DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Optimización de la recolección de residuos en San Carlos de Bariloche

4 de marzo de 2017

Alejandro Antuña
antuna.alejandro@gmail.com

Directora	Co-Director
Dra. Flavia Bonomo	Dr. Guillermo Durán

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

Índice

1. Introducción

1.1. Alcance de la tesis

El presente trabajo se enmarca dentro de un convenio entre la Facultad de Ciencias Exactas y Naturales y la Secretaría de Asuntos Municipales del Ministerio del Interior de la Nación. El objetivo del proyecto es el desarrollo de algoritmos de optimización para mejorar aspectos relacionados con la recolección de residuos en cuatro municipios de la República Argentina. Esta tesis desarrolla en particular el sub-proyecto correspondiente a la ciudad de San Carlos de Bariloche.

Nuestra tarea fue armar recorridos óptimos para la recolección de residuos en la ciudad de San Carlos de Bariloche. Era necesario entender de una forma completa la situación en la que se trabajaba y poder brindar caminos mejorados a fin poder realizar la recolección de una forma más eficiente.

El problema particular para esta ciudad consistió en recorrer todas las cuadras con un camino de distancia mínimo. Este problema es similar a un problema ampliamente estudiando en la literatura académica y conocido en la misma como **problema del cartero chino** que consiste es recorrer todas las cuadras de una ciudad minimizando la distancia total (es decir, minimizando las cuadras por las que se pasa más de una vez).

El modelo utilizado para trabajar fue un grafo, estructura matemática formada por nodos y aristas, donde cada nodo es una esquina de la ciudad conectado a la próxima esquina mediante una arista con dirección y con peso, siendo la dirección el sentido de circulación permitido sobre esa calle y el peso la distancia entre las esquinas.

El hecho de estar modelando las calles de una ciudad hace que el grafo tenga aristas dirigidas para representar el sentido en el que se puede circular. También tenemos cuadras en las que solo hay que pasar, no importa el sentido en el que se las recorra, y otras que sólo es necesario recorrer si mejoran la solución global, pero que no tienen residuos para levantar. Esto hace que tengamos un problema computacionalmente difícil de resolver. En términos teóricos, está demostrado que el problema del cartero chino en un grafo mixto donde no es necesario circular por todas las cuadras es NP-completo[4].

Para cumplir con el objetivo se resolvió un modelo de programación lineal entera en base al problema a optimizar, es decir, se modeló el problema con una serie de variables enteras, desigualdades lineales que involucran esas variables, y una función objetivo lineal a optimizar.

El primer paso para encarar este proyecto fue estudiar cómo se realizaba la recolección actualmente. Nos informaron que la ciudad está demarcada en 17 zonas y que un camión levanta los residuos de cada una de esas zonas en los días corres-

pondientes. No hay contenedores en uso por lo que hay que pasar por todas las cuadras.

En casi toda la ciudad alcanza con recorrer una calle en un solo sentido para juntar los residuos de ambos lados de la misma. Esto con excepción de las rutas y algunas avenidas. Esto nos obliga a marcar las aristas que cumplan una u otra condición y adaptar nuestro modelo a poder no utilizar ciertas aristas.

Dado que gran parte de la ciudad no es céntrica, no hay que preocuparse en limitaciones de giros. Como consideración de seguridad especial nos pidieron circular lo menos posible sobre las rutas.

Con toda esta información se armó el modelo para resolver este problema en particular. Teníamos algo similar al problema del cartero chino con algunas dificultades extra. Tenemos un grafo dirigido debido a las direcciones de las calles y solo es necesario recorrer un subconjunto de las aristas porque en casi todos los casos nos interesa pasar entre dos nodos solo en una dirección.

El último paso importante a resolver fue encontrar una forma de presentar los recorridos de tal manera que el personal de recolección lo pueda entender y utilizar. Necesitábamos encontrar un formato claro y preciso para presentar nuestros resultados.

1.2. San Carlos de Bariloche

La ciudad de San Carlos de Bariloche está ubicada al sureste de la provincia de Río Negro. Siendo parte del Parque Nacional Nahuel Huapi, se encuentra en la rivera del lago Nahuel Huapi, junto a la Cordillera de los Andes.

La ciudad cuenta con una población de 113.450 [1] habitantes y una superficie de 220 km^2

[1] Según el censo de 2010

2. Definiciones

2.1. Grafos

Un grafo G está definido por un par $< V, E >$ donde V es un conjunto finito y no vacío de *vértices* o *nodos* de G. E es un conjunto de *aristas* o *arcos* que son pares no ordenados de nodos.

Decimos que dos nodos v y w son **adyacentes** si hay un arco $(v, w) \in E$.

Si para cada par de nodos hay un arista o arco que los une, el grafo es **completo**.

El **grado** de un nodo v es la cantidad de arcos que tocan el nodo. Principalmente nos interesa si el grado de un nodo es par o impar.

Un **camino** P en un grafo es una secuencia de vértices $v_1, v_2, ..., v_n$ donde $(v_i, v_{i+1}) \in E, i = 1, 2...n - 1$. Se dice **camino cerrado** si empieza y termina en el mismo vértice. Un camino es un **circuito** si empieza y finaliza en el mismo nodo sin repetir arcos.

La **distancia** entre dos nodos es la cantidad de arcos que contiene el camino más corto entre ellos.

Si S es un subconjunto de V, el **subgrafo inducido** G' esta compuesto por los vértices de S y todos los arcos que empiezan y terminan en un nodo de S.

Un **grafo dirigido** G está definido por un par (V, E), donde V es un conjunto finito, el conjunto de nodos de G, y E es un conjunto de pares ordenados de nodos de G, llamados arcos o aristas. La cantidad de nodos se denota por n = $|V|$ y la de arcos por m = $|E|$.

De esta forma, podemos definir la adyacencia, camino y circuito en un grafo dirigido de la misma forma que en un grafo no dirigido, pero respetando el orden de los extremos de los arcos. Un grafo dirigido G es **completo** si para cualquier par de vértices hay un arco en cada dirección que los une.

El **grado interno** de un nodo v es la cantidad de de arcos que ingresan al nodo, es decir lo arcos de la forma $(w, v) \in E$. El **grado externo** es la cantidad de arcos que salen del nodo.

Se dice que un grafo dirigido es **fuertemente conexo** si para cada par de vértices u y v existe un camino de u hacia v y un camino de v hacia u. Dicho de otra manera, se puede llegar desde cada vértice del grafo hasta cualquier otro respetando el sentido de las aristas.

Un **grafo ponderado** tiene una función de ponderación que relaciona el conjunto de vértices o aristas con un conjunto de números. En el caso que los arcos es ten

padreados, la distancia entre dos nodos sera el camino de peso o ponderación total mínima entre ellos.

El costo de pasar por cada arco se puede denotar con una matriz de costos C tal que c_{ij} sea el peso de arco (i, j).

2.2. Camino y circuito Euleriano

Un *camino euleriano* es un camino que pasa por cada arista una y sólo una vez. Un *ciclo* o *circuito euleriano* es un camino euleriano que empieza y termina en el mismo nodo.

Supongamos que tenemos un grafo conexo $G = (V, E)$. Son equivalentes las siguientes afirmaciones:

1. $\forall v \in V$ grado(v) es par
2. G consiste de aristas de una unión disjunta de ciclos y de los vértices de esos ciclos.
3. G tiene un circuito euleriano

Un grafo simple conexo contiene un camino Euleriano si y sólo si el grafo contiene 0 ó 2 vértices de grado impar.

Un grafo dirigido conexo tiene un circuito euleriano si y sólo si los *grados internos y externos* de cada nodo son iguales. En caso de tener sólo un vértice tal que *grado externo de v - grado interno de v* $= 1$ y uno tal que *grado interno de v - grado externo de v* $= 1$ este grafo tiene un camino euleriano.

Si el grafo es mixto, cada vértice tiene que tener grado par de arcos dirigidos y no-dirigidos. Además tiene que ser *balanceado*: para cada $S \subseteq V$, la diferencia entre el número de arcos dirigidos de S hacia $V \setminus S$ y el número de arcos de dirigidos de $V \setminus S$ hacia S tiene que ser igual o menor al número de arcos no-dirigidos que unen S y $V \setminus S$.

2.3. Problemas de ruteo

Con el estudio de *problemas de ruteo de vehículos* se busca la forma óptima de recorrer diferentes puntos en un mapa. Generalmente lo que se busca optimizar es la distancia total recorrida, pero también pueden ser otros aspectos como por ejemplo tiempo, capacidad u orden de precedencia. Todos estos problemas son variantes del *Travelling Salesman Problem*, que consiste en un grafo con una distancia o peso asignada a cada arco y la necesidad de recorrer todos los nodos con un costo mínimo.

En los problemas de ruteo de arcos (ARP por su sigla en inglés, *Arc Routing Problem*) se busca la forma menos costosa de recorrer un subconjunto de arcos de un grafo, posiblemente teniendo en cuenta un conjunto de restricciones. Quizá el problema más famoso de análisis de arcos sea el de los *puentes de Königsberg* resuelto por Euler. Este problema consistía en determinar si se podía recorrer la ciudad utilizando todos los puentes pero solamente cruzando una vez cada uno.

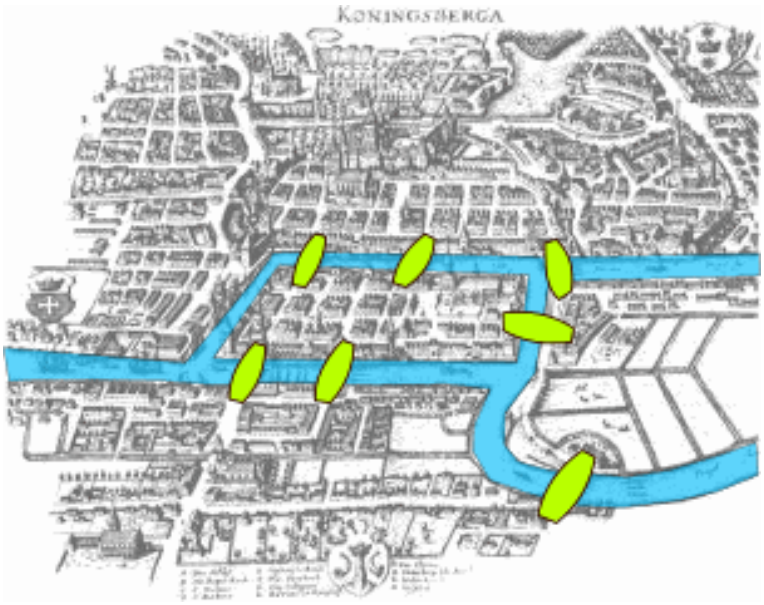


Figura 1: Puentes de Konigsber

Problemas de ruteo de arcos se presentan en múltiples contextos por ejemplo entrega de correo, limpieza de veredas, limpieza de rutas con nieve, colectivos escolares, etc. Se ve que en todos estos ejemplos es importante pasar el frente de todas las viviendas, no sólo tener que ir a puntos determinados.

Cada vez más es importante la eficiencia en logística de transporte. La investigación en el área es uno de los pilares del éxito de la Investigación Operativa, ya que se puede traducir a una gran ganancia económica y ambiental. Proveedores de soluciones y herramientas de ruteo estiman un potencial ahorro de entre 5 % y 30 %[7].

Para poner el problema en perspectiva, en la Unión Europea se calcula que el sector de transporte representa en el 9-10 % del PBI, cuenta con más de 10 millones de empleados y suele representar entre el 10-15 % de los costos de producción de una empresa[7].

