to (i_2 + n) \dots \to i_n \to (i_n + 1) \to i_1$

con $i_k \in V$ para $k = 1, 2 \dots n$

Para reconstruir la distancia del tour completo original, sólo basta sumarle nM a la obtenida al resolver el problema.

Volviendo al TSP tradicional, la formulación mediante programación lineal entera del TSP es la siguiente:

Sea:

$$x_{ijt} = \begin{cases} 1 & \text{si la arista } ij \text{ está en el tour} \\ 0 & \text{caso contrario} \end{cases}$$

Luego, un tour óptimo para n ciudades con pesos c_{ij} debe cumplir:

Minimizar: $z=\sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij}$

Sujeto a:
$$(1)\ \sum_{i=1,i\neq j}^{n} x_{ij} = 1 (j = 1, 2, \ldots, n)$$
$$(2)\ \sum_{j=1,j\neq i}^{n} x_{ij} = 1 (i = 1, 2, \ldots, n)$$
$$(3)\ \sum_{(i,j)\text{ tal que } i\in U, j\in V} x_{ij} \geq 1 (\forall U \subset V \text{ tal que } 2 \leq |U| \leq n-2)$$

Las dos primeras restricciones determinan, respectivamente, que se debe entrar y salir exactamente una vez de cada ciudad; mientras que la tercera prohibe que haya subtours. El principal problema es que la restricción (3) contiene una cantidad exponencial de restricciones, por lo que es de difícil resolución.

Dada su complejidad, una forma de resolver el TSP es utilizando el método de Branch and Bound. Existen diversas formas para realizar la ramificación. La más simple de todas es particionando el conjunto de soluciones factibles en dos subconjuntos, de acuerdo con el hecho de que una arista sea parte del tour o no. La misma es utilizada en [**16**]

Otra estrategia utilizada en [**17**] aprovecha el hecho de que existe un algoritmo eficiente para el problema de asignación (formalmente, encontrar un matching con pesos en un grafo bipartito). De esta manera, comienza resolviendo el problema teniendo en cuenta sólo las dos primeras restricciones, que son condición necesaria pero no suficiente. Sin embargo, esta solución constituye una cota inferior para el óptimo del problema original.

Supongamos entonces que la solución de este problema nos devuelve un tour con longitud menor a n (si no existiera, entonces la misma sería un óptimo para el problema original). Sea entonces $(x_{12}, x_{23}, \ldots x_{k1})$ ese subtour, tal que:

$$x_{12} = x_{23} = \ldots = x_{k1} = 1$$

En la solución del TSP, no todas estas variables pueden valer 1, por lo que podemos particionar el espacio de soluciones en k subconjuntos, agregando en cada caso la restricción adicional de que exactamente una de esas variables valga 0. A su vez, estos problemas son de asignación (basta con poner un costo muy alto en la arista c_{ij} sobre la que queremos excluir de la solución), por lo que se pueden resolver fácilmente.

Capítulo 4

Resolución del problema

En este capítulo presentaremos en detalle la estrategia para tratar el problema. Se explicarán enfoques que por diversos motivos finalmente no se implementaron, ya sea porque fueron modelos de programación lineal entera demasiado grandes para poder resolverse en la práctica, o porque luego de discutirlos con la Municipalidad aparecieron nuevos requerimientos que forzaron a repensar la solución.

En la primera sección veremos un enfoque global inicial, orientado a resolver todo el problema en una única etapa, es decir, asignando cuadras, recorridos de los barrenderos y camiones, y determinando puntos de depósito de montículos de hojas en simultáneo.

En la segunda sección se detallará la estrategia utilizada diviendo el problema en tres etapas distintas: la primera para asignar las cuadras a los barrenderos, la segunda para determinar el orden de barrido de las mismas y los puntos de depósito de montículos, y la última, para determinar el ruteo de los camiones que los recogen.

A su vez, en cada una ellas se explicarán los algoritmos y modelos de programación lineal entera utilizados.

4.1. Estrategia inicial

En un primer momento, se pensó en un modelo matemático de programación lineal entera que resolviera todo el problema en una única etapa, aunque corriendo el modelo para cada una de las zonas por separado. Además de tener en cuenta numerosas restricciones, se realizó una discretización en ventanas temporales, con el objetivo de determinar qué cuadra barre cada operario en esa ventana temporal, y cómo se van generando los montículos para coordinar la recolección con los camiones. Este modelo no tenía en cuenta que la cantidad de barrenderos podía llegar a ser insuficiente, pero la idea era ir corriéndolo incrementando paulatinamente la cantidad hasta encontrar una solución factible.

4.1.1. Conjuntos

- $T = \{1, \ldots, t_f\}$ Conjunto de intervalos de tiempo, que surge de discretizar el tiempo total en base a lo que tarda un barrendero en limpiar una cuadra en promedio. Dado

que la jornada real de trabajo oscila entre 5 y 6 horas, y una cuadra insume entre 10 y 15 minutos, en total se tiene, en el mejor de los casos, 20 ventanas temporales, y 36 en el peor de ellos.

- $V = \{v_1, \dots v_n\}$ Conjunto de esquinas. Debido a la presencia de boulevards, que generan cuatro esquinas por intersección, el cardinal de este conjunto oscila entre 415 y 902.
- $E = \{(v_{i_1}, v_{j_1}), \dots (v_{i_m}, v_{j_m})\} \subset V \times V$ Conjunto de cuadras . Dependiendo de la zona, este número varía entre 500 y 700.
- $B = \{1, \dots k\}$ Conjunto de barrenderos. Según la zona, oscila entre 21 y 35.
- $C = \{1 \dots l\}$ Conjunto de camiones (En la actualidad l=2 para cada una de las zonas).

4.1.2. Parámetros

- $d[v_i, v_j]$, con $v_i, v_j \in V$ Distancia entre dos esquinas.
- *distancia_maxima* Distancia máxima permitida entre el punto de origen y el de fin del recorrido de un barrendero. Se usa para darle cierta compacidad a la solución.
- *cuadras_minimo* Cantidad mínima de cuadras que debe limpiar un barrendero.
- *cuadras_maximo* Cantidad máxima de cuadras que debe limpiar un barrendero. Estos últimos parámetros se usan para que la cantidad de cuadras sea equilibrada entre todos los operarios.

4.1.3. Variables

$$x_{iet} = \begin{cases} 1 & \text{si el barrendero i limpia la cuadra e a tiempo t} \\ 0 & \text{si no} \end{cases}$$

con $i \in B, e \in E, t \in T$.

$$z_{jkt} = \begin{cases} 1 & \text{si el camión j recoge el montículo k a tiempo t} \\ 0 & \text{si no} \end{cases}$$

con $j \in C, k \in V, t \in T$.

$$w_{ikt} = \begin{cases} 1 & \text{si el barrendero i deposita la basura en el montículo k a tiempo t} \\ 0 & \text{si no} \end{cases}$$

con $i \in B, k \in V, t \in T$.

$$u_{ief} = \begin{cases} 1 & \text{si el barrendero i comienza el recorrido en e y termina en f} \\ 0 & \text{si no} \end{cases}$$

con $i \in B, e \in E, f \in E$.

$$s_{jklt} = \begin{cases} 1 & \text{si el camión j está en el montículo k a tiempo t y en t+1 va al montículo l} \\ 0 & \text{si no} \end{cases}$$

con $j \in C, e \in V, l \in V, t = 1 \dots t_f - 1$.

4.1.4. Restricciones

1) Todas las cuadras deben ser barridas.

$$\sum_i \sum_t x_{iet} = 1 \ \forall e \in E$$

2) Cada barrendero puede barrer a lo sumo una cuadra por intervalo de tiempo.

$$\sum_e x_{iet} \leq 1 \ \forall i \in B, \forall t \in T$$

3) Un barrendero sólo puede barrer cuadras lo suficientemente "cercanas" en intervalos de tiempo consecutivos.

$$x_{iet} \leq \sum_{e' \in \delta(e)} (x_{i,e',t-1} + x_{i,e',t-2}) \ \forall i \in B, \forall t \in T, \forall e \in E$$

4) Un camión tiene que tener tiempo suficiente para recorrer el camino hasta el montículo siguiente. Para eso se define el vecindario de un montículo, como todos aquellos montículos que son alcanzables por el camión durante el tiempo de un intervalo.

$$z_{jkt} + \sum_{k \notin \delta'(k')} z_{j,k',t+1} \leq 1 \ \forall j \in C, \forall k \in V, \forall t \in T$$

5) Los carritos tiene una capacidad máxima, y deben descargarse cada cuatro cordones como mínimo.

$$\sum_k \sum_{m=t}^{t+3} w_{ikm} \geq 1 \ \forall i \in B, t = 1 \dots t_f - 3$$

6) Los barrenderos no deben terminar muy lejos de donde empezaron. En este caso v_l es el vértice 1 de la arista e y v_j el segundo de la arista f.

$$d(v_l, v_j) * u_{ief} \leq distancia\_maxima \ \forall i \in B, \forall e, f \in E,$$

7) Restricciones de relación entre las variables.

$$\sum_{f} u_{ief} \geq x_{ie1} \ \forall e \in E, \forall i \in B,$$

$$\sum_{e} u_{ief} \geq x_{if,t_f} - \sum_{f \varepsilon E} x_{if,t+1} \ \forall f \in E, \forall i \in B,$$

$$s_{jklt} \geq z_{jkt} + z_{j,l,t+d} - \sum_{t'=t+1}^{t+d-1} \sum_{k' \varepsilon V} z_{jk't'} - 1 \ \forall j \in C, \forall k, l \in V, t = 1 \ldots t_f - 1$$

con d=t(k,l) el tiempo que tarda un barrendero en ir de k a l.

8) Cada barrendero tiene que limpiar una cantidad de cuadras comprendida en un determinado rango, para que la asignación sea equilibrada.

$$cuadras\_minimo \leq \sum_{e} \sum_{t} x_{iet} \leq cuadras\_maximo \ \forall i \in B$$

9) Si un barrendero depositó en un montículo k, tiene que haber barrido esa esquina en ese intervalo de tiempo. Acá k tiene que ser el segundo vértice de la arista e.

$$w_{ikt} \leq \sum_{\text{e tq tiene un vértice en k}} x_{iet} \ \forall i \in B, \forall t \in T$$

10) Todos los montículos tienen que estar limpios (es decir si algún barrendero depositó en el montículo k, entonces algún camión tiene que haber pasado por ahí después).

$$|B| \sum_{m=t}^{t_f} \sum_{j} z_{jkm} \geq \sum_{i} w_{ikt} \forall i \in B, \forall k \in V, \forall t \in T$$

4.1.5. Función objetivo

Se plantearon diferentes funciones objetivo posibles, destinadas a optimizar distintos aspectos del problema.

Función objetivo 1: Minimizar la distancia recorrida por los camiones.

$$\sum_{j} \sum_{k} \sum_{l} \sum_{t} s_{jklt} d_{kl}$$

Función objetivo 2: Minimizar el recorrido sin barrido de los operarios.

$$\sum_{f} \sum_{e} \sum_{i} u_{ief} d_{v_{e_1} v_{f_2}}$$

Función objetivo 3: Combinación lineal de las dos anteriores.

Si bien esta estrategia era muy interesante desde el punto de vista teórico, tenía una gran cantidad de variables y restricciones. Por ejemplo, considerando sólo la cantidad de variables u_{ief}, que dependen del cardinal de los conjuntos de barrenderos, y del de cuadras (a través de los subíndices e y f), este número supera los 5 millones en el mejor de los casos (es decir, considerando $|B| \times |E| \times |E| = 21 \times 500 \times 500 = 5{,}250{,}000$); mientras que la cantidad de variables x_{iet} puede llegar a ser 882.000 ($|B| \times |E| \times |T| = 35 \times 700 \times 36$) en el peor de los casos.

Otros problemas que podían surgir, era la falta de flexibilidad en los recorridos, es decir no quedaba muy claro si iban a ser sencillos para implementarse; y probablemente la discretización en ventanas temporales hubiera traído algunas imprecisiones con tiempos "muertos" en medio del recorrido, principalmente por la estimación de que todas las cuadras insumen la misma cantidad de tiempo de barrido.

4.2. Estrategia utilizada

Finalmente por los motivos recién mencionados, se realizó la división del problema en tres etapas distintas:

- Asignación de cuadras a barrenderos.
- Definición del recorrido de los barrenderos y asignación de puntos fijos para depositar los residuos de hojas.
- Ruteo de los camiones que levantan los montículos de basura.

La idea de dividir en etapas, a pesar de que probablemente genere soluciones subóptimas, es tener un mayor control sobre la solución, con más flexibilidad para realizar cambios sobre la marcha.

Como ya se mencionó antes, a su vez, las dos primeras etapas también tuvieron más de un enfoque. En el caso de la etapa inicial fue por la aparición de nuevos requerimientos de la Municipalidad; mientras que en el segundo caso, se debió a problemas similares a los del enfoque global, en donde el tamaño del modelo impedía que pudiera resolverse en la práctica.

4.2.1. Primera etapa: Asignación de cuadras a barrenderos

En esta sección detallaremos los dos enfoques planteados para resolver la asignación de cuadras de los barrenderos. El primero, que finalmente fue descartado, involucra la utilización de un algoritmo BFS modificado, con la posterior mejora mediante un algoritmo GRASP; mientras que el segundo involucra el desarrollo de un algoritmo de generación de segmentos para resolver un problema de asignación.

Primera etapa: Enfoque BFS/GRASP

La misma se dividió a su vez en varias subetapas: primero se generó una asignación inicial mediante un algoritmo BFS modificado, que va asignando a cada barrendero una cuadra por vez, y luego con esta solución inicial, utilizar un algoritmo GRASP para mejorar la calidad de la solución. A continuación describiremos las distintas subetapas:

1. **Algoritmo BFS modificado**

Esta primera subetapa está basada en el trabajo realizado en [**18**], que utiliza la idea de ir asignando cuadras progresivamente.

El algoritmo consiste en ir agregando una cuadra adyacente por vez al barrendero con el recorrido más corto hasta el momento, siempre y cuando sea posible hacerlo. Pero además, como forma novedosa, la cuadra a agregar no se elige al azar, sino que se considera aquella que le da más compacidad al recorrido. En este sentido se intenta asignar las cuadras de una misma manzana a un solo barrendero.

A continuación se incluye el pseudocódigo del algoritmo:

Algoritmo 5 Algoritmo BFS modificado

Entrada: Cantidad de barrenderos $n \in \mathbb{N}$
Grafo con vértices dados por las cuadras $G = (V, E)$

```
1: inicializarRecorridos(n)
2: mientras algún barrendero pueda seguir avanzando hacer
3:     barrendero ← elegirPróximoBarrendero()
4:     cuadra ← elegirCuadra(barrendero)
5:     agregarAlRecorrido(cuadra, barrendero)
6: fin mientras
   devolver Recorridos
```

Veamos más detalladamente cada una de las partes. En **inicializarRecorridos** se le asigna a cada uno de los n barrenderos una posición inicial al azar. Sin embargo, para evitar solapamientos en los recorridos, si bien esa posición se elige de manera aleatoria, también se tiene en cuenta que cada operario empiece lo suficientemente alejado del resto.

