Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Algoritmos basados en programación lineal entera para la zonificación de la recolección de residuos

Tesis presentada para optar al título de
Licenciado en Ciencias de la Computación

Nicolás Saravia

Director: Javier Marenco

Buenos Aires, 2017

RESUMEN

La programación de los aspectos logísticos de la recolección de residuos urbanos involucra una serie de problemas de optimización combinatoria de difícil resolución en la práctica. Problemas típicos que aparecen en este contexto son la definición de la flota de camiones para realizar la recolección, la zonificación del área a recorrer (de modo tal que cada camión recorre una zona) y la optimización del recorrido de cada camión en función de consideraciones de tránsito y desgaste de los vehículos.
En esta tesis se propone el estudio del segundo de estos problemas. Consiste en obtener zonas de contornos sencillos, fácilmente recordables para los conductores de la flota de camiones, facilitando el proceso de recolección de residuos. Dicho problema es NP-hard, y en trabajos previos de la literatura ha demostrado ser muy difícil de resolver en la práctica. Por este motivo, se estudia en esta tesis este problema por medio de algoritmos heurísticos.

Palabras clave: PLE, Zonificación, Recolección, Residuos, Optimización, Combinatoria, Heurísticas, Partición, Área, Multigrafo, Contorno, Restricción, Polígonos, OpenStreetMap, SCIP.

AGRADECIMIENTOS

Le quiero dar las gracias a mi tutor, doctor en Ciencias de la Computación Javier Marenco. Gracias por su atención, paciencia, dedicación, motivación, criterio y aliento. Ha hecho fácil lo complicado. La verdad ha sido un privilegio poder contar con su guía y ayuda.

Gracias a todas las personas de la Facultad de Ciencias Exactas y Naturales de la UBA, docentes, ayudantes, compañeros alumnos, por su atención y amabilidad en todo lo referente a mi vida como estudiante de grado.

Gracias a los compañeros de trabajo, por todo su apoyo y motivación.

Y por encima de todo, y con todo mi amor, gracias a los míos por estar incondicionalmente conmigo durante todos estos años. Gracias por su comprensión, apoyo, contención y aliento siempre. Gracias papá Julio, mamá Adriana, hermanos Emilio y Julieta, a mi novia Naty y a mis amigos Zizou y Tomás. Gracias por todo. Los quiero con todo mi corazón.

Índice general

1.. Introducción 1
1.1. Descripción del problema 1
1.2. Trabajos previos 2
1.3. Definiciones 3
1.4. Teoría de complejidad computacional 3
1.5. Programación lineal entera 4
1.6. Coordenadas geográficas 6
1.7. Open Street Map 6
1.8. Contenido de la tesis 7

2.. Capítulo II: Problemática y enfoque de resolución 9
2.1. Requerimientos 9
2.2. Algoritmo propuesto 9
2.3. Representación del mapa de la ciudad 10
2.4. Representación computacional 10
2.5. Algoritmos 11
2.5.1. Parsing: del osm al grafo 11
2.5.2. Filtro de nodos 14
2.5.3. Subdivisión del mapa 15
2.5.4. Generación de polígonos 17
2.5.5. Selección de nodos iniciales 17
2.5.6. PathNodes 17
2.5.7. Búsqueda de vecindad 18
2.5.8. Análisis geométrico en selección de vecinos 20
2.5.9. Restricciones en selección de nodos iniciales 24
2.5.10. Algoritmo generador de polígonos 25
2.5.11. Ubicación del polígono 26
2.5.12. Armado de polígonos 27
2.5.13. Planteo y resolución del modelo 28
2.5.14. Función objetivo 30
2.5.15. Resolución del modelo 31
2.5.16. Extracción de polígonos 31
2.5.17. Algoritmo greedy de inserción de polígonos a la solución 32
2.5.18. Problema encontrado 34
2.5.19. Posprocesamiento de la solución 35
2.5.20. Visualización de polígonos 38

3.. Capítulo III: Experimentos y resultados 41
3.1. Características de las instancias 41
3.2. Heurísticas utilizadas en la generación de polígonos 46
3.3. Armado del modelo de Programación Lineal Entera 47
3.4. Solapamiento de polígonos en la solución 48
3.5. Heurística de merge de polígonos en la solución 49

3.6. Experimentos con heurísticas definidas 50
3.7. Resultados finales 53

4.. Capítulo IV: Conclusiones 57
4.1. Conclusiones 57
4.2. Trabajo a futuro 58

5.. Apéndice 59
5.1. Implementación 59
5.1.1. Configuración 61
5.1.2. Scip-Interfaces/JSCIPOpt 62
5.1.3. Estructuras de datos 65
5.1.4. Hilo de ejecución 66

1. INTRODUCCIÓN

1.1. Descripción del problema

La programación de los aspectos logísticos de la recolección de residuos urbanos involucra una serie de problemas de optimización combinatoria de difícil resolución en la práctica. Problemas típicos que aparecen en este contexto son la definición de la flota de camiones para realizar la recolección, la zonificación del área a recorrer (de modo tal que cada camión recorre una zona) y la optimización del recorrido de cada camión en función de consideraciones de tránsito y degaste de los vehículos. En esta tesis se propone el estudio del segundo de estos problemas, analizando algoritmos heurísticos para la partición en zonas del área a recolectar.
Dados:

1. el mapa de la ciudad en la que se desea recolectar residuos, representado por un multigrafo $G = (V, E)$, con V el conjunto de vértices que representan las esquinas de la ciudad, y E el conjunto de cuadras (aristas dirigidas en el sentido de la calle o avenida),

2. una estimación de la cantidad de residuos promedio a recolectar en cada cuadra,

3. la capacidad máxima de los vehículos de recolección,

el problema consiste en particionar el grafo G en la menor cantidad de zonas conexas de modo tal que los residuos a recolectar en cada zona no excedan la capacidad máxima de los camiones, y principalmente, cada zona tenga un contorno sencillo. Este último requerimiento es relativamente elástico, y está motivado por el hecho de que una zona con un contorno sencillo puede ser fácilmente recordada por los conductores de los vehículos de recolección. En trabajos previos, esta última restricción ha demostrado ser de difícil tratamiento, dado que no parece directo plantear algoritmos que generen automáticamente zonas con contornos sencillos y que al mismo tiempo particionen el mapa de la ciudad. Por este motivo, se propone en esta tesis estudiar este problema por medio de diversos algoritmos heurísticos.
Para esto se utilizarán esquemas de generación de polígonos en conjunto con modelos de programación lineal entera para la selección de zonas que cubran el mapa de la ciudad.
Si se asigna un camión por zona, una restricción posible sobre el tamaño sería que cada zona permitiera al mismo recorrerla íntegra sin tener que realizar ningún viaje de descarga. A su vez el tamaño de la flota de camiones pondrá una cota superior a la cantidad de zonas posibles. En lo que respecta al tiempo que consume la recolección, el mismo estará íntimamente relacionado con la cantidad de cuadras y la cantidad de residuos a recoger; variando este último con la densidad y el nivel adquisitivo de la población, con la época del año, el día de la semana, el horario, la economía, la cercanía de eventos importantes, entre otros factores. Entonces el tamaño de la zona también estará acotado superiormente por la carga horaria.
En caso de trabajar todos los camiones en paralelo, un objetivo deseado es lograr que la recolección en las distintas zonas consuma aproximadamente el mismo tiempo. Adicionalmente, por regla general, estas áreas se suelen delimitar por grandes avenidas y tratando

de que adopten formas geométricas conocidas (esto es, rectangulares, triangulares, trapezoidales) a fin de que los conductores de los camiones puedan recordarlas fácilmente.
El objetivo de este trabajo es estudiar un caso particular de región e implementar un programa basado en heurísticas de programación lineal entera mixta que sea capaz de, dado un mapa y una ubicación tentativa de las distintas áreas, generar un zonificado que cumpla con los requisitos antes mencionados en un tiempo razonable.

1.2. Trabajos previos

Se tomó como referencia de trabajos previos la Tesis de Licenciatura de Marcelo Bianchetti *Algoritmos de zonificación para recolección de residuos* (2015) [7]. En la misma se propone el estudio del problema relacionado a la zonificación del área a recorrer por los camiones recolectores, analizando algoritmos heurísticos para la partición en zonas del área a recolectar. El tamaño y distribución de las zonas va a depender de las características físicas y políticas de la región, las cuales se traducirán en limitaciones; y dentro de lo permitido habrá que buscar la solución que mejor cubra las necesidades.
Como puede verse en la Figura 1.1 el algoritmo propuesto en [7] da una solución al problema planteado, cumpliendo las restricciones impuestas, pero no se logra particionar el mapa en zonas con contornos sencillos, de fácil interpretación para los encargados de la recolección de residuos.

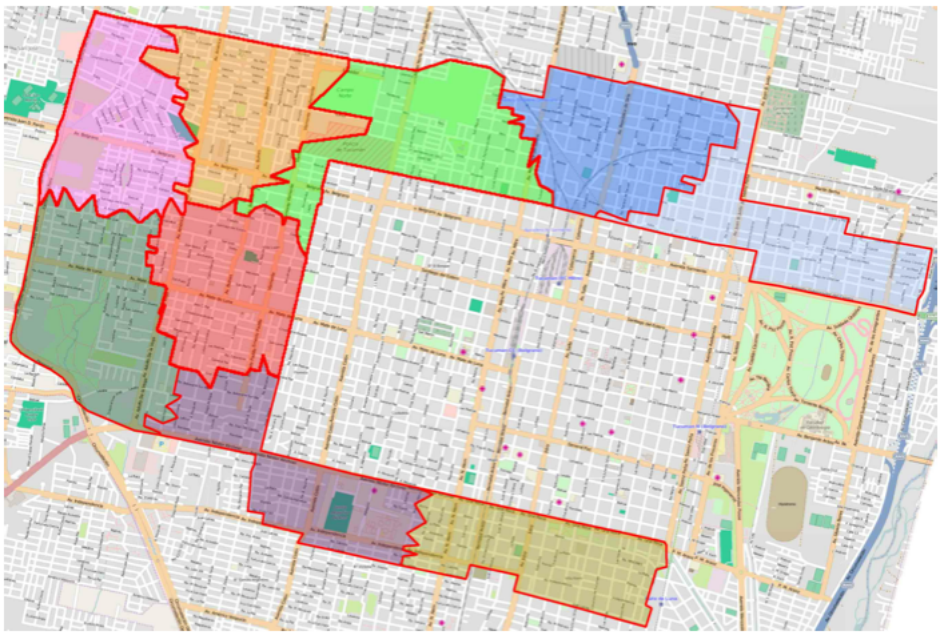

Fig. 1.1: Posible zonificado de la región nocturna de la ciudad de San Miguel de Tucumán

1.3. Definiciones

- Un **grafo** $G = (V, E)$ es un par ordenado en el que V es un conjunto de puntos, nodos o vértices y E es un subconjunto del conjunto de pares no ordenados de elementos distintos de V. Es decir $E \subseteq V \times V$. A su vez, los elementos de E se denominan aristas.

- Un grafo en el cual sus aristas son pares ordenados se dice que es un **digrafo**.

- Un **multigrafo** es un grafo en el que puede haber varias aristas entre un mismo par de nodos distintos.

- En un grafo no dirigido se denota $d(x)$ al **grado** del nodo x, el cual es la cantidad de aristas incidentes al mismo. En un digrafo el grado en cambio se divide en grado de entrada $din(x)$, que es la cantidad de arcos que apuntan hacia el nodo x; y grado de salida, $dout(x)$, que es la cantidad de arcos que parten de x hacia otros nodos. Para este trabajo, se usará la notación $dtot(x) = din(x) + dout(x)$ para el grado total del nodo, es decir, la suma de todos los arcos entrantes o salientes del nodo x.

- Un **camino** en un grafo es una sucesión de aristas $(x_1, y_1), (x_2, y_2)...$ pertenecientes al grafo tal que $y_i = x_{i+1}$. En un digrafo un camino es una sucesión de arcos que cumplen la condición previa, mientras que en un grafo mixto es una sucesión de aristas y/o arcos. Un camino simple es un camino que no pasa más de una vez por el mismo nodo.

- La **longitud** de un camino es la cantidad de aristas que lo componen; y la distancia $d(x, y)$ entre dos nodos x e y se define como la longitud del camino más corto entre x e y.

1.4. Teoría de complejidad computacional

Saber si un algoritmo es mejor que otro puede estudiarse desde al menos dos puntos de vista: un algoritmo es mejor cuanto menos tarde en resolver un problema, o bien es tanto mejor cuanta menos memoria necesite para ello. La idea del tiempo que consume un algoritmo para resolver un problema es denominada *complejidad temporal* y a la memoria que necesita el algoritmo es llamada *complejidad espacial*.
A lo largo de este trabajo, cuando se hable de complejidad, salvo aclaración, se estará haciendo referencia a la temporal. Por ser el tiempo dependiente de la máquina donde se ejecuta el algoritmo, el patrón de medida que se utiliza es la cantidad de instrucciones.
Un algoritmo tiene una entrada cuyo tamaño muchas veces determinará la cantidad de instrucciones que requiera para devolver el resultado. Por lo tanto la complejidad temporal será una función que determine ese número de instrucciones en función del tamaño de la entrada. Dado un problema, se dice que un algoritmo que lo resuelve es un algoritmo eficiente si presenta o tiene una complejidad polinomial.
Es útil clasificar los problemas según su complejidad, pero esto no es necesario hacerlo con todos los problemas, sino con un grupo particular: La teoría de complejidad tiene una rama denominada NP-completitud la cual abarca solamente los problemas de decisión, es decir aquellos cuya respuesta es binaria.
Un problema de decisión es un tipo especial de problema computacional cuya respuesta

es solamente **SI** o **NO** (o, de manera más formal, 1 o 0).
Dicho problema pertenece a la clase P (Polinomial) si existe un algoritmo polinomial que lo resuelve. Un problema de decisión pertenece a la clase NP (Polinomial no-determinísticamente) si, dada una instancia de **SI** y evidencia de la misma, puede ser verificada en tiempo polinomial. Es trivial ver que $P \subseteq NP$, lo que no está resuelto es la validez de la inversa.
Una transformación o reducción polinomial de un problema de decisión α_1 a uno α_2 es una función que transforma en tiempo polinomial una instancia I_1 de α_1 en una instancia I_2 de α_2 tal que I_1 tiene respuesta **SI** para $\alpha_1 \Leftrightarrow I_2$ tiene respuesta **SI** para α_2. El problema de decisión α_1 se reduce polinomialmente a otro problema de decisión α_2, $\alpha_1 \leq_p \alpha_2$, si existe una transformación polinomial de α_1 a α_2. Las reducciones polinomiales cumplen transitividad.
Un problema de decisión α es NP-completo si

- $\alpha \in NP$
- $\forall \beta \in NP, \beta \leq_p \alpha$

En su artículo de 1971 *"The Complexity of Theorem Proving Procedures"* Cook demostró que el problema SAT es NP-completo. Y a partir de entonces, usando la transitividad de las reducciones polinomiales se ha probado la NP-completitud de muchos problemas.
Un problema de decisión α es NP-hard si todo otro problema de NP se puede transformar polinomialmente a α. En la práctica, esta definición a veces se usa por un abuso de lenguaje también para problemas que no son de decisión y cuya versión de decisión es NP-completa.
El problema α es una restricción de un problema β si el dominio de α está incluido en el dominio de β. En este caso se dice además que β es una extensión de α. Adicionalmente, si α es NP-completo, entonces β es NP-hard.
¿Qué hacer si la versión de decisión del problema que se intenta resolver en la práctica es NP-completo? Una posibilidad es utilizar heurísticas. Una heurística es un algoritmo que intenta obtener resultados cercanos al óptimo en un tiempo razonable. Aunque a menudo pueden encontrarse instancias concretas del problema donde la heurística producirá resultados muy malos o se ejecutará muy lentamente. Por lo tanto lo que buscan las heurísticas es obtener buenas soluciones para problemas que en la teoría son muy complejos. Si se desarrollan para un problema concreto suelen denominarse simplemente heurísticas, mientras que los métodos genéricos que dan lugar a una familia de algoritmos heurísticos se denominan metaheurísticas.

1.5. Programación lineal entera

La **Programación lineal** es el campo de la optimización matemática dedicado a maximizar o minimizar (optimizar) una función lineal, denominada función objetivo, de tal forma que las variables de dicha función estén sujetas a una serie de restricciones expresadas mediante un sistema de ecuaciones o inecuaciones también lineales. El método tradicionalmente usado para resolver problemas de programación lineal es el **Método Simplex** [12].
Un problema de Programación Lineal (LP, del inglés Linear Programming) es un problema de optimización con restricciones en el cual se busca un conjunto de valores para variables

continuas $(x_1, x_2, ..., x_n)$ tal que maximice o minimice una función objetivo lineal, satisfaciendo a su vez un conjunto de restricciones lineales (el cual es un sistema de ecuaciones y/o inecuaciones lineales)[24][25].
Matemáticamente, un LP se expresa de la siguiente manera:

$$\begin{gathered}\text{maximizar } z = \textstyle\sum_j c_j x_j \\ \text{sujeto a } \textstyle\sum_j a_{ij} x_j \leq b_i \ (i = 1, 2, ..., m) \\ x_j \geq 0 \ (j = 1, 2, ..., n)\end{gathered}$$

Un problema de Programación Lineal Entera (siglas PLE, IP del inglés: Integer Programming) es un LP en el cual al menos una de las variables está restringida a valores enteros. Generalmente si admite variables enteras y variables continuas, se le da el nombre de problema de Programación Lineal Entera Mixta (PLEM, o MIP del inglés: mixed integer programming), mientras que si todas sus variables son del tipo entero, se lo suele denominar problema de PLE puro. Un **PLEM**, matemáticamente, se define de la siguiente manera:

$$\begin{gathered}\text{maximizar } z = \textstyle\sum_j c_j x_j + \sum_k d_k y_k \\ \text{sujeto a } \textstyle\sum_j a_{ij} x_j + \sum_k g_{ik} y_k \leq b_i \ (i = 1, 2, ..., m) \\ x_j \geq 0 \ (j = 1, 2, ..., n) \\ y_k \in \mathbb{Z}_+ \ \ (\text{k} = 1,\ 2,\ \ldots\ ,\ \text{p})\end{gathered}$$

Con los parámetros de entrada $c_j, d_k, a_{ij}, g_{ik}, b_i \in \mathbb{R}$.
Usando la notación matricial, el mismo PLEM puede ser expresado:

$$\begin{gathered}\text{maximizar } z = c^T x + d^T y \\ \text{sujeto a } Ax + Gy \leq b \\ x \geq 0 \\ y \in \mathbb{Z}_+^p\end{gathered}$$

donde m, n y p son la cantidad de restricciones, variables continuas y variables enteras respectivamente;

- $c = (c_j)$ es un vector de n elementos.
- $d = (d_k)$ es un vector de p elementos.
- $A = (a_{ij})$ es una matriz de $m \times n$.
- $G = (a_{ik})$ es una matriz de $m \times p$.
- $b = (b_i)$ es un vector de m elementos.
- $x = (x_j)$ es un vector de n variables continuas.
- $y = (y_k)$ es un vector de p variables enteras.

Un caso especial de PLE es aquel en el cual todas sus variables se restringen a valores binarios, el cual se define como problema de **Programación Lineal Entera Binaria** (PLEB o BIP del inglés Binary Integer Program). Esta clase de problemas es importante ya que la mayor parte de los problemas de optimización combinatoria se pueden formular

como un PLEB.
Un PL se dice que está expresado de **forma estándar** si se cumple que:

1. La función objetivo es maximizar una combinación lineal de las variables.
2. Todas las restricciones son desigualdades de la forma $\leq$.
3. Cada variable entera está definida sobre un conjunto de enteros consecutivos cuyo límite inferior es 0 y el superior infinito.
4. Cada variable continua es no negativa sin límite superior.

De otro modo, se dice que está expresado en forma no-estándar. Todo problema de maximización puede ser reformulable como uno de minimización.

1.6. Coordenadas geográficas

Las coordenadas geográficas son un sistema de referencia que utiliza las dos coordenadas angulares, latitud (Norte y Sur) y longitud (Este y Oeste). Estas coordenadas se suelen expresar en grados sexagesimales:

- La **longitud** mide el ángulo a lo largo del Ecuador desde cualquier punto de la Tierra. Las líneas de longitud son círculos máximos que pasan por los polos y se llaman meridianos. Para los meridianos, sabiendo que junto con sus correspondientes antimeridianos se forman circunferencias de 40.007,161 km de longitud, 1 grado de dicha circunferencia equivale a 111,131 km.
- La **latitud** es el ángulo que existe entre un punto cualquiera y el Ecuador, medida sobre el meridiano que pasa por dicho punto. La distancia en kilometros a la que equivale un grado de dichos meridianos depende de la latitud, a medida que la latitud aumenta disminuyen los kilómetros por grado.