2.3.1. CPP: Problema del cartero chino

Para resolver problemas como el de los puentes de Kônigsberg, Euler buscaba tener caminos cerrados, que no repitieran aristas. Pero en problemas dentro de un

contexto de optimización más reales no suele ser posible encontrar grafos que cumplan esa propiedad. El comúnmente conocido como **problema del cartero chino** es el que busca un camino cerrado de distancia mínima que pase por cada arista *por lo menos* una vez. Lo importante es cubrir todas las aristas en la distancia mínima, repitiendo las aristas que sean necesarias para lograr recorrer la distancia mínima.

El problema del cartero chino (CPP por sus siglas en inglés) es un problema ampliamente estudiado en la teoría de grafos. Su nombre lo obtuvo por Alan Goldman del U.S. National Bureau of Standards después de leer un paper de Kwan Mei-Ko, matemático chino que estudió esta problemática en 1960. [10]. El problema dice que hay un cartero que necesita entregar cartas a todas las casas de un barrio. Para caminar lo menos posible quiere encontrar el camino más corto que pase por le frente de todas las viviendas.

En teoría de grafos esto se podría expresar como que dado un grafo ponderado donde cada arco tiene un peso positivo queremos encontrar una camino cerrado de peso mínimo que pase todas las aristas. Expresado de otra forma, para el grafo $G = (V, A)$ con una matriz de costos $C = c_{ij}$ asociada a A se busca el camino cerrado de costo mínimo que recorra todos los arcos de A. Llamaremos a éste camino una **postman route**.

En el caso de tener una ruta euleriana en el grafo, éste sería el camino óptimo ya que pasaría una sola vez por cada arista. Entonces si cada nodo del grafo es de grado par y el grafo es conexo, el CPP se reduce a simplemente encontrar un camino euleriano, que sabemos existe en el grafo.

Armar una ruta dado un grafo euleriano requiere tiempo polinomial[4], Edmonds y Johnson[3] proponen el algoritmo *end-pairing* con complejidad $O(|V|)$ para un grafo dirigido. En el caso de un digrafo se puede utilizar el algoritmo *spanning arborescence* (van Aardenne-Ehrenfest y de Brujin), también en tiempo polinomial.

Por otro lado, si tenemos una postman route para el grafo G, sabemos que cada arista de G está en el camino *por lo menos* una vez. Sea c_e la cantidad de veces que se usa el arco e en el camino. Generamos un grafo G', que es el **grafo aumentado** de G, agregando $c_e - 1$ copias del arco e en G'. Ahora el camino del cartero en G pasa a ser un camino euleriano en G'. En el grafo G' cada nodo tendrá grado par.

Vemos que los algoritmos para resolver el CPP tienen dos etapas bien definidas. Primero generar un grafo aumentado de costo mínimo y después, recorrer ese grafo.

Si G es un grafo no-dirigido, un grafo aumentado Euleriano mínimo se puede derivar en tiempo polinomial resolviendo un problema de matcheo, utilizando el algoritmo de Edmonds[13].

Si el grafo es dirigido se lo modela como un problema de flujo de costo mínimo. En[3] se propone un algoritmo polinomial.

Entonces vemos que el CPP tanto para grafos como digrafos se puede resolver

tiempo polinomial.

Cuando el CPP es sobre un grafo mixto, ya estamos dentro de los problemas NP-Hard[4]. La dificultad está en encontrar el grafo aumentado. Vimos que para que un grafo mixto sea euleriano tiene que ser par y balanceado. Ford y Fulkerson presentan un método para resolver esta variante. Las mejores soluciones para resolver este problema son con programación lineal entera aplicando técnicas de branch-and-cut.[4][5]

2.3.2. Variantes del CPP

Las problemáticas del mundo real generan diferentes variantes del CPP. Cada una de ellas está muy estudiada académicamente, algunas de las más conocidas son:

Windy Postman Problem (**WPP**) introduce la variante que el valor del arco (i, j) puede diferir del arco (j, i). Con esto se pueden modelar variantes donde no es lo mismo recorrer en una dirección que en la otra, por ej, el viento, o para nuestro caso, que sea más caro hacer una subida que una bajada.

El WPP es de complejidad *NP-Hard*. En[6] se demuestra la equivalencia entre WPP y un problema de CPP para grafo mixto.

Rural postman problem (**RPP**) es una variante en la cual algunos arcos no son obligatorios. Esto es, para el grafo $G = (V, A)$ con su matriz de costos, queremos el camino cerrado mínimo que recorra todos los arcos de R tal que $R \subseteq A$. En la vida real, casi todos los problemas de ruteo de arcos son RPP con alguna restricción adicional.

El RPP es *NP-Hard*[5] tanto para grafos dirigidos como no-dirigidos, a menos que $R = A$ en cuyo caso sería igual al CPP. La forma de resolverlo es similar a la que se utiliza para el CPP. Primero hay que encontrar un grafo aumentado y después armar un camino euleriano sobre ese grafo. La parte de armar el camino sabemos que está bien resuelta y requiere tiempo polinomial.

Dada la complejidad del RPP hay varias heurísticas para intentar aproximar soluciones, principalmente utilizando algoritmos de matcheo o de spanning tree. En este trabajo nos vamos a enfocar en soluciones exactas. En todos esos casos se utilizan técnicas de programación lineal entera, con técnicas de branch-and-cut o branch-and-bound.

En[5] presentan esta tabla como resumen de las técnicas presentadas para resolver el problema.

Problem	Exact Algorithms	Heuristic Algorithms
Undirected **RPP**	NP-hard. Branch-and-bound ILP based algorithm: 24 instances solved ($9 \leq \|V\| \leq 84$, $13 \leq \|A\| \leq 184$, $4 \leq \|R\| \leq 78$). Corberán and Sanchis (1991b) solve 23 of these instances without branching	Heuristic with worst-case ratio of 3/2 (Frederickson 1979)
Directed **RPP**	NP-hard. Branch-and-bound ILP based algorithm (Christofides et al. 1986): 23 instances solved ($13 \leq \|V\| \leq 80$, $24 \leq \|A\| \leq 180$, $7 \leq \|R\| \leq 74$)	Christofides et al. (1986) proposed a heuristic: on 23 instances, the average deviation from optimality was 1.3% and 10 instances were solved optimally

2.4. Programación lineal entera

En la sección anterior vimos que es necesario aplicar técnicas de programación lineal entera para poder resolver un RPP sobre un grafo mixto.

La programación lineal es el campo de la optimización matemática dedicado a optimizar, maximizando o minimizando, una función lineal. Esta función objetivo tiene variables sujetas a una serie de restricciones expresadas mediante un sistema de inecuaciones también lineales.

Un *problema de programación lineal entera* es un problema de *programación lineal* con la restricción adicional de que algunas de las variables deben tomar valores enteros. Si todas las variables deben tomar valores enteros decimos que se trata de un problema de programación lineal entera puro, en caso contrario decimos que es mixto . Diremos que una variable es binaria si sólo puede tomar los valores 0 y 1. Una gran variedad de problemas combinatorios pueden ser planteados como problemas de programación lineal entera.

En su forma más simple los modelos de *programación lineal entera* tienen la siguiente forma:

Dada un matriz $A \in R^{nxm}$, vectores $b \in R^m$ y $c \in R^n$, y un subconjunto $I \subseteq N = \{1, ..., n\}$ se busca resolver

$$c^* = min\{c^T x | Ax \leq b, x \in R^n, x_j \in Z, \forall j \in I\}$$

El conjunto $x \in \{R^n | Ax \leq b, x \in R^n, x_j \in Z, \forall j \in I\}$ es el conjunto de las soluciones factibles. Decimos que es la solución óptima si $c^T x^* = c^*$

Una solución que toma valores no enteros es una relajación lineal:

$\check{c} = min\{c^T x | Ax \leq b, x \in R^n\}$

Por más que el problema general de programación lineal entera pertenece a la clase de problemas NP-Hard, hay formas eficientes de encarar cierto tipo de problemas para lograr soluciones en tiempos razonables.

Una forma simple de obtener una solución óptima es enumerar todas las solucio-

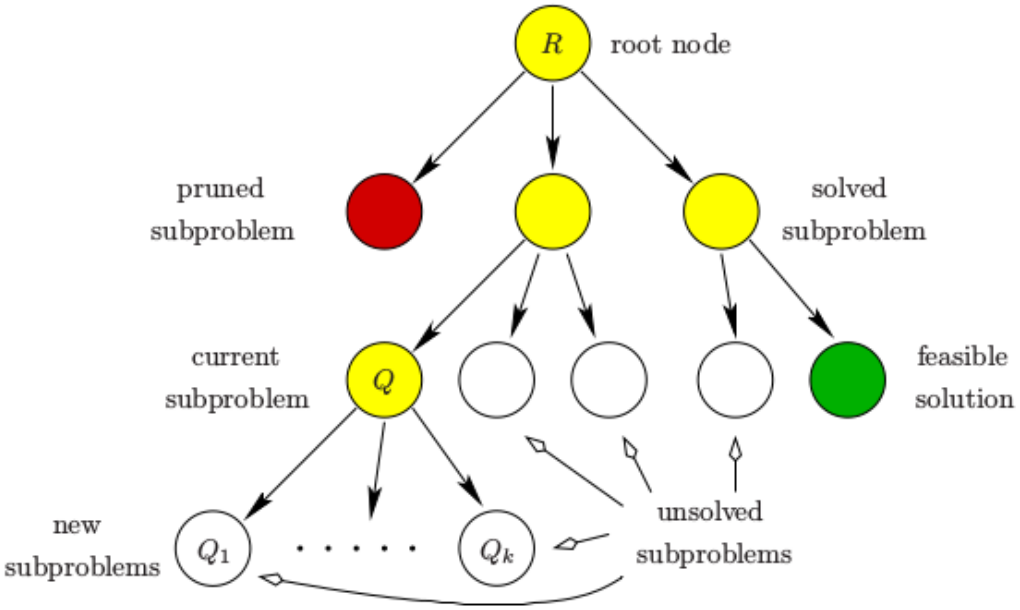


Figura 2: Árbol de branch-and-bound[2]

nes factibles y quedarse con una que minimice la función objetivo. Pero dado que la cantidad de combinaciones suele ser exponencial en la cantidad de variables es necesario buscar formas más eficientes de llegar a una solución.

Un **solver** es un software que toma como entrada el modelo a utilizar junto con todo el conjunto de datos necesario para poder optimizar la función objetivo utilizando técnicas de optimización lineal. En el caso del solver elegido se utiliza el algoritmo de *branch and bound*.[1][2][8]

Branch and Bound es un procedimiento muy utilizado para resolver problemas de optimización. La idea es dividir sucesivamente el problema en subproblemas más chicos hasta llegar a subproblemas que sean fáciles de resolver. La mejor solución de todos los subproblemas será la mejor solución global.

Partir el problema en subproblemas más chicos se llama *branching*. A medida que se aplica esta técnica se va generando un *branching tree* donde cada nodo representa un subproblema. La raíz del árbol corresponde al problema inicial y cada hoja será una instancia más "fácil", que ya se resolvió o que se dividió en subproblemas que tienen que ser procesados. La imagen 2 muestra como va avanzando el algoritmo.

