Departamento de Investigación Operativa
Instituto de Computación
Facultad de Ingeniería
Universidad de la República

Mayo 2017

Distribución de Suministros en Logística Humanitaria

Informe de proyecto de grado

Integrantes:
Fabricio Andrés Gregorio de Oliveira
Santiago Bartesaghi Monza
Santiago Fernández Mendy

Tutores:
Pedro Piñeyro
Omar Viera

Resumen

En las operaciones de logística humanitaria, las decisiones de ruteo de vehículos toman un rol fundamental. Ante desastres ya sean naturales o causados por el hombre, diferentes organismos o países suelen enviar suministros a los afectados. Problemas de ruteo similares son ampliamente tratados con fines comerciales en donde en general el objetivo principal es minimizar costos, en términos de distancia o tiempo utilizado. En la logística humanitaria esto se torna más complejo debido a la presencia de otros objetivos adicionales, como ser disminuir el sufrimiento de la población. En la realidad, las organizaciones que brindan ayuda deben lidiar con más de un objetivo simultáneamente, por lo que es normal que no exista una solución que sea óptima a la misma vez para cada uno de los objetivos que componen el problema. Esto se debe a la existencia de conflictos entre objetivos, que harán que la mejora de uno de ellos dé lugar a un empeoramiento de algún otro.

Tomando como punto de partida un trabajo en el que se exploran los objetivos de eficiencia, eficacia y equidad de forma individual, se logra implementar un sistema que permite resolver instancias del *Problema de Ruteo de Vehículos con Time Windows y Split Delivery* tomando estos objetivos como parte de un problema multi-objetivo. El problema se plantea como un *Set Partitioning Problem* y se implementan dos algoritmo genéticos: uno para la generación de rutas factibles y otro para la resolución del problema multi-objetivo.

La ejecución del sistema desarrollado retorna un conjunto de soluciones de buena calidad, cercanas al frente de Pareto.

Dada la naturaleza del problema, la ejecución del sistema desarrollado retorna un conjunto con los mejores compromisos obtenidos, los cuales pueden ser visualizados gráficamente por el usuario. Las pruebas realizadas muestran que los algoritmos genéticos implementados son competitivos respecto a GLPK y CPLEX, ya que se logró obtener resultados óptimos o muy cercanos a ellos en todos los casos, utilizando un tiempo computacional varios órdenes menor.

Palabras claves: Logística humanitaria, ruteo de vehículos, algoritmo genético, multi-objetivo, post-desastre, investigación operativa.

Índice

1 Introducción

Los desastres naturales no son algo nuevo para la humanidad. Se ha convivido con ellos desde que el hombre tiene uso de razón y múltiples regiones en el mundo han sido víctimas de ellos. El daño que los mismos producen a la sociedad y economía de un país puede llegar a ser exorbitante, como por ejemplo el producido por los terremotos de Haití y Chile en 2010, o el tsunami de Indonesia en 2012. Es de vital importancia reconocer que los mismos ocurren cada vez más seguido y por lo general las consecuencias son cada vez mayores [31]. Es esto lo que motiva ampliamente a investigar sobre logística humanitaria, específicamente en la distribución de suministros post desastre, de modo de poder contribuir a este gran tema que afecta a todos.
La logística humanitaria provee un marco de organización para controlar la adquisición, almacenamiento y distribución de suministros de forma eficiente ante estas situaciones. Tomar las mejores decisiones en tiempo real puede significar la diferencia entre la vida y la muerte para miles de personas, y es por eso que desarrollar un algoritmo eficiente que pueda resolver la distribución de suministros es de gran importancia.
Es importante mencionar que la logística humanitaria es algo relativamente nuevo y que actualmente cuenta con más atención por parte de diferentes investigadores. Mucho antes de que la misma comenzara a ganar terreno, la logística comercial y militar ya estaban asentadas. Es por esta razón que gran parte del conocimiento obtenido de las mismas se reutiliza [5].