1.7. Open Street Map

OpenStreetMap (también conocido como OSM) es un proyecto colaborativo para crear mapas libres y editables en el cual los mapas se crean utilizando información geográfica capturada con dispositivos GPS móviles, ortofotografías y otras fuentes libres. Sus usuarios registrados pueden subir sus trazas desde el GPS y crear y corregir datos vectoriales mediante herramientas de edición creadas por la comunidad OpenStreetMap. Cada semana se añaden 90.000 km de nuevas carreteras con un total de casi 24.000.000 km de viales (febrero de 2011), eso sin contar otros tipos de datos (pistas, caminos, puntos de interés, etc.). El tamaño de la base de datos (llamada planet.osm) se situaba en febrero de 2011 por encima de los 205 gigabytes (14 GB con compresión bzip2), incrementándose diariamente en unos 10 megabytes de datos comprimidos.

1.8. Contenido de la tesis

- En el **Capítulo II** del presente informe se realizará en primer lugar un planteo detallado del problema a tratar, se enumerarán las características de diversas regiones a zonificar junto con las limitaciones y requerimientos formales e informales.
  En segundo lugar, se presentará el algoritmo propuesto en esta tesis, se estudiarán sus estructuras asociadas junto a la teoría que hay detrás de ellas. Se analizarán exhaustivamente todos los componentes de este algoritmo heurístico.
  Se enumerarán las diversas dificultades encontradas, y cómo fueron superadas, en pos de resolver la problemática planteada y optimizar ciertos parámetros, en especial la complejidad temporal en los algoritmos propuestos y estudiados.

- En el **Capítulo III** se incluyen diversos experimentos realizados con el algoritmo descripto en el capítulo anterior. Se estudian y testean resultados obtenidos de la aplicación del algoritmo a diversas ciudades. También se analiza cómo varían estos resultados de acuerdo a parámetros seleccionados (tamaño de zonas, minimización en cantidad de solapamientos en la zonificación, etc).
  Se priorizan tiempos de ejecución y calidad de resultados obtenidos, los cuales serán expuestos en gráficos a lo largo del capítulo.

- En el **Capítulo IV**, analizando los experimentos y resultados obtenidos, se enuncian las conclusiones que se desprenden de dicho proceso. Asimismo, se enumera una serie de posibles trabajos a futuro, que puedan tomar como puntapié inicial el presente trabajo de investigación, encontrando mejoras y nuevas alternativas en vías de resolver la problemática vinculada al proceso de zonificación en regiones determinadas.

2. CAPÍTULO II: PROBLEMÁTICA Y ENFOQUE DE RESOLUCIÓN

2.1. Requerimientos

El objetivo principal de este trabajo es el particionado del mapa de una región determinada de acuerdo a ciertos parámetros que se desean optimizar. Esto es, por ejemplo, el tamaño de las zonas en las que es particionado el mapa, la cantidad de zonas definidas, la cantidad de solapamiento entre dichas zonas. Más allá de estos objetivos, lo que principalmente se busca es obtener zonas de contorno sencillo, a fin de que sean fácilmente delineables y memorizables por parte de las personas involucradas en la recolección de residuos.
Para esto se utilizarán esquemas de generación de polígonos en conjunto con modelos de programación lineal entera en la selección de zonas que cubran el mapa de la ciudad.

2.2. Algoritmo propuesto

Se propone en esta tesis un algoritmo basado en heurísticas para encontrar solución a la problemática previamente mencionada. Dicho algoritmo consiste inicialmente en el procesamiento del archivo osm provisto por OpenStreetMap. Por un lado del parsing del mismo, mapéandose dicho osm a un multigrafo en el cual se representará el mapa determinado, donde los vértices representan las esquinas y las aristas las cuadras del mismo. Por otro lado, se debe realizar un proceso de filtrado del grafo, restringiendo su alcance a los límites administrativos de la ciudad representada en cuestión, evitando procesamientos innecesarios que bajen la performance total.
Sumado a esto, se subdivide el mapa para de esta manera reducir los tiempos en la verificación de cubrimiento de aristas por polígonos al momento de agregar restricciones al modelo de programación lineal entera planteado.
Se plantea una heurística de generación de polígonos basada en la selección de pares de nodos denominados *iniciales* que se explicará con mayor detalle posteriormente. Luego del armado de un conjunto de polígonos, se plantea y resuelve un modelo de programación lineal entera obteniendo, en el caso de encontrar solución factible, un subconjunto de polígonos que satisfacen las respectivas restricciones. Se procesan los polígonos solución, incluyendo transformaciones del plano cartesiano a geocoordenadas para ser visualizadas correctamente sobre el mapa. Se aplican algoritmos heurísticos en el procesamiento de la solución, con el objetivo de adaptarla y mejorar la calidad de la misma, en lo que respecta a tamaño y formas de polígonos. Se recortan polígonos para evitar solapamientos entre ellos, y se los une o combina evitando residuos chicos en la visualización de la solución.
A continuación se muestra el pseudocódigo correspondiente al algoritmo que resuelve la problemática en cuestión, luego en las secciones posteriores se visualizarán los pseudocódigos y se describirán los respectivos comportamientos correspondientes a cada paso del algoritmo.

```
1: procedure ALGORITMO PROPUESTO
2:     Parsing de osm
3:     Heurística de generación de polígonos
4:     Armado y resolución de modelo de PLE
5:     Extracción de polígonos de la solución
6:     Operaciones con polígonos
7: end procedure
```

2.3. Representación del mapa de la ciudad

En una primera instancia, el mapa M de la región a estudiar se modelará como un **multigrafo** $G = (V_G, E_G)$. Dicho grafo cumplirá que:

- sus arcos $E_G \subseteq (V_G \times V_G)$ serán los segmentos de calles y avenidas mano única, y su sentido estará dado por el orden de sus nodos,
- el conjunto de vértices V_G representará entonces los extremos e intersecciones de segmentos de caminos vehiculares. Dichos nodos tendrán asociados datos referidos a su localización (coordenadas latitud y longitud).

En el caso de avenidas o calles doble mano, habrá una arista saliente y entrante en cada nodo perteneciente. En el caso de calles o avenidas de mano única solo un arco dirigido en el mismo sentido.

2.4. Representación computacional

Inicialmente el grafo que se utilizó para la primera versión del algoritmo contenía todos los vértices y aristas obtenidos del parseo del archivo **osm** de OpenStreetMap, no se aplicaba ningún tipo de filtrado o reducción del tamaño de la entrada. Al trabajar con mapas, para representar curvas suelen utilizarse varios segmentos rectos, por lo que una cuadra (segmento de camino vehicular determinado por dos intersecciones con otros caminos) quizás comprenda un conjunto de aristas o arcos. Por este motivo, inicialmente se obtuvo una estructura demasiado grande, sumado a que el conjunto de nodos V incluía nodos fuera del casco urbano de la ciudad o fuera del límite administrativo de la misma que para el proceso de zonificación eran innecesarios. El principal inconveniente al tener una estructura amplia era el aumento en el tiempo de ejecución del algoritmo.
Para reducir el número de variables, los algoritmos implementados en esta tesis trabajan con un **modelo simplificado** en el cual se eliminan los nodos inútiles del grafo original; aquellos nodos cuya geolocalización se encuentra fuera de los límites administrativos de la ciudad y no intervienen en la zonificación de la misma (ver **algoritmo de filtrado** del grafo original). Sumado a la reducción en número de cantidad de vértices en V_G, naturalmente, también se reduce la cantidad de aristas en E_G.
A continuación se menciona la estructura utilizada en la representación del grafo $G = (V_G, E_G)$ que modela a cada mapa de una determinada ciudad:

- G está representado a través de la clase **RoadGraph**, la cual guarda toda la información necesaria en la resolución del problema.

- Dicha clase contiene por un lado el conjunto de nodos o vértices V_G, representado mediante una instancia $HashMap < Long, GraphNode >$ del lenguaje de programación **Java**. De esta manera, para acceder a la información del nodo, la cual se almacena en un objeto de la clase **GraphNode**, sólo es necesario su número de identificación (**id**). Esto permite dicho acceso en tiempo promedio **O(1)**, reduciendo la complejidad temporal total del algoritmo.

- El atributo **adyLst** guarda la información correspondiente a las adyacencias para cada nodo (esquina) en el mapa. Se representa a través de una instancia de la clase Map, la cual relaciona para cada nodo, indexado con su id, su correspondiente lista enlazada de adyacencias.

- Cada adyacencia de un nodo es representada con la clase definida **AdyacencyInfo**. Guarda información referida al *id* del nodo vecino, *distancia* o *longitud* que los separa, si el segmento es mano única o doble mano, el *tipo* de camino (mano única o doble) y el *nombre* de la calle en dicho segmento.

- El conjunto de referencias a nodos que forman parte del contorno de la ciudad o límite administrativo de la misma.

- El conjunto de nodos, propiamente dichos, integrantes del límite/contorno de la ciudad.

2.5. Algoritmos

A continuación se enumeran los diversos algoritmos auxiliares utilizados durante el algoritmo presentado en la Seccion 2.2.

2.5.1. Parsing: del osm al grafo

Los datos del mapa M del municipio vienen dados en formato .osm (de OpenStreetMap), el cual consta de un conjunto de nodos $N_M = \{n_1, ..., n_j\}$, un conjunto de caminos $W_M = \{w_1, ..., w_k\}$, y un conjunto de restricciones R_M.

- Cada **nodo** $n_i \in N_M$ tiene asociado un par de coordenadas $(latitud_i, longitud_i) \in \mathbb{R}^2$ y un conjunto de atributos.

- Un **camino** $w_i \subseteq N_M \times ... \times N_M$ (cantidad de veces igual al largo del camino) está definido como una sucesión ordenada de un subconjunto del conjunto de nodos y un listado de atributos. Dependiendo de éstos, el camino representará una calle doble mano, una avenida, un río, una vía de tren, el perímetro de un edificio público, etc. En caso de ser una calle mano única, el orden de los nodos indicará el sentido de la misma.

- Las **restricciones** $r_i \subseteq W_M \times N_M \times W_M$ determinan los giros prohibidos de un camino a otro camino a través de un nodo.

La conversión de este formato al grafo G sobre el cual se trabajará vendrá dada por:

- el conjunto de aristas dirigidas $E_G \subseteq (V_G \times V_G) \subseteq (N_M \times N_M)$ cumplirá que para toda arista $(x, y) \in E_G$ existe un camino $w_i \in W_M$ tal que $w_i = (n_1, ..., x, y, ...n_n)$ y cuyos atributos indican avenida o calle residencial de doble sentido.
- el conjunto de vértices $V_G \subseteq N_M$ cumplirá que si existen $v, y \in V_G$ con $v \neq y$ y adyacentes entre sí, es decir con aristas salientes y entrantes entre ellos, entonces $(v, y) \in E_G$ y $(y, v) \in E_G$.

Al trabajar con coordenadas geográficas, la distancia euclideana en km entre dos nodos n_i y n_j estará dada por:

$$d = 2 \cdot \arctan2(\sqrt{a}, \sqrt{1-a}) \cdot r \tag{2.1}$$

En donde se define

$$a = \sin^2\left(\frac{dLat_{i,j}}{2}\right) + \cos(lat_i) \cdot \cos(lat_j) \cdot \sin^2\left(\frac{dLon_{i,j}}{2}\right) \tag{2.2}$$

$$r = 6373Km(el\ radio\ de\ la\ Tierra) \tag{2.3}$$

$$dLat_{i,j} = (latitud_j - latitud_i) \tag{2.4}$$

$$dLon_{i,j} = (longitud_j - longitud_i) \tag{2.5}$$

Como se mencionó con anterioridad, se trabajará con diversos mapas de ciudades en un formato específico. Cada uno de ellos es descargado de **OpenStreetMap** en el formato con extensión **.osm**.
Se cuenta con el método **osmGraphParser** de la clase RoadGraph, el cual es el encargado de realizar la traducción. Se encarga de leer el archivo con formato osm de entrada. Durante este proceso, al encontrarse con distintos *labels*, se van llenando estructuras definidas en la representación del grafo.
Si encuentra el label *node* creará una instancia de la clase GraphNode, la cual completará con información leída debidamente a continuación. Es decir, si se encuentra el label *id* completará el atributo correspondiente en el GraphNode. Por otro lado, si se lee el label *lat*, completrá con dicho valor leído el atributo lat del nodo. De la misma manera al leerse el label *lon*. Más precisamente:

Algorithm 1 Parsing de nodos

```
1: tempNode ← createGraphNode()
2: attributesCount ← parser.getAributtesCount()
3: for 0:attributeCount do
4:     if parser.getAttributeName() = id then
5:         tempNode.setId(parser.getAttributeValue())
6:     end if
7:     if parser.getAttributeName() == "lat" then
8:         tempNode.setLat(parser.getAttributeValue())
9:     end if
10:    if parser.getAttributeName() == "lon" then
11:        tempNode.setLon(parser.getAttributeValue())
12:    end if
13: end for
```

Del mismo modo, se realiza un similar procedimiento para obtener información de cada camino. Los caminos transitables contienen el atributo "highway" con un nombre asociado que determina qué tipo de camino es. Es decir se consideran únicamente aquellos *highways* con su valor asociado. También información sobre si la calle o avenida es mano única, su nombre, etc. Cabe mencionar que durante este proceso, se filtran aquellos caminos que no sean calles o avenidas urbanas. Se descartan aquellas pertenecientes a caminos rurales, de ferrocaril, bicisendas, etc. También si es un camino que forma parte del borde administrativo de la ciudad. En dicho caso se agregan los nodos atravesados por dicho camino al conjunto de nodos pertenecientes a la *boundary zone* de la ciudad.

Algorithm 2 Parsing de caminos

```
1: for 0:attributeCount do
2:     if parser.getAttributeName() ="k"∧ parser.getAttributeValue() ="highway"
   then
3:         String v ← parser.getAttributeValue() + 1
4:         if v ∉ filteredWays then
5:             tempWay.setType(v)
6:         end if
7:     end if
8:     if parser.getAttributeName() ="k"∧ parser.getAttributeValue() ="name" then
9:         if isBoundary(parser.getAttributeValue()) then
10:            Guardar referencias zona límite del mapa de la ciudad
11:        else
12:            tempWay.setName(parser.getAttributeValue() + 1)
13:        end if
14:    end if
15:    if parser.getAttributeName() ="k"∧ parser.getAttributeValue() ="oneway" then
16:        tempWay.setOneWay(parser.getAttributeValue() + 1)
17:    end if
18:    if parser.getAttributeName() ="id" then
19:        tempWay.setId(parser.getAttributeValue())
20:    end if
21: end for
22: allWays.add(tempWay)
```

Luego, cada camino obtenido es procesado. Se completan las listas de adyacencias para cada nodo y la información correspondiente, la distancia euclideana entre nodos, el nombre de la calle que los une y el sentido de la misma.
Si se llegara a encontrar el tag k con valor *name*, hay dos posibilidades. La primera es que se trate de un camino que forme parte del contorno de la ciudad. En dicho caso se agregan sus referencias al conjunto de caminos que forman parte de dicho límite. Por el otro lado, en caso de no tratarse del límite, se setea el valor del nombre al camino corriente.
Cabe mencionar que el parser lleva un control acerca de cuándo trata con un camino frontera y cuándo no, esto lo realiza leyendo previamente otros tags contenidos en el archivo osm (algoritmo de Parsing de caminos).
Finalizada la lectura y creación de caminos del mapa, se procede a realizar el procesamiento de los mismos. Se arman y completan las listas de adyacencias correspondientes a cada

nodo. Es importante mencionar que si dos nodos son vecinos, ambos estarán incluídos en sus respectivas listas de adyacencias.

Algorithm 3 Procesamiento de caminos

```
1: for each i in 0,...,waysCount-1 do
2:     tempWay ← allWays_i
3:     firstNode ← tempWay.getFirstNode()
4:     for 1:tempWay.refsCount - 1 do
5:         nextNode ← tempWay.getNode_i
6:         length ← CalculateLength(firstNode, nextNode)
7:         adyList_firstNode ← nextNode
8:         adyList_nextNode ← firstNode
9:         firstNode ← nextNode
10:    end for
11: end for
```

2.5.2. Filtro de nodos

Como se mencionó previamente, se trabajó inicialmente con un grafo obtenido a través del parseo del archivo osm de una región determinada, el cual contenía nodos que eran útiles e influyentes en la solución, y por otro lado, otros innecesarios, cuya geolocalización se encontraba fuera de los límites administrativos de la ciudad.
Debido a esta gran cantidad de nodos la versión inicial del algoritmo debía procesar inútilmente una entrada demasiado grande, incrementándose notoriamente su tiempo de ejecución. Probando como entradas ciudades de tamaño relativamente considerable, los resultados fueron inaceptables en la práctica.
El algoritmo de filtrado propiamente dicho consta de dos pasos principales. Por un lado, se filtran sólo aquellos nodos pertenecientes al área de la ciudad en cuestión. Se recorre cada elemento contenido en el conjunto de nodos de la ciudad y se eliminan aquellos cuya geolocalización, es decir coordenadas en latitud y longitud, queda fuera del área de la misma (definida durante el parseo del archivo osm).
Por otro lado, como segunda parte, se actualizan las listas de adyacencias para cada nodo de la ciudad. Se eliminan aquellos nodos vecinos cuya ubicación está fuera de sus límites.

A continuación, los algoritmos propiamente dichos:

Algorithm 4 Filtrado de nodos

```
1: FilterOnlyNodesInCity()
2: FilterOnlyCityAdyacentsNodes()
```

Algorithm 5 Filtrado de nodos pertenecientes a la ciudad

```
1: for v ∈ V_G do
2:     if ¬nodeIsIncludedInCity(v) ∧ ¬nodeIsIncludedInBoundary(v) then
3:         removeNode(v)
4:     end if
5: end for
```

Algorithm 6 Filtrado de adyacentes que estén en la ciudad

```
1: for v ∈ nodesInCity(G) do
2:     adjacents ← getAdjacents(v)
3:     for adj ∈ adjacents do
4:         if ¬nodeIsIncludedInCity(adj) then
5:             removeNodeInList(adj, adjacents)
6:         end if
7:     end for
8: end for
```

2.5.3. Subdivisión del mapa

En un primer momento, se aplicó el algoritmo sobre la totalidad del grafo que representa a todo el mapa de una determinada ciudad.
Con el correr de las pruebas, se hizo notoriamente evidente el aumento en los tiempos de ejecución. Esto se debía a que al considerarse el grafo de la ciudad en su totalidad, la cantidad de polígonos generados por el algoritmo era muy elevada. Al momento del armado del modelo del problema para el solver, en particular la restricción de cubrimiento de aristas, que más tarde se detallará, era necesario comparar cada arista del grafo con cada uno de los polígonos generados, haciendo que la cantidad de operaciones realizadas en la operación fuera considerablemente elevada, concretamente $O(m \times n)$, siendo m la cantidad de aristas de G, y n la cantidad de polígonos generados por el algorimo.
Rápidamente se hizo notoria la necesidad de plantear una mejor solución con respecto a la performance del algoritmo, que fuera capaz de disminuir este procesamiento costoso. En numerosos casos, una arista era comparada con polígonos a una gran distancia en el mapa. Por ejemplo, el caso de una arista ubicada aproximadamente en la esquina superior izquierda del mapa con polígonos en el extremo opuesto.
Visto esto último, y con el objetivo de reducir la cantidad de polígonos con los que una arista es comparada, se propone trabajar con el grafo del mapa en forma particionada. El mapa es subdivivido en una cantidad de cuadrantes del mismo tamaño. Regiones más acotadas y específicas dentro de todo el mapa en las que cada nodo, arista y polígono son ubicados para el posterior armado del modelo.
Básicamente, el algoritmo realiza las siguientes acciones para dividir el mapa sobre el que se va a trabajar:

- Se define la clase **MapQuadrantsGenerator**, la cual es la encargada de la subdivisión del mapa en cuadrantes o regiones.

- Los cuadrantes se van generando desde la esquina superior izquierda. Se barre cada "fila" de cuadrantes hasta llegar al extremo derecho del mapa total. Luego, se cambia de fila y se continúa repitiendo el proceso.
- La cantidad de cuadrantes será determinada por el algoritmo tomando como referencia las cuatro esquinas del mapa. Es decir, se toman las esquinas con mayor latitud y longitud del mapa para referenciar los extremos del mismo. Luego sobre dichos puntos se realizan cálculos para dividir el mapa en partes del mismo tamaño.
- Arbitrariamente, la clase utiliza dos factores de división. Uno en ancho (longitudes) y otro en alto (latitudes). Se testeó el rendimiento del algoritmo con diversos valores y se terminó seleccionando el factor de 1.7 (en km) en ambos casos.