Si nos limitamos al branching estaríamos haciendo una búsqueda de fuerza bruta de la solución óptima. Pero como se mencionó previamente, enumerar todas las soluciones no es una buena decisión ya que pueden ser infinitas o exponenciales en la cantidad de variables. El propósito del proceso de *bounding* es el de no tener que recorrerlas a todas, eliminando (prunning) subproblemas que se puede probar no van a contener una solución óptima. Para poder hacer esto hay que tener buenas

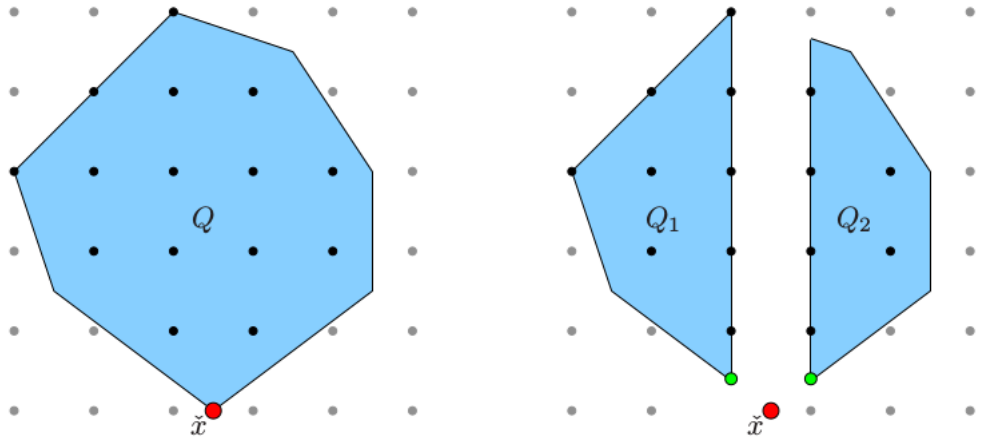


Figura 3: Branching sobre una variable no entera[2]

cotas inferiores y superiores. Las cotas inferiores (en un problema de minimización) se calculan con la ayuda de relajaciones, que tienen que ser fáciles de resolver.

Además de generar la cota inferior para usar durante el procedimiento de bounding, la relajación lineal también se suele utilizar como guía para el proceso de branching. Una estrategia es partir el dominio de un variable entera, $x_j, j \in I$ con valor fraccional $\check{x}_j \notin Z$ en dos partes, creando dos subproblemas a partir de Q (figura 3): $Q_1 = Q \cap \{x_j \leq \lfloor \check{x}_j \rfloor\}$ y $Q_2 = Q \cap \{x_j \geq \lfloor \check{x}_j \rfloor\}$.

2.5. Descripción del problema a resolver

El proyecto que se nos entrega es armar recorridos con distancia óptima para la recolección de residuos en la ciudad de San Carlos de Bariloche. El primer paso era entender de la forma más completa posible, cómo era que realizaban la recolección en la actualidad. Era importante entender el problema a resolver y qué tipo de restricciones nos íbamos a encontrar.

Mediante una serie de reuniones con personal de la municipalidad de San Carlos de Bariloche, fuimos recolectando los datos necesarios para entender la recolección actual. Para realizar los recorridos tenían zonas fijas. Cada zona tiene un horario y días en los cuales un camión realiza el retiro de los residuos. Este camión sale del corralón y luego se dirige al Basurero Municipal para realizar la descarga. Nos informan que todos los camiones comienzan y finalizan sus recorridos en los mismos lugares. Querían mantener las mismas zonas ya que hace años las utilizaban y todo el personal las conocía, no querían que las modificásemos.

La gente saca sus residuos a tachos en las puertas de sus casas ya que en ninguna parte de Bariloche hay contenedores. Esto obliga al camión o los recolectores a pie a pasar por todas las cuadras de la ciudad.

Nos hicieron especial hincapié en la seguridad del personal de recolección. Nos mencionaron la preocupación por tener accidentes de tránsito y también en las lesiones de los recolectores que corren detrás de los camiones, principalmente causados por el terreno irregular de la ciudad. Respecto a los accidentes de tránsito habían tomado una medida, que era minimizar la entrada y salida hacia las rutas y avenidas de mayor tránsito, como las avenidas Bustillo o Pioneros, dos de las más utilizadas en gran parte de Bariloche. Entre ambas suman más de 32km de distancia y hay que tener en cuenta que hay viviendas a lo largo de las mismas. Como parte de la solución para minimizar su uso, pusieron un camión que exclusivamente recorre éstas rutas, de esta manera el mismo nunca debe salir del camino.

Un punto que nos interesaba tener en cuenta era si debido al poco tránsito de muchas zonas de Bariloche, se podía asumir que con pasar en una sola dirección de una calle doble mano se levantaban los residuos de ambos lados de la vereda. Sobre esto se nos informó que con excepción de las avenidas más grandes, en todos lados se podía levantar los residuos de ambas manos con sólo pasar en una dirección. En las zonas céntricas los recorridos se hacían por la noche por lo que en ese caso tampoco era un problema tener que hacer ambas manos de las calles que no eran principales.

En Bariloche gran parte de las calles son de tierra y hay muchos desniveles. Nos indicaron que en algunas calles el camión no puede circular, pero no tenían registro de cuáles eran esas calles, sólo que los choferes sabían cuales eran. En esas calles personal a pie junta todas las bolsas y lo acerca a las esquinas.

Preguntamos también cuanto afecta el clima y el turismo, ya que no son factores menores para el funcionamiento de la ciudad. Sobre el clima lo que hace el chofer es decidir por cuáles calles puede circular cuando hay nieve y por cuáles no. Sobre el turismo, en algunas zonas, principalmente hoteleras, se juntan muchos más residuos. Nos dicen que si un camión se llena hay dos opciones: o se vuelve con el camión para vaciarlo y se retoma el recorrido o se dejan algunas áreas sin completar para que lo haga el camión de otra zona que tenga espacio. Les parecía bien este sistema y no querían cambiarlo.

Pedimos si nos podían entregar los recorridos actuales, pero esto no fue posible ya que no tienen registro de los mismos. Cada chofer de una zona conoce el camino a utilizar ya que lo recorren hace años.

Todo esto nos deja con algunos datos concretos pero muchas áreas que son ambiguas para poder ser modeladas. La sensación es que hay muchas decisiones que tienen que tomar los choferes, lo que tampoco ayuda a poder tener recorridos fijos. El terreno en función del clima o el turismo, hace que se modifiquen constantemente los recorridos, los choferes dejen calles sin recorrer y tengan que cambiar su recorrido. Nos hicieron mucho hincapié en lo "artesanal" de muchas decisiones que se toman, y que es invaluable tener experiencia haciendo la recolección.

3. Modelado del problema

3.1. Estrategia de resolución

3.1.1. Consideraciones generales

Teniendo la descripción del problema a resolver nos interesaba separar los datos que sabemos no van a ser modificados por cuestiones externas para que sean parte de nuestro modelo. Otros, afectados por factores como el clima o el turismo, son muy difíciles de modelar, principalmente porque no tenemos datos para poder agregarlos.

Entonces, los datos firmes que tenemos son:

- Hay zonas fijas.
- Cada casa saca la basura a su propio tacho. No hay contenedores en ningún lado. Esto nos obliga a pasar por todas las cuadras de la ciudad.
- Por seguridad hay que minimizar el uso de rutas y avenidas.
- Algunas calles el camión las puede hacer solo en un sentido (ej. Pendientes pronunciadas)
- El camión no suele entrar a las calles sin salida. Decidimos excluirlas a todas. Que el personal de recolección decida si entra o no, no va a afectar al recorrido óptimo.
- Con excepción de las calles principales, se juntan los residuos de ambos lados de la vereda pasando una sola vez por la cuadra. En rutas y avenidas es obligatorio circular en ambos sentidos.

El hecho de tener que pasar por todas las cuadras nos indica que para obtener un camino óptimo podríamos utilizar alguna variante del **problema del cartero chino**, a diferencia de un enfoque del estilo del viajante de comercio. No nos importa recorrer puntos en particular, algo que sería importante en el caso de que Bariloche tuviese contenedores, sino que hay que pasar por la puerta de todas las casas.

Para resolver el problema necesitamos convertir un mapa en un grafo dirigido. De esta manera se puede representar toda la información del mapa junto con la dirección de las calles. Vamos a necesitar agregar información a este grafo, poniendo datos sobre los nodos o aristas para que el solver tenga la información que necesite y poder obtener los recorridos.

3.1.2. Mapas

Necesitábamos obtener mapas que fueran lo más completos y actualizados posible y que nos permitan llevarlos a un formato apto para ser utilizado computacionalmente. La municipalidad no tenía mapas digitales para entregarnos por lo que se analizaron las opciones disponibles. Google Maps[9] era la primera opción, tiene buenos mapas pero no permite exportarlos a un formato abierto, por eso se decidió utilizar **Open Street Maps** (**OSM**), que también tenía mapas muy completos de la zona.

OpenStreetMap[11] es un proyecto colaborativo para crear mapas libres y editables. Los mapas son creados por colaboradores utilizando información geográfica capturada con dispositivos GPS móviles, orto-fotografías y otras fuentes libres. Esta cartografía, tanto las imágenes creadas como los datos vectoriales almacenados en su base de datos, se distribuyen bajo Licencia Abierta de Bases de Datos[14].

OSM tenía buenos mapas de Bariloche, brindaba una interfaz completa y fácil de utilizar tanto en una versión web como una de escritorio y también permitía exportar los mapas en formato XML. Todo esto lo hacía perfecto para nuestros requerimientos.

3.2. Estructuras de datos

Para poder trabajar necesitábamos llevar los mapas a grafos. El primer paso fue lograr convertir el XML de la exportación de OSM a un grafo. La estructura del XML utilizada es la siguiente, y se encuentra detallada en su sitio[12]:

- Un bloque con *nodos*. Cada nodo tiene sus coordenadas y tags con información adicional.
- Un bloque con *caminos*. Cada camino tiene una secuencia de nodos en cada dirección que sea necesaria. También están todos los tags para anotaciones sobre los mismos.
- Un bloque con *relaciones*. Éstas especifican relaciones entre objetos de OSM. No son de interés para nuestro trabajo.

Para poder trabajar con el solver, era necesario pasarle toda la información geográfica, de tránsito y de zonas, junto con la información recolectada en las reuniones con el personal de la municipalidad. Era necesario tener estructuras que representen toda esta información.

Como estructura principal con la cual trabajar se buscaba un grafo dirigido con anotaciones en el que cada esquina sea un nodo y la calle que une dos esquinas

una arista. Esta arista debe respetar la dirección de la calle, en el caso de ser doble mano se agregará una más para lograr tener un arista para cada dirección. Con este primer paso, lograríamos tener un esquema que represente toda la información sobre las calles, sus direcciones y que nos permita trabajar programáticamente. En las siguientes figuras vemos un ejemplo simple de cómo se adaptará un mapa a un grafo.

Figura 4: Mapa de una manzana

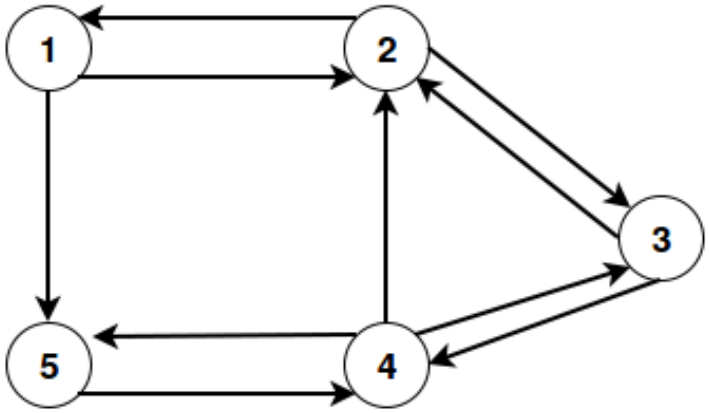


Figura 5: El grafo generado de la manzana de la figura anterior.