En **elegirPróximoBarrendero()**, como ya se dijo anteriormente, se elige a quien tenga la mínima distancia recorrida y pueda seguir avanzando. Es decir, aquel cuyo trayecto no se encuentre encerrado por el de otros.

elegirCuadra() consiste en tomar la cuadra, entre todas las adyacentes a la actual, que brinda mayor compacidad al recorrido. Sin embargo, a la hora de implementarlo, para mejorar la eficiencia del algoritmo, también se consideraron las cuadras adyacentes a la cuadra inicial. Para garantizar la compacidad, se tuvo en cuenta el concepto físico de un "centro de masa" de cada recorrido. Para calcularlo primero se consideró el punto medio de cada una de las cuadras, y se sumó coordenada a coordenada cada uno de ellos, para finalmente dividirlo por la cantidad de calles. Una vez obtenido esto, se midió la distancia del punto medio de cada una de las cuadras adyacentes a la actual, para elegir la más cercana a dicho centro.

Finalmente, se agrega esa cuadra al recorrido del barrendero con la función **agregarAl-**

Recorrido.

2. **Postprocesado**

Dado que en el algoritmo anterior termina cuando ningún barrendero puede seguir avanzando, puede ocurrir que algunas cuadras queden sin asignar. Es por eso que fue necesario realizar un postprocesado. La idea básica de esto es buscar al barrendero con menor cantidad de metros en su recorrido y que tiene alguna cuadra adyacente aún no asignada. La principal diferencia con lo que se realizó en el BFS modificado, es que antes las cuadras que se iban agregando sólo eran adyacentes a uno de los extremos del recorrido, mientras que ahora, no existe tal restricción, y las mismas pueden ser adyacentes a cualquier cuadra del camino.

Nuevamente se tuvo en cuenta el concepto de centro de masa, por lo que la cuadra elegida para agregar al recorrido es aquella que está más cercana al baricentro del mismo.

A continuación se encuentra el pseudocódigo del algoritmo:

Algoritmo 6 Algoritmo BFS

Entrada: Recorridos

```
1: mientras pueda extenderse algún recorrido hacer
2:     barrendero ← minimoRecorridoExtensible(Recorridos)
3:     cuadra ← elegirCuadraNoAgregada(barrendero)
4:     agregarAlRecorrido(cuadra,barrendero)
5: fin mientras
       devolver Recorridos
```

3. **Algoritmo GRASP**

Las soluciones obtenidas mediante el paso anterior, si bien garantizan que todas las cuadras estén asignadas, aún no eran tan balanceadas con respecto a la cantidad de metros asignada a cada barrendero, como se deseaba. Fue así que surgió la utilización de un algoritmo de GRASP.

Como ya vimos en 3.8, la idea es generar soluciones iniciales mediante un procedimiento semi-greedy. El método de BFS modificado da una solución inicial factible, que está en el conjunto de las mejores, y como las posiciones iniciales desde donde comienza cada barrendero se determinan al azar, en cada iteración podemos obtener soluciones distintas.

Con respecto a la búsqueda local, las soluciones vecinas se obtienen de la siguiente forma. Primero se calcula quién es el barrendero con menos metros asignados, y luego se buscan las cuadras adyacentes a su recorrido (es decir, que pertenezcan a otro barrendero). Así, se genera una nueva solución asignándole una de esas cuadras vecinas al barrendero con menos metros, y eliminándola del camino del barrendero que la tenía originalmente. Naturalmente, los recorridos que quedan podrían no ser conexos. Por este motivo, cuando se guarda una solución mejor en este procedimiento de búsqueda local hay que tener en cuenta dos requisitos:

- **Conexión de los dos recorridos modificados:** Por la forma en que se eligió la cuadra adyacente al recorrido, la conexión para el barrendero mínimo se da de manera trivial. Pero no sucede lo mismo con el recorrido del barrendero al cual se le

extrajo la cuadra. Una manera de verificar que se cumpla esto es aplicar el algoritmo BFS desarrollado en la sección 3.7.

- **Dispersión con respecto al centro de masa:** Como se quiere cierta compacidad de cada recorrido, una forma posible es medir la distancia del punto medio de cada cuadra al centro de masa del recorrido, y considerar la máxima de esas distancias como medida de dispersión. Luego para que la solución vecina sea mejor, se debe tener en cuenta que esta distancia no empeore en ninguno de los dos recorridos modificados.

Al elegir siempre al barrendero con menos metros asignados, y asignarle más cuadras, la brecha que existe entre él y el barrendero con mayor recorrido va a mejorar en la mayoría de los casos.

Finalmente, por disposión de la municipalidad, que consideraba que iba a ser más sencillo asignar a cada operario manzanas enteras en lugar de cuadras, se volvió a plantear esta etapa con una resolución totalmente distinta.

Primera etapa: Enfoque problema de asignación

Para este primer paso, se tomó como base el trabajo realizado en [**19**]. En el mismo se implementa un algoritmo de división en segmentos para el Censo del 2010 en la provincia de Buenos Aires. Al igual que en el presente trabajo, la cantidad de segmentos (en ese caso las casas que le corresponden a cada censista), es demasiado grande como para resolver un simple problema de asignación. Es por eso que se van construyendo subconjuntos, de manera creciente y progresiva, y se intenta resolver el problema de asignación partiendo de ese conjunto base, hasta finalmente hallar una solución factible.

El problema de asignación de barrenderos tiene algunas diferencias con este último trabajo: acá no se tienen restricciones tales como que un segmento no pueda contener cuadras separadas por una avenida, pero sí se le agrega la dificultad de que ahora las cuadras tienen distintas frecuencias de barrido.

A continuación se verá con más detalle el algoritmo implementado.

1. Identificación de manzanas

Por una cuestión de simplicidad, tanto para la implementación por parte de los operarios, como computacionalmente, se tomó como unidad mínima un segmento formado por una manzana entera. Este fue un requerimiento específico por parte de la municipalidad.

En 7 se muestra el pseudocódigo del algoritmo para identificarlas.

La función **buscarVecinosCuadra(cuadra,esquina1)** se encarga de encontrar las cuadras adyacentes a cuadra que comparten con esta última a esquina1, y que además son perpendiculares a ella.

Por su parte, la función **nombreCalleTransversal** busca el nombre de la calle perpendicular a vecino1 (cabe aclarar que no es la identificación dada a una cuadra, sino el nombre de la calle en el sentido corriente) . Naturalmente, dado que vecino1 también es

Algoritmo 7 Algoritmo de identificación de manzanas

Entrada: C= {Conjunto de cuadras}

```
1: Manzanas ← []
2: para cuadra en C hacer
3:     si estaMarcada(cuadra)==FALSE entonces
4:         [esquina1 , esquina2] ← esquinas(cuadra)
5:         Vecinos1 ← buscarVecinosCuadra(cuadra,esquina1)
6:         Vecinos2 ← buscarVecinosCuadra(cuadra,esquina2)
7:         para vecino1 en Vecinos1 hacer
8:             para vecino2 en Vecinos2 hacer
9:                 si nombreCalleTransversal(vecino1) == nombreCalle-
   Transversal(vecino2) & estaMarcada(vecino1) == FALSE &
   estaMarcada(vecino2) == FALSE entonces
10:                    manzana=[cuadra,vecino1,vecino2]
11:                    ultimaCuadra=cerrarPoligono(manzana)
12:                    si estaMarcada(ultimaCuadra)== FALSE entonces
13:                        manzana=manzana + ultimaCuadra
14:                        Manzanas = Manzanas + manzana
15:                    fin si
16:                fin si
17:            fin para
18:        fin para
19:    fin si
20: fin para
    devolver Manzanas
```

perpendicular a la cuadra original, se descarta el nombre de la calle a la cual pertenece esta última.

Finalmente, en **cerrarPoligono** se busca la cuadra que une a vecino1 y vecino2.

Con este algoritmo ya explicado, pasemos a ver algunos de los parámetros que tiene cada manzana. Para el presente trabajo, se tuvo en cuenta una doble clasificación:

1. **Dificultad temporal**: Debido a que no todas las manzanas tienen el mismo tamaño, el tiempo que demanda cada una es variable. Para estimarlo, se tuvo en cuenta la longitud de cada cuadra en la manzana junto con un parámetro de estimación de "tiempo"de barrido de cuadra, calculado de la siguiente manera:

   $$\frac{\text{Longitud de la cuadra en metros} * \text{Minutos que insume barrer una cuadra de 100 mts.}}{\text{100 metros}}$$

   Dado que la velocidad depende mucho de cada barrendero, resulta muy difícil determinar este parámetro con exactitud. La estimación provista por la municipalidad es que cada cuadra insume entre 10 y 15 minutos en barrerse, por lo que como veremos más adelante, se propusieron soluciones con distintas estimaciones.

2. **Frecuencia de barrido**: Debido a distintas características geográficas, como la cantidad de árboles, el hecho de si una calle es una avenida o no, o la cercanía a la frontera de la ciudad, algunas cuadras necesitan un barrido más frecuente. Luego de discutirlo con la municipalidad, se decidió poner un parámetro 3-ario:

   -Cuadras barridas los días lunes, martes, miércoles, jueves y viernes, es decir, diariamente.

   -Cuadras barridas día por medio.

   -Cuadras barridas una vez por semana.

   Esta clasificación también fue provista por la municipalidad. Dado que en este enfoque estamos trabajando con manzanas, y el parámetro anterior nos fue dado por cada cuadra, finalmente se resolvió hacer un "redondeo" en la frecuencia de barrido. Por ejemplo, si una manzana estaba compuesta por tres cuadras de barrido diario, y una de frecuencia día por medio, a los efectos del modelo, la manzana entera pasa a considerarse de barrido diario. Es decir, en general el criterio utilizado fue considerar la frecuencia de la cuadra que debe barrerse más seguido. Finalmente, con este redondeo, la última categoría se terminó eliminando. En el capítulo 1 ya habíamos visto el mapa con la categorización por cuadra, que permite obtener una idea de cómo es el caso con las manzanas.

   Con respecto a la categoría de manzanas que se barren día por medio, para la resolución del modelo el grupo se dividió en dos mitades, y se determinó un ciclo de barrido alternado, a completar en dos semanas. Es decir, si por ejemplo consideramos una manzana de uno de los grupos, la misma se barre los días lunes, miércoles y viernes, y a la semana siguiente martes y jueves; mientras que una manzana perteneciente a la otra mitad, se barre primero martes y jueves, y a la semana siguiente lunes, miércoles y viernes.

Estos dos parámetros, así como los vecinos de cada manzana, fueron guardados como datos para resolver el modelo.

2. **Algoritmo de generación de segmentos**

Una vez identificadas las manzanas, el paso siguiente fue realizar este algoritmo de generación de segmentos en forma progresiva, e ir resolviendo el modelo de asignación. A grandes rasgos, la idea es comenzar generando el conjunto integrado por segmentos de una sola manzana. Si no hay solución factible, que por las características del problema, ocurrirá siempre, se genera un nuevo conjunto de segmentos, ahora conformado no sólo por los manzanas unitarias, sino por todos los que contienen exactamente dos manzanas adyacentes entre sí, y así se itera el procedimiento hasta encontrar una solución factible. A su vez, también se impone una cantidad n máxima de iteraciones (en este trabajo tomamos $n = 8$) Este número surge de que por una cuestión práctica, no queremos generar segmentos que tengan más de ocho manzanas. Para una mejor comprensión, en 8 está el pseudocódigo del algoritmo.

Algoritmo 8 Algoritmo de generación de segmentos

Entrada: C= {Conjunto de manzanas}, n= número máximo de iteraciones

```
1:  B_1 = {Conjunto de manzanas unitarias}
2:  para i=2, ...,n hacer
3:      B_i = B_{i-1}
4:      para s ∈ B_{i-1} hacer
5:          para m ∈ B ∩ N(s) hacer
6:              B_i = B_i ∪ {s + m}
7:          fin para
8:      fin para
9:      solución=resolverModelo()
10:     si solución es factible entonces
11:         Terminar el algoritmo
12:     fin si
13: fin para
        devolver solución
```

Finalmente, una vez obtenidos los segmentos, el modelo que se resuelve para asignar los recorridos es el siguiente:

• **Conjuntos:**

$M = \{m_1, \dots m_n\}$: Conjunto de manzanas.

$S = \{\{m_1\}, \dots \{m_n\}, \dots, \{m_{i1}, m_{i2}, \dots, m_{ij}\}\}$: Conjunto de segmentos.

• **Parámetros** :

cantidad_barrenderos: Cantidad de barrenderos de la zona.

• **Variables** :

$$x_s = \begin{cases} 1 & \text{si s es parte de la solución} \\ 0 & \text{si no} \end{cases}$$

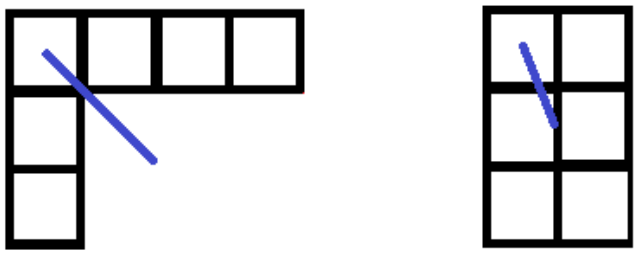
con $s \in S = B_i$ en la iteración i.

• **Función objetivo** :

Se pensaron dos criterios distintos de valuaciones para medir la calidad de la solución.

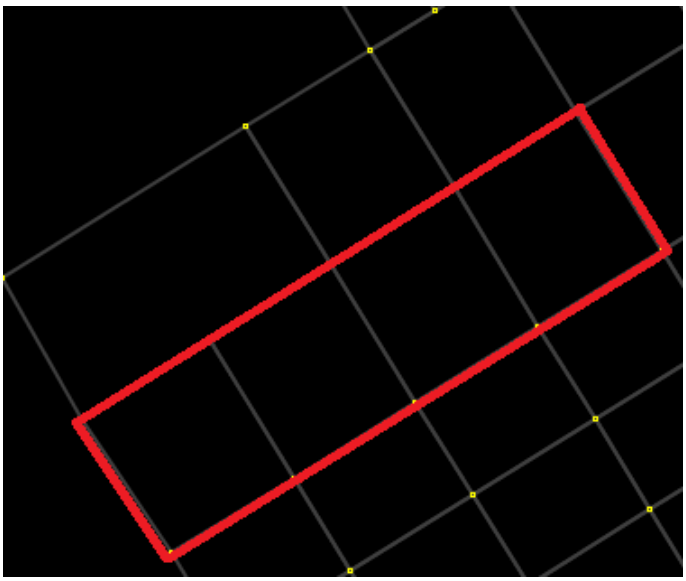
1. Minimizar $\sum_{s \in S} val_{1s} x_s$

   Aquí la valuación de cada segmento es una medida de la "compacidad" o "dispersión" del mismo. Para eso, se calcula primero el centro de masa del segmento completo, y luego se hace lo mismo pero con cada una de las manzanas. Finalmente, se calcula la distancia máxima entre el centro de masa del segmento y el de cada una de las manzanas. Ese valor es el que determina val_1. A continuación se tienen dos segmentos totalmente distintos pero con la misma cantidad de manzanas. La valuación de cada uno está dada por la longitud del segmento azul. Si bien el conjunto de manzanas de la derecha es ligeramente mejor, ambas medidas no difieren en mucho.