El siguiente pseudocódigo del algoritmo resume el proceso descripto:

Algorithm 7 Generador de cuadrantes del mapa

```
1: factorWidth ← 1.7
2: factorHeight ← 1.7
3: cityWidth ← getCityWidthDistance()                ▷ se calcula el ancho de la ciudad
4: cityHeight ← getCityHeightDistance()              ▷ se calcula el alto de la ciudad
5: quadrantsWidthCount ← (cityWidth/factorWidth)     ▷ cantidad de cuadrantes a lo ancho
6: quadrantsHeightCount ← (cityHeight/factorHeight)  ▷ cuadrantes a lo alto
7: offsetLat ← (cityHeight/quadrantsHeightCount)     ▷ desplazamiento en latitud
8: offsetLon ← (cityWidth/quadrantsWidthCount)       ▷ desplazamiento en logitud
9: for 0:quadrantsHeightCount-1 do                   ▷ movimiento en latitud
10:     for 0:quadrantsWidthCount-1 do               ▷ movimiento en longitud
11:         quadTemp.add(maxLatTemp, minLongitTemp)
12:         quadTemp.add(maxLatTemp, minLongitTemp + offsetLon)
13:         quadTemp.add(maxLatTemp − offsetLat, minLongitTemp + offsetLon)
14:         quadTemp.add(maxLatTemp − offsetLat, minLongitTemp)
15:         cityQuadrants.add(quadTemp)              ▷ movimiento a lo ancho
16:         minLongitTemp ← minLongitTemp + offsetLon
17:     end for
18:     maxLatTemp ← maxLatTemp − offsetLat          ▷ movimiento a lo alto
19: end for
20: CalculateCityQuadrantsAreas(cityQuadrants)
```

Algorithm 8 CalcularAreasDeCuadrantes

```
1: cityQuadsAreas ← newArea[quadrantsCount]
2: for each i in 0,...,quadrantsCount-1 do
3:     quad_temp ← cityQuadrants_i
4:     quadPoints ← quad_temp.getPoints()
5:     newArea ← calculateArea(quadPoints)
6:     cityQuadsAreas_i ← newArea
7: end for
```

2.5.4. Generación de polígonos

En geometría, un **polígono** es una figura plana compuesta por una secuencia limitada de segmentos rectos consecutivos que cierran una región en el plano. Estos segmentos son llamados lados, y los puntos en que se intersectan se llaman vértices. El interior del polígono es llamado **área**. El polígono es el caso bidimensional del politopo, figura geométrica general definida para cualquier número de dimensiones.
Respecto de la generación de todos los polígonos pertenecientes al mapa de una ciudad, se implementó una clase abstracta **PolygonsGenerator** encargada de, como su nombre lo indica, generar todos los polígonos posibles sobre la estructura del grafo de la ciudad, respetando ciertas restricciones impuestas para optimizar dicho proceso.
En este trabajo se decidió implementar una clase abstracta en la generación de polígonos para abstraer la forma en la que se generan dichos polígonos del propósito de la generación en sí lo que permitió lograr un menor acoplamiento de clases, logrando en el futuro, minimizar el impacto en el caso de realizar modificaciones en la estrategia utilizada al generar los polígonos.
La clase concreta encargada del proceso es **PolygonsGeneratorFromFilteredEntries**. En las próximas secciones se describe con detalle el procedimiento de generación.

2.5.5. Selección de nodos iniciales

Selecciona aquellos nodos ubicados en la ciudad que cumplan ciertas restricciones. Esto se debe a que en un primer momento, y a medida que se fue testeando el proceso, se hizo notoria la generación de una cantidad considerable de polígonos que, si bien erán válidos e intuitivos de ver en el mapa, no aportaban a una resolución aceptable del problema. Generalmente eran polígonos muy pequeños, que abarcaban por ejemplo una o dos manzanas, o eran de formas muy poco regulares, dificilmente aplicables en la práctica por los conductores de camiones. Con el transcurso de las pruebas, se fueron aplicando restricciones en la selección de estos nodos a los que en este trabajo se los llama *iniciales*. Son pares de nodos seleccionados por el algoritmo, a partir de los cuales comienza a recorrer todas las direcciones posibles hasta encontrar dos intersecciones más, en el caso de ser posible. En el caso de encontrarlas, cierra el polígono con los cuatro puntos extremos del mismo. En el caso de no poder encontrar las dos intersecciones mencionadas, eventualmente llegará un punto en el que el algoritmo no pueda continuar avanzando en ninguna dirección posible. En dicho caso, el algoritmo determina que no hay posibilidad de formar nuevo polígono, descarta la selección de nodos iniciales y comienza una nueva búsqueda a partir de la selección de un nuevo par de iniciales.

2.5.6. PathNodes

En cuanto a estructuras de datos utilizadas por el algoritmo, se cuenta con dos arrays **pathNode1** y **pathNode2** (por cada vértice inicial seleccionado en cada iteración) de tamaño fijo (por cada una de las direcciones de avance) compuesta por *LinkedLists* de instancias de la clase $AdyacencyInfo$ (con información correspondiente a cada nodo integrante del path). Cada $pathNode$ guarda el camino de nodos a partir de un nodo inicial seleccionado en cada una de las direcciones posibles. Luego, el algoritmo se encarga de utilizarlos en la detección de posibles intersecciones entre caminos. Específicamente se buscan dos intersecciones, logrando de esta manera establecer cuatro puntos esquinas del

nuevo polígono (intersecciones y vértices iniciales). Se contempla el caso de aquellos nodos iniciales con un grado menor (2 o 3). En dicha situación se tendrán pathNodes vacíos en alguna/algunas dirección/direcciones.

2.5.7. Búsqueda de vecindad

A partir de cada uno de los dos vértices de inicio, este procedimiento avanza de ser posible en las diferentes direcciones (paths) posibles. Utiliza el método **PuedeAvanzar**, el cual intenta avanzar en por lo menos alguna de ellas. En el caso en que no pueda hacerlo por ninguna, y al no haber encontrado dos intersecciones necesarias para formar el polígono, descarta los nodos iniciales y selecciona un par nuevo dando inicio a una nueva iteración.
El algoritmo realiza dos chequeos para determinar si es posible avanzar en cada una de las direcciones. Por un lado, toma el nombre inicial de la calle por donde avanza la primera vez a partir del nodo inicial y parado en el último nodo de cada camino, busca entre todas las direcciones (aristas dirigidas que salen del vértice), aquella cuyo nombre de calle coincida (**Búsqueda por nombre de calle**).

Algorithm 9 Búsqueda por nombre de calle

```
1: for i ∈ 0 : pathsCount − 1 do
2:     lastNode ← pathNode[i].getLastNode
3:     nameStreet ← lastNode.getNameStreet
4:     adyacents ← lastNode.getAdyacents
5:     for ady ∈ adycents do
6:         if ady.nameStreet = nameStreet then
7:             adyacentFound = ady
8:             exit
9:         end if
10:    end for
11: end for
```

Se muestran los siguientes gráficos para hacer más entendible el funcionamiento descripto:

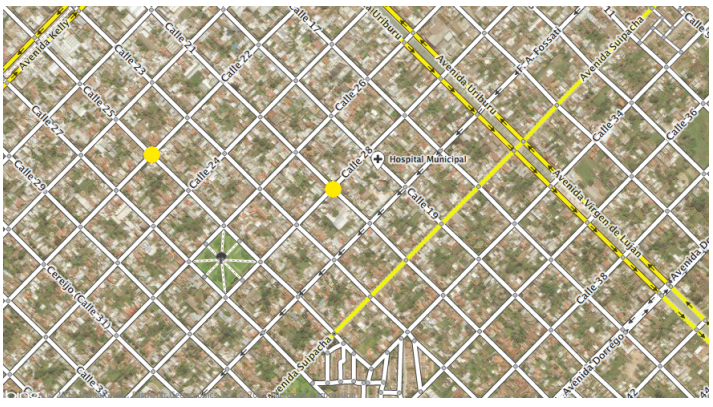

Fig. 2.1: Selección de nodos iniciales

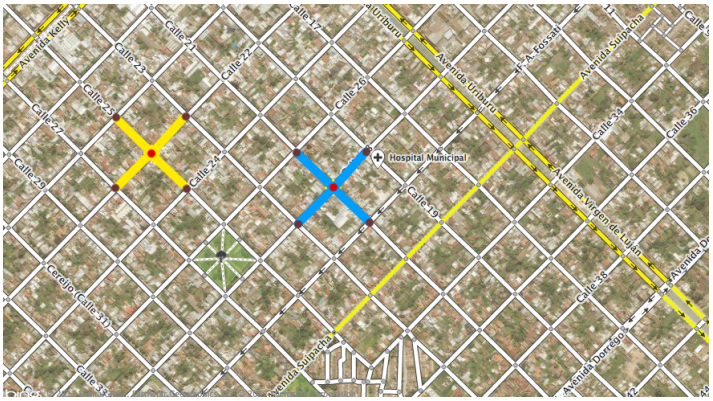

Fig. 2.2: Primer avance en cada dirección

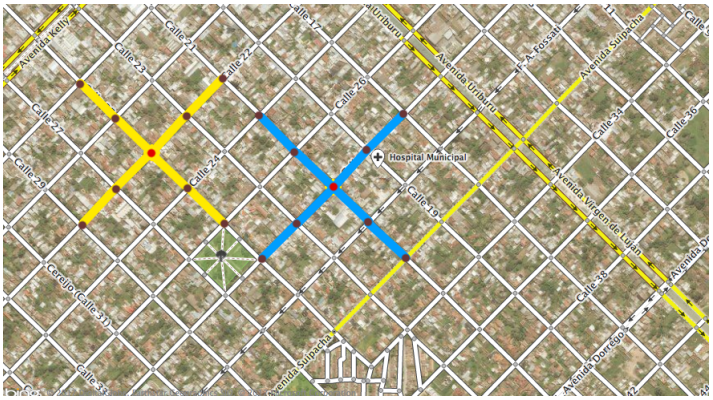

Fig. 2.3: Segundo avance en cada dirección

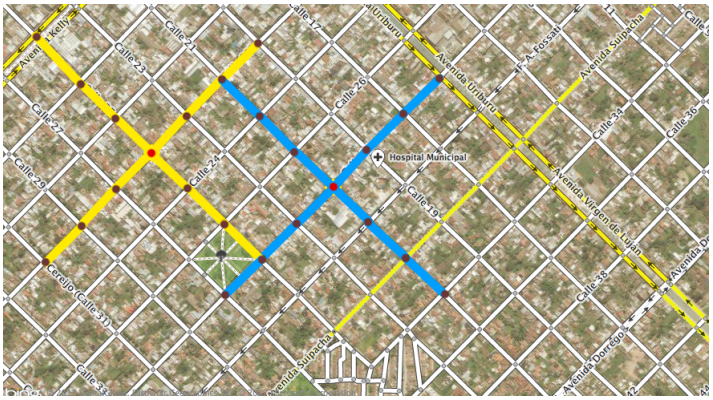

Fig. 2.4: Tercer avance en cada dirección

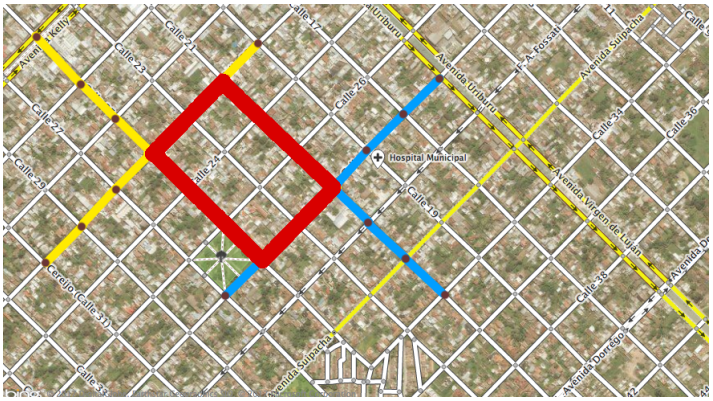

Fig. 2.5: Armado del polígono encontrado

2.5.8. Análisis geométrico en selección de vecinos

Claramente la variante presentada previamente no contempla los casos probables en que una calle cambie de nombre a partir de una determinada cuadra.

Fig. 2.6: Caso en el que una calle cambia de nombre. Ciudad de Junín

Se hizo necesario entonces agregar otro tipo de búsqueda complementaria de vecindad en la misma dirección, que sea independiente del análisis de los nombres de las aristas. Para ello, el trabajo se basó en el siguiente concepto geométrico:

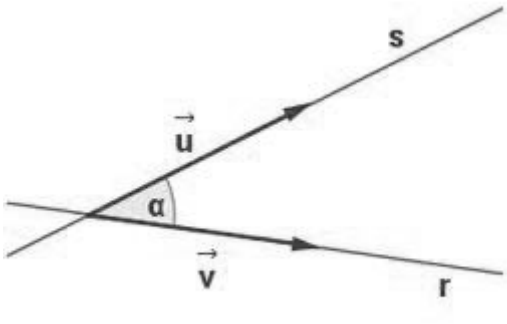

Fig. 2.7: Ángulo formado entre las rectas r y s

El ángulo formado entre dos rectas (Figura 2.7), es el menor de los ángulos que se forma de éstas. Podemos obtener la medida de este ángulo tanto por sus vectores directores o por sus pendientes. Veamos a continuación la representación de dos rectas y la fórmula para hallar el ángulo mediante sus vectores:
Sea u y v los vectores que se pueden ver en la **Figura 2.8**, el ángulo α comprendido entre ambos se calcula de la siguiente forma:

$$cos\ \alpha = \frac{|u_1 \cdot v_1 + u_2 \cdot v_2|}{\sqrt{u_1^2 + u_2^2} \cdot \sqrt{v_1^2 + v_2^2}} \tag{2.6}$$

Como se mencionó previamente, también se puede obtener el ángulo a partir de las pendientes de las rectas. Por ejemplo dadas dos rectas con pendientes m y m', verificaremos de la siguiente forma:

$$tan\ \alpha = \left|\frac{m' - m}{1 + mm'}\right| \tag{2.7}$$

En el presente trabajo, se tomó la decisión de utilizar la segunda versión en el cálculo del ángulo formado entre dos rectas.
Cada recta representa a cada arista. El algoritmo calcula la pendiente de la última arista de un camino. Obtiene de cada *pathNode* el último y anteúltimo nodo del mismo (en el caso inicial, el pathNode tendrá únicamente cada nodo adyacente al inicial respectivo. Por lo tanto se asumen implícitamente los nodos iniciales como cabezas de cada pathNode), luego calcula la pendiente de la recta (arista) que pasa por ellos utilizando sus respectivas coordenadas en el plano (realiza una conversión de las coordenadas geográficas a puntos del plano en $\mathbb{R}^2$).
Tomando esta pendiente m_1 como base, se calculan las pendientes m_2 entre el último nodo del camino y cada uno de sus vecinos. Luego, comparando cada par de pendientes se calcula el ángulo formado entre la recta del camino (con pendiente m_1) con la recta al nodo vecino correspondiente (con pendiente m_2). En el caso de los nodos de grado 2 el de verificación para el ángulo difiere, de esta manera se evitan los casos de las esquinas (ángulos cercanos a los 90 grados). Finalmente se analiza el valor de cada ángulo. Se establece el rango para el cual es aceptable la vecindad de un nodo:

- Ángulos cuyo módulo oscila entre los 0° y 45° para aquellos nodos de grado 3 o 4.

- Ángulos cuyo módulo oscila entre los 0° y 50° para aquellos nodos de grado 2. Con esta restricción, se evita la selección de nodos de grado 2 que sean esquinas del mapa.

Al mismo tiempo, también se verifica que el vecino cumpla la condición de que no haya estado previamente visitado. A continuación se presenta el pseudocódigo correspondiente al algoritmo:

Algorithm 10 Búsqueda por ángulo formado entre vectores de dirección

```
1: for i ∈ 0 : pathsCount − 1 do
2:     lastNode ← pathNode[i].getLastNode
3:     preLastNode ← pathNode[i].preLast
4:     m1 ← CalculatePend(preLastNode, lastNode)
5:     adyacents ← lastNode.getAdyacents
6:     for ady ∈ adycents do
7:         m2 ← CalculatePend(lastNode, ady)
8:         angle ← CalculateAngle(m1, m2)
9:         if (ánguloEnRango ∧ ady ∉ visitedNodes) ∨ (nodoGrado2 ∧ ánguloEnRango
   ∧ ady ∉ visitedNodes) then
10:            result ← ady
11:            exit
12:        end if
13:    end for
14: end for
```

En las primeras versiones, los nodos iniciales eran descartados al no poder avanzar en ninguna de las direcciones. Bastaba con no poder avanzar en al menos una de ellas para descartar la selección de iniciales.
Sin embargo, de esta forma, si bien la complejidad temporal era menor, había polígonos válidos que eran descartados en determinados casos, según la ciudad con la que era testeado el algoritmo.
Existían casos en los que no se podía continuar avanzando en una dirección determinada pero sí en las demás, permitiendo igual la formación de un polígono válido para ser considerado parte de la solución. Como el algoritmo no podía continuar avanzando en dichas direcciones estos casos eran descartados, obteniendo finalmente soluciones muy malas, con "agujeros" en el mapa, es decir, regiones del mismo que no eran cubiertas por ningún polígono de la solución. Al considerar la búsqueda de nodos adyacentes por análisis de pendientes, se añadió al algoritmo la posibilidad de considerar una mayor porción del dominio, perfeccionando su búsqueda de polígonos y de esta manera, mejorando notoriamente la calidad de las soluciones (ver mapas comparativos de las *Figuras 2.9 y 2.10*).

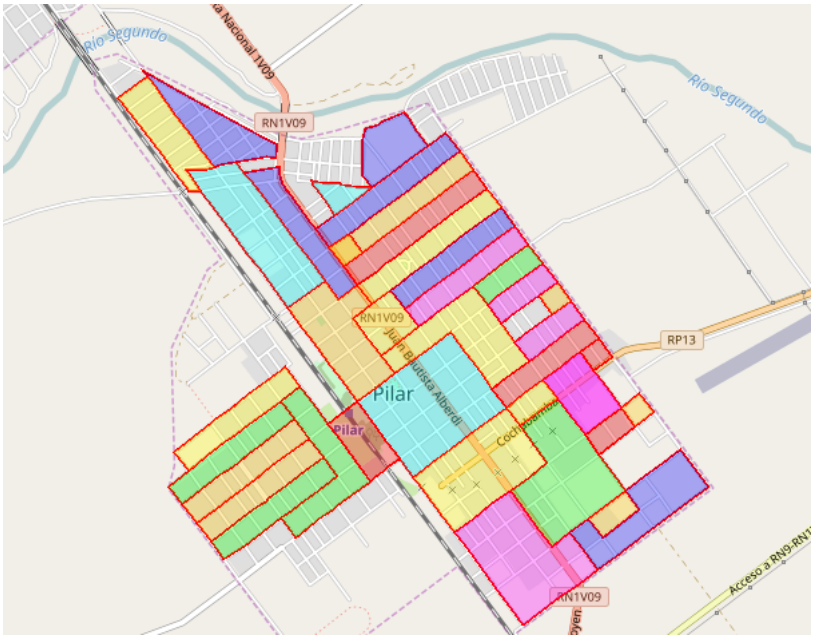

Fig. 2.8: Solución obtenida en el caso en el que únicamente se realiza búsqueda de vecinos por mismo nombre de calle. Ciudad de Pilar, provincia de Córdoba

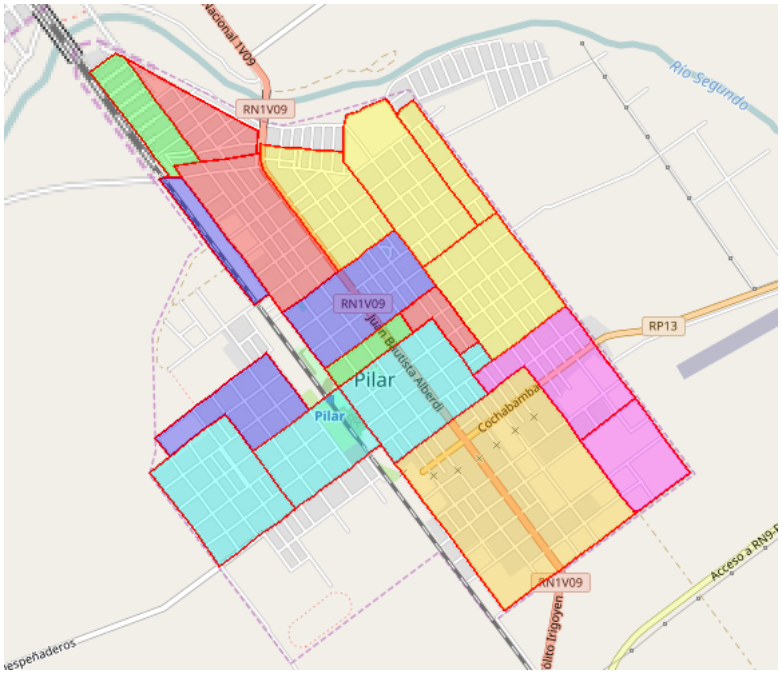

Fig. 2.9: Solución obtenida en el caso en el que se añade la búsqueda de vecinos mediante análisis de ángulos entre pendientes de vectores de dirección. Ciudad de Pilar, provincia de Córdoba

2.5.9. Restricciones en selección de nodos iniciales

Se restringe la selección de pares de nodos iniciales. Son **seleccionables** si verifican las siguientes restricciones:

- **Distancia:** se verifica que ambos nodos estén a una distancia euclideana, en un rango especificado el cual se fue modificando en virtud de testear el comportamiento del algoritmo y la calidad de las soluciones obtenidas. En una primera versión del algoritmo, dicha restricción no era aplicada. Debido a esto, la cantidad de polígonos obtenidos era mucho mayor y por ende también el tiempo de ejecución. Al restringir la distancia mínima y máxima a la que pueden estar los nodos iniciales, se reduce el conjunto de polígonos a generarse.