En el mapa vemos dos manzanas. Vemos que *Ojos del Salado* y *Copahue* son doble mano y las otras no.

El siguiente paso era agregar toda la información necesaria sobre cada nodo o arista. Por ej, gracias a la información geográfica que nos brinda OSM tenemos las coordenadas de latitud y longitud de cada punto en el mapa, con esto pudimos calcular la distancia de cada arista.

En cada arista sería necesario guardar la siguiente información:

- Distancia en metros.
- Si es parte de la zona a resolver.
- Si es doble mano. [2]
- Si es obligatorio hacer ambas manos de esta cuadra, en caso que sea de ambos sentidos y no alcance recorrerlas en una sola dirección.

Siguiendo con el ejemplo anterior y suponiendo que la zona a resolver se encuentra a la izquierda de *Tupungato* marcaríamos cada arista como dentro de la zona o no. Debido a que ninguna calle en el ejemplo es una avenida o calle principal alcanza con hacer las que son doble mano en un solo sentido. También podemos calcular la distancia entre las esquinas con las coordenadas que nos brinda OSM. Obtendríamos una tabla (Cuadro 1) de la siguiente manera:

Arista	Distancia	*Arista* $\in$ *zona*	Doble mano	Oblig. ambos sentidos
$1 \to 2$	130	Si	Si	No
$2 \to 1$	130	Si	Si	No
$2 \to 3$	80	No	Si	No
$3 \to 2$	80	No	Si	No
$3 \to 4$	85	No	Si	No
$4 \to 3$	85	No	Si	No
$4 \to 5$	100	Si	Si	No
$5 \to 4$	100	Si	Si	No
$1 \to 5$	140	Si	No	No
$4 \to 2$	80	Si	No	No

Cuadro 1: Grafo para mapa

3.3. Problema del cartero chino

Como vimos en la sección 2.3.1 en su definición más básica el problema del cartero chino nos sirve para poder determinar dado un grafo ponderado, el camino más corto que visite a todas las aristas del mismo. Para modelar nuestro problema necesitamos poder introducir las siguientes variantes:

- Las aristas tienen un peso.
- Las aristas son dirigidas.

[2] Dado que tenemos el conjunto de aristas, este dato es redundante. De todos modos por cuestiones de simplificar se precalcula.

- Hay aristas que no es obligatorio utilizar.

El hecho de tener un grafo dirigido pero tal que algunas calles doble mano alcanza con recorrerlas en un sólo sentido, indica que es un CPP mixto. Que tengamos aristas que no es obligatorio utilizar nos indica que es un Rural Postman Problem.

Entonces para nuestra situación tendríamos que *encontrar el camino con la distancia total más corta que visite todas las aristas obligatorias del grafo, respetando la dirección de las aristas y pudiendo utilizar o no las aristas que no son obligatorias.*

Debido a que partimos de un **RPP**, ya sabemos que vamos a tener una complejidad *NP-Hard*[5].

3.4. Modelo de programación lineal entera

El modelo más simple para el cartero chino es el siguiente: siendo N el conjunto de nodos del grafo, A el conjunto de aristas, c_{ij} el costo de recorrer la cuadra (arista) que une los nodos i y j y x_{ij} la cantidad de veces que se pasa por la misma.

Lo que se quiere minimizar es

$$\sum_{i,j \in A} c_{ij} x_{ij} \qquad x_{ij} \in Z^{+} \tag{1}$$

Sujeto a una restricción que garantice el flujo, para garantizar que la solución sea un ciclo.:

$$\forall i \in N(\sum_{j \in N(i,j) \in E} x_{ij} = \sum_{j \in N(j,i) \in E} x_{ji}) \tag{2}$$

Dado que cada cuadra es utilizada por lo menos una vez por la restricción sobre x, hasta este punto garantizamos tener un conjunto de aristas que tenga distancia mínima tal que se pueda armar un recorrido.

Para nuestra solución necesitábamos poder agregar restricciones para el resto de los parámetros a tener en cuenta. Hay aristas que si bien no son obligatorias, quizás nos acortan la distancia total a recorrer si son utilizadas. También hay que permitir, a menos que se indique lo contrario, hacer las calles doble mano en un solo sentido.

Siendo Z las aristas que están dentro de la zona a recorrer, IyV el conjunto de aristas que, perteneciendo a una calle doble mano, es obligatorio hacer ida y vuelta. UNA_MANO es un conjunto con todas las aristas que pertenecen a calles de una sola dirección.

$RUTA\_FIJA$ contiene todas las aristas que es obligatorio tomar. Su propósito se explicará en la sección 3.5.

$$\forall (i,j) \in A$$

$$\begin{aligned}
&\text{if } (i,j) \in RUTA_FIJA \rightarrow x_{i,j} \geq 1 && \text{(3a)}\\
&\text{else if } (i,j) \notin Z \rightarrow x_{i,j} \geq 0 && \text{(3b)}\\
&\text{else if } (i,j) \in IyV \rightarrow x_{i,j} \geq 1 && \text{(3c)}\\
&\text{else if } (i,j) \in UNA_MANO \rightarrow x_{i,j} \geq 1 && \text{(3d)}\\
&\text{else } x_{i,j} \geq 0 && \text{(3e)}\\
& && \text{(3f)}
\end{aligned}$$

Si una arista pertece a una ruta fija (3a), es una calle de una sola mano (3d) o una cuadra donde es obligatorio circular en ambos sentidos (3c), forzamos a que se utilice. Si en cambio vemos que no pertece a la zona a resolver (3b) o no hay ninguna restricción en particular, podemos no utilizarla.

Agregamos una restricción más, para que las calles doble mano se hagan en un solo sentido, a menos que sea obligatorio ir en ambos sentidos:

En (4a), si el arista es doble mano y no es obligatorio hacer ida y vuelta, entonces alcanza con ir en uno de los sentidos para cumplir la desigualdad.

En (4b), obligatorio ir en ambos sentidos, utilizar esa arista.

$$\forall (i,j)/(i,j) \in Z$$

$$\begin{aligned}
&\text{if } (j,i) \in Z \wedge (i,j) \notin IyV \rightarrow x_{ij} + x_{ji} \geq 1 && \text{(4a)}\\
&\text{else if } (i,j) \in IyV \rightarrow x_{ij} \geq 1 && \text{(4b)}
\end{aligned}$$

Hay un par de consideraciones a tener en cuenta en este momento. Muchas soluciones a este tipo de problemas, donde se intenta conseguir recorridos para vehículos de carga, toman dos factores en se decidió no tener en cuenta en el presente trabajo: doblar a la izquierda y doblar en U.

Para atacar ambos problemas lo que se suele hacer es quitar el nodo de cada esquina y reemplazarlo por una combinación de nodos y aristas que representen cada una de la opciones de movimiento en ese momento. De esta forma se modela la posibilidad de doblar para cada dirección posible, o seguir derecho. Con pesos sobre las nuevas aristas se puede hacer que sea más barato seguir en alguna dirección en particular. Esto agrega complejidad al modelado y al trabajo con el solver, es muy necesario para ciudades mas densas y con más tránsito donde por ejemplo no es

posible (o muy caro por cuestiones de tiempos de semáforos y tránsito) doblar a la izquierda.

Sobre la problemática para doblar a la izquierda, fue algo que se le preguntó a la gente encargada de los recorridos en la primer reunión. Nos informaron que eso no era un problema para Bariloche, con la excepción de las rutas y muy pocas avenidas. Pero las rutas son una zona en si mismas, por lo que las sacamos por completo del problema. La mayor área de la ciudad esta compuesta por zonas con poco tránsito por lo que no tiene ningún problema el camión para doblar en cualquier dirección. Reafirma esto el hecho de que nos indiquen que se levantan residuos de ambas manos de las calles sólo circulando en una sola dirección. Como se verá en la próxima sección, la forma en que se armaron los recorridos intenta minimizar la cantidad de veces que se dobla, y en casos en que es necesario no doblar, se marca ese tramo como una ruta fija y se lo recorre en linea recta.

Una vez se tomó la decisión de no poner restricciones al doblado, notamos que el modelo se simplificaba mucho, tanto en especificación como en tiempos de corrida, que se mantenían en los pocos segundos. Por este motivo se decidió analizar que tanto afectaba el doblar en U. Lo mismo que con doblar, en la mayoría de las zonas el camión puede doblar en U, o lo resuelven haciendo a pié la cuadra que los lleva a esa situación. Las incidencias de doblar en U en las soluciones eran muy bajas, entre 0 y 3 por zona. Se habló esto con el personal de recolección para que tomen la decisión ellos sobre como resolverlo, si dado una vuelta la manzana, o si es posible, doblando efectivamente en U o haciendo la cuadra a pié para simplificar. El personal estaba de acuerdo con esta decisión y prefirió soluciones lo mas simple posibles.

3.5. Armado de los recorridos

Luego de las reuniones con el personal de recolección teníamos el dato de que los choferes saben los recorridos de memoria, no usan ningún tipo de anotación ni tienen la tecnología (GPS, notepad, etc) para poder ver los recorridos. Por esto mismo se consideró que era importante que los recorridos entregados sean lo más fáciles de entender y memorizar.

El solver nos entrega la solución al problema indicando cuántas veces se pasa por cada arista respetando la restricción sobre el flujo. Esto no indica cómo armar ese camino, sino solamente qué arcos utilizar. La primera aproximación para armar un camino fue tomar un arco que contenga el nodo inicial y a partir de ahí ir tomando aristas adyacentes aleatoriamente hasta llegar nuevamente al nodo inicial. Por la restricción de flujo esto está garantizado que va a ocurrir.

La siguiente función arma un ciclo que pasa por (no necesariamente todas) las aristas que nos entrega el solver para tener camino óptimo. Toma como parámetros un multiconjunto que contiene todas las aristas que el solver selecciona, donde cada arista pertenece al conjunto tantas veces como tenga que ser utilizada.

Listing 1: Armado de caminos. Primera implementación

```
1   input: arcos: multiconjunto aristas
2          inicial : nodo para comenzar el ciclo
3   output: una lista de nodos, que representa un ciclo
4
5  arco ← arco \ arco_i sea igual al nodo inicial
6  ciclo ← lista [arco_i, arco_j]
7  quitar_arco(arcos, arco)
8
9  while ciclo_primero != ciclo_ultimo
10     arco ← arco / arco_i sea igual a ciclo_ultimo
11     ciclo ← agregar nodo arco_j
12     quitar_arco(arcos, arco)
```

Luego de llamar a la función anterior, tenemos un *ciclo*. Lo más probable es que éste no contenga todos los arcos a utilizar. En ese caso vamos a llamar a esa función hasta consumir todos los arcos, pasando como parámetro de entrada el conjunto de arcos al que le quitamos cada arista que fuimos utilizando.