2. Minimizar $\sum_{s \in S} val_{2s} x_s$

   La función anterior le otorga un valor alto a segmentos que también pueden llegar a ser deseables, como el caso de una hilera de manzanas:



Una alternativa a este problema es calcular el área de la cápsula convexa del segmento, y compararla con el área real del segmento. Para eso fue necesario, entonces, calcular la cápsula convexa mediante el algoritmo de Jarvis March ya descripto, y usar la fórmula del determinante de Gauss para calcular el área.

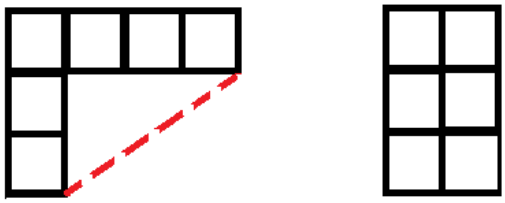
La comparación entre ambas áreas se midió de dos formas distintas:

•$\text{val}_{2s} = \frac{\text{Área de la cápsula convexa}}{\text{Área real del segmento}}$:

Esta valuación siempre es mayor o igual que 1. En ese valor es donde alcanza el óptimo, ya que por simplicidad se quiere que la cápsula convexa del segmento sea igual al mismo. También se pensó una variante similar de esta valuación calculando el cociente inverso, y obviamente pasando el problema a uno de maximización.

•val_{2s} = Área de la cápsula convexa - Área de real del segmento:

En este caso, el valor óptimo de un segmento es 0. A continuación se puede ver el mismo ejemplo visto en la primera valuación. El segmento de la derecha en este caso es óptimo, mientras que en el de la izquierda, como puede observarse a partir de la línea punteada roja que agrega el área de la cápsula convexa, la diferencia entre las dos áreas es igual a la mitad del área del segmento original. Con esta medición, la brecha entre ambos segmentos es mucho más marcada que antes.



- **<u>Restricciones</u>** :

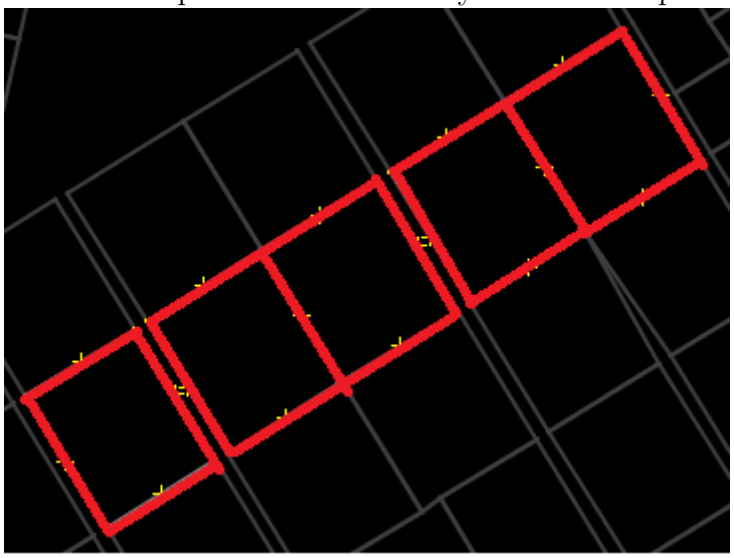
1. Cada manzana pertenece a un único segmento.

   $\sum_{s:m \in S} x_s = 1$ para todo $m \in M$

2. La cantidad de segmentos no puede exceder la cantidad de barrenderos disponibles. En general, para resolver el modelo, se impuso que esta cota sea lo suficientemente grande, por ejemplo de alrededor de 80 barrenderos. Si bien esto pareciera que va a generar soluciones con un excedente de operarios, como la cantidad efectiva de barrenderos es igual a la cantidad de segmentos utilizados, el modelo contempla esto en la función objetivo, y como veremos en el análisis de la solución, en todos los casos esa cota está lejos de alcanzarse.

   $\sum_{s \in S} x_s \leq cantidad\_barrenderos$

Este modelo devuelve, para cada barrendero, el conjunto de manzanas que le corresponde barrer. Sin embargo, debido a la presencia de boulevards, el conjunto de aristas, es decir cuadras, no dan como resultado un grafo conexo. A continuación, puede observarse un ejemplo de segmento formado por cinco manzanas y con tres componentes conexas.



Como para el paso siguiente, que consiste en encontrar el orden de barrido de las cuadras, es necesario que el grafo sea conexo, una vez terminado el modelo, se realiza un algoritmo de postprocesado del subgrafo dado por el segmento. La idea del mismo se basa en el hecho de que las aristas a agregar para dar la conexión cumplen dos condiciones:

• No pertenecen al subgrafo

• Son adyacentes a dos aristas que sí están en el subgrafo, pero que no son vecinas entre sí.

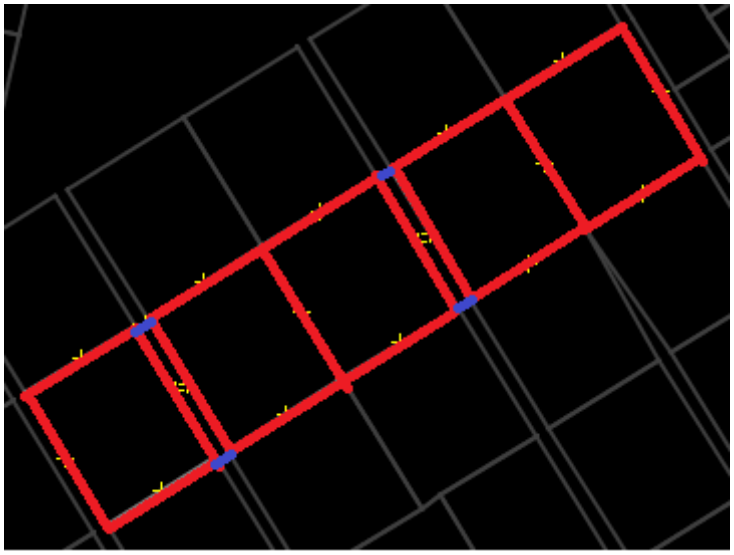
A partir de esto, se aplica el algoritmo 9.

Aquí se observa el output que devuelve el algoritmo en el ejemplo, con las nuevas aristas marcadas en azul:

Algoritmo 9 Algoritmo de completamiento del grafo

Entrada: Subgrafo con las aristas correspondientes a manzanas de un barrendero dado

```
1: para cuadra en Subgrafo hacer
2:     vec1 ← vecinos(cuadra)
3:     para vecino1 en vec1 hacer
4:         si vecino1 no está en Subgrafo entonces
5:             vec2 ← vecinos(vecino1)
6:             para vecino en vec2 hacer
7:                 si vecino2 no está en vec1 y vecino2 está en Subgrafo entonces
8:                     agregarASubgrafo(vecino1)
9:                 fin si
10:            fin para
11:        fin si
12:    fin para
13: fin para
       devolver Subgrafo
```



4.2.2. Segunda etapa: Recorrido de los barrenderos y asignación de montículos

Una vez definidas las manzanas que se le asignarán a cada uno de los operarios, se corre el modelo del Problema del Cartero Chino visto en 3.1.3 para crear el grafo de aumento. Este modelo determina cuáles son las cuadras que van a ser recorridas más de una vez, con el objetivo de cumplir con la eulerianidad. En un primer momento, podría pensarse que pedir que sea un ciclo euleriano y no un camino es demasiado exigente. Sin embargo, en la práctica las únicas aristas replicadas son pequeños segmentos que surgen de la presencia de boulevards, y sólo agregan entre 10 y 20 metros a la distancia total recorrida por el barrendero. Esto es bastante razonable, y muestra que la decisión de resolverlo de esta forma no tiene una contribución negativa en la solución.

Como ya mencionamos anteriormente, esta etapa también tuvo dos estrategias distintas. Si bien ambas comienzan determinando previamente las aristas replicadas, la primera consiste en realizar un modelo de programación lineal entera que determine en simultáneo el orden en que se recorren esas cuadras y los puntos de depósitos de montículos; mientras que la segunda estrategia involucra utilizar el algoritmo de Hierholzer para determinar el orden de recorrido de las cuadras, y sólo después de haber obtenido esto, aplicar un modelo de PLE más simplificado, que sólo sirve para fijar los puntos de acumulación de montículos.

Estrategia 1: Modelo de PLE exclusivo

Una vez obtenido el grafo de aumento, en lugar de usar algoritmos como el de Fleury o Hierholzer para determinar el orden de las cuadras, se intentó aplicar el siguiente modelo de programación lineal entera que resuelve en simultáneo el recorrido y fija los puntos para depositar los montículos. El mismo tiene un especial cuidado con aquellas cuadras que sólo se recorren caminando (aristas replicadas), y al igual que el modelo global, realiza una discretización en tiempos. La diferencia es que ahora esta discretización es una forma simbólica utilizada más bien para establecer el orden de barrido, por lo que no trae problemas de imprecisiones. Otro aspecto en que también difiere, es que como se decidió desfasar el horario de los recorridos de los camiones, ya no es importante en qué momento se genera un montículo.

1. **Conjuntos**

- $T = \{1, \ldots, t_f\}$ Conjunto de intervalos de tiempo.
- $V = \{v_1, \ldots v_n\}$ Conjunto de esquinas.
- $B = \{1, \ldots k\}$ Conjunto de barrenderos de la zona.
- $E_i = \{(v_{i_1}, v_{j_1}), \ldots (v_{i_m}, v_{j_m})\} \subset V \times V$ Conjunto de cuadras asignadas del barrendero i.
- $E = \{(v_{i_1}, v_{j_1}), \ldots (v_{i_m}, v_{j_m})\} \subset V \times V$ Conjunto de cuadras.

2. **Parámetros**

- d_e $e \in E$: Distancia desde la base de la zona hasta la cuadra e.
- $$a_e = \begin{cases} 1 & \text{si la cuadra e solo se recorre caminando} \\ 0 & \text{si no} \end{cases}$$
- C : Cantidad de cuadras que se pueden barrer hasta llenar un carrito.
- $max\_aristas\_replicadas$: Máximo entre todos los barrenderos, de la cantidad de aristas replicadas.

3. **Variables**

$$x_j = \begin{cases} 1 & \text{si la esquina j se usa para depositar un montículo} \\ 0 & \text{si no} \end{cases}$$

con $j \in V$.

$$y_{iet} = \begin{cases} 1 & \text{si el barrendero i pasa por la cuadra e a tiempo t} \\ 0 & \text{si no} \end{cases}$$

con $i \in B, e \in E_i, t \in T$

$$w_{ijt} = \begin{cases} 1 & \text{si el barrendero i deposita la basura en el montículo j a tiempo t} \\ 0 & \text{si no} \end{cases}$$

con $i \in B, j \in Vt \in T$

$$z_{ijt} = \begin{cases} 1 & \text{si el barrendero i comienza barriendo una cuadra desde la esquina j a tiempo t} \\ 0 & \text{si no} \end{cases}$$

con $j \in V, t \in T$ (es decir por ejemplo si se barre la cuadra (i, j) a tiempo t, en el sentido $j \to i$ entonces $z_{mjt} = 1$ y $z_{mit} = 0$).

4. **Función objetivo**

Combina optimalidad del punto de origen del recorrido (ya que sería deseable reducir el tiempo que se tarda al salir de la base hasta que se comienza a barrer), junto con la minimización en la cantidad de lugares para depositar montículos.

c_1 y c_2 son constantes ponderadoras, a calibrar empíricamente.

Minimizar $c_1 \sum_{i \in B} \sum_{e \in E} d_e y_{ie1} + c_2 \sum_{j \in V} x_j$

5. **Restricciones**

1) Las esquinas iniciales deben ser únicas.

$$\sum_{j \in V} z_{ij1} = 1 \ \forall i \in B$$

2) Conexión del recorrido: si un barrendero barre una esquina j a tiempo t, tiene que haber barrido una esquina vecina a j en el tiempo anterior.

$$z_{ijt} \leq \sum_{j' \in N(j)} z_{i,j',t-1} \ \forall i \in B, \forall j \in V, \forall t > 1$$

3) No se puede recorrer más de una esquina por tiempo.

$$\sum_{j \in V} z_{ijt} \leq 1 \ \forall i \in B, \forall t \in T$$

4) Todas las cuadras tienen que barrerse una única vez.

$$\sum_{t \in T} y_{iet} = 1 \ \forall e \in E_i, \forall i \in B$$

5) Los barrenderos pueden llevar a lo sumo la carga de C cordones. Las cotas impuestas en los intervalos de tiempo se realizaron con el objetivo de disminuir la cantidad de restricciones.

$$\sum_{t \in [t_1,t_2]} \sum_{e \in E} a_e y_{iet} \leq C + C \sum_{t \in [t_1,t_2]} \sum_{j \in V} w_{ijt}$$

$\forall i \in B, \forall t_1 < t_2, t_1 + C + max\_aristas\_replicadas + 1 > t_2$

6) Si un barrendero depositó, entonces tiene que haber pasado por ahí en ese momento. En este caso, el vecindario no se toma por ser la esquina opuesta de una cuadra, sino que se utiliza N', que toma todos los que se encuentran a una distancia muy cercana (a parametrizar). El principal motivo para considerar esto, tiene que ver con que debido a la presencia de boulevards, lo que en la práctica sería una única esquina dada por la intersección de dos calles, al procesar el grafo, en realidad está dada por cuatro nodos distintos.

$$w_{ijt} \leq \sum_{j' \in N'(V)} z_{ij't} \forall i \in B, \forall j \in V, \forall t \in T$$

7) Si un barrendero depositó entonces la esquina se usa como montículo.

$$x_j \geq w_{ijt} \ \forall i \in B, \forall j \in V, \forall t \in T$$

Este modelo, sin embargo tenía una cantidad demasiado grande de variables y restricciones como para poder resolverse. Por ejemplo, la cantidad de variables, tomada a partir de las características promedio de la zona 1, ($|E_i| = 30$ para todo i, $|B| = 30$, $|T| = 32$ y $|V| = 600$), asciende a 1.100.000.