Las primeras investigaciones sobre problemas de ruteo de vehículos datan de los años 1800. Matemáticos formularon el problema TSP (Travelling Salesman Problem) que busca resolver lo siguiente: "Dado un conjunto de ciudades y la distancias entre ellas, minimizar el costo de visitar todas las ciudades exactamente una vez y retornar a la ciudad de origen".
Recién en la década de 1930 este problema fue considerado matemáticamente por primera vez por Merrill Flood, que buscaba resolver un problema de ruteo para autobuses escolares [28]. Poco tiempo después, Hassler Whitney de la universidad de Princeton introdujo a este problema el nombre que lleva actualmente (Travelling Salesman Problem). Hoy en día es uno de los problemas más estudiados dentro del campo de la optimización y es utilizado como benchmark para varios métodos de optimización [14].
En el problema TSP se basan una amplia gama de problemas incluyendo el también muy conocido problema VRP (Vehicle Routing Problem). El problema VRP es un problema de optimización combinatoria y de programación entera que plantea la siguiente pregunta:
"¿Cuál es el conjunto óptimo de rutas para una flota de vehículos que debe satisfacer las demandas de un conjunto dado de clientes?".
Fue originalmente introducido en un artículo de G.B. Dantzig y J.H. Ramser en 1959 en donde se plantea una aproximación algorítmica y fue aplicado para entregas de gasolina. Luego, en 1962, Clarke y Wright mejoraron el trabajo anterior utilizando una aproximación "greedy" conocida como algoritmo de ahorros [15][35].

Los problemas TSP y VRP son problemas que pertenecen a la clase NP-hard, lo que significa que al aumentar el tamaño del problema el costo computacional de la obtención de la solución aumenta exponencialmente. Es por esta razón que presentan un interés práctico y académico y de mucha importancia dentro de las áreas de Investigación Operativa y la Ciencias de la Computación. Hoy en día para problemas VRP que tienen muchas restricciones o nodos, no se puede obtener una solución óptima en un tiempo razonable. En consecuencia, se utilizan heurísticas y metaheurísticas para encontrar soluciones casi óptimas en el orden de segundos o minutos.
En particular, dado que los problemas relacionados al VRP son de la clase NP-hard, se dejará de lado la búsqueda de la solución óptima (aunque se requiere que sean casi óptimas) a cambio de que las soluciones sean obtenidas en períodos cortos de tiempo. Las técnicas de metaheurísticas, a diferencia de las heurísticas, poseen mecanismos para escapar de soluciones óptimas locales en su intento por encontrar la solución óptima global. Comparadas con las heurísticas, las metaheurísticas encuentran soluciones muy superiores, con esfuerzos computacionales mayores pero aceptables desde el punto de vista práctico [19].

El término "logística" tiene diferentes significados para diferentes organizaciones y personas. Por ejemplo, existen la logística operacional en las fuerzas armadas que actúa como puente entre la logística estratégica y la logística táctica o en el sector de negocios que es utilizada como un marco de trabajo para la planificación [5].
En [47], se define a la logística humanitaria como el proceso de planificar, implementar y controlar de manera eficiente, el flujo y almacenamiento de materiales y de información relacionada. Esto se hace desde el punto de origen al punto de consumo, con el propósito de satisfacer las necesidades de los beneficiarios y aliviar el sufrimiento de la población vulnerable.
En [1] se define a la logística comercial como aquella parte de la actividad empresarial que tiene como finalidad la previsión, organización y control del flujo de materiales (materias primas, productos semielaborados y productos terminados), desde las fuentes de aprovisionamiento hasta el consumidor final.
A simple vista se podría decir que son muy similares pero existen grandes diferencias como por ejemplo el tipo de público que involucran, el entorno y el tiempo en los cuales se desarrolla la actividad. En la logística humanitaria se busca ayudar a la población afectada, y esto debe realizarse en el menor tiempo posible ya que puede haber vidas que dependen de ello.

Como mayores diferencias en la logística humanitaria se destacan [5]:

- Impredecibilidad de la demanda, en términos de tiempo, localización, tipo y cantidad.
    - Tiempo: No hay margen para satisfacer las necesidades. Se debe actuar de inmediato.

    - Localización: No se tiene una red definida, sino que se debe crear en el momento, teniendo en cuenta aspectos geográficos y partes afectadas por el desastre.
    - Tipo y cantidad: Lo que se necesite puede variar ampliamente entre los destinatarios de los suplementos.
- Falta de recursos para afrontar el desastre. Por lo general nunca se está lo suficientemente preparado en fondos, tecnología, recursos humanos, entre otros.
- Gran dependencia del ámbito social: se necesitan voluntarios y donantes dispuestos a ayudar y facilitar los materiales necesarios.
- No se busca maximizar ganancias, sino minimizar el sufrimiento de la población. El objetivo de este tipo de logística tiene fines sociales.