Cada ciclo que obtenemos lo unimos con el resultado anterior simplemente buscando un nodo que pertenezca a ambos, sea ese nodo n_r. Se quita este nodo de el recorrido acumulado y se *inserta* el nuevo ciclo en ese lugar. Por ser un ciclo sabemos que va a comenzar y finalizar en el mismo nodo, por lo que no modifica la conectividad de recorrido acumulado.

$$recorrido = a_1, a_2, ...n_r..., a_1 \tag{5a}$$
$$nuevo\_ciclo = b_1, b_2, ..., b_{r-1}, n_r, b_{r+1}..., b_1 \tag{5b}$$
$$recorrido' = a_1, a_2, ...n_r, (n_{r+1}, ..., b_1, b_2, ...n_{r-1}, n_r), ..., a_1 \tag{5c}$$

De esta forma obtenemos un recorrido completo para toda la zona. Supongamos que tenemos un mapa que genera el grafo de la figura 6. Con el método anterior se genera un recorrido como el de la figura 7. Vemos que por más que ese recorrido cumple con el objetivo buscado, tiene el problema de ser muy aleatorio en como recorre las cuadras.

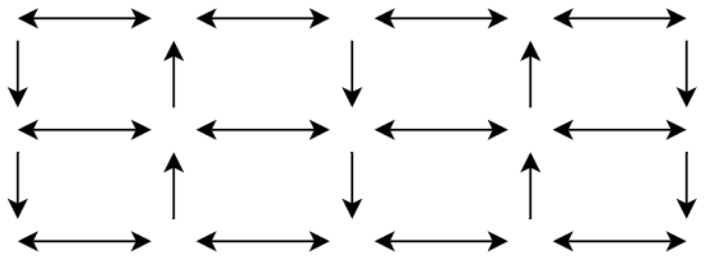


Figura 6: Grafo del recorrido de ejemplo

Para intentar mejorar este problema de los recorridos se intentó crear algunas heurísticas que hagan los caminos más “fáciles” de seguir.

Figura 7: Recorrido armado aleatoriamente.

Notamos que hay dos problemas principales. La falta de orden en el recorrido y el doblar constantemente. El orden para los recorridos sería que idealmente se vaya completando cada sector que se comienza. Para esto se decidió partir las zonas en sub-zonas de menor tamaño. Encontramos otro beneficio de esto, permite al personal de recolección tomar sub-zonas como referencia para mejorar alguno de sus recorridos en casos de no utilizar el recorrido completo.

Para hacer esas particiones, se seleccionaron zonas que no perjudiquen el resultado final. Por ej, marcar barrios que estén de cada lado de una ruta, que están claramente autocontenidos y se los resolvía completamente como si fuesen una zona en si mismos. De esta forma nos aseguramos que una vez se comienza ese barrio se lo finaliza. Haciendo una comparación con la distancia total recorrida de esta forma contra hacer toda la zona junta, nunca se incrementa el total en más de unas pocas cuadras.

El tratar de no doblar consiste en evitar ir avanzando aleatoriamente haciendo un constante “zig-zag” o volviendo hacia atrás y retomando sin ningún criterio. Idealmente queremos que la solución que tengamos sea lo más parecida a un recorrido que haría un humano. Por ej, en una red de calles cuadriculada (figura 6), hacer primero las verticales y después las horizontales. Notamos que hacer esto automáticamente es de una complejidad alta, mas aún en Bariloche que no toda la ciudad es cuadriculada e incluso en las zonas céntricas hay curvas o subidas pronunciadas que dificultan aún más la tarea.

De todas formas se intentaron implementar algunas heurísticas para mejorar los recorridos. La primera era: una vez que estemos circulando por una calle, si podemos seguir por la misma en vez de doblar, hacerlo. Para lograr esto podemos utilizar el nombre de la calle al que pertenece cada arista. Esa información la tenemos

disponible gracias a las estructura de datos con las que se trabajó.

Un problema a resolver fue que, dado que OSM contiene información cargada por voluntarios, los nombres de las calles no suelen estar escritos de una forma consistente. A medida que se fue mapeando, se fueron completando los mismos y nos encontramos en la mayoría de los casos con inconsistencias. Por ejemplo, el nombre de la calle es San Martín por algunas cuadras, después S. Martín, luego San Martin (sin acento) y luego S. Martín nuevamente.

Para esto se agregaron algunos pasos al armado de los recorridos. En vez de ir seleccionando al azar un arco que este conectado al camino actual, generamos una lista de candidatos ordenados por que tan "parecido" es el nombre al nombre de la calle a la que pertenece la última arista utilizada.

Listing 2: Armado de caminos, con orden de candidatos

```
1   input: arcos: multiconjunto aristas
2          inicial : arista para comenzar el ciclo
3   output: una lista de nodos, que representa un ciclo
4
5  arco ← arco \ arco_i sea igual al nodo inicial
6  ciclo ← lista [arco_i, arco_j]
7  quitar_arco(arcos, arco)
8
9  while ciclo_primero != ciclo_ultimo
10     nombre_calle = nombre calle del arco (ciclo_ultimo, ciclo_anteultimo)
11     candidatos ← lista todos los arcos / arco_i sea igual a ciclo_ultimo
12     ordenar candidatos en función de distancia levenshtein contra nombre_calle
13     arco ← candidatos_0
14     ciclo ← agregar nodo arco_j
15     quitar_arco(arcos, arco)
```

Figura 8: Recorrido armado con heurística de nombre de calles.

Con esto logramos una mejora notable. En el ejemplo que estamos utilizando se logra una solución muy parecida a lo que una persona probablemente haría. Notar que la distancia es exactamente la misma, se usó el mismo set de datos que nos

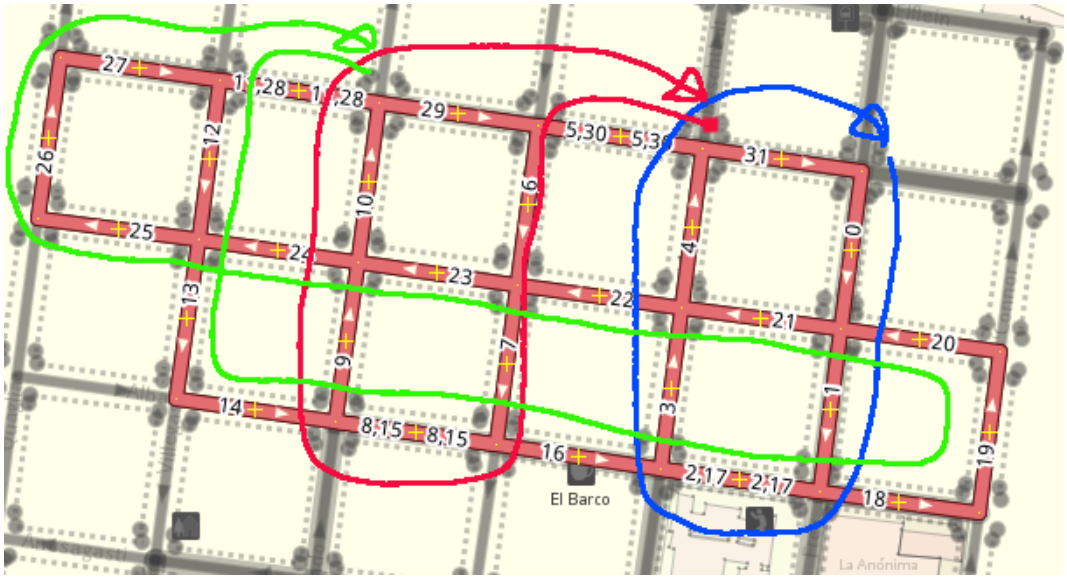


Figura 9: Recorrido armado con heurística de nombre de calles, ciclos resaltados.

entrega el solver, se observa que cada cuadra se hace una vez, con excepción de las mismas cuatro que se repiten en ambos mapas. Pero ahora el recorrido es mas "humanoz menos aleatorio.

En los casos del mundo real, aparecen algunas complicaciones. Un problema ahora sí viene del solver y se encuentra en las calles doble mano en las que no nos interesa la dirección en la que se la recorren. El solver no necesariamente va a intentar recorrer la calle en una dirección en particular. Puede que en la misma calle tome algunas cuadras en una dirección y otras en la opuesta. Esto hace que los trazados tengan un componente de aleatoriedad que los hace difícil de seguir y memorizar.

Para resolver esto fue necesario forzar al solver a tomar ciertas aristas en calles doble mano.

Notamos que este tema empezaba a extenderse mucho, hay muchas mejoras que se pueden a hacer, pero ya escapaban el alcance de este trabajo. Se decidió tomar algunas opciones razonables de implementar dado el cronograma de entregas de los recorridos.

Una posibilidad que no era muy compleja para implementar en este momento, era poder marcar caminos sobre el mapa y forzar que se utilicen. Por ej, si en una zona hay dos calles principales hacer que se hagan de una sola vez, en un solo sentido. Esto ayuda no sólo a que el camino sea más "simple" de memorizar y entender, sino que es más simple para el camión saber que en esa calle no tiene que doblar una vez que la toma, ya que la hace de punta a punta (dentro de la zona).

Para esto se hicieron algunos cambios a todas las etapas. Primero era necesario marcar los caminos sobre el mapa. Esto nos lo permite la herramienta JOSM, sim-

plemente agregamos tags sobre los nodos para indicar el camino, poniendo un orden sobre algunos nodos. En la etapa previa a llamar al solver juntamos los caminos fijos y pasamos esa nueva estructura al solver. Esto es la $RUTA\_FIJA$ que se vio en la sección anterior, y se fuerza todas esas aristas sean parte de la solución.

Otro procedimiento que permitió mejorar mucho las soluciones fue generar los recorridos varias veces, tomando aleatoriamente el orden de las aristas que nos entrega el solver. De estar forma se obtienen diferentes resultados, obviamente con la misma distancia total. Pero con esto podemos descartar algunos caminos que quedan más "desprolijos" desde un punto de vista de facilidad de comprensión del mismo. Este punto se podría mejorar mucho con más input de parte de los operarios.

4. Implementación

Para al resolución del problema se utilizaron las siguientes tecnologías:

- SCIP (Solving Constraint Integer Programs) es el solver utilizado para correr los modelos de programación lineal.
- ZIMPL es el lenguaje para traducir el modelo matemático de un problema lineal a un formato que SCIP puede procesar.
- python es el lenguaje de programación que se utiliza para procesar y generar toda la información que utilizamos, desde la traducción del XML a grafo, generación de las tablas, código de ZIMPL y finalmente para generar los mapas de los recorridos con la solución que obtenemos del solver.
- bash como lenguaje de scripting para correr los programas auxiliares sobre linux.
- Makefiles para controlar la ejecución de todo el proceso para generar los mapas.
- JOSM y librerías para manipular mapas de OSM, todas herramientas de Open Street Maps para manipular mapas, exportarlos y poder trabajar con ellos desde python.

Todo se desarrolló y ejecutó sobre Linux, en un procesador i5-3210M @ 2.50GHz, con 8Gb de RAM.

Para la implementación era importante cuidar los tiempos de corrida del solver, ya que es el único lugar donde hay complejidad computacional. En todas las otras etapas se priorizó la simpleza del código y del uso de las herramientas al tiempo de corrida ya que esos pasos son necesarios una sóla vez para generar los datos, por ej, generar un grafo con todos los datos de los mapas y nunca requiren de tiempos mayores a uno o dos segundos en total.

4.1. Mapas

Obtención de los mapas: Para obtener los mapas se utilizó la herramienta web de OSM[3]. Con ésta se seleccionó un área un poco mayor a la zona a resolver y se la exportó. Era necesario siempre un área mayor a la zona a resolver para poder utilizar calles fuera de la zona si era necesario para mejorar la solución.