Estrategia 2: Modelo de PLE utilizando previamente Hierholzer

Surgió entonces el planteo de un modelo alternativo, esta vez utilizando previamente el algoritmo de Hierholzer sobre cada uno de los subgrafos dados por el recorrido de un barrendero. Notar que el hecho de recorrer ordenadamente todas las cuadras exactamente una vez, comenzando y terminando desde un mismo punto, equivale a encontrar un tour euleriano en un grafo par. Luego, dado que replicamos aristas, y por ende el grafo es par, Hierholzer permite obtener el tour, con la ventaja de tener complejidad lineal en la cantidad de aristas.

Con este cambio, entonces, algunas de las variables, como y e z, y restricciones vistas en el modelo anterior, carecen de sentido. De hecho, las variables y y z, ahora pasan a ser un parámetro (obtenido luego de correr Hierholzer) y por ende, las cuatro primeras restricciones son eliminadas.

1. Conjuntos

- $T = \{1, \ldots, t_f\}$ Conjunto de intervalos de tiempo.
- $V = \{v_1, \ldots v_n\}$ Conjunto de esquinas.
- $B = \{1, \ldots k\}$ Conjunto de barrenderos de la zona.
- $E_i = \{(v_{i_1}, v_{j_1}), \ldots (v_{i_m}, v_{j_m})\} \subset V \times V$ Conjunto de cuadras asignadas del barrendero i.
- $E = \{(v_{i_1}, v_{j_1}), \ldots (v_{i_m}, v_{j_m})\} \subset V \times V$ Conjunto de cuadras.

2. Parámetros

- $$a_e = \begin{cases} 1 & \text{si la cuadra e solo se recorre caminando} \\ 0 & \text{si no} \end{cases}$$

  $e \in E_i$, $\forall i \in B$.

- $$y_{iet} = \begin{cases} 1 & \text{si el barrendero i pasa por la cuadra e a tiempo t} \\ 0 & \text{si no} \end{cases}$$

  con $i \in B, e \in E_i, t \in T$

- $$z_{ijt} = \begin{cases} 1 & \text{si el barrendero i comienza barriendo desde la esquina j a tiempo t} \\ 0 & \text{si no} \end{cases}$$

  con $j \in V, t \in T$ (es decir por ejemplo si se barre la cuadra (i, j) a tiempo t, en el sentido j-i entonces $z_{mjt} = 1$ y $z_{mit} = 0$).

3. Variables

$$x_j = \begin{cases} 1 & \text{si la esquina j se usa para depositar un montículo} \\ 0 & \text{si no} \end{cases}$$

con $j \in V$.

$$w_{ijt} = \begin{cases} 1 & \text{si el barrendero i deposita la basura en el montículo j a tiempo t} \\ 0 & \text{si no} \end{cases}$$

con $i \in B, j \in Vt \in T$

4. Función objetivo

Dado que ahora se aplica previamente el algoritmo de Hierholzer, y por lo tanto y es simplemente un parámetro, la primera parte se eliminó. Si bien es cierto que Hierholzer devuelve un ciclo que podría empezarse desde cualquier nodo, la idea de este modelo era reducir la cantidad de variables y restricciones, por lo que dejar ese grado de libertad no era algo deseable.
Minimizar $\sum_{j \in V} x_j$

1) Los barrenderos pueden llevar a lo sumo la carga de C cordones.

$$\sum_{t \in [t_1, t_2]} \sum_{e \in E} a_e y_{iet} \leq C + C \sum_{t \in [t_1, t_2]} \sum_{j \in V} w_{ijt}$$

$\forall i \in B, \forall t_1 < t_2, t_1 + C + max\_aristas\_replicadas + 1 > t_2$

2) Si el barrendero depositó en j a tiempo t, entonces tiene que haber pasado por ahí o muy cerca del montículo en el momento.

$$w_{ijt} \leq \sum_{k \in N'(j)} z_{ikt}, \forall i \in B, \forall j \in V, \forall t \in T$$

3) Si un barrendero depositó, entonces la esquina se usa como montículo.

$$x_j \geq w_{ijt}, \forall j \in V, \forall i \in B, \forall t \in T$$

Sin embargo, nuevamente la cantidad de variables y restricciones era demasiado grande, como para poder resolverse. Así que finalmente se procedió a correr el modelo por separado para cada barrendero, con la diferencia que ahora lo que cambian son los conjuntos, reemplazados por subconjuntos del modelo anterior:

- $T = \{1, \ldots, t_f\}$ Conjunto de intervalos de tiempo.
- $V = \{v_1, \ldots v_n\}$ Conjunto de esquinas del barrendero.
- $B = \{i\}$

- $E_{=}\{(v_{i_1}, v_{j_1}), \ldots (v_{i_m}, v_{j_m})\} \subset V \times V$ Conjunto de cuadras asignadas del barrendero.

Dado que con esta solución podría llegar a perderse mucha optimalidad (en la parte de los resultados se analizará esto con más detalle), se diseñó también una heurística muy simple que si encuentra dos montículos de barrenderos distintos, muy cercanos entre sí, unifica todo en un único montículo y actualiza los recorridos en coherencia con este cambio.

Algoritmo 10 Algoritmo postprocesado de montículos

Entrada: Rutas= Rutas de los barrenderos
Entrada: listaMonticulos = Lista de montículos de los barrenderos

```
1: para barrendero hacer
2:     para barrenderoVecino hacer
3:         para monticulo en monticulos(barrendero) hacer
4:             para monticuloVecino en monticulos(barrenderoVecino) hacer
5:                 si distancia(monticulo,monticuloVecino)<20  monticulo != monticu-
   loVecino entonces
6:                     listaMonticulos ← borrarMonticulo(monticulo)
7:                     Rutas ← agregarMonticulo(barrendero,monticulo)
8:                 fin si
9:             fin para
10:        fin para
11:    fin para
12: fin para
13:
       devolver Rutas, listaMonticulos
```

4.2.3. Tercera etapa: Recorrido de los camiones

Una vez fijados los puntos de acumulación de montículos de hojas, comienza esta etapa que determina en qué orden van a ser recolectados por los camiones de la zona, de manera tal que se reduzca la distancia recorrida. Como ya dijimos, luego de conversarlo con la gente de la Municipalidad, y a sugerencia del equipo académico, se decidió que el turno de los camiones estuviera desfasado completamente con el de los barrenderos.

Naturalmente, al ser camiones y no personas, al problema se le suman las restricciones impuestas por las reglas de tránsito, que impiden que los camiones giren a la izquierda si hay un semáforo, realicen giros en U o que se recorra una calle de contramano.

Recordemos que para todos las zonas se tienen dos camiones distintos. Luego, por una cuestión de simplicidad, se separaron los montículos en dos mitades iguales divididas según la posición geográfica: una mitad dada por los montículos ubicados más al norte, y otra por los montículos ubicados más al sur.

Dado que la distancia cartesiana no resulta adecuada para medir distancias que recorre un vehículo, y la distancia de Manhattan no contempla el caso en que los giros a la izquierda no están permitidos, lo que se hizo fue aplicar el algoritmo de Dijsktra para calcular distancias. De esta forma se obtuvo el camino mínimo de un montículo a otro, de manera que se respeten las restricciones de giro.

Si bien Dijkstra se corrió n veces (cada vez tomando como nodo fuente un punto de depósito de montículo distinto), al utilizarse el grafo completo, en realidad se obtuvo la distancia desde ese montículo hacia todos los nodos del mapa (incluso aquellos en donde no se deposita nada).

Para resolver el problema de esta etapa, hay que tener en cuenta dos cosas: como se quiere obtener un circuito hamiltoniano de costo mínimo entre montículos, hay que crear un grafo auxiliar cuyo conjunto de vértices esté formado exclusivamente por ellos, y tal que el costo de cada arista sea igual a la longitud del camino mínimo obtenido con Dijkstra sobre el grafo original; a su vez como un circuito hamiltoniano se calcula sobre un grafo no dirigido, y en este caso el costo de ir de un montículo i a otro j no es el mismo que ir de j a i (es decir, es un TSP asimétrico), es necesario crear un nodo ficticio por cada nodo original, como ya se vio anteriormente en la sección dedicada al TSP. De esta forma, tenemos un grafo con tantos nodos como el doble de la cantidad de montículos.

Una vez calculado esto, se generó un archivo de input de Concorde, con las aristas y sus respectivos costos, del grafo recién mencionado.

Luego de obtener la solución de Concorde (software al cual se hará alusión en la sección 5.4), fue necesario volver a reconstruir el camino entre cada par de montículos, pero ahora en el grafo original. Esto se realizó a partir del output *padres* obtenido en Dijkstra: se comienza desde el nodo final, y se busca en quién es el predecesor. Este procedimiento se itera hasta que finalmente el predecesor es igual al montículo anterior determinado por la solución de Concorde. Dado que esto devuelve el camino en orden inverso, finalmente se invierte el orden del mismo.

Por último, para mostrar la solución obtenida de manera gráfica, se crea un archivo .osm, del cual hablaremos en el capítulo siguiente en la parte de procesamiento del mapa.

Capítulo 5

Implementación

En el presente capítulo, detallaremos cómo fue la estructura creada para representar el mapa de la ciudad, especificando qué elementos fueron necesarios para crear el grafo y que funciones sirvieron de apoyo para realizar dicha tarea.

También se explicarán detalles importantes de cada una de las etapas, que surgen exclusivamente de cuestiones geográficas de esta ciudad en particular. En adición a esto, se presenta muy brevemente una introducción a Concorde, dónde obtener el software y qué formato debe tener el archivo de input.

5.1. Procesamiento del mapa

Para poder determinar los recorridos, naturalmente fue necesario obtener el mapa de la ciudad. El mismo se extrajo de la base de datos de OpenStreetMap [1].

La información allí obtenida es de licencia libre, y a través del software JOSM (Java OpenStreetMap Editor), también gratuito, se puede visualizar y editar de manera sencilla los datos contenidos en el archivo de formato .osm. De esta forma la división en zonas de la ciudad fue una tarea sumamente simple. En [**20**] puede encontrarse un tutorial de uso de este software, mientras que en la figura 5.1 se muestra la interfaz del mismo.

El formato .osm contiene, entre otros datos, la siguiente información que fue de utilidad:

- **Esquinas o nodos**: Cada uno de ellos cuenta con sus respectivas coordenadas de geolocalización.
- **Caminos**: Pueden ser calles, ríos, vías de ferrocarril, etc. En el caso de las calles, contienen el sentido y nombre, junto con las esquinas que las componen.

Todos los elementos recién mencionados cuentan con un ID, y eventualmente con alguna etiqueta de identificación. Esto resultó de vital importancia a la hora de distinguir los semáforos, que imponen restricciones en los giros de los camiones.

Sin embargo, el archivo también presenta otros datos que no fueron relevantes, y por lo tanto tuvieron que ser filtrados, como por ejemplo nodos que representaban comercios,

[1] Puede verse aquí: https://www.openstreetmap.org

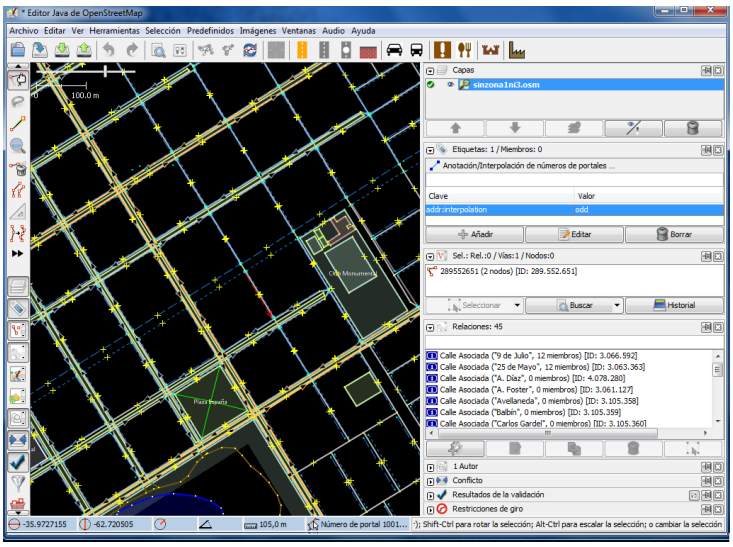


Figura 5.1: Interfaz de JOSM

```
<?xml version='1.0' encoding='UTF-8'?>
<osm version='0.6' generator='JOSM'>
  <bounds minlat='-35.9742206' minlon='-62.7602577' maxlat='-35.9473693' maxlon='-62.720561' origin='CGImap 0.6.0 (28372
thorn-01.openstreetmap.org)' />
  <node id='1804546171' timestamp='2017-06-24T02:31:34Z' uid='73016' user='dada' version='1' changeset='11995099' lat='-35.9759669'
lon='-62.7365945' />
  <node id='1804546197' timestamp='2017-06-24T02:31:34Z' uid='73016' user='dada' version='1' changeset='11995099' lat='-35.9765125'
lon='-62.7376792' />
  <way id='10' action='modify' timestamp='2017-11-03T22:43:45Z' uid='211657' user='dada' version='2' changeset='13740507'>
    <nd ref='1804546171' />
    <nd ref='1804546197' />
  </way>
</osm>
```

Figura 5.2: Archivo .osm

escuelas, paradas de transporte, etc, y caminos que eran vías del ferrocarril, arroyos, etc. Aquí [2] se encuentra el listado de todos los tags que se utilizan por convención para identificarlos, y que sirvió de referencia para eliminar la información irrelevante.

En 5.2, y a modo de ejemplo, se incluyen fragmentos de un archivo.

Como puede observarse, se tiene:

- Dos encabezados: uno XML con la codificación, y otro OSM.
- Los bordes del mapa: la latitud y longitud, máximas y mínimas, del área donde están ubicados los nodos.
- Los nodos, con su correspondiente ID, posición geográfica y algunos datos extra que surgen de la edición del archivo.
- Los calles, formadas por una lista de nodos, cuyo orden determina el sentido de las mismas. En el caso en que sea de doble sentido, al final de la misma se le agrega un etiqueta o tag con la información correspondiente.

[2] http://wiki.openstreetmap.org/wiki/Map_Features

5.1.1. Estructura de datos

El primer paso realizado fue procesar el archivo de texto plano .osm para generar una estructura de grafo.

Se usó la librería de Python imposm.parser, que permite trabajar con las distintos TADs ya mencionados que están presentes en el archivo (nodes, ways, etc). Más específicamente se usó el parser (también llamado analizador sintáctico) OSMparser.