- **Desigualdad:** ambos nodos deben ser distintos. Claramente es válida al cumplirse la restricción anterior (por cota inferior).

- **Grado:** se usan aquellos nodos de grado 2, 3 o 4. Inicialmente sólo se utilizaban aquellos nodos de grado 4 pero con el correr de las pruebas, se hizo notorio que los polígonos generados no cubrían la totalidad del área de la ciudad. Principalmente las regiones de las esquinas, nodos de grado 2 y grado 3 que no eran seleccionados, dejando así porciones de la región del mapa sin ser cubiertas.

- **No vecindad:** se verifica que los nodos seleccionados no sean vecinos entre sí. El algoritmo chequea que no estén en cada una de las respectivas listas de adyacencias.

- **No en la misma dirección:** se verifica que no se encuentren en una misma dirección. Se asegura que ambos nodos no estén sobre una misma calle. Esto se realiza comparando el String con el nombre de cada calle. A pesar de la posibilidad y del hecho de que una calle cambie de nombre, los resultados obtenidos son aceptables y no es motivo para interferir en la generación de polígonos.
  Sumado a esto, el supuesto caso en el que ambos nodos se encuentren sobre una misma calle y que ésta cambie su nombre, es contemplado en el análisis, ya que el algoritmo intentará avanzar en todas las direcciones posibles (sobre el camino que une ambos nodos) y en este caso va a encontrar los dos vértices intersección pero luego al momento de validar las mismas, descubrirá que son nodos vecinos, descartando el caso. A continuación se muestra un ejemplo gráfico donde ocurre esta situación

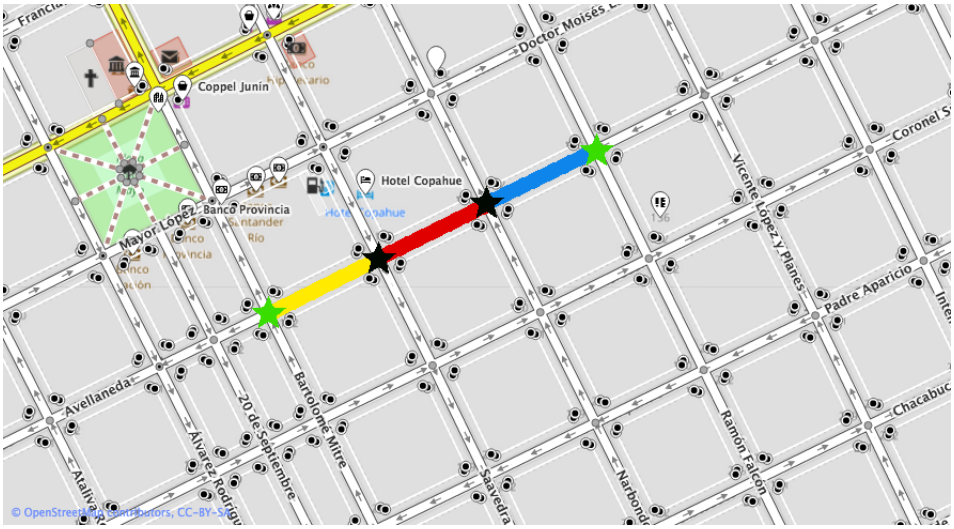

Fig. 2.10: Caso en el que se intersecan los caminos de búsqueda. Las estrellas negras corresponden a las dos intersecciones encontradas, como son vecinas, el algoritmo descarta este caso

2.5.10. Algoritmo generador de polígonos

Con respecto a la generación de polígonos, como se mencionó previamente, el procedimiento se basa en la selección de pares de nodos iniciales, nodos que cumplen la condición de ser *seleccionables*, es decir, verifican las restricciones antes mencionadas. Avanza en todas las direcciones posibles a partir de cada uno de dichos nodos iniciales formando caminos(paths) hasta, en el caso de ser posible, encontrar dos nodos de intersección entre los mismos. En dicho caso, se determina la formación de un nuevo polígono el cual se agrega al conjunto de polígonos generados para el cuadrante del mapa correspondiente. Por otro lado, en el caso de no poder encontrarse las dos intersecciones necesarias y no poder continuar avanzando por ningún path, el algoritmo termina descartando el par de iniciales y da inicio a una nueva búsqueda a partir de la selección de un nuevo par de nodos.
La versión final del algoritmo encargado de la generación de polígonos es la siguiente:

```
1: procedure Generador de polígonos
2:     for v ∈ Nodes do
3:         for u ∈ Nodes do
4:             if theyAreSelectables(u,v) then
5:                 nodosVisitados1 ← v
6:                 nodosVisitados2 ← u
7:                 agregarAdyacenciasIniciales(v)
8:                 agregarAdyacenciasIniciales(u)
9:                 cantIntersect ← checkForInitialsIntersect(v, u)
10:                while cantIntersect < 2 ∧ puedeAvanzarEnAlgunaDireccion do
11:                    avanzarEnLosPathsQueSePueda
12:                    nodosIntersec ← agregarNodosEnInterseccionDePaths
13:                    cantIntersect ← cantidadInterseccionesEntrePaths()
14:                end while
15:                if cantIntersect = 2 ∧ interseccionesValidas(nodosIntersec) then
16:                    armarPoligono
17:                    agregarPoligonoACuadrantesRespectivos
18:                end if
19:            end if
20:        end for
21:    end for
22: end procedure
```

2.5.11. Ubicación del polígono

Como se mencionó anteriormente, se hizo necesaria la subdivisión del multigrafo del mapa en diversos cuadrantes con motivos de optimizar la complejidad temporal del algoritmo general.
La idea consiste en ubicar los polígonos generados en los cuadrantes que subdividen al mapa para, de esta forma, reducir los cálculos al momento de verificar si una arista es cubierta por al menos un polígono de la solución.
Así, esta comparación se hará solamente con el conjunto de polígonos ubicados en un cuadrante en particular (el mismo al que pertenece la arista) y no con la totalidad, optimizando de esta forma el proceso.
Entonces, una vez generado cada polígono se procede a setear su cuadrante de ubicación. Para ello se toma arbitrariamente como referencia el primer nodo inicial seleccionado ($entry1$) y se obtiene el índice del cuadrante al que pertenece. Dicho índice es seteado al nuevo polígono y luego almacenado en el conjunto de polígonos generados para dicho cuadrante. Se contempla el caso en el que un polígono se interseca con mas de un cuadrante y se resuelve de la manera descripta anteriormente. Si bien se pierde calidad en las soluciones obtenidas, el algoritmo disminuye sus tiempos de procesamiento obteniendo de igual forma soluciones aceptables en la práctica.

2.5.12. Armado de polígonos

Una vez encontrados los dos vértices intersección para un par de nodos iniciales, el algoritmo generador determina que ha encontrado un polígono válido potencialmente integrante de la solución. Sin embargo solo dispone de los *pathNodes*, los nodos iniciales y los nodos de la intersección. Entonces se da inicio al proceso encargado del ensamblaje y creación del polígono (instancia de *MapPolygon*) hallado.
Como los vértices iniciales seleccionados por el algoritmo cumplen las restricciones previamente mencionadas en el proceso de selección, especificamente la de no encontrarse sobre una misma dirección, sabemos por lo tanto que deben estar ubicados como vértices opuestos si se viera al polígono con forma similar a un cuadrado o rectángulo.
Por otra parte, se cuenta con los mapeos (HashMap) de distancias a cada uno de los nodos iniciales. Es decir, se guarda durante el proceso de generación de polígonos las distancias al nodo inicial para cada nodo integrante del path. Cabe aclarar que dicha distancia guarda la cantidad de nodos intermedios desde el nodo inicial al nodo en cuestión. Por ejemplo, si se tiene el siguiente path: $v_0, v_1, v_2, v_3, v_4...$, el HashMap *distancesToNode* va a almacenar para cada nodo empezando desde v_1 su distancia (en cantidad de saltos) hasta el nodo inicial v_0.
Es decir, se tendrá lo siguiente:

- $distancesToNode(v_1) = 1$
- $distancesToNode(v_2) = 2$
- $distancesToNode(v_3) = 3$
- $distancesToNode(v_4) = 4$
- Idem para el resto de los nodos integrantes del path

Al tener esta información guardada para cada integrante de los caminos, se cuenta entonces con la distancia de los nodos intersección a cada uno de los nodos iniciales. Es decir, sean i_1, i_2 los nodos en la intersección de caminos para el par de nodos iniciales v, u. Se obtienen las distancias correspondientes, necesarias en el armado de los caminos contorno del nuevo polígono.
Por otro lado, al verificar si se intersecan los caminos recorridos por el generador de polígonos, se almacena en otro HashMap llamado *nodosInterseccionEnCaminos* el id del nodo intersección junto al par que contiene los números de caminos de los nodos iniciales en los que se intersecan. Por ejemplo, si se tiene el siguiente valor en el HashMap $(v_1, (1, 2))$. Esto se traduce como que el camino 1 correspondiente al pathNodes del nodo inicial 1 y el camino 2 del pathNodes del nodo inicial 2, se intersecan en el nodo v_1. Por lo tanto v_1 pertenece a la intersección.
Finalmente con toda esta información, se procede al armado del nuevo polígono. Se establece un orden arbitrario para agregar los nodos al polígono. Se comienza agregando uno de los nodos intersección. Luego utilizando un método auxiliar, se seleccionan de los respectivos caminos la distancia de nodos existente desde el nodo intersección a cada uno de los nodos iniciales. El pseudocódigo que se muestra a continuación explica el proceso descripto:

```
1: procedure ARMADO DE POLIGONOS
2:     intersect1 ← i1
3:     intersect2 ← i2
4:     dimensiones ← CalcularDistancias(intersection, distToNode1, distToNode2)
5:     distancesTo1Int1 ← dimensiones.get(0)
6:     distancesTo2Int1 ← dimensiones.get(1)
7:     distancesTo1Int2 ← dimensiones.get(2)
8:     distancesTo2Int2 ← dimensiones.get(3)
9:     caminoInt1 − 1 ← getCaminoInt1 − 1
10:    caminoInt1 − 2 ← getCaminoInt1 − 2
11:    caminoInt2 − 1 ← getCaminoInt2 − 1
12:    caminoInt2 − 2 ← getCaminoInt2 − 2
13:    nuevoPol.add(intersect1)
14:    agregarKesimosElem(intersect1, caminoInt1 − 1, distancesTo1Int1, nuevoPol)
15:    nuevoPol.add(initNode1)
16:    agregarKesimosElem(initNode1, caminoInt2 − 1, distancesTo1Int2, nuevoPol)
17:    nuevoPol.add(intersect2)
18:    agregarKesimosElem(intersect2, caminoInt2 − 2, distancesTo2Int2, nuevoPol)
19:    nuevoPol.add(initNode2)
20:    agregarKesimosElem(initNode2, caminoInt1 − 2, distancesTo2Int1, nuevoPol)
21: end procedure
```

Fue necesario establecer un orden en el armado ya que inicialmente, si bien el algoritmo generador de polígonos devolvía correctamente todos los nodos integrantes del polígono, en ciertos casos lo hacía en distinto orden, muchas veces en reverso, rasgo que al momento de la visualización era muy notorio.

2.5.13. Planteo y resolución del modelo

Una vez generados los polígonos, se procede al planteo del modelo de programación lineal para luego su resolución. Es decir, del conjunto total de polígonos generados para un mapa determinado, la selección del subconjunto de ellos que resuelven el problema planteado respetando ciertas restricciones.
El objetivo al plantear el modelo es el de minimizar la cantidad de polígonos que formarán parte de la solución al problema. Dichos polígonos tienen que cumplir ciertas restriciones. En este trabajo se consideraron las siguientes restricciones:

- **Cubrimiento de aristas:**
  Los polígonos solución al problema, en su totalidad, deben cubrir todas las aristas del mapa de la ciudad. Es decir, no debe quedar arista que una nodos de la ciudad sin ser cubierta por al menos un polígono de la solución. En este trabajo, se utilizará la siguiente definición de cubrimiento de aristas:

  Sea $G = (V, E)$ el multigrafo que modela al mapa de una ciudad determinada, sea $e \in E$, tal que $e = (u, v)$ con $u \in V \wedge v \in V$ y sea p un polígono generado. Entonces p *cubre a* e si y sólo si sus dos nodos extremos u y v están incluídos en p.

Por otro lado, en este trabajo un nodo o vértice pertenece a un polígono cuando queda contenido en el área del mismo. Durante el planteo del modelo a resolver, el solver con esta restricción no encontraba solución **factible**.

Problemas de factibilidad

Analizando cada caso en particular, se evidenció que en ciertos mapas, las calles cortadas no eran cubiertas por ningún polígono generado (ver Figura 2.11).
Esto ocasionaba un problema al momento de resolver el modelo en búsqueda de solución. Los problemas eran no factibles debido a esta falta de cubrimiento para estas aristas con dicha particularidad.
Para resolver el inconveniente, se modificó el algoritmo encargado de insertar la restricción al modelo, de manera tal que se contemplen dichos casos y poder "salvar" el modelo encontrando solución factible que cubra la mayor parte del conjunto de aristas del grafo.

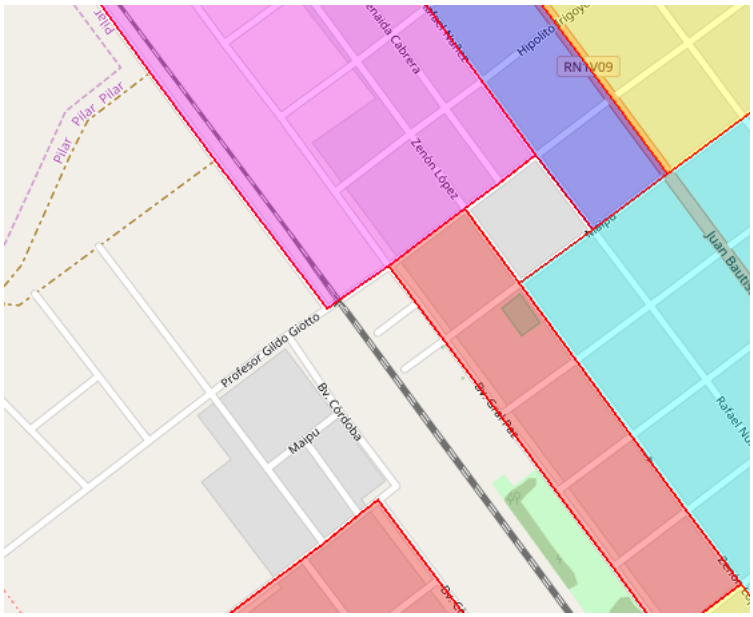

Fig. 2.11: Ejemplo en el que existen aristas no cubiertas por la solución

Finalmente, la restricción planteada en el modelo es:

$$\forall e \in E \setminus E', \sum_{i=0}^{n} (x_i \cdot c_{i,e}) \geq 1 \tag{2.8}$$

donde se define:

- $c_{i,e} = \begin{cases} 1 & \textit{si el iésimo polígono cubre la arista e.} \\ 0 & \textit{si no} \end{cases}$
- $x_i = \begin{cases} 1 & \textit{si el iésimo polígono está en la solución.} \\ 0 & \textit{si no} \end{cases}$

También se definen los siguientes elementos:
$E' = C \cup NC$ con
C= conjunto de aristas que son "cortadas o calles sin salida".
NC= conjunto de aristas no cubiertas por ningún polígono.
n= cantidad de polígonos generados.
Donde $C \subseteq NC$.

- **Límite en cubrimiento para cada arista**
  Dada la cantidad de polígonos generados en la mayoría de las pruebas realizadas, se hizo evidente el gran solapamiento entre polígonos. Se encontraron situaciones en las que una arista era cubierta por numerosos polígonos miembros de la solución hallada.
  Al mismo tiempo, la visualización de dicha solución era muy poco intuitiva, ya que se mostraban numerosos polígonos solapados, haciendo muy dificultosa su interpretación y su implementación en la realidad, ya que no es esperable que haya regiones del mapa en común a ser recorridas por los camiones recolectores.
  Debido a esta situación, se hizo necesario introducir una nueva restricción que limitara la cantidad de polígonos solapados y por ende, la cantidad de polígonos que cubren a cada arista del multigrafo. Se intenta minimizar el solapamiento.
  A continuación la restricción agregada al modelo:

$$\forall j \in E, (p-1) \cdot y_j \geq (\sum_{i \in P_j} x_i) - 1 \tag{2.9}$$

donde:

- p es la cantidad de superposiciones permitidas (parametrizable).
- P_j es el conjunto de polígonos generados que cubren a la arista j.
- $\sum_{i \in P_j} x_i$ es la cantidad de polígonos de la solución que cubren a la arista j.
- x_i como se mencionó anteriormente indica con 1 si el iésimo polígono está en la solución y 0 si no.
- $y_j = \begin{cases} 1 & \textit{si la arista j está cubierta por más de un polígono} \\ 0 & \textit{si no} \end{cases}$

2.5.14. Función objetivo

Como se explicó en secciones previas, el objetivo de esta tesis es obtener una zonificación del mapa de una determinada ciudad para dar una solución factible y práctica al problema de la recolección de residuos. Es sabido que se pueden encontrar numerosas soluciones a dicho problema, sin embargo es necesario imponer ciertas restricciones para optimizarlas. La principal de ellas es encontrar una solución que minimice la cantidad de polígonos que forman parte de ella, minimizando con ello la cantidad de recursos destinados a cumplir con la recolección (por ejemplo, cantidad de camiones y personal que presten el servicio, desgaste de los camiones, costos de mantenimiento y operabilidad, etc.). Se desea minimizar la cantidad de particiones al mapa, optimizando de esta manera la cantidad de camiones requeridos para la recolección de residuos respetando las restriciones previamente mencionadas. En simultáneo, se prioriza la minimización de solapamiento de zonas

particionadas, evitando así tener regiones cubiertas por muchos polígonos de la solución. Finalmente la función objetivo del modelo es la siguiente:

$$min \sum_{i=0}^{n} x_i + \alpha \cdot \sum_{j \in E} y_j \tag{2.10}$$

donde α representa una constante muy chica (0.001) para priorizar la minimización de la cantidad de polígonos por sobre la limitación en el overlap de polígonos.

2.5.15. Resolución del modelo

En este trabajo el modelo de **Programación Lineal Entera** fue implementado mediante la clase *System Solver*.
Dicha clase utiliza la interfaz con el solver SCIP provista en https://github.com/SCIP-Interfaces/JSCIPOpt para construir un modelo de programación lineal entera el cual resolverá, en el caso de ser factible, el problema a tratar.
Para ello, se le deberá brindar a SCIP, mediante su interfaz, una función objetivo, la cual priorizará la minimización en la cantidad de polígonos en la solución siempre y cuando se satisfagan ciertas restricciones impuestas al modelo, tal como se mencionó previamente.
Cada restricción será agregada como productos escalares (restricciones lineales) entre vectores definidos con variables binarias, representando por un lado la pertenencia de polígonos a la solución y por el otro la cobertura para cada arista por parte de cada uno de los polígonos generados.
Tanto la función objetivo, como las restricciones planteadas al modelo, deben respetar la sintaxis establecida por la interfaz con SCIP. Cabe destacar la facilidad y simplicidad en la utilización de la interfaz, proveyendo métodos intuitivos y más claros en lugar de tener que realizar llamadas de instrucciones de bajo nivel agregando código más extenso y poco claro en la implementación.

2.5.16. Extracción de polígonos

Una vez obtenida una solución factible para el modelo de programación lineal planteado, es necesario extraer los polígonos integrantes de la misma y procesarlos para su presentación. Inicialmente, los polígonos eran agregados en un conjunto para luego ser visualizados.
El problema encontrado al realizar esto fue que si bien el solver encontraba una solución factible al modelo, la misma presentaba numerosos solapamientos entre los polígonos, lo cual no era aplicable a la práctica.
Debido a esto, se hizo necesario el planteo de una solución alternativa que evite dichos solapamientos. Para ello, se pensó en un algoritmo goloso basado en la inserción individual de polígonos a la solución, comparando y recortando el área de los mismos para de esa forma quitar la superficie intersecada con las superficies de otros polígonos ya pertenecientes a la solución.
Se planteó que previo al comienzo de realizar las inserciones mencionadas, se ordenasen los polígonos de acuerdo a la superficie de sus áreas en forma descendente para de esta forma, intentar insertar a la solución polígonos que abarquen una mayor región del mapa,

evitando solapamientos iniciales. La idea es que los solapamientos se produzcan al insertar polígonos de menor superficie, minimizando su procesamiento y adaptación posterior al agregárselos al conjunto solución.

2.5.17. Algoritmo greedy de inserción de polígonos a la solución

Una vez ordenada la lista de polígonos (por tamaño de área), el algoritmo greedy consiste en comparar cada polígono que se inserta al conjunto resultado final.
En cada inserción se compara el polígono a agregar contra todos los polígonos ya pertenecientes a la solución. En el caso de solaparse con algún polígono, se "recorta" el área solapada y se analizan los diversos casos posibles.