Una vez realizada la exportación, tenemos un mapa en formato XML[4] que contiene entre otras cosas la información que vamos a utilizar para trabajar:

[3] http://www.openstreetmap.org
[4] http://www.w3.org/TR/REC-xml

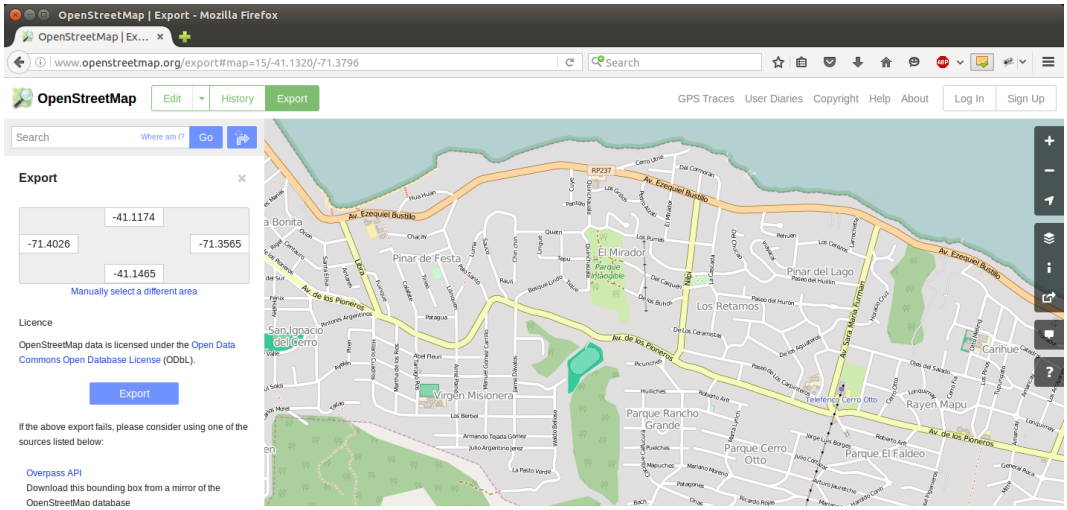


Figura 10: Exportando mapa desde la aplicación web de OSM.

- coordenadas de cada punto de una calle. Cada calle en OSM está compuesta de múltiples puntos, cuando es una esquina comparten las dos calles que se cruzan al mismo punto. No todos los puntos son intersecciones, también se utilizan para demarcar calles que no son rectas como una sucesión de rectas, como se en la figura 11.
- nombre de las calles con su numeración y todos los puntos que la componen.
- sentido de las calles.

Sobre el mapa que exportamos agregamos información manualmente (en sección 4.5 se ve un ejemplo), que es importante para poder modelar una solución. Para esto, se usó la herramienta de escritorio para JOSM[5]. Esto nos permitía agregar tags sobre cualquier elemento del mapa. Como primer paso se marcaba toda el área que componía a la zona en cuestión, agregando el tag 'zona' a todos lo elementos dentro de la misma. Luego se agregaron los datos necesarios para cumplir todos las etapas del desarrollo. Por ej. se marcaron la cuadras donde era necesario ir en ambos sentidos para hacer la recolección: solo en las avenidas principales, rutas y algunas calles que nos informaron. También se marcaron las calles por donde no hay que pasar, para excluirlas completamente de la solución, por ej. en caso de calles muy angostas donde el camión no entra o en algunas con pendientes muy pronunciadas.

4.2. Mapa a grafo

Para parsear ese XML utilizamos la librería osmparser[6] escrita en python. Luego llenamos algunas estructuras que se utilizaron en varias etapas del trabajo:

[5]https://josm.openstreetmap.de
[6]http://imposm.org/docs/imposm.parser/latest/

Las principales son estos diccionarios:

- zona: conjunto de todos los puntos sobre rutas dentro de la zona
- caminos[id calle]: {'$tags$': tags de OSM sobre la calle, '$refs$' : todos los puntos de la calle}
- coordenadas[id nodo]: (latitud, longitud)
- rutas_fijas[id ruta fija]: lista de nodos por donde pasa la ruta

Utilizando estos datos podemos armar una nueva estructura que nos diga para cada cuadra, el nombre de la calle a la que pertenece y la distancia de esquina a esquina. Ésta estructura junto el conjunto de coordenadas son las principales para poder aplicar CPP ya que tendremos tanto un conjunto de nodos como una tabla con todas las aristas, su dirección y la longitud (peso) de la misma.

La estructura que queremos armar es un diccionario con los siguientes datos:

aristas[($nodo_a$, $nodo_b$)] = {'way':id de la calle, '$distancia$':peso del arista)}

Para esto buscamos todas las esquinas del mapa en la función *armar_aristas* buscando todas las intersecciones entre calles. En ese caso estamos seguros de tener una esquina, entonces se agrega el arista desde ese lugar hasta la próxima esquina de la calle al conjunto de aristas. Ahí también se guarda el id de la calle a la que pertenece junto con su largo en metros y si tiene parámetros éstos también.

En la primera implementación cada nodo del mapa era un nodo de nuestro grafo. OSM modela las calles con puntos, en el caso de un trazado cuadriculado con calles rectas cada punto es una esquina por lo que casi todos los nodos del grafo que represente ese mapa serían una esquina. Pero pronto se notó que las curvas están representadas como una secuencia de rectas, por este motivo teníamos muchos más nodos que esquinas en nuestro mapa. En Bariloche, a causa de su geografía, casi ninguna calle es recta con excepción de algunas zonas céntricas. La representación sería la siguiente:

Vemos que para llegar de A a B, lo que podría ser dos nodos y un arista, son 8 nodos y 7 aristas. De C a D tenemos 9 nodos y 8 aristas. Esto hace que tengamos muchos nodos que no son esquinas agregando muchas aristas que no aportan nada a la solución pero que complejizaban el problema. Esto se hace a nivel del grafo que representa el mapa, por lo que no hay problema con las distancias, solo se quitan todas las aristas de la curva y se agrega una nueva con la distancia igual a la suma de todos los segmentos.

Todo lo mencionado anteriormente, se resuelve entre los métodos *generar_arista* y *agregar_arista*.

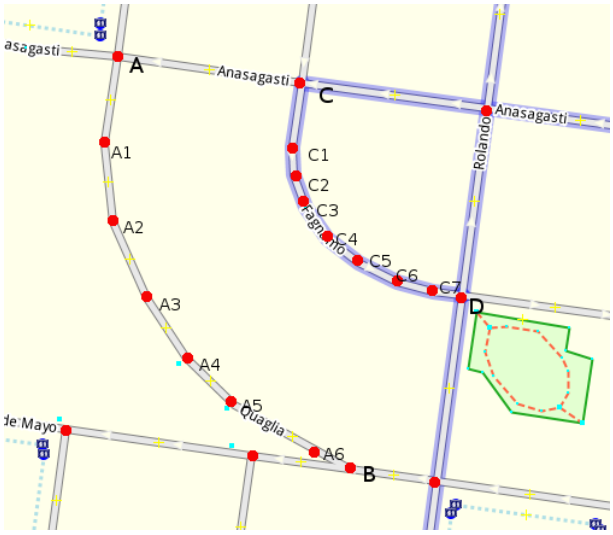


Figura 11: Zona céntrica de Bariloche con curvas

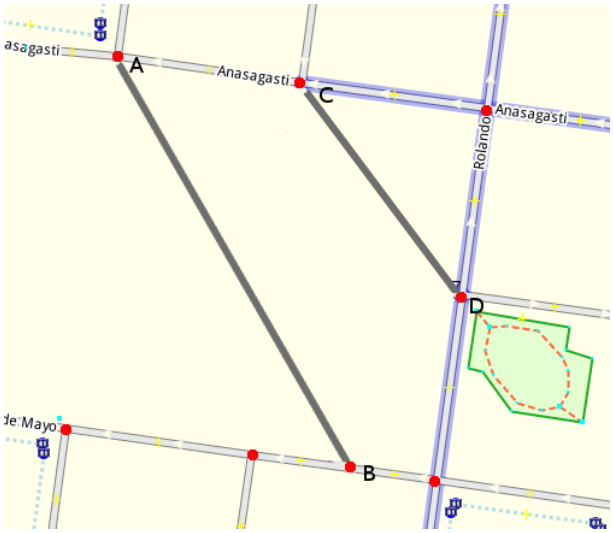


Figura 12: Zona céntrica de Bariloche con curvas simplificadas

```
def generar_aristas(self):
    for c1 in self.caminos:
        # camino es la lista de nodos de la calle c1
        camino = self.caminos[c1]['refs']

        actual = 0
        for i in range(1, len(camino)):
            for c2 in self.caminos:
                # si es el mismo no lo miramos
                if c1 == c2:
                    continue
                # si c2 y c1 se tocan es una esquina, cargo el
                # arista y la distancia para llegar
                # de actual a i sobre el camino c1
```

```
            if camino[i] in self.caminos[c2]['refs']:
                self.agregar_arista(c1, camino[actual:i+1])
                actual = i
        # ver si es la punta y lo agregamos
        if actual != i and i == (len(camino)-1):
            self.agregar_arista(c1, camino[actual:i+1])
```

El método *generar_aristas* recorre todas las calles del mapa y *agregar_arista* toma un segmento del camino entre dos esquinas y el id de la calle en la cual nos encontramos. Calcula la distancia total del camino entre las esquinas y descarta todos los nodos intermedios.

```
def agregar_arista(self, cid, segmento_camino):
    p1 = segmento_camino[0]
    p2 = segmento_camino[-1]

    # nos interesa guardar si es obligatorio recorrer
    # el arista de ida y vuelta. Revisamos sus tags
    ambos = self.caminos[cid]['tags'].get(
                'idayvuelta', 'no') == 'yes'

    # agrego el arista para ir de p1 -> p2
    self.aristas[(p1, p2)] = {
        'way': cid,
        'distancia': self.distancia(segmento_camino),
        'idayvuelta': ambos,
    }

    # si no es de una mano agrego p2 -> p1
    if self.caminos[cid]['tags'].get('oneway', 'no') == 'no':
        self.aristas[(p2, p1)] = {
            'way': cid,
            'distancia': self.distancia(segmento_camino),
            'idayvuelta': ambos,
        }
```

El último paso es armar el archivo que que consume SCIP para poder obtener el circuito optimizado. Volcamos toda la información del grafo y sus atributos en un txt donde cada línea contiene:

- id $nodo_1$ arista
- id $nodo_2$ arista
- distancia en mts (peso) del arista
- bool que indica si ida y vuelta son obligatorias

- bool que indica si es de una mano
- obligatorio pasar (pertenece a la zona)
- arista fija (forzar que este arco sea parte de la solución)

Generamos otro archivo que donde se escriben todos los nodos del grafo.

4.3. RPP

Con el paso anterior logramos dos archivos que utilizaremos para la resolución del **problema del cartero chino rural**.

En un archivo tenemos todos los nodos que representan a todas las esquinas del mapa. En otro una tabla con todos los parámetros explicados en la sección anterior. La implementación se intentó mantener de la forma más simple posible y ver si se lograban corridas en tiempos aceptables sin complejizar el uso del solver.

```
set V := { read LOS_NODOS as "<1n>" };
set P := { <i,v> in V*V };

param d[V*V] :=
    read TABLA as "<1n,2n> 3n" default 0;

param ida_y_vuelta[V*V] :=
    read TABLA as "<1n,2n> 4n" default 0;

param una_mano[V*V] :=
    read TABLA as "<1n,2n> 5n" default 0;

param obligatorio_pasar[V*V] :=
    read TABLA as "<1n,2n> 6n" default 0;

param ruta_fija[V*V] :=
    read TABLA as "<1n,2n> 7n" default 0;

set E := { <i,j> in V*V with d[i,j] > 0 };
```

En esta primera parte simplemente se levantan la lista de nodos y la tabla con todos los parámetros que utilizamos para modelar el problema. Se generan estructuras para cada atributo que vamos a utilizar.