La primera etapa de asignación de recorridos fue modelada mediante un grafo donde cada cuadra está representada por un vértice. Pero además, como veremos más adelante, al generar un programa para mostrar los recorridos en forma gráfica, fue esencial guardar también la información sobre las esquinas. Es por eso que finalmente se optó por generar el tipo abstracto de datos "Grafo" que contiene:

- **Esquinas**: Con su latitud, longitud, y la lista de calles a las cuales pertenece.
- **Cuadras**: Con el nombre de la calle a la que pertenece, el ID de las esquinas que la delimitan, su longitud, paridad de la numeración y la lista de cuadras adyacentes.

Para almacenar estos datos, se usó la estructura namedtuple de la librería Collections.

El algoritmo para generar el grafo comienza agregando las calles que están cargadas en el archivo .osm a un diccionario, entendiéndose una calle como un conjunto de nodos o esquinas, y que puede incluir también a una avenida en el sentido corriente. Este paso, a pesar de que las calles no se utilizan en ningún momento de la resolución del presente trabajo, es fundamental para luego establecer el vecindario de cada nodo y para determinar el nombre de la calle a la que pertenece cada cuadra.

Una vez agregadas las calles, se procede a generar un diccionario con todas las esquinas con su respectiva información (latitud, longitud, etc).

Por último se genera el diccionario con todas las cuadras y sus respectivos datos (nombre, paridad, etc, pero todavía sin las aristas vecinas) Esta parte fue la más complicada del algoritmo, ya que fue necesario tener un especial cuidado con la presencia de boulevards, ya que el lado de la calle que está pegado al mismo no necesita ser barrido, y por ende tampoco agregado grafo. Para eso se creó una función especial que identifica los boulevards. Si la cuadra cae en ese caso, entonces no se duplica (es decir, no es necesario que estén los dos cordones), y se identifica con una "u" (de única) en el ID. Caso contrario, se la duplica y se identifica con una 'p' o 'i' que surge de si es la cuadra con numeración par o impar. Cabe destacar que salvo el caso de los boulevards, la información brindada por OSM tiene una única cuadra que representa las dos manos de la misma. Finalmente una vez que todas las cuadras están agregadas, se actualizan los vecinos de cada una de las cuadras.

En el caso de los camiones, el grafo fue generado por una clase distinta. Básicamente el algoritmo para crearla fue bastante parecido pero con las siguientes diferencias:

- **Esquinas**: También con su respectiva latitud, longitud, la lista de calles a las que pertenecen y la lista de cuadras a la que pertenece, pero ahora se le suma un booleano que indica si tiene un semáforo.

- **Cuadras**: Con su respectivo nombre, las esquinas que la determinan y la longitud. Sin embargo, dado que ahora es un grafo dirigido, el ID fue determinado de la siguiente forma: "ID de la esquina 1, ID de la esquina 2", en donde el orden de las esquinas sí importa.

- Se realizó una función de validación de giro, que indica si se puede pasar de una cuadra a otra. Los criterios para determinar esto fueron: verificar que no es la misma cuadra pero en distinto sentido (giro en U), y en el caso de una calle con semáforos verificar que el giro sea a la derecha. Para eso, se utilizó la fórmula para calcular el ángulo (siempre en sentido antihorario) entre dos cuadras (pensadas como dos vectores). Si el mismo no está en un rango cercano a 90º, se decide que el giro no esté permitido.

5.1.2. Función distancia

Al igual que en [**3**], para calcular la distancia entre dos nodos se usó la función de [**21**]: dados dos puntos $p_1 = (lat_1, lon_1)$ y $p_2 = (lat_2, lon_2)$ con coordenadas que representan la latitud y longitud del mismo, se define:

$$a(p_1, p_2) = \left(sen\left(\tfrac{lat_2 - lat_1}{2}\right)\right)^2 + cos(lat_1)cos(lat_2)\left(sen\left(\tfrac{lon_2 - lon_1}{2}\right)\right)^2$$

Luego la distancia entre p_1 y p_2 es:

$$distancia(p_1, p_2) = 6371000 * atan2(\sqrt{a}, \sqrt{1-a}),$$

donde 6371000 es el radio de la Tierra en metros, y la función atan2 está dada por:

$$\text{atan2}(y, x) = \begin{cases} \arctan(\frac{y}{x}) & \text{si } x > 0, \\ \arctan(\frac{y}{x}) + \pi & \text{si } x < 0, y \geq 0, \\ \arctan(\frac{y}{x}) - \pi & \text{si } x < 0, y < 0, \\ \frac{\pi}{2} & \text{si } x = 0, y > 0, \\ -\frac{\pi}{2} & \text{si } x = 0, y < 0, \\ \text{indefinida} & \text{si } x = 0, y = 0. \end{cases}$$

5.2. Detalles importantes de la primera etapa

Si bien en la sección 4.2.1 se vio el algoritmo general para identificar manzanas, dado que el mapa no es un damero perfecto, existen casos especiales que la función debe identificar y resolver específicamente.

En la figura 5.3 se ve uno de ellos. En este caso, existen esquinas a las que sólo inciden tres aristas (en lugar de cuatro como ocurriría normalmente). Si el algoritmo logra identificar este escenario, automáticamente sigue avanzando en línea recta.

También hubo otros casos en donde la manzana no era un polígono cerrado, como por ejemplo en 5.4. Dado que eran sólo dos, se optó por agregarlas manualmente.

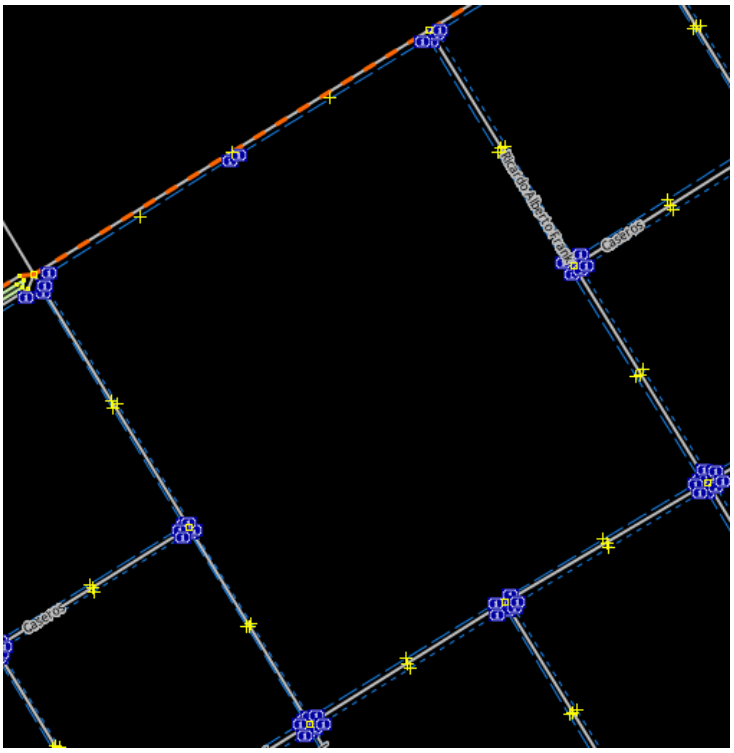


Figura 5.3: Caso especial: cuadra compuesta por dos aristas

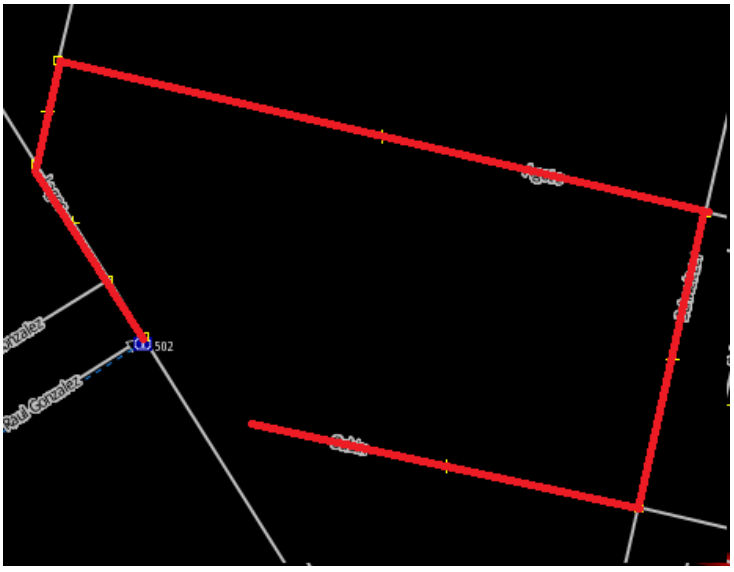


Figura 5.4: Caso especial: polígono no cerrado

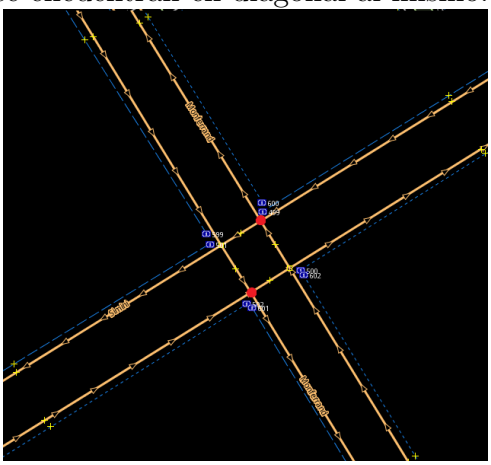
Figura 5.5: Caso especial:boulevard

Por último, algunas manzanas si bien eran polígonos cerrados, sólo estaban formadas por tres aristas (formando un triángulo). Podría haberse tratado como un caso especial, pero dado que eran pocas, también se optó por agregarlas a mano.

A la hora de implementarse, también se tuvo un especial cuidado en situaciones como 5.5. En el momento de la creación del grafo, algunas de las aristas en realidad no representaban cuadras, sino que eran segmentos que se formaban producto de la presencia de boulevards. Por una cuestión de conexión, no resultaba práctico excluirlas al generar ese grafo. Pero sí resultaba importante filtrarlas a la hora de identificar manzanas. Es por eso que empíricamente se descartaron las cuadras cuya longitud era menor a 19 metros. Es decir, cada vez que el algoritmo detectaba una cuadra que caía en ese caso, la manzana no se agregaba al conjunto total de manzanas. En la imagen, los polígonos marcados en rojo son ejemplos que no deben ser considerados manzanas, como sí debe serlo el azul.

5.3. Detalles de la segunda etapa

Como se indicó en el modelo II resuelto aplicando previamente Hierholzer, en la restricción 2 se utiliza un tipo de vecindario especial, que apunta a identificar aquellos cuatro nodos de una misma intersección que surgen por la presencia de boulevards. Dado que todos ellos tienen un ancho de menos de 20 metros, se tomó como parámetro de distancia esa longitud. La forma para identificar estos vecinos fue la siguiente: primero se buscan los vecinos del vértice en cuestión que estén a menos de 20 m. Hasta este paso, todavía están excluidos los vértices que se encuentran en diagonal al mismo:



Por lo tanto, es necesario buscar a aquellos vértices que son vecinos de los vecinos del vértice original, pero que se encuentren a 29 m del mismo (distancia obtenida mediante el teorema de Pitágoras). De esta manera pueden obtenerse los tres vértices deseados.

5.4. Concorde

Concorde [3] es el programa aquí utilizado para resolver las instancias de TSP. Fue creado por David Applegate, Robert Bixby, Vasek Chvatal y William Cook en el Instituto de Tecnología de Georgia. Es de licencia libre para uso académico, y su última versión data de 2003. Sin embargo, hasta la fecha es el programa más eficiente, ya que permite resolver en forma óptima instancias de hasta 1000 elementos. Concorde utiliza programación lineal entera para resolver en forma exacta instancias del TSP. Para eso se vale de un solver, que puede ser Cplex o QSOpt [4] (desarrollado por los mismos autores, y también de licencia libre).

Concorde permite utilizar diversos algoritmos y heurísticas para resolver el problema: branch and cut, la heurística Chained Lin-Kernighan, entre otras.

Para resolver este problema se usó un algoritmo de cut and branch con ramificación en profundidad. Esta variante es recomendada para los casos en que no hay una buena heurística para producir una solución inicial, ya que crea una solución completa, y por ende una cota superior.

Concorde admite diversos formatos de archivo para el input: .tsp, .dat, .edg, .mas. En

[3] Puede obtenerse aquí: http://www.math.uwaterloo.ca/tsp/concorde/downloads/downloads.htm

[4] Puede descargarse aquí: http://www.math.uwaterloo.ca/∼ *bico/qsopt/downloads/downloads.htm*

general todos contienen la misma información: la cantidad de nodos y y los costos de las aristas, aunque difieren en la forma de escritura. Por ejemplo en el formato .tsp la información se ingresa a partir de una matriz de costos, mientras que los .dat admiten ingresar la información de los nodos a través de sus coordenadas o en formato de aristas.

En este caso, se eligió un archivo .dat, con el último formato mencionado, que tiene la siguiente estructura:

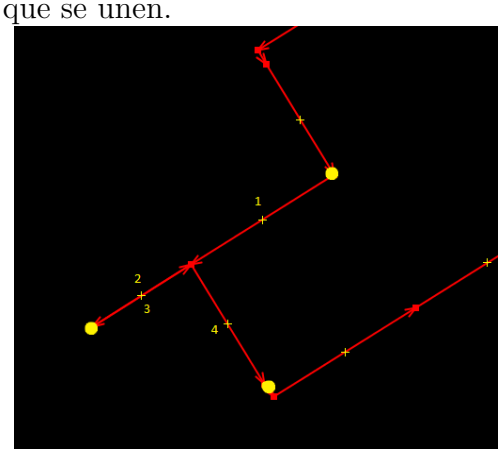
$$\#\text{nodos } \#\text{aristas}$$
$$v_{i_1}\ v_{i_2}\ c(v_{i_1}, v_{i_2})$$
$$\dots$$
$$v_{i_{n-1}}\ v_{i_n}\ c(v_{i_{n-1}}, v_{i_n})$$

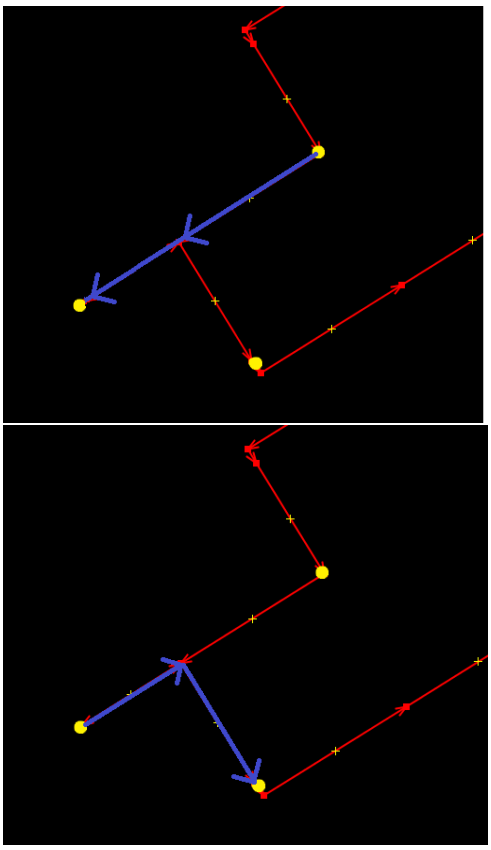
La elección de este formato sólo responde al hecho de que era más sencillo interpretar la solución de Concorde que con un archivo de coordenadas.