- Si el área una vez recortada queda vacía, significa entonces que ese polígono queda totalmente solapado por la solución, por ende su área ya es cubierta por la misma, por lo tanto es descartado. Si bien esta situación no se presentó en las soluciones encontradas por el solver, se decidió tener igualmente contemplado el caso.
- Otra posibilidad ocurre cuando el área del polígono no sufre modificación alguna, es decir, el polígono no se solapa con la solución, caso en el que es agregado en su totalidad a la misma.
- Por último se pueden mencionar los casos en los que el área del polígono es modificada pero una parte de ella no se solapa con ningún otro, en dichos casos el algoritmo debe proceder a recortar el área dejando solamente la porción no solapada. Para ello utiliza los métodos provistos por la clase **Area** de Java *subtract* e *intersect*. Se contemplan los casos en los que el área obtenida del recorte resulta singular o no. En el primero de los casos, se considerará a la misma como una única zona a ser agregada. Para el segundo caso, se crea y agrega a la solución un nuevo polígono por cada área en la que fue separado el original, mediante los subpaths almacenados.

A continuación, el algoritmo en cuestión:

```
1: procedure ALGORITMOGREEDYDEINSERCION(orderedPolygonList, polygonsInSolution)
2:     for p ∈ orderedPolygonList do
3:         compWithSolModifIfNecAndAdd(pPolygon, polygonsInSolution)
4:     end for
5: end procedure
```

Donde se define:

```
1: procedure COMPWITHSOLMODIFIFNECANDADD(aPol, polsInSolution)
2:     initPolArea ← aPolygon.getArea()
3:     for p ∈ polygonsInSolution do
4:         checkOverlapAndCutPolygon(aPolygon, p)
5:     end for
6:     areaModified ← (initPolArea! = aPolygon.getArea())
7:     if areaModified then
8:         if !aPolygon.getArea().isEmpty then
9:             modifyPolygonPoints(aPolygon)
10:            processingPolygonAndAdd(aPolygon, polsInSolution)
11:        end if
12:    else
13:        polygonsInSolution.add(aPolygon)
14:    end if
15: end procedure
```

```
1: procedure CHECKOVERLAPANDCUTPOLYGON(pol1,pol2)
2:     if intersect(pol1,pol2) then
3:         pol1.getArea().substract(pol2.getArea())
4:     end if
5: end procedure
```

```
1: procedure ProcessingPolygonAndAdd(aPolygon, polygonsInSolution)
2:     if thePolygonWasModified then
3:         subpathX ← aPolygon.getSubpathX
4:         subpathY ← aPolygon.getSubpathY
5:         for s ∈ subpathX do
6:             area ← CalculateArea(subpathX.get(s), subpathY.get(s))
7:             polPoints ← Pair(subpathX.get(s), subpathY.get(s))
8:             newPol ← CreatePolygon(newId, polPoints, area)
9:             polygonsInSolution.add(newPol)
10:        end for
11:    else
12:        xPoints ← aPolygon.getXpoints()
13:        yPoints ← aPolygon.getYpoints()
14:        area ← CalculateArea(xPoints, yPoints)
15:        polPoints ← Pair(xPoints, yPoints)
16:        newPol ← CreatePolygon(newId, polPoints, area)
17:        polygonsInSolution.add(newPol)
18:    end if
19: end procedure
```

2.5.18. Problema encontrado

Cabe mencionar un inconveniente encontrado al realizar diversas pruebas con diversos mapas de ciudades.
Dicho problema consistía en que el algoritmo no contemplaba aquellas situaciones en las que al recortarse el área de un polígono, ésta resultaba dividida en dos o más polígonos. Para el algoritmo, es un único polígono con lo cual la falla se hizo notoria al momento de la visualización de la solución.
Básicamente se mostraba en la zonificación del mapa en la que existían polígonos mal formados, producto de una mala representación, de contornos irregulares, solapandose porciones con otros polígonos de la solución.
Haciendo foco en esta situación, se descubrió que dicha representación fallida se debía a que el visualizador de polígonos consideraba al polígono recortado como un polígono homogéneo, recorriendo los nodos de su contorno en orden secuencial realizando trazos sobre el mapa. Al producirse un salto entre los nodos, ya que el polígono se dividía, para el visualizador era simplemente un trazo más entre nodos mal considerados consecutivos, provocando así trazas erróneas en la visualización de JMapViewer.
Encontrada e identificada esta situación problemática, se pensó en la siguiente solución intuitiva:
La clase Area de Java proporciona el método **getPathIterator**. Dicho método devuelve, como lo indica su nombre, un iterador sobre el contorno del área en cuestión. Lo que era necesario entonces era poder identificar los diversos tipos de segmentos, identificando aquellos casos en los que se cerraba el contorno, formando así uno de los polígonos en los que era dividido el polígono original. Otra modificación que se hizo necesaria fue la de agregar en la clase MapPolygon la cantidad de subpaths de nodos integrantes del área del mismo. Es decir, en el caso de ocurrir la situación con la problemática mencionada, el

algoritmo formará los diversos subpaths descubiertos al iterar sobre el contorno del área. Por último seteará dichos subpaths a los correspondientes atributos de la instancia.
Cabe mencionar que fue necesario considerar dos tipos de subpaths en cada instancia de MapPolygon, uno por cada coordenada del plano (x e y).
De esta manera por ejemplo si se tuviera un polígono con el siguiente único subpath fomado por los puntos en $\mathbb{R}^2$,

$[(x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4)]$

en la instancia de MapPolygon se tendrán los siguientes atributos:

$subpathsX = [x_1, x_2, x_3, x_4]$
$subpathsY = [y_1, y_2, y_3, y_4]$

Se aclara que subpathsX y subpathsY siempre tendrán las mismas longitudes ya que al momento de agregarse un nuevo subpath, el iterator devuelve por definición cada punto del plano dividido en sus coordenadas x e y, las cuales son agregadas a las respectivas listas.

Al momento de la visualización de polígonos de la solución, el Viewer simplemente recorrerá el conjunto de polígonos pertenecientes a la solución y los agregará pasando los subpaths de puntos (en ambas coordenadas) al mapViewer.

2.5.19. Posprocesamiento de la solución

Una vez obtenida la solución al problema, en la que se eliminaron los solapamientos entre los polígonos que la integran, se notó, como en la imagen a continuación, que el recorte de los mismos dejaba pequeños polígonos residuo (Figura 2.12).

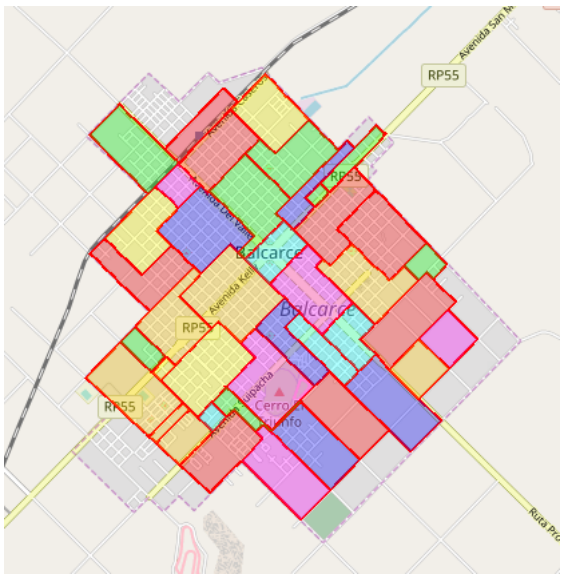

Fig. 2.12: Zonificación de Balcarce sin procesamiento de la solución

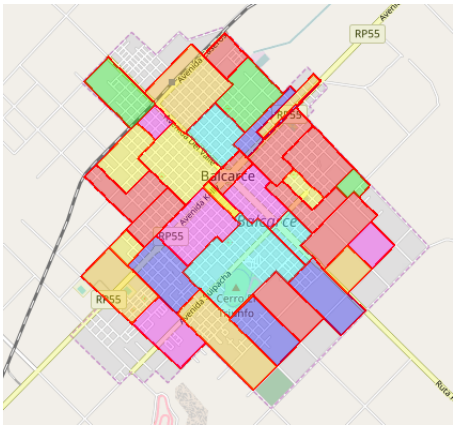

Fig. 2.13: Zonificación de Balcarce con procesamiento de solución

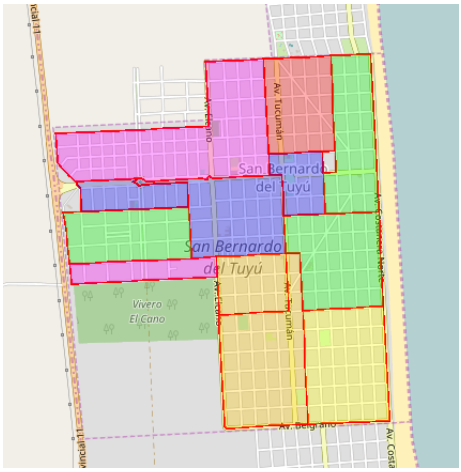

Fig. 2.14: Zonificación obtenida para la ciudad de San Bernardo del Tuyú tras la ejecución del algoritmo

Estos polígonos en la práctica no aportan mucho ya que su superficie es muy pequeña. Por este motivo se hizo necesario un posprocesamiento de la solución, que se encargue de unir dichos polígonos pequeños a poligonos vecinos, de manera tal de obtener una zonificación más razonable y pareja en términos de tamaños de zonas.
Básicamente el algoritmo se encarga de verificar que el tamaño de área de cada polígono perteneciente a la solución esté por encima de un umbral mínimo definido y calculado previamente.
Se estableció para este valor mínimo la mitad del promedio de tamaño de área de la solución, de manera tal que se garantice que luego del procesamiento, las áreas de los polígonos de la solución tendrán un tamaño igual o superior al mismo. Al mismo tiempo y como se mencionó, para aquellos polígonos cuyo tamaño de área sea inferior al umbral, el algoritmo se encarga de buscar un polígono vecino que cumpla ciertas restricciones para poder unir sus áreas. Entre estas restricciones se puede mencionar que el vecino sea aquel que posea el menor tamaño de áerea y que pueda seguir expandiendose, ya que tampoco se desea obtener polígonos de tamaño demasiado grande.
Al añadir este procesamiento por un lado se logra obtener mejores soluciones a las encon-

tradas por el algoritmo greedy, se automatiza la zonificación del mapa, pero por el otro lado se añade una mayor complejidad temporal. A continuación se presenta el algoritmo en cuestión:

```
1: procedure procesarYMergeDePolsChicos(polygonsInSol)
2:     areaOp ← new AreaOperator()
3:     avg ← areaOp.calculateAreaSizeAverage
4:     cantExpand ← new HashSet
5:     while not(allPolygonsInAcceptableSize) ∧ canExpandPolygons do
6:         checkAndMergePolygons(polygonsInSol,areaOp,avg,cantExpand)
7:     end while
8: end procedure
```

```
1: procedure checkAndMergePolygons(polsInSol,areaOp,avg,cantExp)
2:     for p ∈ polsInSol do
3:         if canExpand(p) then
4:             areaSize ← areaOp.getAreaSize(p)
5:             if areaSize ≤ 0.5·avg then
6:                 minAreaSizeNeighbor ← searchMinAreaSizeNeighbor
7:                 if minAreaSizeNeighbor ≠ NULL then
8:                     mergePolsAreaAndUpdatePoints(minAreaSizeNeighbor,p)
9:                     remove(p)
10:                else
11:                    cantExpand.add(p)
12:                end if
13:            else
14:                cantExpand.add(p)
15:            end if
16:        end if
17:    end for
18: end procedure
```

```
1: procedure mergePolsAreaAndUpdatePoints(minAreaSizeNeighbor,pol)
2:     minAreaSizeNeighbor.getPolArea().add(pol.getPolArea())
3:     modifyPolygonsPoints(minAreaSizeNeighbor)
4: end procedure
```

2.5.20. Visualización de polígonos

Obtenidos y procesados los polígonos que integran la solución del problema, se debe proceder a mostrar dichos resultados en forma gráfica, intuitiva.
Para cumplir con tal fin, en este trabajo se decidió utilizar el componente de Java **JMapViewer**, el cual permite una fácil integración en la visualización de un mapa en formato OSM dentro de una aplicación Java.
Permite incluír en la aplicación controles de zoom, desplazamiento sobre el mapa, insertar marcadores en coordenadas geográficas, etc (ver Figura 2.16).

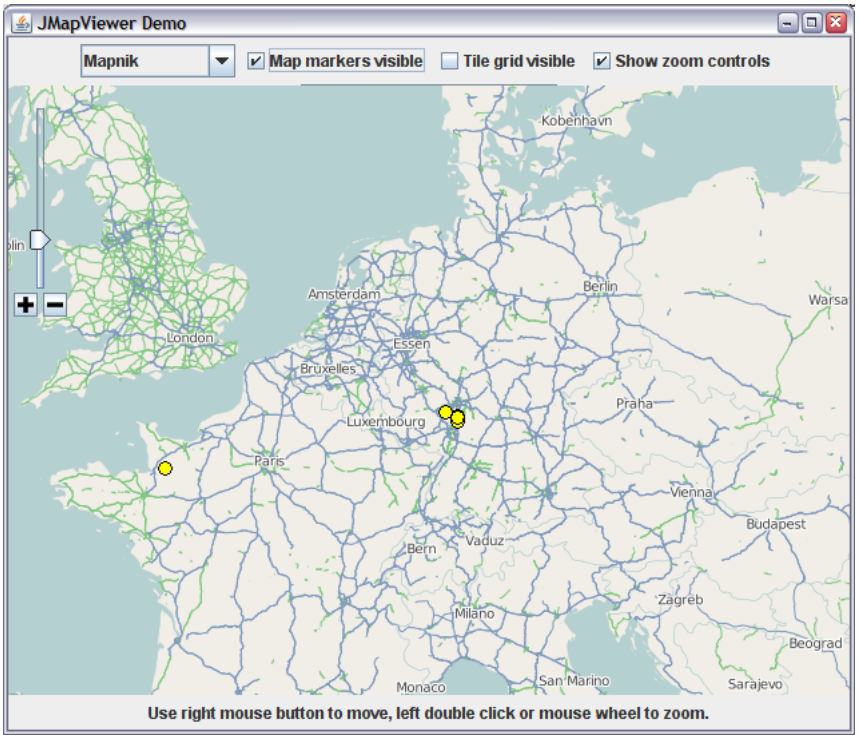

Fig. 2.15: Ejemplo de visualización con JMapViewer

Con respecto a la implementación, en el trabajo se creó una clase **Viewer** encargada del proceso de visualización de polígonos sobre un mapa. Dicha clase utiliza la librería JMapViewer en el proceso.
Cuando se crea una instancia de la clase, ésta es inicializada con una lista de coordenadas como parámetro.
Esta lista contiene los pares ordenados (Latitud,Longitud) correspondiente a las coordenadas geográficas de cada uno de los vértices integrantes de cada polígono.
Es decir, el algoritmo encargado de la visualización de la solución, se encarga de recorrer para cada polígono de la misma, la lista de puntos que conforman su contorno. Al hacerlo, crea instancias de la clase **Coordinate** de Java, seteandoles los valores correspondientes a las latitudes y longitudes de cada vértice integrante del polígono.
Cabe aclarar que dichos puntos se encuentran almacenados en los atributos **xPoints** e **yPoints** de la clase MapPolygon. Los primeros guardarán coordenadas correspondientes a la posición sobre la dimensión x (horizontal) en el plano, los segundos sobre la dimensión y (vertical).

Utilizando la clase **CoordinatesConversor**, se convierten dichas coordenadas del plano a coordenadas geográficas, para ser pasadas al Viewer. El método encargado de llamar a estas operaciones es **operateWithPolygons** de la clase PolygonsOperator.

Con respecto a la conversión de coordenadas en el plano a coordenadas geográficas, se utiliza la clase auxiliar **CoordinatesConversor**. Esta clase se encarga de los mapeos entre puntos del plano y puntos geográficos y viceversa. En el caso de la visualización de los polígonos solución en el mapa, utiliza los métodos de conversión **convertPointToLatitude** y **convertPointToLongitude**. Por otro lado, define métodos de conversión a la inversa. Dichos métodos emplean las siguientes fórmulas de conversión:

- **Conversión de geocoordenadas a puntos del plano**

  1. **Latitud a $\mathbb{R}^2$**

$$\log\left(\tan\left(\frac{\pi}{4}+\frac{latitud}{2}\right)\right)\cdot rad \tag{2.11}$$

  2. **Longitud a $\mathbb{R}^2$**

$$longitud\cdot rad \tag{2.12}$$

Donde la variable *rad* almacena el radio de la Tierra en metros (aproximadamente 6378137 metros).
Como se indicó anteriormente, la clase también se encarga de la conversión de coordenadas en el plano a coordenadas geográficas. Los métodos encargados de la conversión emplean para tal fin, las siguientes ecuaciones de conversión:

- **Conversión de puntos del plano a geocoordenadas**

  1. **De $\mathbb{R}^2$ a latitud**

$$\arctan\left(e^{\frac{yPoint}{rad}}\right)\cdot 2-\frac{\pi}{2} \tag{2.13}$$

  2. **De $\mathbb{R}^2$ a longitud**

$$\frac{xPoint}{rad} \tag{2.14}$$

3. CAPÍTULO III: EXPERIMENTOS Y RESULTADOS

Para decidir el algoritmo definitivo que resolviera de la mejor forma el problema de zonificación sobre el mapa de una ciudad determinado fue necesario estudiar las respuestas de diversas heurísticas con sus variantes, parámetros y criterios para diferentes tamaños de mapa y de zona.
Primero se estudiaron las características de las diversas regiones a zonificar, entre ellas, su tamaño, el tipo de dibujo de la misma, etc. Posteriormente se hizo un análisis riguroso de los resultados de aplicar las heurísticas; y por último se eligió la mejor y con ella se resolvió el zonificado.

3.1. Características de las instancias

- **Dimensión de las zonas**
  Analizando los distintos mapas de diversas ciudades de la Argentina, los tiempos de procesamiento del algoritmo zonificador varían considerablemente de acuerdo a las características de los maps en cuestión. Dichos tiempos van desde los pocos minutos (tres o cuatro para ciudades de tamaño chico) a varias horas en ciudades de importancia, como por ejemplo Rosario y Junín.
  Debido a esto, como se mencionó en el capítulo anterior, el algoritmo subdivide el mapa, minimizando costos en el procesamiento. Cuanto mayor es el tamaño de la ciudad, mayor es la cantidad de subdivisiones de la misma. Dicha subdivisión está relacionada con los factores de altura y ancho seteados para el tamaño de cada cuadrante.
  En este trabajo se probó el algoritmo con distintos mapas de ciudades, muy diversos en cuanto a su tamaño, distribución, dibujos y demás características. Con el correr de las pruebas, se fueron notando los distintos resultados obtenidos al setearse distintos parámetros vinculados al algoritmo.
  A continuación se mencionan cómo estos cambios fueron influyendo tanto en los resultados obtenidos (características de la solución) como en los tiempos de procesamiento y utilización de memoria.
  Por ejemplo al correr el programa para el mapa de la ciudad de Junín, el tiempo total de procesamiento hasta obtener una solución del problema era superior a una hora y media de reloj. Como fue mencionado en el capítulo anterior, las pruebas iniciales comenzaron a realizarse con mapas de ciudades con poco procesamiento previo, agregando restricciones al modelo de programación lineal entera en las cuales se compara si cada arista del grafo es contenida en alguno de todos los polígonos generados. Estas comparaciones agregan un elevado costo computacional al algoritmo, haciéndolo muy poco práctico en la resolución de la problemática.

- **Factores de subdivisión del mapa**
  Después de haber realizado numerosas pruebas con valores diversos para factores de subdivisión, se llegó a la conclusión de que el valor 1.7 km tanto para ancho y alto es bastante aceptable en cuanto al tiempo consumido por el algoritmo en obtener la solución al problema y también en cuanto al tamaño de región (cuadrante) a

considerar al momento de realizar comparaciones durante el armado de restricciones para el solver. La calidad de las soluciones (formas de polígonos) varía de acuerdo al seteo de dichos factores, debido a cómo el algoritmo arbitrariament ubica cada uno de los polígonos a su respectivo cuadrante. Al modificar el tamaño de los cuadrantes cambian las asignaciones para algunos de ellos.
Sumado a esto, la cantidad de polígonos pertenecientes en las soluciones obtenidas es menor o igual a las de los casos en que ambos factores superan los 2 km (casos 6 y 7).