La función a minimizar es la distancia total recorrida, dado que pasemos por todas las cuadras y que se cumplan con las restricciones del problema:

```
minimize fobj: sum <i,j> in E: (d[i,j]*x[i,j]);
```

En la implementación más simple del cartero chino tendríamos la siguiente restricción:

```
var x[<a,b> in E] integer >= 1
```

Vamos a darle un nuevo enfoque a esto usando parte de los parámetros que tenemos:

```
 var x[<a,b> in E] integer >=
# si previamente fijamos el arista hay que tomarla
if ruta_fija[a,b] == 1
    then 1
else
# si no es parte de la zona podemos no usarla
    if obligatorio_pasar[a,b] == 0
       then 0
    else
# tomamos el arista si es obligatorio ir en ambos sentidos.
        if ida_y_vuelta[a,b] == 1
            then 1
            else
# tomamos el arista si es calle de solo una mano
            if una_mano[a,b] == 1
                then 1
            else
# estamos en calle doble mano,
# podemos tanto tomar esta arista como la inversa.
                then 0
            end
        end
    end
end;



subto flujo:
    # respetar flujo
    forall <i> in V:
        (sum <j> in V with <j,i> in E: x[j,i]) ==
            (sum <j> in V with <i,j> in E: x[i,j]);

subto noIyV:
    # si pertece a la zona y es obligatorio ida y vuelta
    # hay que recorrer cada arista, sino con una alcanza.
    forall <i,v> in E:
        if obligatorio_pasar[i,v] == 1 then
            if d[v,i] > 0 then
                if ida_y_vuelta[i,v] == 0 then
```

```
                    x[i,v] + x[v,i] >= 1
                end
            else x[i,v] >= 1
        end
    end;
```

4.4. Entregables

La idea era intentar entregar los recorridos con una presentación que sea lo más fácil de utilizar. Queríamos que pudieran tener los recorridos en un camión y entender cómo llevarlo a cabo con la menor cantidad de distracciones posibles. En un principio se analizó algún formato digital, posiblemente con una animación para ver los recorridos, pero nada de eso iba a ser útil para los choferes. Los mismos no tienen ningún dispositivo móvil como una tablet, celular o computadora dentro del camión. Entonces lo mejor era armar los entregables de tal forma que se puedan imprimir. Se decidió entregar cada zona en formato de mapa junto con una tabla que ayuda a seguir el recorrido.

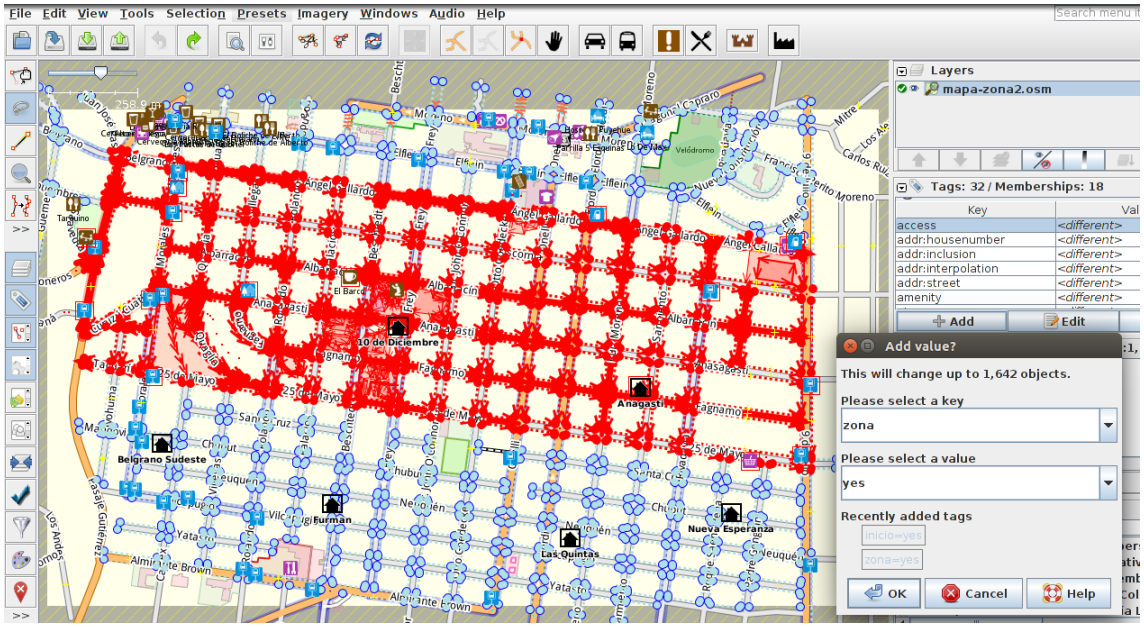
Dado que el recorrido de una zona tiene cientos de cuadras, marcarlas todas en el mismo mapa hacía que el mismo quede difícil de entender. Se tomó la decisión de no mostrar mas de 100 cuadras del recorrido al mismo tiempo, por lo que generaremos tantos mapas como sean necesarios, mostrando sólo 100 cuadras del recorrido en cada unos.

Dado que OSM no tenía información muy buena sobre las alturas de las calles usamos la ubicación de una cuadra como un número sobre el mapa. Encontramos este formato fácil de entender y de seguir. También tiene la ventaja de que estando en cualquier punto se puede entender tanto cómo seguir como cuál fue el camino para llegar hasta ese lugar. Esto cumplía con la necesidad de que se pueda utilizar un segmento de nuestro recorrido para ayudar en la optimización de alguna parte del recorrido en particular.

4.5. Resolución de una zona

Como ejemplo vamos a ver la Zona 2 de Bariloche. Ésta es una zona céntrica. El primer paso fue recortar el mapa en OSM (utilizando la herramienta JOSM) y marcar todo lo que era la zona con el tag para que el solver lo pueda utilizar. En este caso se pone zona=yes como tag a todos los elementos que nos interesan.

Para obtener una recorrido más prolijo fijamos algunos caminos, a fin de hacer un dibujo más parecido a lo que haría una persona, recorriendo primero horizontalmente y después las que cortan. Los puntos que marcamos son los que se pueden ver en la figura 13.



Una vez que teníamos esto listo, corrimos todos los pasos de nuestra aplicación para obtener los siguientes datos para entregar. Figuras 14 a 16.