Para más información sobre la instalación y uso, se puede consultar [**22**].

5.5. Detalles de la tercera etapa

Dado que algunas de las zonas de la ciudad presentan una gran cantidad de cuadras de doble mano (calles sin boulevard), representadas por un mismo *way* , esto supuso un obstáculo para el algoritmo desarrollado en esta etapa. Si bien Dijkstra calculaba los caminos correctamente sin realizar giros en U, como a la hora de reconstruir el tour se unían caminos obtenidos de diversas corridas del algoritmo, existían algunos casos en donde esa "unión" violaba esta restricción. A continuación se muestra un ejemplo en donde ocurre. Los círculos amarillos representan los puntos de depósito de montículos, mientras que los números representan el orden en que se recorre. En las últimas dos imágenes se observan los dos caminos que se unen.





La solución a este problema fue entonces colocar una marca en la arista en donde se daba el giro en U, y volver a correr una versión modificada de Dijkstra, sólo desde el montículo problemático, pero ahora tomando también como input esta arista marcada, para que cuando genere el camino no la utilice. Para ello, se modificó la función que calcula los sucesores de un nodo para que no devuelva ese sucesor en particular. Dado que la morfología de un mapa permite tener muchos caminos entre dos puntos de distancia similar, la optimalidad del camino por lo general no se vio afectada, como sí hubiera sucedido si se hubiera implementado un algoritmo para dar una vuelta manzana hasta llegar al punto previo.

Capítulo 6

Análisis y Resultados finales

La exposición de los resultados se realizará separándola en cada una de las etapas. Para el caso de la primera, que todavía se encuentra en proceso de implementación en la Ciudad, se mostrarán los resultados de todas las zonas con sus diversas parametrizaciones, mientras que las otras dos etapas se analizarán con un caso en particular, simplemente para mostrar el potencial de mejora que se puede alcanzar una vez que se lleve a la práctica la totalidad de la solución. Este análisis acotado también se fundamenta en el hecho de que la resolución de las últimas dos etapas no involucra tanta libertad en la parametrización, como sí la tiene la etapa inicial.

6.1. Etapa 1

Como ya se mencionó anteriormente, en el modelo de programación lineal de esta etapa se utilizaron tres valuaciones distintas. Dado que para calcular las áreas de cada segmento generado se utilizó el sistema de coordenadas dado por la latitud y longitud, inicialmente la magnitud de las mismas rondaban el orden de 10^{-5}. Por este motivo, se reescalaron las valuaciones multiplicando por 10^6.

Con respecto a la generación de segmentos, dado que la cantidad de ellos era muy grande, se impuso una restricción al momento de aceptar un segmento factible, descartando aquellos cuya relación entre el área de la cápsula convexa y el área real del segmento era muy grande. Empirícamente se tomó como constante de tolerancia de dicha relación 1,5. De esta forma, la zona 1, con 200 manzanas, pasó de generar 803.000 segmentos a tan sólo 225.000. Por su parte, la zona 2, con 253 manzanas pasó de 895.000 a 200.000, y finalmente la zona 3, con 166 manzanas, pasó de 129.000 segmentos a tener 73.000. Todos estos valores están tomados en promedio a partir de las distintas corridas realizadas.

En la figura 6.1 se observan los resultados obtenidos por el modelo con la valuación que tenía en cuenta la distancia máxima entre el centro de masa del segmento y el de las manzanas. Los cuadrados adyacentes pintados con el mismo color corresponden al segmento asignado a cada operario. Como podemos ver, la mayoría de los segmentos están formados por tres o cuatro manzanas que se ajustan bastante a la forma de un cuadrado (aunque incompleto en algunos casos), lo que sugiere que esta valuación logra captar la esencia de lo que queríamos, que era tener poca dispersión de los recorridos.

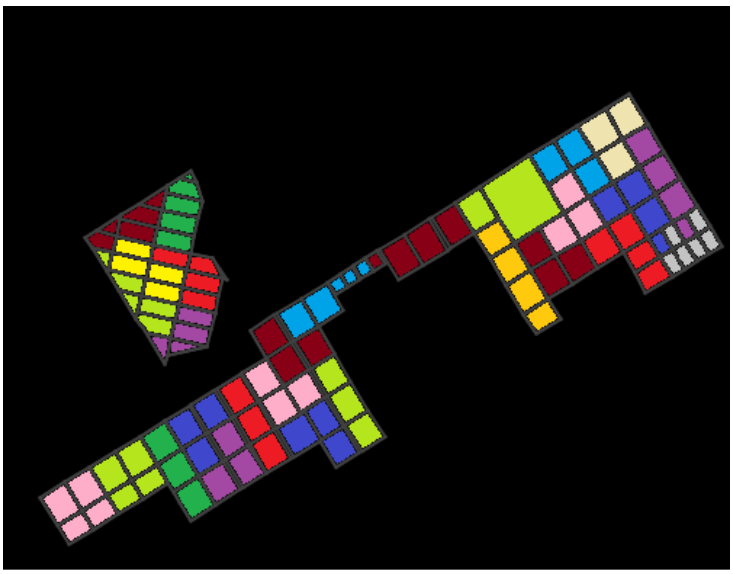


Figura 6.1: Resultados de un fragmento de la zona 3 con la valuación que mide la expansión del segmento.

En 6.2 se encuentran los resultados de utilizar la valuación que está dada por el cociente entre el área del segmento y el área de la cápsula convexa. Los resultados indican que esta estrategia no parece ser demasiado buena a la hora de encontrar segmentos cuya cápsula convexa sea similar a él, probablemente por cuestiones numéricas.

Por último, en 6.3 pueden observarse los resultados con la valuación dada por la diferencia entre el área de la cápsula convexa del segmento y el área de este último. La misma tiende a obtener segmentos más "alargados" que las anteriores. Para la Municipalidad, este tipo de solución era más deseable, ya que les resultaba más fácil monitorear que se cumpla con el barrido, y por lo tanto, finalmente fue la elegida.

Los resultados mostrados en estas últimas tres imágenes pertenecen a un fragmento de la zona 3 (correspondiente a las manzanas que se barren diariamente). Debido a las características morfológicas de esta zona en particular, que limitaban bastante la generación de segmentos, en este caso las manzanas se categorizaron en tres grupos distintos. La misma se acordó con la municipalidad, y está íntimamente relacionada con el tamaño de la manzana.

- Categoría 1: Manzanas que insumen 40 minutos.
- Categoría 2: Manzanas que insumen 50 minutos.
- Categoría 3: Manzanas que insumen 60 minutos.

En este caso la proporción de manzanas con respecto al total fue aproxidamente 50 %, 25 % y 25 % para las categorías 1, 2 y 3 respectivamente.

Esta categorización sólo se utilizó para la zona 3, mientras que en las otras dos zonas sí se usó el parámetro temporal ya mencionado.

Con respecto a la categorización de las manzanas según la frecuencia, la zona 3 es la única que tiene una subzona que se barre diariamente y otra que se barre día por medio. Luego,

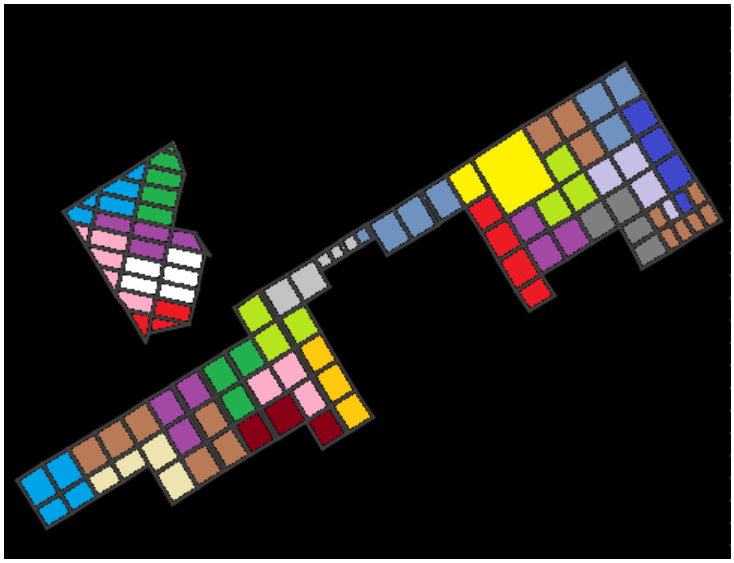


Figura 6.2: Resultados de un fragmento de la zona 3 con la valuación que calcula el cociente entre el área del segmento y el de la cápsula convexa.

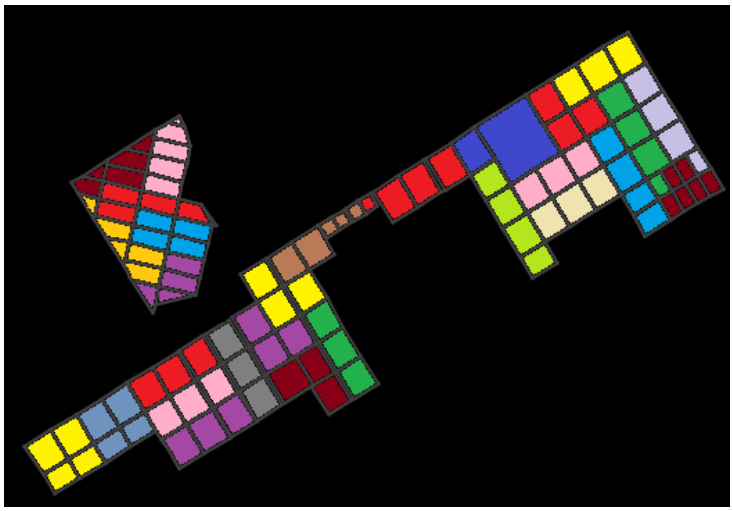


Figura 6.3: Resultados de un fragmento de la zona 3 con la valuación que calcula la diferencia entre el área del segmento y el de la cápsula convexa.

para resolver el modelo, lo que se hizo fue tratarlo como si fueran dos zonas separadas (tanto en el proceso de generación de segmentos, como en el modelo de PLE).

En el caso de las otras dos zonas, sólo hay dos o tres manzanas que tenían una frecuencia que no era diaria. Esto provocaba que al correr el modelo las mismas no estuvieran incluídas en ningún segmento (ya que el tiempo que tardaba en barrerse un segmento formado sólo por ellas era muy poco y por lo tanto no era considerado como factible). Finalmente, con el previo acuerdo de la municipalidad, las mismas se pasaron a la categoría de barrido diario.

Con respecto a la dificultad de cada manzana, dado que el tiempo de barrido varía dependiendo de cada operario, para la zona 1 se consideraron tres parámetros distintos (a testearse empíricamente), para así elegir el que mejor se ajuste a la realidad. Por una cuestión de simplicidad a la hora de correr el modelo, en lugar de variar el parámetro $\frac{\text{Longitud de la manzana} \times \text{Cantidad de minutos que se tarda en barrer una cuadra}}{100}$, este número se fijó en 0.11, y lo que se fue parametrizando fueron los límites inferior y superior del tiempo que insume barrer un determinado segmento, como para que sea considerado válido. Es decir, si estoy entre 240 y 360 considero todos los segmentos que puedo barrer entre 240 y 360 minutos, con la velocidad de barrido fija. De esta forma, para la zona 1, los límites fueron:

- Entre 320 y 400 minutos.
- Entre 300 y 360 minutos.
- Entre 240 y 350 minutos.

En las figuras 6.4, 6.5 y 6.6 se muestran los resultados finales obtenidos para la zona 1, con 28, 31 y 35 barrenderos respectivamente. La variación en la cantidad surge de cada uno de los parámetros recién mencionados. Es decir, la instancia parametrizada entre 320 y 400 minutos, que era considerando una velocidad mayor de barrido, se corresponde con la que tiene menor cantidad de barrenderos, y análogamente, las otras dos se pueden identificar a través de una relación inversamente proporcional entre el tiempo y la velocidad de barrido.

A diferencia de la zona 1, que previo a este trabajo era la más particular por la baja cantidad de barrenderos que tenía en proporción con la cantidad de cuadras; las zonas 2 y 3 se corrieron sólo con dos parámetros distintos. En el caso de la zona 2, los mismos fueron los siguientes:

- Entre 260 y 360 minutos.
- Entre 240 y 300 minutos.

Mientras que para la zona 3, se seleccionaron estos:

- Entre 240 y 320 minutos.
- Entre 200 y 260 minutos.

Los parámetros de la zona 3 son un poco menores que en las otras zonas, producto de las características morfológicas que surgen de tener dos frecuencias distintas, que hacen que las manzanas queden un poco más "aisladas".

En 6.7 y 6.8, se observan las soluciones con 36 y 40 barrenderos para la zona 2.

Por otra parte, en 6.9, 6.11 están los resultados para las manzanas de barrido diario en la zona 3, con 15 y 19 barrenderos respectivamente; mientras que en 6.10 y 6.12 están los resultados de las manzanas de barrido día por medio, con 5 y 6 barrenderos respectivamente. Notar que en realidad, en esas imágenes, la cantidad de segmentos es exactamente el doble de la de barrenderos mencionados. Esto se relaciona con el hecho de que el esquema de barrido se completa en dos semanas, por lo que un mismo barrendero barre dos segmentos distintos dependiendo del día de la semana. En este caso, la división de los segmentos en dos grupos se realizó tomando por un lado la mitad de segmentos más al oeste, y por otro los que están más al este. Con estos números, la cantidad de barrenderos total de esa zona es de 20 y 25, según la parametrización elegida entre las mencionadas anteriormente.