A continuación se mostrará una serie de experimentos realizados en los cuales varían los valores de los factores. En cada caso se muestra la solución obtenida junto al tiempo consumido por el algoritmo en obtenerla. Es muy importante aclarar que dichas pruebas fueron realizadas sin la ejecución del algoritmo de posprocesamiento descripto anteriormente. En las siguientes secciones también se darán a conocer los resultados aplicando dicha modificación

Caso 1
Altura: 1 km Ancho: 1 km

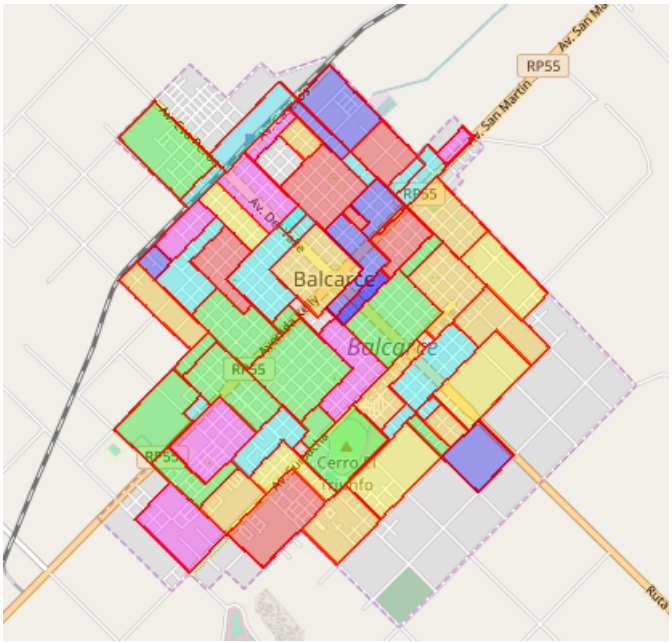

Fig. 3.1: Solución encontrada para la ciudad de Balcarce. Caso 1

Caso 2
Altura: 1 km Ancho: 1.5 km

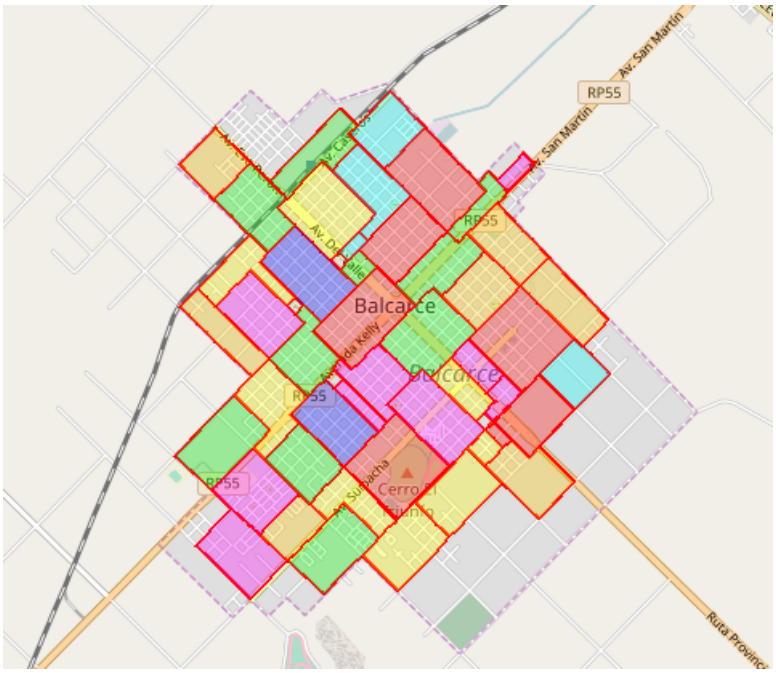

Fig. 3.2: Solución encontrada para la ciudad de Balcarce. Caso 2

Caso 3
Altura: 1.5 km Ancho: 1 km

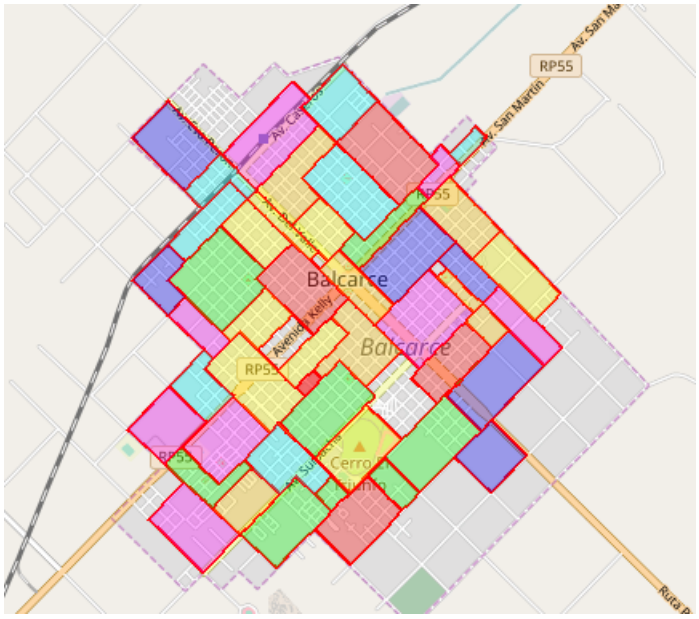

Fig. 3.3: Solución encontrada para la ciudad de Balcarce. Caso 3

Caso 4
Altura: 1.5 km Ancho: 1.5 km

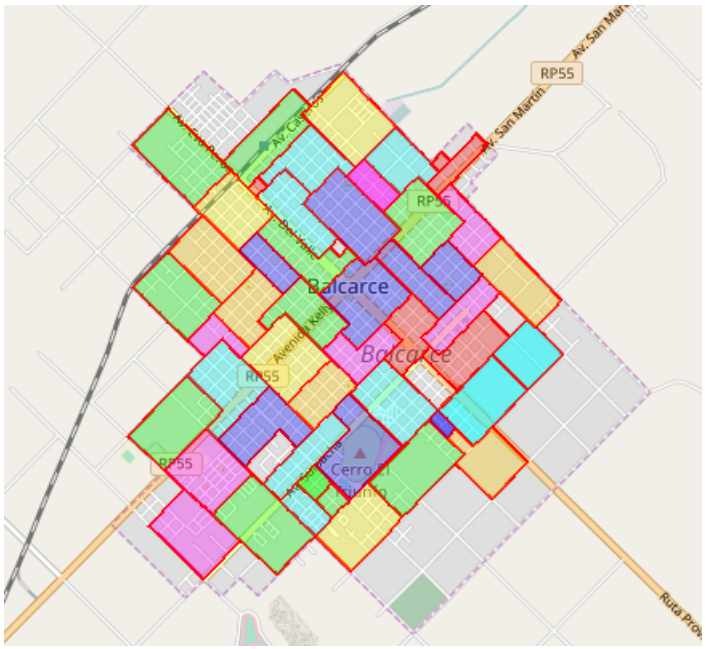

Fig. 3.4: Solución encontrada para la ciudad de Balcarce. Caso 4

Caso 5
Altura: 1.7 km Ancho: 1.7 km

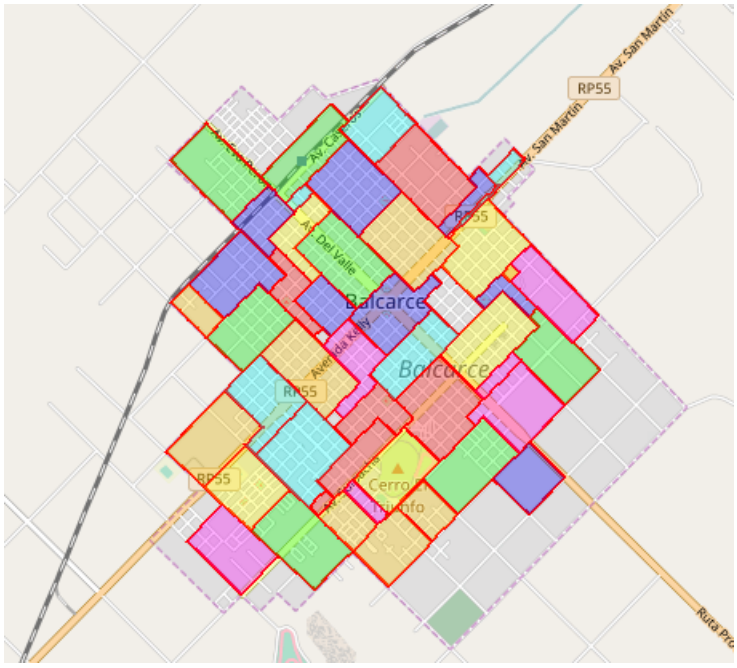

Fig. 3.5: Solución encontrada para la ciudad de Balcarce. Caso 5

Caso 6
Altura: 2 km Ancho: 2 km

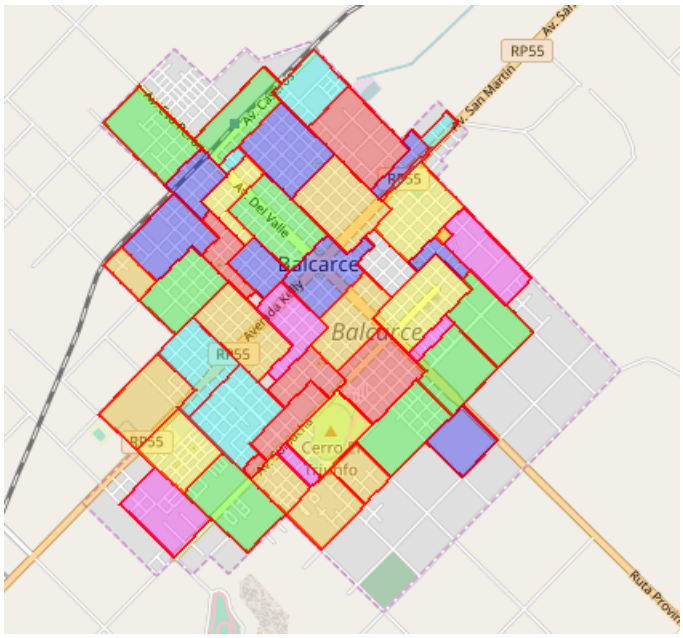

Fig. 3.6: Solución encontrada para la ciudad de Balcarce. Caso 6

Caso 7
Altura: 3 km Ancho: 3 km

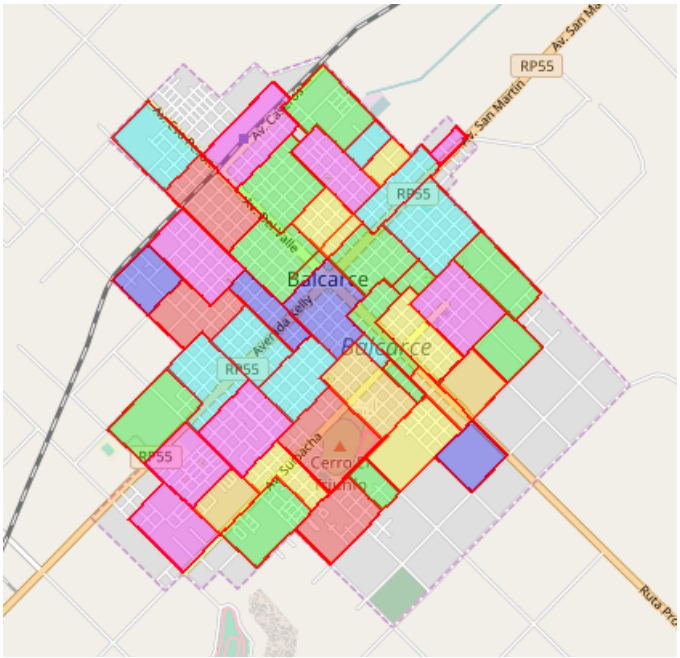

Fig. 3.7: Solución encontrada para la ciudad de Balcarce. Caso 7

Caso	Alto(km)	Ancho(km)	#Pols total	#Pols en sol.	Tiempo(min)
1	1	1	40064	64	05:42
2	1	1.5	40064	52	05:01
3	1.5	1	40064	54	05:25
4	1.5	1.5	40064	51	07:16
5	1.7	1.7	40064	48	11:52
6	2	2	40064	48	11:02
7	3	3	40064	48	14:39

Tab. 3.1: Resultados obtenidos sobre la ciudad de Balcarce.

- **Análisis de resultados** Tras ejecutar los casos mencionados, se encontró que, como se aprecia en la tabla, a medida que se incrementan las dimensiones de los cuadrantes (incrementando ambos factores), los tiempos de procesamiento también se incrementan.
  Por otro lado, puede verse que a partir de una dimensión determinada, la solución en cuanto a la cantidad de polígonos no varía si se incrementa la misma. Por ejemplo, en el caso del mapa de Balcarce, el caso 7 que setea una dimensión de 3 km en ancho y alto, obtiene una solución de similar calidad a la del caso 5 (misma cantidad de polígonos). La diferencia es que para el caso 5, el tiempo de procesamiento es inferior.
  Con respecto a la "calidad" de zonas obtenidas, puede verse en los distintos casos, que todas las zonas son de contorno "sencillo", uno de los objetivos principales de este trabajo. No hay demasiada variación en cuanto a los tamaños de las mismas en superficie y es correcto que así sea ya que, como se explicó en el capítulo anterior, dichos tamaños están relacionados con el proceso heurístico en la selección de nodos durante la generación de polígonos.

3.2. Heurísticas utilizadas en la generación de polígonos

Como se mencionó en el Capítulo 2, el algoritmo generador de polígonos se encarga de seleccionar en cada nueva iteración un par de nodos a partir del cual comienza a formar un posible nuevo polígono.
Precisamente es este proceso de selección de nodos el encargado de determinar el tamaño que tendrán los polígonos a generar ya que como pudo verse, establece un rango de distancia euclidea para dicho par de nodos.
Durante este trabajo se realizaron distintas pruebas en poder establecer dicho rango, que permitiera la generación de polígonos con un tamaño aceptable y a la vez, obtener una solución al problema de calidad, es decir, con la mínima cantidad y con formas que resulten intuitivas en la práctica.
Se pudo ver que establecer este rango es fundamental, ya que influye no sólo en la calidad de las soluciones sino también en los tiempos de procesamiento.
Uno esperaría, a priori, que a mayor distancia entre nodos iniciales, el algoritmo genere polígonos de un mayor tamaño, por consiguiente el tiempo consumido en esta tarea se incrementará debido a que la cantidad de nodos intermedios que deberá verificar en cada una de las direcciones en las que avance será mayor hasta poder

encontrar las posibles intersecciones necesarias en la formación de polígonos.
Se realizaron distintas pruebas para analizar esto último. Se probó el algoritmo para diversas ciudades, en particular Junín, se modificaron los valores en el rango de distancias en la selección de nodos, y se chequearon los tiempos totales consumidos junto a la calidad de las soluciones obtenidas, tanto en cantidad de polígonos pertenecientes a cada una de ellas, como así también a las formas de los mismos, que tan intuitivos resultan en la práctica. Se pudo observar a simple vista, que al reducir la distancia en la selección de nodos iniciales el tamaño de polígonos era mucho menor y al mismo tiempo, se particionaba el mapa en una mayor cantidad de zonas. Es decir, por un lado se optimizan los tiempos de ejecución pero la calidad de las soluciones obtenidas no es del todo la esperable (muchos polígonos y de tamaño muy chico, poco práctico). Por otro lado, al aumentarse la distancia en la selección de iniciales, la cantidad de polígonos se reduce, su tamaño es más esperable sin embargo empeora la performance del algoritmo, los tiempos de ejecución se incrementan considerablemente.
Se realizaron numerosas pruebas basadas, como se mencionó, en diferentes seteos de tamaños hasta lograr obtener una configuración *aceptable*, es decir, resultados rápidos y de calidad.

3.3. Armado del modelo de Programación Lineal Entera

Durante el proceso de planteo y armado del modelo de programación lineal entera (PLE) a ser resuelto por el solver SCIP, se deben agregar las diversas restricciones a considerar. Una de ellas es la que plantea en este trabajo la cobertura por parte de los polígonos integrantes de la solución de todas, o casi todas, las aristas del grafo de la ciudad.
En esta tarea, como se mostró en el Capítulo 2, el algoritmo realiza para cada arista del grafo, una comparación contra todos los polígonos generados en el mapa.
En este punto se puede encontrar un vínculo con la subdivisión del mapa de la ciudad en cuadrantes, más precisamente con la cantidad y tamaño de los mismos, establecidos como se mencionó anteriormente por los factores de ancho y alto.
Al realizar diversas pruebas, se pudo notar claramente que a mayor tamaño de cuadrantes, la superficie del mapa abarcada por los mismos contendrá una mayor cantidad de polígonos generados, con lo cual cada arista se deberá comparar contra un conjunto mayor de polígonos, ocasionando incremento en el tiempo de ejecución del algoritmo al momento de agregar las restricciones al modelo del solver.
Por el contrario, al tener cuadrantes que abarquen una menor superficie, se realizarán menos comparaciones de cobertura para cada arista.
Sumado a esto, como también pudo mostrarse en los resultados obtenidos al realizar experimentos con diversos factores en el tamaño de los cuadrantes del mapa, a mayor dimensión de los mismos la calidad de las soluciones no muestra cambios significativos, es más, en varios de ellos la cantidad de polígonos integrantes de la solución no varía. Por el contrario sí lo hacen los tiempos de procesamiento, incrementándose significativamente.

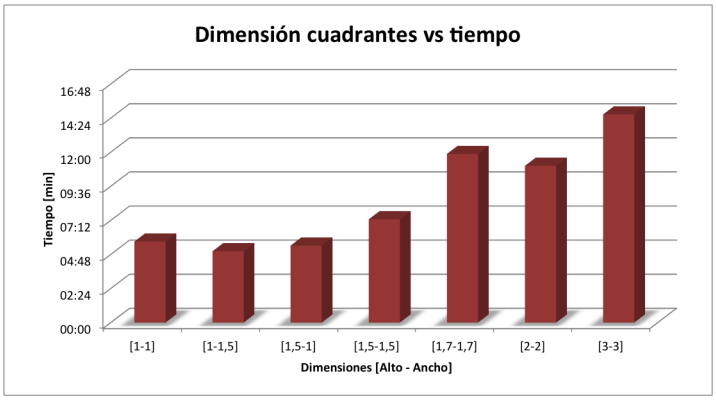

Fig. 3.8: Gráfico que muestra la relación entre las dimensiones de los cuadrantes de subdivisión del mapa y los tiempos de procesamiento.

3.4. Solapamiento de polígonos en la solución

El solver al resolver el modelo encuentra soluciones que se caracterizan por tener muchos polígonos solapados, es decir, muchas intersecciones entre ellos.
Si bien la superficie del mapa de la ciudad determinada queda cubierta por la solución no es esperable tener un solapamiento tan grande. Claramente se hace necesario resolver este problema, y tener soluciones que zonifiquen el mapa al mismo tiempo que cubran la mayor parte de su superficie y en la que no haya solapamiento entre ninguno de los polígonos obtenidos tal como fue descripto en el capítulo previo (Algoritmo greedy de inserción).
Como se mostró, se puede deducir que al tener mucho solapamiento de polígonos en la solución, el costo del algoritmo será mayor, ya que se deberá procesar a dicha solución para poder presentarla en forma adecuada.
Luego de realizar diversas pruebas, se fue comprobando que el ordenamiento previo por tamaño de áreas (de mayor a menor) no provocaba mejoras significativas en cuanto a tiempos de procesamiento pero sí en cuanto a la calidad de soluciones obtenidas.
Más precisamente, el caso en el que los polígonos no eran ordenados de acuerdo a su tamaño de área, los tiempos eran relativamente más bajos que cuando existía un ordenamiento previo. Sin embargo en el trabajo se decidió optar por la solución que ordena a los polígonos ya que en la zonificación gráfica, la solución obtenida era de una mayor calidad.

Esto último puede apreciarse de una mejor forma en la Tabla 3.3:

Ciudad	Orden previo	Rango(km)	#Polígonos	Tiempo (min)
Balcarce	NO	0.8-1.1	44	16:11
Balcarce	SI	0.8-1.1	46	17:20
Balcarce	NO	0.9-1.1	44	10:45
Balcarce	SI	0.9-1.1	48	10:52
Junín	NO	0.8-1.1	120	48:53
Junín	SI	0.8-1.1	129	44:02
Junín	NO	0.9-1.1	118	20:47
Junín	SI	0.9-1.1	118	21:59

Tab. 3.2: Resultados obtenidos al ejecutar pruebas con orden y sin orden de polígonos previo

3.5. Heurística de merge de polígonos en la solución

Como se explicó en la sección previa, se implementó un algoritmo (opcional) encargado de procesar las soluciones obtenidas.
Dependiendo de la ciudad a procesar, los factores en el rango determinado en la selección de nodos durante la generación de polígonos, el ordenamiento previo de los mismos por tamaño de área y la cantidad de solapamiento en la solución obtenida determinarán e influirán notoriamente en los tiempos de posprocesamiento.
Es decir, dichos factores determinarán si las soluciones obtenidas contendrán muchos o pocos polígonos recortados de superficie muy pequeña, los cuales necesitarán ser combinados a polígonos vecinos de mayor tamaño, dejando una solución con polígonos de tamaño mayor o igual a un umbral establecido como fue mencionado previamente.
Luego de realizar numerosas pruebas, se encontraron diversos problemas derivados de los diferentes casos a considerar, se logró obtener un algoritmo de merge que mejora el tamaño de polígonos y devuelve una solución de mayor calidad.
A modo de comparación, a continuación se visualizan dos figuras pertenecientes al mapa de la ciudad de San Bernardo del Tuyú, provincia de Buenos Aires. En una de ellas, no fue aplicado el algortimo de merge, en cambio en la otra sí. Se puede ver a simple vista, la diferencia en la calidad de soluciones obtenidas en ambos casos.