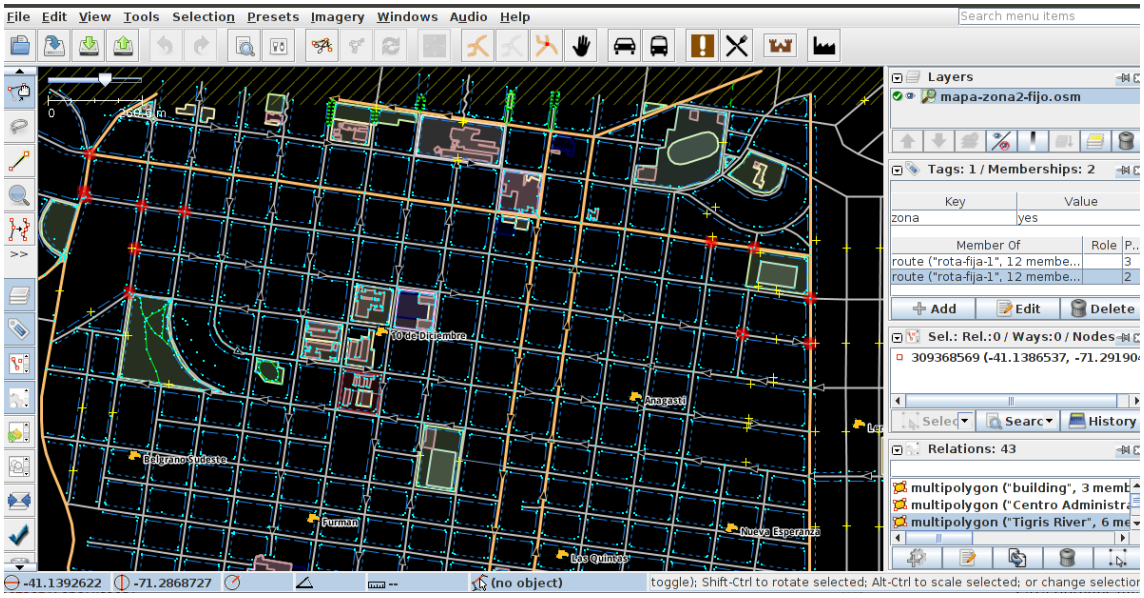


Figura 13: Puntos fijos para marcar camino obligatorio

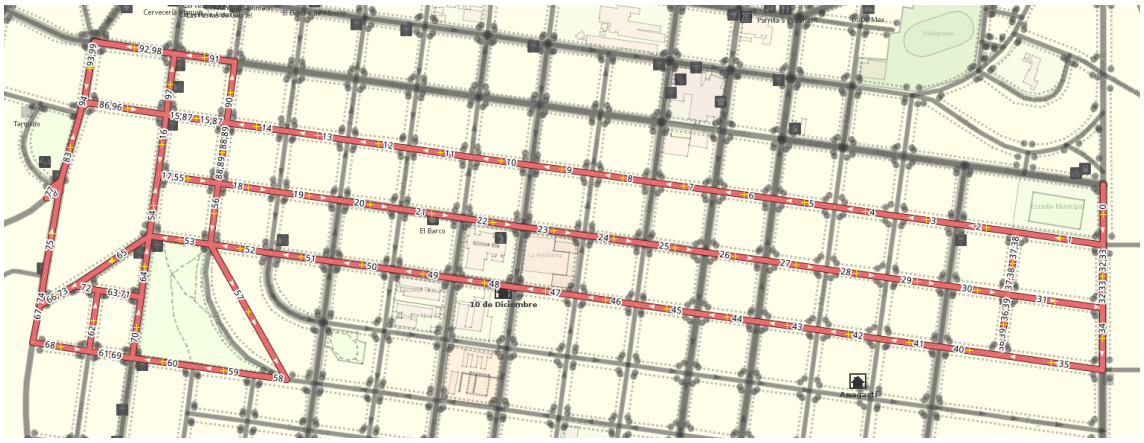


Figura 14: cuadras 0-99 del recorrido

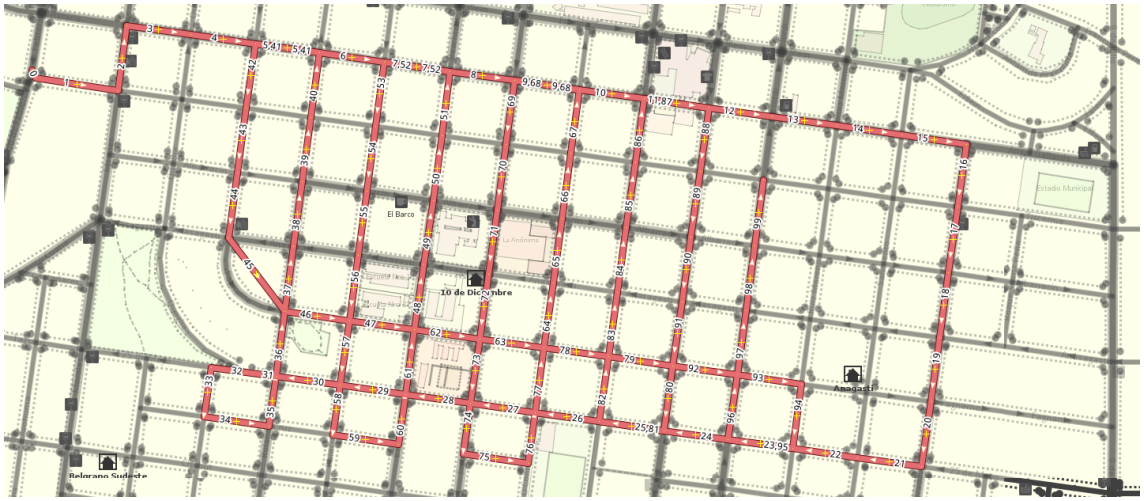


Figura 15: cuadras 100-199 del recorrido

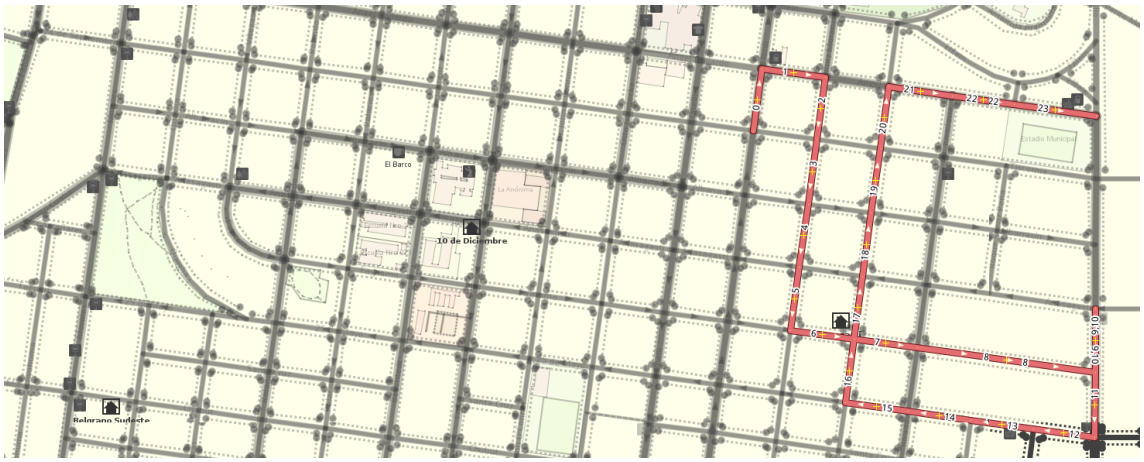


Figura 16: cuadras 200- hasta el fin del recorrido

Algunas comparaciones de tiempos entre hacer caminos fijando parte del recorrido contra no hacerlo: con los caminos fijos se tardó 2.36 segundos para una distancia total de 25.66 kilómetros. Sin los caminos fijos se resolvió en 1.27 segundos con la misma distancia total.

9 de Julio	0
Discernía	1 - 15
Morales	16
Albarracín	17 - 31
9 de Julio	32 - 34
Anasagasti	35
Roque Sáenz Peña	36 - 39
Anasagasti	40 - 53
Morales	54
Albarracín	55
Quaglia	56 - 57
25 de Mayo	58 - 60
Tacuarí	61
Ayohuma	62
Castañares	63
Morales	64
Curuzú Cuatiá	65 - 66
Pasaje Gutiérrez	67
Tacuarí	68 - 69
Morales	70
Castañares	71 - 72
Curuzú Cuatiá	73
Pasaje Gutiérrez	74 - 75
Saavedra	76
Av. de los Pioneros	77 - 78
20 de Febrero	79
Av. de los Pioneros	80
Saavedra	81
20 de Febrero	82 - 84
24 de Septiembre	85 - 86
Morales	87
Belgrano	88
20 de Febrero	89 - 90
24 de Septiembre	91 - 92
Tiscornia	93
Quaglia	94 - 96
Ángel Gallardo	97
Belgrano	98
20 de Febrero	99

Zona 2 parte 1

24 de Septiembre	0 - 1
Morales	2
Ángel Gallardo	3 - 15
Rivadavia	16 - 20
25 de Mayo	21 - 32
Villegas	33
Santa Cruz	34
Rolando	35 - 40
Ángel Gallardo	41
Villegas	42 - 44
Fagnamo	45 - 47
Beschtedt	48 - 51
Ángel Gallardo	52
Palacios	53 - 58
Santa Cruz	59
Beschtedt	60 - 61
Fagnamo	62 - 63
Dr. John O'Connor	64 - 67
Ángel Gallardo	68
Frey	69 - 74
Santa Cruz	75
Dr. John O'Connor	76 - 77
Fagnamo	78 - 79
Onelli	80
25 de Mayo	81
Otto Goedecke	82 - 86
Ángel Gallardo	87
Onelli	88 - 91
Fagnamo	92 - 93
Ruiz Moreno	94
25 de Mayo	95
Elordi	96 - 99

Zona 2 parte 2

Elordi	0
Ángel Gallardo	1
Ruiz Moreno	2 - 5
Fagnamo	6 - 8
9 de Julio	9 - 11
25 de Mayo	12 - 15
Sarmiento	16 - 20
Ángel Gallardo	21 - 23

Zona 2 parte 3

5. Resultados y conclusiones

Los recorridos entregados en este informe permiten hacer la totalidad de las cuadras, transitando la menor distancia posible. Se entregaron los resultados en un formato que no requiere de ningún dispositivo electrónico para ser entendidos y utilizados con facilidad.

Las zonas grandes están resueltas por subzonas y sugerimos un camino óptimo para cada subzona. Esto permite ir retocando partes más chicas de los recorridos actuales, en función de lo sugerido por nuestros modelos. Dado que se notó cierta reticencia a cambiar la forma de hacer los recorridos por el personal de la municipalidad, esto les permite tomar parte de nuestros resultados sin tener que modificar por completo los caminos actuales. Posiblemente de esta manera se pueda tener una transición menos brusca y que permita una adopción de los cambios.

Dada la falta de documentación de los recorridos actuales, no se contaba con ningún tipo de dato para poder comparar con los recorridos presentados. En Bariloche contábamos con la colaboración de Elias Neuman y Lautaro Lobos como contactos con la municipalidad. Después de meses de idas y vueltas lograron poder subir a camiones y acompañaron en el recorrido a los recolectores en cuatro de las zonas, relevando el recorrido realizado, velocidades promedio, y características particulares a tener en cuenta.

Resumimos la información comparada para esos cuatro recorridos. En los cálculos de distancia no están contemplada la ida desde el corralón a la zona ni desde la zona al vertedero, una vez finalizado el recorrido.

Zona	recorrido propuesto	recorrido actual	porcentaje sin recorrer	dist. sin recorrer
1	36.21 km	25.2 km	30 %	6.4 km
2	25.9 km	26 km	7 %	1.8 km
3	31.26 km	25.6 km	19 %	4 km
7	30.25 km	24.4 km	22 %	5.2 km

Los recorridos actuales relevados son cercanos a un camino óptimo *para el sub-*

conjunto de cuadras recorrido, pero dejan cuadras sin recorrer (en algún caso hasta un 30 %). Con esto notamos una heurística utilizada en los recorridos actuales para acortar los caminos: Intentar pasar una sóla vez por cada cuadra completando el resto a pie como método de no tener que agregar cuadras repetidas al recorrido. De todas formas podemos afirmar que no sólo no son caminos completos, sino que más allá de ser buenos para las cuadras que sí recorren, tampoco son óptimos para ese subconjunto. Se puede ver que el aporte computacional para resolver completitud y distancia mínima es muy importante.

En la actualidad muchos tramos deben ser recorridos a pie por los recolectores, con el esfuerzo físico extra y el incremento en la posibilidad de lesiones que ello conlleva, o dejando directamente cuadras donde no se recogen los residuos perjudicando a los vecinos de la ciudad. Hay que notar que no hay ninguna garantía de que pase alguien por esas cuadras, pudiendo varias de ellas quedar con los residuos sin recoger.

En comparación de distancias, nuestros caminos, que recorren la totalidad de las cuadras, son más largos que los actuales. Pero si agregamos a los recorridos actuales la distancia total de las cuadras no recorridas es donde vemos lo bueno de nuestra propuesta. Armar un camino que pueda hacer un camión cumpliendo las reglas de tránsito siempre va a ser más largo que el camino actual más la distancia sin recorrer ya que a muchas cuadras tendrían que recorrerlas más de una vez para que sea un camino factible.

En la etapa de adquisición de datos del proyecto nos habían remarcado dos temas importantes que querían resolver. Uno era no dejar cuadras olvidadas con residuos para no recibir quejas de la vecinos. La otra era que tenían problemas con las lesiones de los operarios que corren detrás del camión juntando bolsas. Nuestra solución resuelve ambos problemas ya que tienen una forma ordenada de pasar por todas las cuadras de la ciudad.

Una posible continuación de este trabajo sería analizar como armar recorridos para que sean aún más fáciles de seguir, respetando las distancias totales. Se trabajó sobre esto pero notamos que hay muchos puntos sobre los cuales se puede seguir trabajando.

Viendo los recorridos actuales, después de poder medirlos notamos que es una decisión que toman el hecho de no pasar con el camión por todas las cuadras. Entonces, asumiendo que uno va a hacer un número de cuadras a pie para acortar tiempos y distancia, se podría buscar caminos óptimos que no repitan ninguna cuadra permitiendo dejar algunas sin recorrer con el camión. Este tipo de soluciones se podría ir ajustando con alguna cota sobre la cantidad de distancia total que están dispuestos a caminar. De esta forma obtendríamos caminos significativamente más cortos en distancia y tiempo total de recorrido a costa de hacer un número de cuadras a pie.

Referencias

[1] Tobias Achterberg. Scip: solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, 2009.

[2] Tobias Achterberg, Timo Berthold, Thorsten Koch, and Kati Wolter. Constraint integer programming: a new approach to integrate cp and mip. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 6–20, 2008.

[3] Jack Edmonds and Ellis L. Johnson. Matching euler tours and the chinese postman. *Mathematical Programming*, 5:88–124, 1973.

[4] H. A. Eiselt, Michel Gendeaeu, and Gilbert Laporte. Arc routing problems, part 1: The chinese postman problem. *Operations Research*, page 231–242, 1995.

[5] H. A. Eiselt, Michel Gendeaeu, and Gilbert Laporte. Arc routing problems, part 2: The rural postman problem. *Operations Research*, pages 399–414, 1995.

[6] Meigu Guan. On the windy postman problem. *Discrete Applied Mathematics*, pages 41–46, 1984.

[7] Geir Hasle, Knut-Andreas Lie, and Ewald Quak. *Geometric Modelling, Numerical Simulation, and Optimization.* Applied Mathematics at SINTEF, 2007.

[8] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 3(28):497–520, 1960.

[9] Google Maps. `http://maps.google.com`.

[10] Kwan Mei-Ko. Graphic programming using odd or even points. *Chinese Math*, (1):273–277, 1962.

[11] OpenStreetMap. `http://www.openstreetmap.org`.

[12] OpenStreetMap. `http://wiki.openstreetmap.org/wiki/OSM_XML`.

[13] Amy Shoemaker and Sagar Vare. Edmonds' blossom algorithm. *Standford University, CME*, 323, 2016.

[14] Open Database License (ODbL) v1.0. `http://opendatacommons.org/licenses/odbl/1.0/`.