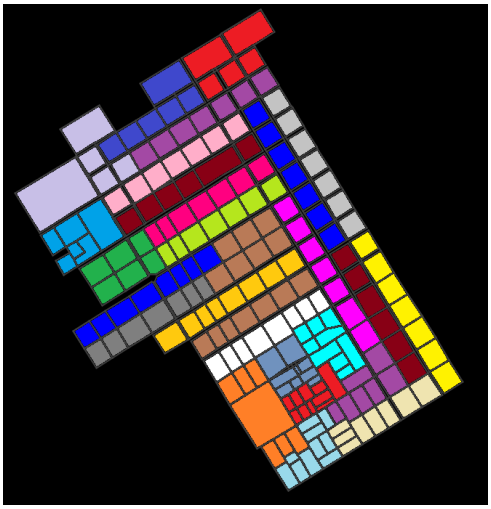


Figura 6.4: Resultados de la zona 1 con 28 barrenderos

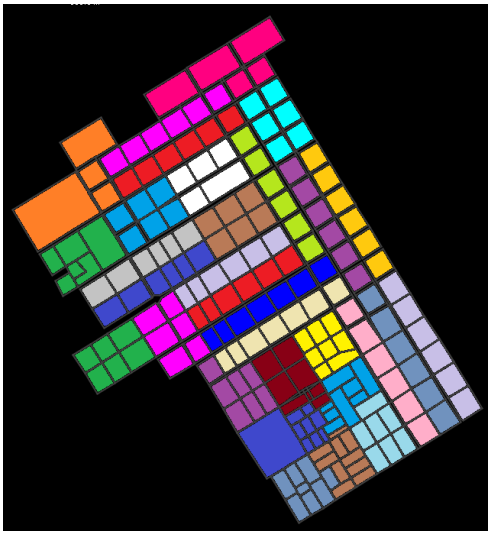


Figura 6.5: Resultados de la zona 1 con 31 barrenderos

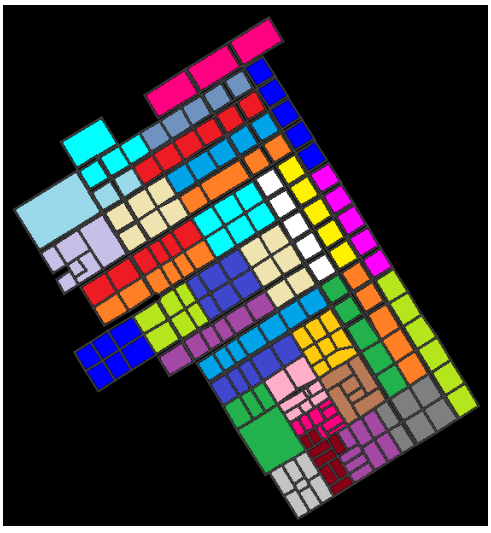


Figura 6.6: Resultados de la zona 1 con 35 barrenderos

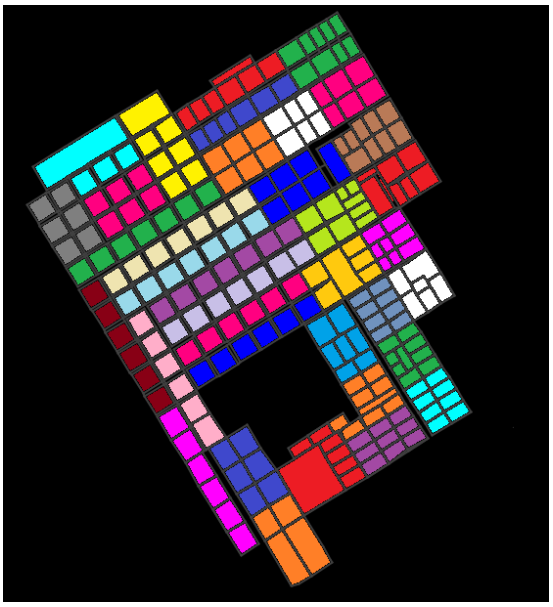


Figura 6.7: Resultados de la zona 2 con 36 barrenderos

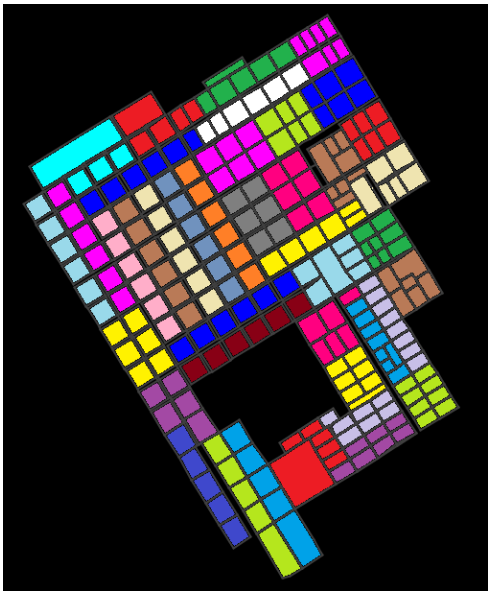


Figura 6.8: Resultados de la zona 2 con 40 barrenderos

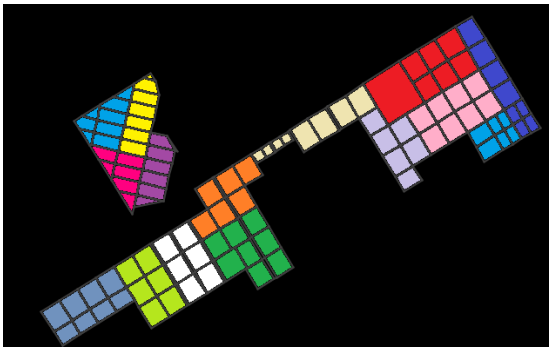


Figura 6.9: Resultados de la zona 3 con 20 barrenderos- Manzanas de frecuencia diaria

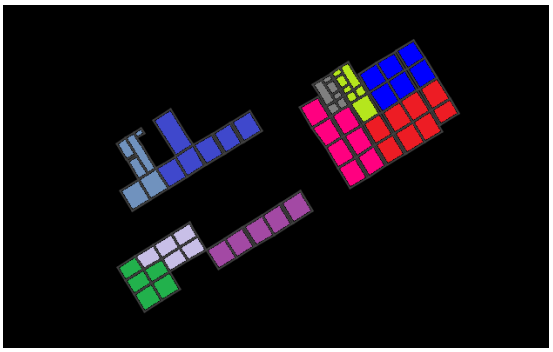


Figura 6.10: Resultados de la zona 3 con 20 barrenderos- Manzanas de frecuencia media

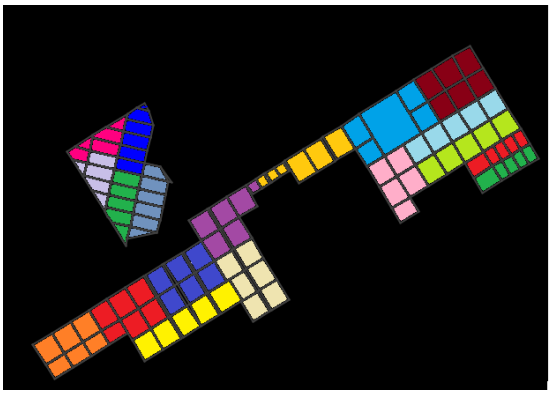


Figura 6.11: Resultados de la zona 3 con 25 barrenderos- Manzanas de frecuencia diaria

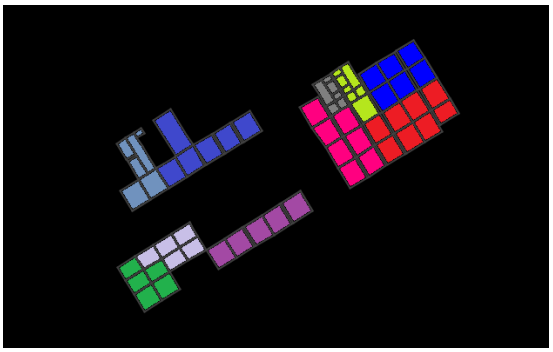


Figura 6.12: Resultados de la zona 3 con 25 barrenderos- Manzanas de frecuencia media

Un detalle a mencionar, es que en general los conjuntos obtenidos, al ser bastante compactos, son fáciles de identificar para los barrenderos. Este hecho resulta bastante apreciado por parte de los responsables técnicos de Trenque Lauquen.

A continuación podemos observar una tabla con algunos resultados numéricos, que indican cómo varía la proporción de barrenderos antes y después del presente trabajo.

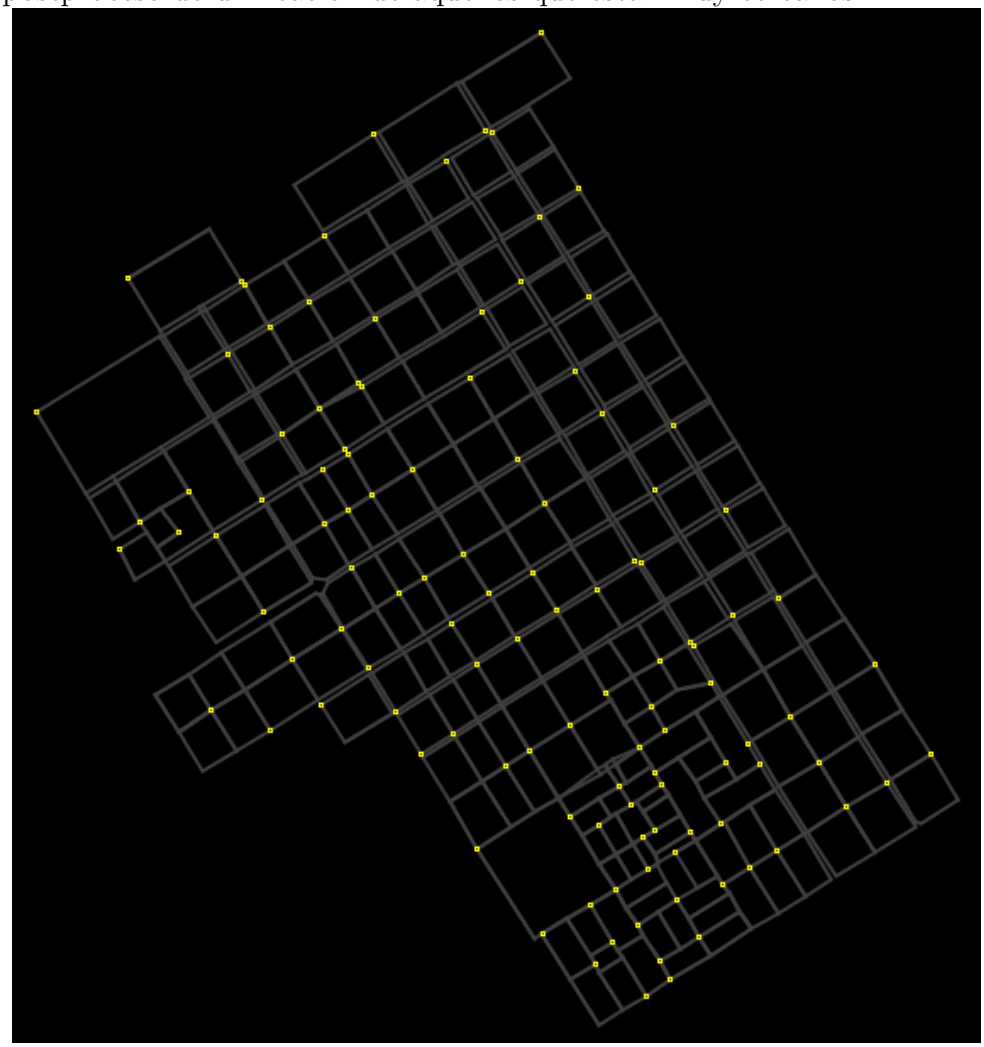
	Zona 1	Zona 2	Zona 3	Total
Cant. de operarios antes	21	35	28	84
Cant. de operarios después (promedio)	30	38	23	91
Variación en la cantidad de operarios	+43 %	+ 8 %	-18 %	+8 %

Como se ve, las estimaciones en la cantidad de operarios por zona no estaban realizadas de manera proporcional a la cantidad de cuadras. Mediante estos resultados, se puede llegar a la conclusión de que en la zona 3 este número estaba sobreestimado, mientras que en la zona 1 estaba considerablemente subestimado. Si bien esto se veía a simple vista, lo importante de este modelo, es que permite conocer qué tan alejado se está de un número más adecuado. La estimación promedio indicaría que son necesarios unos 7 barrenderos extra, lo que constituye un 8 % más de empleados.

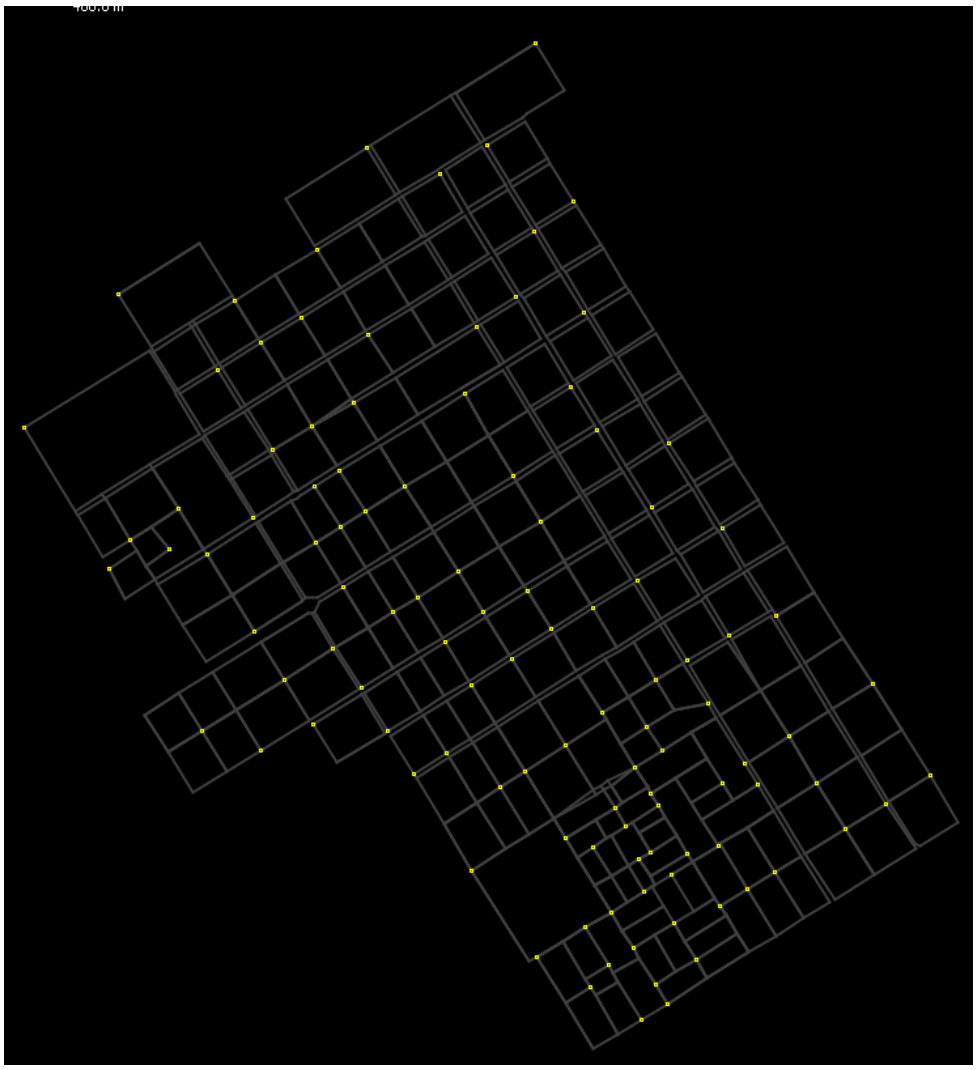
6.2. Etapa 2

Dado que en la actualidad la etapa I está en pleno proceso de implementación, y por ende aún no se eligió el modelo que mejor se ajusta a la realidad, los resultados de esta etapa y de la siguiente se mostrarán solamente con el ejemplo de la figura 6.5, que corresponde a la zona 1, con 31 barrenderos.