Centrándose en los tiempos de respuesta, en Campbell et al. (2008) [9] se propone cambiar la función objetivo clásica de VRP por dos nuevas funciones que se adecúan más a los objetivos que tiene la logística humanitaria. Los mismos son minimizar el tiempo máximo de llegada y minimizar el tiempo promedio de entrega. A partir de esto se pueden obtener mejores tiempos de servicio que aquellos obtenidos como resultado del objetivo tradicional. En Huang et al. (2011) [22] se extiende el trabajo realizado por Campbell et al. (2008) [9] para considerar la distribución equitativa de suministros, dado que este es un punto crítico para la logística humanitaria [7]. Proponen métricas de equidad alternativas donde la equidad es caracterizada en términos de la disparidad entre niveles de servicio entre los destinos. Los niveles de servicio son caracterizados por la cantidad y velocidad de la entrega. En el trabajo futuro se menciona que en la práctica los objetivos propuestos deben ser tratados concurrentemente. Esto tiene sentido ya que en la realidad se quiere ser eficaz y equitativo en el alivio a los damnificados, pero teniendo en cuenta que se disponen recursos limitados es importante tener en cuenta la eficiencia.

En la realidad, se intenta encontrar un punto intermedio entre los objetivos, y es por esto que es de gran importancia considerar al problema como uno multi-objetivo. Tomando como punto de partida el trabajo realizado por [22] en el que se exploran los objetivos eficiencia, eficacia y equidad de forma individual, se plantea desarrollar un sistema que permita resolver instancias de *Problema de Ruteo de Vehículos con Time Windows y Split Delivery* tomando estos objetivos como parte de un problema multi-objetivo. La resolución de los mismos debe realizarse en un tiempo computacional corto (segundos o pocos minutos) y se deben obtener buenas soluciones. El sistema deberá presentar el resultado al usuario final de manera amigable.

El problema se plantea como un *Set Partitioning Problem* (SPP), el cual utiliza un subconjunto de todas las rutas posibles. Para esto es necesario implementar un algoritmo que lo genere. Se implementan dos variantes de un algoritmo genético, capaces de generar rutas con y sin *Split Delivery*, basándose en los trabajos realizados por Alvarenga et al. (2007) [3] y Alvarenga (2005) [2].

Una vez obtenido un conjunto de rutas factibles, para la resolución del problema multi-objetivo se utiliza NSGA-II, un algoritmo genético ampliamente utilizado para resolver este tipo de problemas. Este algoritmo busca combinaciones de rutas para generar las soluciones. Como resultado se obtiene un conjunto de Pareto que consiste en un conjunto con las mejores soluciones obtenidas [17].

Se utilizan las instancias propuestas por Solomon para VRPTW [40], las cuales son utilizadas como referencia en múltiples trabajos de ruteo de vehículos [3][22][45]. Además, se generan instancias propias que sólo pueden ser resueltas mediante el uso de *Split Delivery*.
Se realizan pruebas para la generación y selección de rutas para cada objetivo de forma individual en instancias chicas, obteniendo el resultado óptimo o muy cercano a él en todos los casos. Adicionalmente, partiendo de las rutas generadas por el algoritmo generador, se prueba la selección de rutas para el problema multi-objetivo para instancias grandes, obteniendo resultados muy satisfactorios.

Como resultado del proyecto, se logra desarrollar un sistema que permite obtener soluciones multi-objetivo de calidad para problemas SDVRPTW. Para instancias de 100 nodos el tiempo de resolución es de pocos minutos.
La arquitectura del sistema permite la implementación de algoritmos alternativos, ya sea para la generación y/o la selección de rutas, para poder fácilmente compararse contra los que fueron desarrollados. La visualización de los resultados permite al usuario el entendimiento de la solución obtenida por el sistema y la posibilidad de verificarla manualmente.

Este artículo consiste de 8 Secciones, incluyendo la introducción actual. En la Sección 2 se define el problema en cuestión. En la Sección 3 se definen los requerimientos tanto funcionales como no funcionales. En la Sección 4 se detalla la solución propuesta para resolver el problema. En la Sección 5 se presenta el diseño y la implementación de la solución. En la Sección 6 se detallan las pruebas planificadas para comprobar la calidad de la solución. En la Sección 7 se exponen los resultados obtenidos. En la Sección 8 se presentan las conclusiones del proyecto.