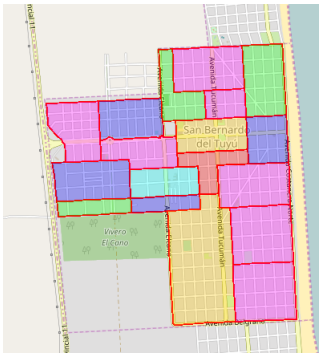

Fig. 3.9: Zonificación de San Bernardo del Tuyú sin ejecución del algoritmo de procesamiento de solución

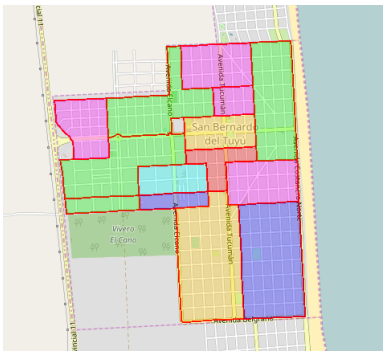

Fig. 3.10: Zonificación de San Bernardo del Tuyú tras la ejecución del algoritmo de procesamiento de solución

3.6. Experimentos con heurísticas definidas

Se realizaron experimentos con las heurísticas definidas previamente (algoritmo greedy de inserción y heurística de merge de polígonos), se analizaron y compararon los resultados obtenidos.
A continuación se muestra una tabla con los resultados obtenidos tras correr la aplicación analizando la heurística de generación de polígonos para distintos casos particulares sobre el mapa de la ciudad de Junín.

Caso	Rango (km)	Dif (km)	#Polígonos	Tiempo (min)
1	0,8 - 1	0,2	132	22:49
2	0,5 - 0,8	0,3	167	36:50
3	0,7 - 1	0,3	127	42:12
4	0,9 - 1	0,1	100	09:52
5	1 - 1,2	0,2	86	24:04
6	1,2 - 1,3	0,1	80	08:38
7	1,4 - 1,5	0,1	51	11:02

Tab. 3.3: Resultados obtenidos al ejecutar los casos para el mapa de la ciudad de Junín.

Como puede verse de los resultados de la tabla 3.3, cuanto menor sea la distancia en el rango, es decir la distancia (en km) entre los valores mínimo y máximo establecidos, menor es el tiempo de procesamiento al obtener la solución al problema. Otro aspecto importante a destacar es la mejora en la calidad de soluciones (menor cantidad de polígonos) para rangos superiores a 1 km. Esto es relativo, y en la práctica resultó ventajoso al experimentar con mapas de ciudades de superficie importante, ejemplo Junín, Olavarría, Balcarce. Para mapas más pequeños en superficie no se notaron mejoras significativas en este aspecto.

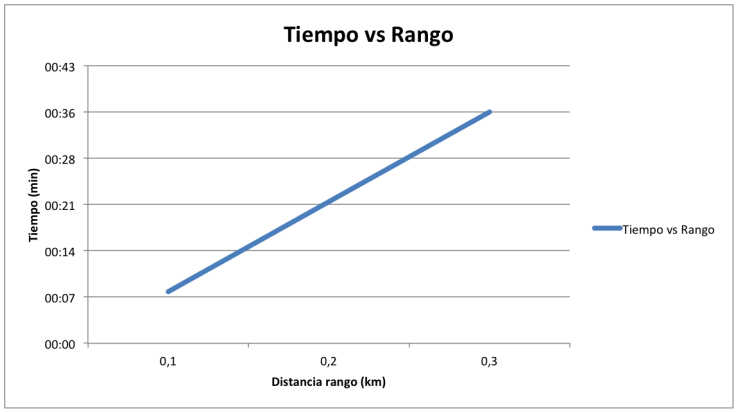

Fig. 3.11: Gráfico que muestra cómo a medida que aumenta la distancia en el rango se incrementa el tiempo de procesamiento del algoritmo para el mapa de Junín

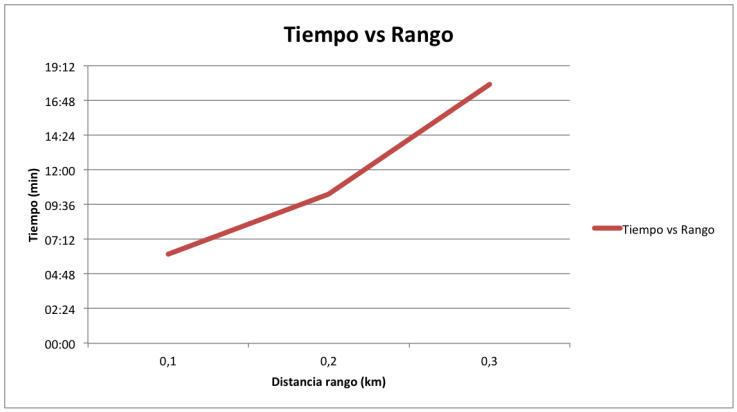

Fig. 3.12: Gráfico que muestra cómo a medida que aumenta la distancia en el rango se incrementa el tiempo de procesamiento del algoritmo para el mapa de Balcarce

Por otro lado, se realizaron las mismas pruebas exhibidas previamente (mismos parámetros, mismas ciudades) pero ahora utilizando el algoritmo heurístico de merge de polígonos en la solución.
Se obtuvieron diversos resultados (tiempos de ejecución, cantidad de polígonos en la solución), los cuales fueron plasmados en la Tabla 3.4:

Caso	Rango (km)	Dif (km)	#Polígonos	Tiempo (min)
1	0,8 - 1	0,2	37	10:19
2	0,5 - 0,8	0,3	68	15:14
3	0,7 - 1	0,3	63	17:56
4	0,9 - 1	0,1	38	06:10
5	1 - 1,2	0,2	59	10:16
6	1,2 - 1,3	0,1	35	06:00
7	1,4 - 1,5	0,1	28	05:10

Tab. 3.4: Resultados obtenidos al ejecutar los casos para el mapa de la ciudad de Balcarce.

Se puede observar que a mayor distancia entre los nodos iniciales seleccionados mayor el tiempo de procesamiento. Caso contario, a menor distancia entre ellos, menores tiempos (Figuras 3.11 y 3.12).

Otro aspecto importante a destacar de los análisis de los resultados, es la relación entre las magnitudes en dichas distancias con los tiempos de procesamiento.

- A menor magnitud en las distancias, mayores tiempos que, como vimos antes, crecen proporcionalmente con las distancias entre nodos *iniciales*.
- Por otro lado, a medida que aumentan las magnitudes disminuyen los tiempos de procesamiento que, como también se mencionó, disminuyen proporcionalmente con las distancia entre los *iniciales* durante la generación de polígonos.

Otro experimento realizado fue observar cómo a medida que aumenta la magnitud del rango en la selección de nodos iniciales la cantidad de polígonos integrantes de la solución disminuye en forma directamente proporcional. Cabe aclarar que las distancias en los rangos testeados fue siempre la misma (0,1 km), lo único que ha variado, como se ha mencionado antes, ha sido la magnitud en las mismas.

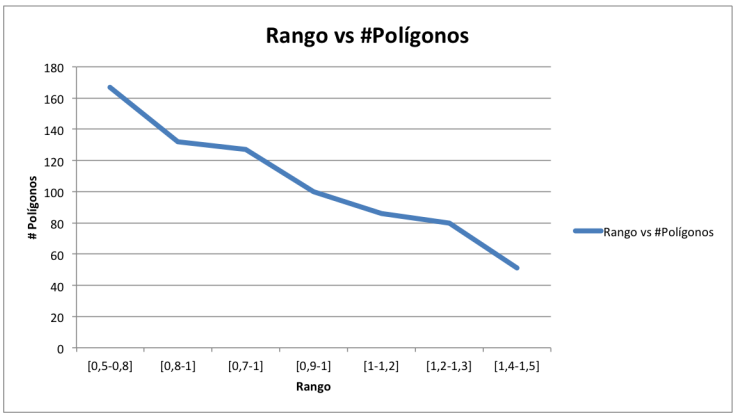

Fig. 3.13: Gráfico que muestra cómo a medida que disminuye la distancia en el rango y al mismo tiempo aumentan los valores en el mismo, disminuye la cantidad de polígonos en la solución para el mapa de Junín.

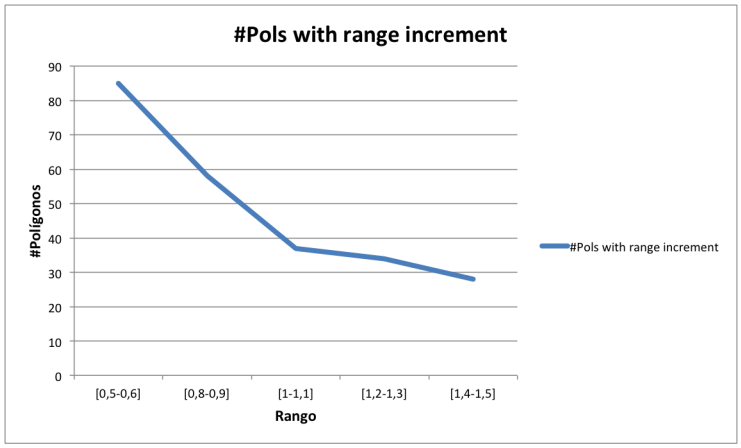

Fig. 3.14: Idem. Ciudad de Balcarce.

Por otro lado, se puede mostrar en el siguiente gráfico cómo varía la cantidad de polígonos para distintos valores en rangos de distancia entre nodos iniciales durante el proceso de generación de polígonos. Se puede ver que a medida que aumenta dicho rango aumenta notoriamente la cantidad de polígonos integrantes en la solución. Además dicho aumento se ve afectado también por las magnitudes en los valores de rangos. A medida que aumentan los mismos, disminuye la cantidad de polígonos en la solución (Figura 3.14). Del mismo modo, puede verse en la Figura 3.15 cómo varía la cantidad de polígonos en la solución a medida que se modifica el rango de distancia entre nodos iniciales en forma integrada. Se puede notar cómo varía dicha cantidad de acuerdo a lo mencionado en las figuras previas.

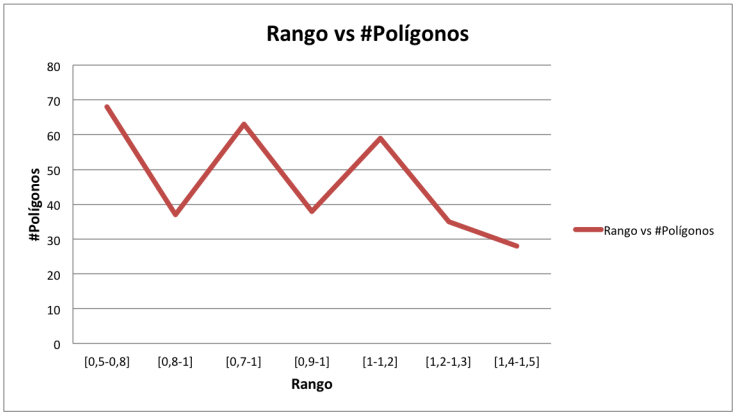

Fig. 3.15: Gráfico que muestra cómo varían los valores en la cantidad de polígonos a medida que varían las distancias en rangos. Ciudad de Balcarce.

3.7. Resultados finales

A continuación se mostrará una serie de gráficos con resultados obtenidos tras la ejecución del algoritmo propuesto. Se probó exhaustivamente su funcionamiento sobre distintos tipos de mapas de ciudades con dibujos y tamaños variados y se analizó la relación entre la calidad de soluciones obtenidas y los tiempos de ejecución.

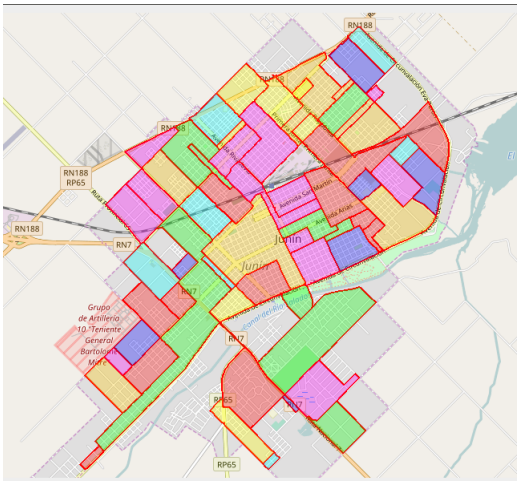

Fig. 3.16: Solución obtenida para la Ciudad de Junín, provincia de Buenos Aires.

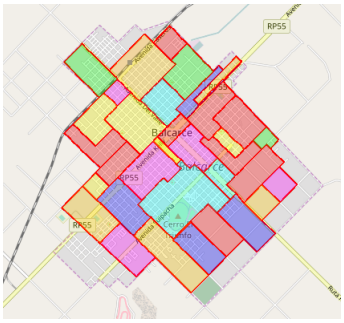

Fig. 3.17: Solución obtenida para la Ciudad de Balcarce, provincia de Buenos Aires.

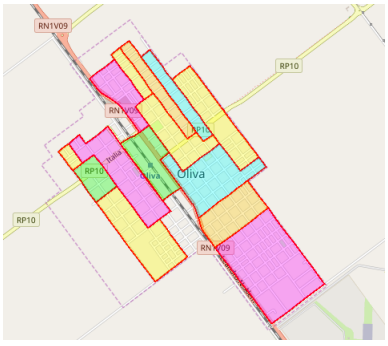

Fig. 3.18: Solución obtenida para la Ciudad de Oliva, provincia de Córdoba.

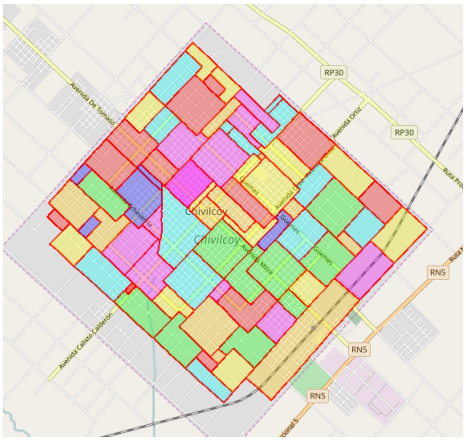

Fig. 3.19: Solución obtenida para la Ciudad de Chivilcoy, provincia de Buenos Aires.

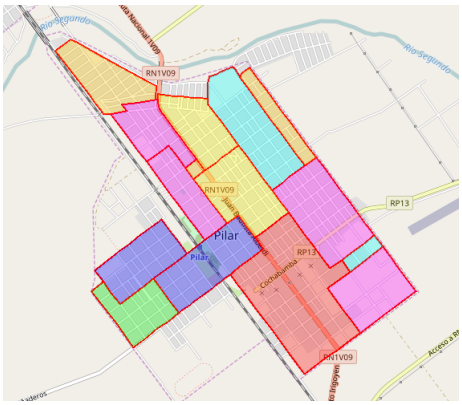

Fig. 3.20: Solución obtenida para la Ciudad de Pilar, provincia de Córdoba.

4. CAPÍTULO IV: CONCLUSIONES

4.1. Conclusiones

A lo largo del presente trabajo se estudió la posibilidad de dar una buena heurística que a partir de un mapa y una flota de camiones recolectores, resolviera el zonificado del primero a fin de asignar cada área a cada uno de los elementos del segundo, para que éstos realizaran la labor de la recolección de forma cooperativa cubriendo toda la región de interés.
De las heurísticas desarrolladas en este informe se enumeraron las más relevantes junto con algunos de los resultados obtenidos con las mismas para determinados valores de parámetros, y se compararon entre ellas con respecto al tiempo consumido y la calidad de los resultados al aplicarlas sobre distintos tamaños y cantidades de áreas.
Las heurísticas desarrolladas consistieron en la selección de pares de nodos, a los que llamamos *iniciales* y que cumplen ciertas restricciones en su selección, e ir luego recorriendo los caminos salientes de cada uno de ellos (en todas las direcciones posibles) hasta encontrar potenciales nodos intersección en dichos caminos (Generación de polígonos).
Más precisamente se busca hallar dos intersecciones que permitan establecer la región formada por los cuatro nodos (los dos iniciales y los dos de intersección).
Se ha comprobado en este trabajo la estrecha relación entre el rango de distancias establecido durante la selección de nodos iniciales en la generación de polígonos para cada mapa y el tiempo de procesamiento del algoritmo.
Otro factor influyente es, obviamente, el tamaño del grafo analizado. A mayor tamaño de ciudad, mayores tiempos de procesamiento. Con el transcurso del trabajo, se fueron resolviendo numerosos inconvenientes relacionados principalmente con los tiempos de procesamiento, calidad en las soluciones obtenidas, casos particulares en dibujos de mapas determinados, etc.
Una de la pricipales heurísticas desarrolladas para bajar tiempos de resolución consistió en subdividir el mapa de una ciudad en cuadrantes, para que al momento de agregar restricciones al modelo de programación lineal entera, más precisamente la restricción de cobertura para las aristas del grafo, se redujera el conjunto de polígonos a comparar contra cada arista. Esta modificación trajo una mejora importante en cuanto al tiempo consumido por el algoritmo en encontrar y devolver una solución factible al problema.

Otra conclusión importante a destacar, luego de realizar numerosas pruebas, variando valores en factores de subdivisión y selección de nodos, fue la de descubrir cómo a medida que se disminuye la distancia en la selección de nodos iniciales se reduce la cantidad de polígonos generados y por consiguiente esto provoca una importante caída en los tiempos de armado y resolución del modelo de PLE.

En lo que respecta a los requerimientos, mediante las heurísticas desarrolladas, se ha logrado cumplir con el más importante, el relacionado con la obtención de áreas producto de la zonificación de formas sencillas, con contornos claros y bien definidos. Esto es muy valioso al momento de llevar a la práctica el proceso de recolección de residuos. Esta mejora era la buscada tomando como base tesis anteriores, en las que se mencionaba como trabajos a

futuro, precisamente la obtención de polígonos, zonas de formas más sencillas y armónicas.

4.2. Trabajo a futuro

Cuando de heurísticas y metaheurísticas se trata, el panorama que se abre entre variantes y mejoras no pareciera tener fin. Evidentemente en este trabajo no sólo no se han cubierto todas las heurísticas posibles para resolver el problema planteado, sino que ni siquiera se han podido cubrir completamente las pocas enunciadas.
Si bien en este trabajo se probó gran cantidad de combinaciones de parámetros, se ha abierto un sinfin más de posibles variaciones y modificaciones a analizar en trabajos posteriores.
Entre esas modificaciones, se pueden mencionar:

- Tratar de optimizar la heurística encargada de *eliminar* aquellos polígonos pequeños, producto del recorte realizado por el algoritmo propuesto. Si bien se ha planteado una heurística encargada de mejorar las áreas de dichos polígonos (combinandólos con aquellos vecinos de área mínima), se puede continuar mejorando este aspecto en futuros trabajos.

- Por otro lado, sería deseable poder en el futuro, setear la cantidad de polígonos en las que se desea particionar el mapa de una determinada ciudad, de manera tal que la heurística no sea quien la determine. De esta forma, se ajustaría mejor a los requerimientos de la realidad, los cuales no fueron contemplados al momento de implementar la solución propuesta en este trabajo, como por ejemplo la cantidad de camiones disponibles en la flota, distancias de recorridos para cada camión en cada zona, etc.

- Con respecto a la heurística encargada de la generación de polígonos, se puede plantear a futuro una nueva estrategia, simplificando su adaptación a la implementación aquí presentada. Esto se debe a la utilización de un patrón de diseño de software **Strategy**, permitiendo agregar una nueva clase que implemente una nueva heurística de generación de polígonos sin tener impacto sobre la implementación de otras clases ya implementadas.

- Es deseable en el futuro, poder garantizar la totalidad en el cubrimiento de las aristas del mapa, obteniendo de esta manera, un modelo de programación lineal entera (PLE) con solución factible.
  En este trabajo, como se explicó previamente, para obtener un modelo con solución factible fue necesario adaptar el armado del mismo, más precisamente se dejaron sin cubrir ciertas aristas para de esta manera poder obtener solución factible al problema.

- Mejoras en la interfaz con el ususario, permitiendo que el mismo pueda visualizar información relacionada a un polígono seleccionado, por ejemplo, superficie del mismo, datos relacionados al promedio en cantidad de residuos a recolectar, distancia de recorrido del mismo, etc.

5. APÉNDICE

5.1. Implementación

Se desarrolló software para resolver el problema utilizando los distintos algoritmos ya mencionados en las secciones previas. Básicamente se implementó un **jar** el cual a través de un *fileChooser* permite adjuntar un archivo con extensión *osm* exportado previamente desde el sitio de OpenStreetMap.
Al realizar dicha operación se dispara la ejecución de distintos algoritmos encargados del procesamiento del input (parsing, armado de grafo, subdivisión del mapa, filtros aplicados), de la generación de polígonos, del armado y resolución del modelo de programación lineal entera (utilizando librería de Scip para Java) y posprocesamiento de la solución obtenida (recorte de polígonos, merge de polígonos *chicos*).