A continuación se observa cómo quedan determinados los 117 montículos antes de realizarse el postproceso de unificación de aquellos que están muy cercanos:

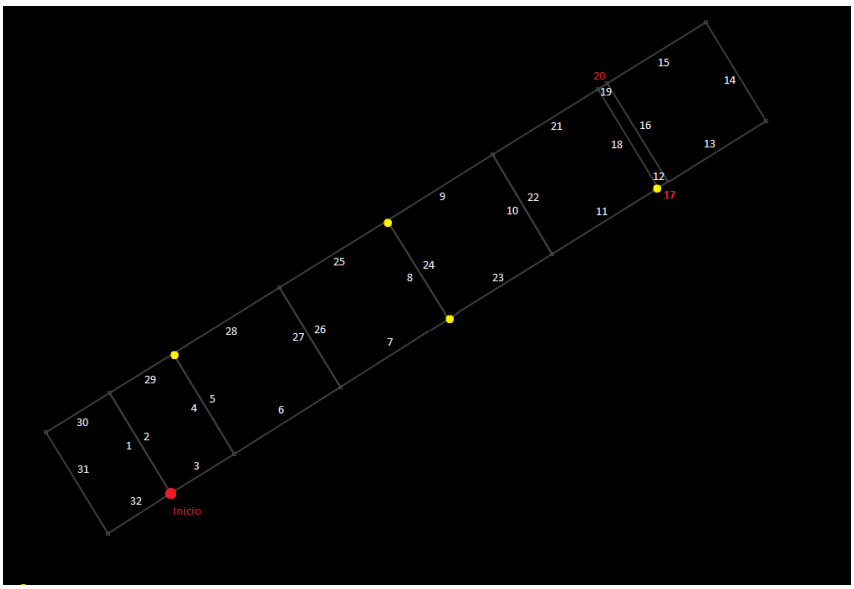


Luego del postprocesado, la cantidad pasó a ser 111 (es decir, se redujo en un 4,2 %). En la siguiente figura puede observarse cómo están determinados los puntos de acumulación de hojas una vez realizado el algoritmo de postprocesado. Los montículos que se fusionaron en algunos casos pertenecían a una misma intersección, pero eran utilizados por dos barrenderos distintos. Este problema obviamente surge del hecho de resolver el modelo en muchas subzonas distintas, y demuestra la necesidad de realizar este postprocesado.



Anteriormente al presente trabajo, la municipalidad depositaba montículos en todas las intersecciones, por lo que era ineficiente. Por ejemplo en la zona 1 había aproximadamente unos 240 montículos, es decir más del doble que la cantidad obtenida en este trabajo.

En general el modelo planteado demostró ser muy eficiente. Por ejemplo, para un recorrido de 7 manzanas (28 cuadras), sólo utiliza cuatro montículos distintos (aunque obviamente el barrendero deposita en cada uno de ellos más de una vez). A continuación puede observarse este ejemplo en particular. Las cuadras se numeran por el orden que son barridas, los puntos amarillos son los puntos fijos para depositar montículos, y las números en rojo representan cuadras que están duplicadas para obtener la eulerianidad del ciclo, y por ende sólo se recorren caminando.

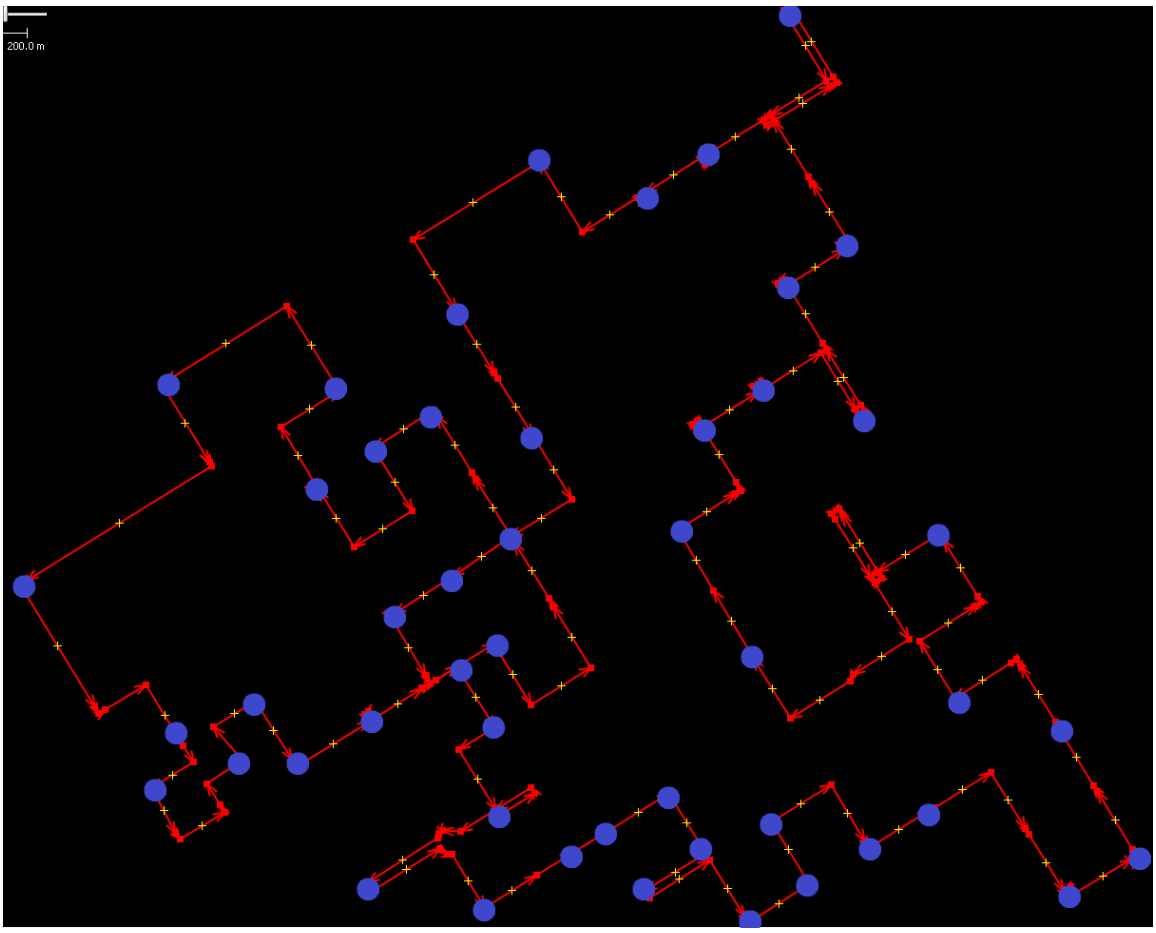


En el resto de los caminos, la situación es bastante similar: sólo se utilizan 3 o 4 montículos por barrendero, en donde se deposita múltiples veces.

6.3. Etapa 3

A continuación se muestran los resultados obtenidos para los recorridos de los dos camiones de la zona 1. Ambas instancias del TSP fueron resueltas en pocos segundos (en ambos casos fueron menos de 3). Debido a la creación de nodos ficticios, ambas instancias tuvieron un total de 112 nodos.

En la primera imagen, se encuentra el recorrido de la zona sur. Dado que los camiones se guardan en un depósito alejado de la ciudad, ubicado al noreste de la región urbana, el tour comienza en un montículo de la parte superior derecha del mapa. Lo mismo ocurre para la zona norte, ubicada en la segunda imagen. En ambas, los montículos a recoger están marcados con azul.



Previamente a este trabajo, dado que todas las esquinas se utilizaban como depósitos de montículos de hojas, los camiones fijaban su recorrido pasando por todas las cuadras de la ciudad en forma zigzagueante. Este nuevo recorrido, a pesar de parecer un poco más complejo a la vista, supone un gran ahorro en la distancia recorrida.

En términos numéricos, anteriormente se recorría un total de aproximadamente 32,2 km para los dos camiones en conjunto. El nuevo recorrido sólo insume 28,4 km entre ambos (14,7 km y 13,7 km para la zona sur y norte respectivamente), con lo cual se produjo un ahorro del 12 % en la distancia total.

Capítulo 7

Conclusiones y trabajo a futuro

Una idea interesante desde el punto de vista teórico, pero quizás un poco difícil de realizar en la práctica, hubiera sido distinguir a cada barrendero por su velocidad de barrido. Con esta información, para cada manzana se podría haber cronometrado el tiempo que ese trabajador tarda en limpiarla, y luego, dividiendo por una constante de velocidad de barrido de dicho operario, podría haberse obtenido una estimación absoluta para la dificultad temporal de cada manzana. De esta forma, se podrían haber conseguido soluciones mucho más exactas al ajustar el parámetro por la constante de cada barrendero.

Otro elemento que también podría haberse considerado sería sumarle a cada segmento una penalidad temporal según que tan alejado esté el centro de masa del punto de salida del barrendero. En la práctica, algunos de los barrenderos guardan los carritos en sus propias casas, con lo cual el punto de salida puede llegar a ser variable, y por ende no tuvo mucho sentido incluirlo en el modelo que aplicamos. Pero si se realizara la distinción por barrendero sí tendría un poco más de sentido.

En cuanto a los resultados obtenidos en este trabajo, la asignación eficiente de cuadras de los barrenderos era el objetivo principal para la Municipalidad. Como ya se mencionó, en la actualidad la ciudad se encuentra en plena fase de implementación de la etapa I. Mediante este trabajo se pudo reestimar la cantidad necesaria de operarios para cada una de las zonas, que previamente estaban desbalanceadas.

Paulatinamente se va incorporando personal al equipo de operarios, y se van implementando los nuevos recorridos presentados en este trabajo. Hasta el momento los resultados han sido satisfactorios teniendo en cuenta fundamentalmente la percepción de los responsables de organizar la tarea en la Municipalidad.

Un aspecto que probablemente se tenga en cuenta más adelante, es asociar a la dificultad de la cuadra o manzana cambios dados por motivos estacionales. Por ejemplo, una cuadra que tiene muchos árboles va a insumir más tiempo de barrido durante el otoño, época en se produce la caída de hojas. En estos casos, por ejemplo, una solución posible podría llegar a ser multiplicar el tiempo que insume barrerla por un factor de corrección adecuado.

Bibliografía

[1] Eglese R., Murdock H. *Routeing Road Sweepers in a Rural Area, J. Opl Res. Soc. Vol. 42, No.4, pp. 281-288, 1991*

[2] Bodin L., Kursh S. *A Computer-Assisted System for the Routing and Scheduling of Street Sweepers. Operations Research 26(4):525-537, 1978*

[3] Braier G., Durán G., Marenco J. and Wesner F. *"An integer programming approachto a real-world recyclable waste collection problem in Argentina". Waste Management and Research 35 (5) (2017), 525-533.*

[4] Bianchetti M., Durán G., Koch I., Marenco J. *"Algoritmos de zonificación para el problema de la recolección de residuos urbanos: El caso de estudio de una ciudad argentina". Revista Ingeniería de Sistemas (Universidad de Chile), Vol. 31 (2017), 81-110.*

[5] Flavio Bonomo, Guillermo Durán, Federico Larumbe y Javier Marenco. *"A method for optimizing waste collection using mathematical programming: a Buenos Aires case study" - Waste Manag Res 2012 30: 311 originally published online 1 April 2011*

[6] Kwan Mei-Ko. "Graphic programming using odd or even points", Chinese Mathematics 1 273-277, 1962

[7] Alexanderson, Gerald *"Euler and Königsberg's bridges: a historical view". Bulletin of the American Mathematical Society. 43 (4): 567, July 2006.*

[8] Edmonds J., Johnson E. *"Matching, Euler Tours and the chinese postman". Mathematical Programming 5, 88-124 (1973)*

[9] Frederickson, G. *"Approximation Algorithms for Some Routing". Problems J. Assoc. Comput. Mach. 26, 538-554 (1979)*

[10] Christofides, N. *"Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem". Report Nº388, Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh*

[11] Vasek Chvátal. *Linear Programming - McGill University, 1983*

[12] *http://artofproblemsolving.com/wiki/index.php?title=Shoelace_Theorem*

[13] Mauricio G.C. Resende, Celso C. Ribeiro. *Optimization by GRASP - Springer, 2016*

[14] Jesús Yordá Pérez. *El problema del cartero chino - Máster Interuniversitario en Técnicas Estadísticas (USC, Universidade Vigo, Universidade da Coruña) 2014-2015*

[15] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein *Introduction to Algorithms, MIT*

[16] Roy Jonker, Tom Volgenant *Transforming asummetric into symmetric traveling salesman problem- Operations Research Letters, Noviembre 1983*

[17] Little, Murty, Sweeny, Karel. *An Algoritm for the Traveling-Salesman Problem - OR, 11(1963) 972-89*

[18] Eastman, W. L. *Linear Programming with Pattern Constraints - Ph, D. Thesis, Report No. BL. 20, The computation Laboratory, Harvard University, 1958.*

[19] Francisco Javier Pigretti, Javier Marenco, Diego Delle Donne, Guillermo Durán. *Algoritmo de redistribución de barridos para la Ciudad de Salta - Informe Final- Instituto de Cálculo, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires (2015).*

[20] Flavia Bonomo, Diego Delle Donne, Guillermo Durán, Javier Marenco. *Automatic Dwelling Segmentation of the Buenos Aires Province for the 2010 Argentinian Census (2013) Interfaces 43(4):373-384.*

[21] *Java OpenStreetMap Editor Basic Manual*

https://ma.ellak.gr/edu/mod/resource/view.php?id=1088

[22] A. Hedges. *http://andrew.hedges.name/experiments/haversine/*

[23] Documentación Concorde

http://www.math.uwaterloo.ca/tsp/concorde/downloads/codes/src/970827/README

[24] Christos H. Papadimitriou, Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity - Dover Publications, Inc. 1982*

[25] *TSP con ventanas temporales*

https://acrogenesis.com/or-tools/documentation/user_manual/manual/tsp/tsptw.html#ferreira2010

[26] H.A Eiselt, Michel Gendreau, Gilbert Laporte *"Arc Routing Problems, Part I: The Chinese Postman Problem". Operation Research 43(2):231-242 (1995)*