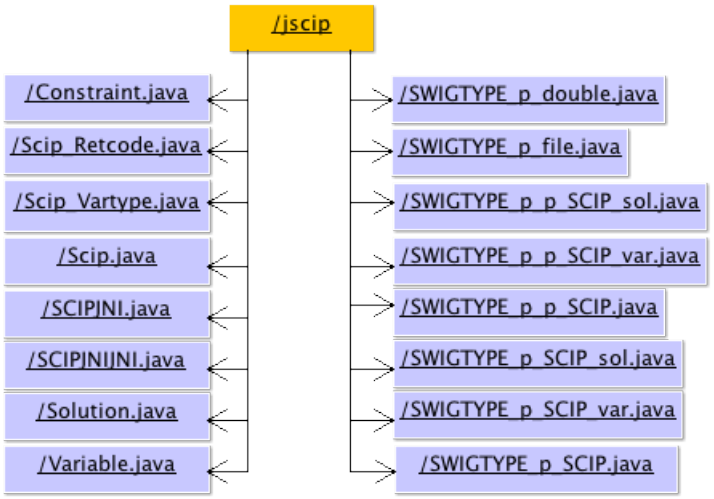

Fig. 5.1: Clases componentes del paquete **jscip**. Implementación de la interfaz de SCIP en Java.

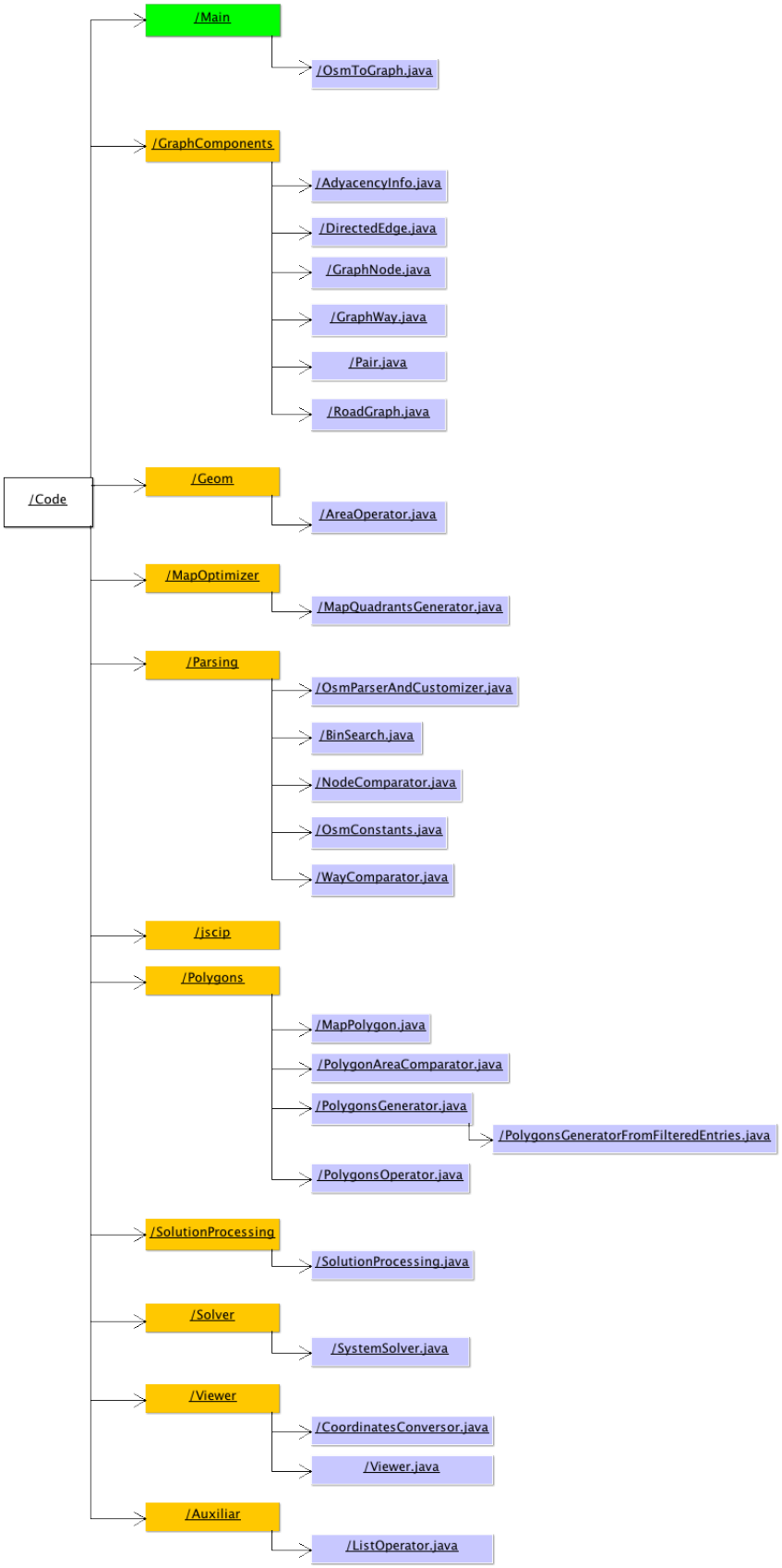

Fig. 5.2: Estructura de paquetes del código.

A continuación se expone con detalle el software desarrollado, en particular los requisitos necesarios, archivos de configuración, las distintas rutinas y subrutinas, los canales de comunicación, salidas intermedias y métodos de debug, para la obtención de los resultados obtenidos en esta tesis.

5.1.1. Configuración

El código que resuelve el problema presentado en los capítulos anteriores, está desarrollado en su totalidad en lenguaje *Java*[5], por lo que para la ejecución es necesario tener instalado dicho entorno, el cual contiene ciertas bibliotecas necesarias. Éstas son:

- *collections-generic-4.0.1*
- *colt-1.2.0*
- *concurrent-1.3.4*
- *j3d-core-1.3.1*
- *jgraph-5.13.0.0*
- *jgrapht-core-0.9.2*
- *jgrapht-demo-0.9.2*
- *jgrapht-ext-0.9.2-uber*
- *jgrapht-ext-0.9.2*
- *jgraphx-2.0.0.1*
- *JMapViewer*
- *kxml2-2.3.0*
- *scip*
- *stax-api-1.0.1*
- *vecmath-1.3.1*
- *wstx-asl-3.2.6*

Antes de ejecutar el programa, es necesario asegurarse de guardar el archivo de exportación de OpenStreetMap (con el mapa a procesar) con el mismo nombre de la ciudad correspondiente. Por ejemplo si se deseara zonificar el mapa de la ciudad de San Clemente del Tuyú, el archivo se debe guardar con el mismo nombre, en caso contrario el programa fallará al momento de definir los límites de la zona administrativa durante el proceso de Parsing del algoritmo.

El *solver* utilizado para plantear y resolver el modelo de PLE (Programación Lineal Entera) es *SCIP*[14] en su versión *3.2.1*.

5.1.2. Scip-Interfaces/JSCIPOpt

En este trabajó se utilizó como interfaz de SCIP para el lenguaje Java la provista por el proyecto alojado en https://github.com/SCIP-Interfaces/JSCIPOpt.git.
Esto permitió la creación y resolución del modelo de programación lineal entera (PLE) para cada problema planteado. Para ello es necesario siempre añadir al proyecto la librería **scip.jar** obtenida al compilar los fuentes provistos en el proyecto. El paquete ya contiene una interfaz para SCIP creada con SWIG (*Simplified Wrapper and Interface Generator*) [10].

Para construir un modelo usando JSCIPOpt siempre es necesario:

- import jscip.*; en el inicio del archivo Java. Realiza la importación de todas las clases necesarias para el uso de la interfaz.
- Crear una instancia del solver e inicializar la estructura de datos C interna mediante

```
Scip scip = new SCIP(); scip.create("Example");
```

- Las clases más importantes son **Scip.java**, **Variable.java**, **Constraint.java**, y **Solution.java**. Representan las estructuras de datos en $C\ SCIP$, $SCIP\_VAR$, $SCIP\_CONS$, $SCIP\_SOL$ y proveen cierta funcionalidad básica. A continuación se menciona un pequeño ejemplo de uso:

```
Variable vars = new Variables[2];
vars[0] = scip.createVar("x",1.0,2.0,-1.0,SCIP_VARTYPE_INTEGER);
vars[1] = scip.createVar("y",3.0,4.0,-2.0, SCIP_VARTYPE_CONTINUOUS);

double[] vals = {1.0, -3.6};
Constraint lincons =
        scip.createConsLinear("lincons",vars,vals,-scip.infinity(),10.0);
scip.addCons(lincons);
scip.releaseCons(lincons);

scip.solve();
scip.free();
```

Podemos mencionar algunos ejemplos de modelos simples creados mediante la interfaz descripta:

- Modelo con restricciones lineales

```
import jscip.*;

{
   public static void main(String args[])
   {
      // load generated C-library
      System.loadLibrary("jscip");

      Scip scip = new Scip();

      // set up data structures of SCIP
      scip.create("LinearExample");

      // create variables (also adds variables to SCIP)
      Variable x =
        scip.createVar("x",2.0,3.0,1.0,SCIP_VARTYPE_CONTINUOUS);
      Variable y =
        scip.createVar("y",0.0,scip.infinity(),-3.0,SCIP_VARTYPE_INTEGER);

      // create a linear constraint
      Variable[] vars = {x, y};
      double[] vals = {1.0, 2.0};
      Constraint lincons =
                scip.createConsLinear("lincons",vars,vals,-scip.infinity(),10.0);

      // add constraint to SCIP
      scip.addCons(lincons);

      // release constraint (if not needed anymore)
      scip.releaseCons(lincons);

      // set parameters
      scip.setRealParam("limits/time", 100.0);
      scip.setRealParam("limits/memory", 10000.0);
      scip.setLongintParam("limits/totalnodes", 1000);

      // solve problem
      scip.solve();

      // print all solutions
      Solution[] allsols = scip.getSols();

      for( int s = 0; allsols != null && s < allsols.length; ++s )
         System.out.println(solution (x,y)=
           (+scip.getSolVal(allsols[s],x)+,+scip.getSolVal(allsols[s], y)+)
           with objective value+scip.getSolOrigObj(allsols[s]));

      // release variables (if not needed anymore)
      scip.releaseVar(y);
      scip.releaseVar(x);

      // free SCIP
      scip.free();
   }
```

```
}
```

- Modelo con restricciones cuadráticas

```
import jscip.*;

public class Quadratic
{
   public static void main(String args[])
   {
      // load generated C-library
      System.loadLibrary("jscip");

      Scip scip = new Scip();

      // set up data structures of SCIP
      scip.create("LinearExample");

      // create variables (also adds variables to SCIP)
      Variable x =
           scip.createVar("x", 2.0, 3.0, 1.0, SCIP_VARTYPE_CONTINUOUS);
      Variable y =
           scip.createVar("y", 0.0, 1.0, -3.0, SCIP_VARTYPE_BINARY);

      // create quadratic constraint: x^2 + 2xy - y^2 + x + y <= 11
      Variable[] quadvars1 = {x, x, y};
      Variable[] quadvars2 = {x, y, y};
      double[] quadcoefs = {1.0, 2.0, -1.0};
      Variable[] linvars = {x, y};
      double[] lincoefs = {1.0, 1.0};

      Constraint quadcons =
        scip.createConsQuadratic("quadcons", quadvars1, quadvars2,
                quadcoefs, linvars, lincoefs, -scip.infinity(), 11);

      // add constraint to SCIP
      scip.addCons(quadcons);

      // release constraint (if not needed anymore)
      scip.releaseCons(quadcons);

      // solve problem
      scip.solve();

      // print the best solution
      Solution sol = scip.getBestSol();

      if( sol != null )
      {
         System.out.println(best solution (x,y)=
           (+scip.getSolVal(sol,x)+,+scip.getSolVal(sol,y)+)with objective value
            +scip.getSolOrigObj(sol));
      }

      // release variables (if not needed anymore)
      scip.releaseVar(y);
      scip.releaseVar(x);

      // free SCIP
      scip.free();
   }
}
```

5.1.3. Estructuras de datos

Como se mencionó y explicó con mayor detalle en capítulos anteriores, en lo que respecta a las estructuras de datos utilizadas en el proyecto, se diseñaron diversas clases agrupadas en paquetes de acuerdo a su funcionalidad.
Se cuenta con las clases que intervienen durante el proceso de parsing y mapeo del mapa determinado en formato *osm* exportado de la página oficial de OpenStreetMap [11] al multigrafo de representación, en el proceso clave y principal de generación de polígonos, en el armado y resolución del modelo de PLE por el Solver, en la visualización de la solución y en el posprocesamiento de la misma.
Además se desarrollaron otras estructuras para la representación de los grafos y subgrafos. Por ejemplo la representación del **RoadGraph** el cual representa la información de adyacencias entre nodos mediante listas de adyacencias implementadas mediante la clase **Map** de Java, listas enlazadas para representar conjuntos de nodos y ejes, diccionarios que relacionan **id** con **GraphNodes**.

5.1.4. Hilo de ejecución

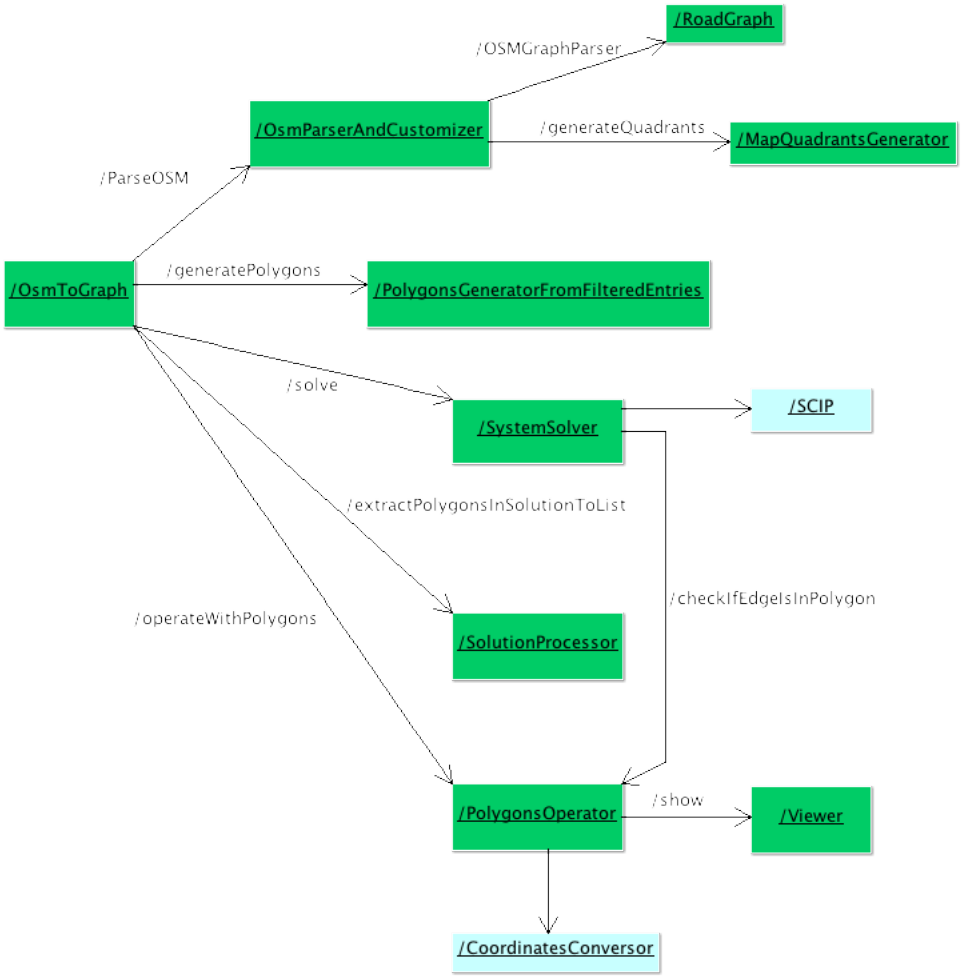

Fig. 5.3: Hilo de ejecución, dependencias entre principales clases.

La clase principal *OsmToGraph* es la encargada de iniciar la ejecución de las demás clases intervinientes durante el proceso. Implementa un Filechooser, el cual permite la carga de archivos con extensión *osm* exportados de la página oficial de OpenStreetMap.
Una vez realizado el attach del archivo de extensión *osm*, se procede al parsing del mismo, mediante la invocación al método *ParseOSM* de la clase *OSMParserAndCustomizer*. Durante dicho método se invocan los métodos *OSMGraphParser* de la clase **RoadGraph**, con el cual se procesan los caminos parseados y se arma la estructura del multigrafo al cual se mapea, y *generateQuadrants* de la clase **MapQuadrantsGenerator**, encargado de la subdivisión del mapa en cuadrantes.
Luego se da inicio al proceso de generación de polígonos mediante el envío del mensaje *generatePolygons* a la clase **PolygonsGeneratorFromFilteredEntries**. Como fue explicado en capítulos previos, los polígonos son generados y ubicados en los respectivos cuadrantes. La heurística aplicada se encarga de seleccionar pares de nodos *seleccionables*

que cumplen ciertas restricciones y luego intenta avanzar en las posibles direcciones con origen dichos nodos hasta lograr obtener intersecciones que permitan la formación de polígonos.
Obtenido el conjunto total de polígonos generados se procede al armado y resolución del modelo de Programación Lineal Entera por parte del Solver. Para ello se llama al método *solve* de la clase **SystemSolver**. En dicho proceso, al crear las restricciones al modelo, se debe garantizar entre ellas, el cubrimiento de la mayor parte de las aristas del grafo. Para ello, se debe chequear cuando un polígono generado cubre a una determinada arista, invocando al método $checkIfEdgeIsInPolygon$ de la clase **PolygonsOperator**.
La llamada al método $extractPolygonsInSolutionToList$ de la clase **SolutionProcessor** se encarga, como lo indica su nombre, del procesamiento de la solución. Aplica el algoritmo *greedy* para la inserción de polígonos en el conjunto solución, de manera de cortar aquellos polígonos solapados con los ya pertenecientes a la misma.
Además se realiza el *merge* de aquellos polígonos de tamaño demasiado pequeño, residuos en su mayoría del proceso de recorte de áreas, para de esta forma obtener una solución con polígonos de áreas bastante similares, es decir, acotando los tamaños de área de los mismos.
Finalmente, se llama al método $operateWithPolygons$ de **PolygonsOperator** para el procesamiento de los diversos polígonos de la solución, mapeando coordenadas del plano a coordenadas geográficas (clase **CoordinatesConversor**) e invocando al **Viewer** para su correcta visualización (mediante el método *show*).

Bibliografía

[1] B. Kim, S. Kim, and S. Sahoo. *Waste collection vehicle routing problem with time windows.* Computers and Operations Research, 33:3624–3642, 2006.

[2] C. Arribas, C. Blazquez, and A. Lamas, *Urban solid waste collection systems using mathematical modelling and tools of geographic information systems.* Waste Management & Research, 24(4):355–363, 2010.

[3] D. Eisenstein and A. Iyer. *Garbage collection in Chicago: a dynamic scheduling model.* Management Science, 43:922–933, 1997.

[4] F. Bonomo, G. Durán, F. Larumbe, and J. Marenco, *A method for optimizing waste collection using mathematical programming: A Buenos Aires case study.* Waste Management & Research, 30(3):311–324, 2012.

[5] Java Platform Standard Edition 7, *https://docs.oracle.com/javase/7/docs/api/*, 2017

[6] J. Yeomans, G. Huang, and R. Yoogalingam, *Combining simulation with evolutionary algorithms for optimal planning under uncertainty: An application to municipal solid waste management planning in the regional municipality of Hamilton-Wentworth.* Journal of Environmental Informatics, 2(1):11–30, 2003.

[7] M. Bianchetti, *Algoritmos de zonificación para recolección de residuos.* Tesis de Licenciatura, Departamento de Computación, FCEyN, UBA (2015).

[8] M. Mourao and M. Almeida. *Lower-bounding and heuristic methods for a refuse collection vehicle routing problem.* European Journal of Operational Research, 121:420–434, 2000.

[9] N. Chang, H. Lu, and L. Wei.*GIS technology for vehicle routing and scheduling in solid waste collection systems.* Journal of Environmental Engineering, 123:901–933, 1997.

[10] Simplified Wrapper and Interface Generator (SWIG), *http://www.swig.org/*, 2017

[11] The OpenStreetMap Foundation, *https://www.openstreetmap.org*, 2017

[12] The Simplex Method. *Wikipedia. https://en.wikipedia.org/wiki/Simplex_algorithm*

[13] Zuse Institute Berlin, *http://scip.zib.de*, 2017

[14] Zuse Institute Berlin, *http://zimpl.zib.de*, 2017