2 Definición del problema

El problema que se va a abordar en el proyecto consiste en la extensión de un trabajo sobre ruteo de vehículos en logística humanitaria [22], de forma de poder aplicar sus conceptos en un terreno más cercano a la realidad. Dicho trabajo, a su vez, utiliza Campbell et al. (2008)[9] como punto de partida.

En Campbell et al. (2008) [9] se propone sustituir la función objetivo clásica de VRP (minimizar la distancia total recorrida) por dos nuevas funciones que se centran en los tiempos de respuesta y por lo tanto son más adecuadas a los objetivos que tiene la logística humanitaria. Los mismos son minimizar el tiempo máximo de llegada y minimizar el tiempo promedio de entrega. De esta forma se pueden obtener mejores tiempos de servicio que aquellos obtenidos como resultado del objetivo tradicional. Se demuestra que minimizar la distancia total recorrida resulta en mayores tiempos de servicio.

En Huang et al. (2011) [22] se extiende el trabajo realizado por Campbell et al. (2008) [9] para considerar la distribución equitativa de suministros, dado que este es un punto crítico para la logística humanitaria [7]. Proponen métricas de equidad alternativas donde la equidad es caracterizada en términos de la disparidad entre niveles de servicio entre los destinos. Los niveles de servicio son caracterizados por la cantidad y velocidad de la entrega.
El foco está puesto en la distribución de última milla, teniendo en cuenta que muy comúnmente las decisiones son tomadas ad hoc, lo que lleva a la ineficiente utilización de los recursos, respuesta lenta y entregas inadecuadas o desiguales.
Se definen y formulan métricas de *performance* en la distribución de recursos, enfocándose en la eficiencia (costos de transporte), eficacia (respuesta rápida y adecuada) y equidad (ser justos con todos los destinos). Ésta última es la que está directamente relacionada a la logística humanitaria y es la que trae consigo grandes desafíos para poder resolverse de manera eficiente.

2.1 Modelo

El modelo propuesto se basa en el *Last Mile Distribution Problem* (LMDP). Dado un grafo $G = (N^0, A)$ *completo y simétrico*, N^0 es el conjunto de nodos, incluyendo el depósito $(i = 0)$ y las locaciones de demanda $i \in (1..N)$. Los costos de viaje $c_{ij} \ \forall \ (i,j) \in A$, son no negativos y satisfacen el teorema de desigualdad triangular. Los ruteos comienzan en el tiempo 0. La demanda se mide en pallets, como si fuera un solo recurso, pero cada pallet puede estar compuesto de diferentes tipos de suplementos. $K = \{1,..,K\}$ representa la flota de vehículos que operan desde el depósito para servir a los receptores de ayuda, C es la capacidad de cada vehículo medido en pallets y la demanda puede ser distribuida entre múltiples

vehículos. La demanda de cada destino es menor que la de un vehículo $d_i \leq C \quad \forall\, i \in N$ y la demanda de todos los destinos es a lo sumo igual que lo que pueden distribuir todos los vehículos $KC \geq \sum_{i \in N} d_i$. Los vehículos solo pueden hacer un viaje, visitar cada destino una sola vez y no regresan al almacén a recargar.
Para medir la eficiencia se toma en cuenta el costo de recorrido en función del tiempo utilizado. La eficacia se mide como la suma de los tiempos de arribo de cada pallet. La equidad se mide tomando en cuenta la disparidad de los niveles de servicio (velocidad y cantidad de entrega) entre los distintos destinos.

El problema LMDP involucra dos tipos de decisiones:

1. Decisiones de ruteo. Para cada vehículo, los clientes que debe visitar.
2. Decisiones de distribución. El número de pallets que se entregan en cada cliente.

El LMDP se formula con una extensión de la formulación de partición de conjunto de VRP, la cual asume que el conjunto *R={1,..,R}* de rutas factibles está dado. Cada ruta $r \in R$ representa un conjunto de nodos y el orden en que son visitados. Con esta información, se pueden obtener los siguientes parámetros para cada ruta:

- El costo de viaje c_r es el tiempo requerido para recorrer la totalidad de la ruta.
- El parámetro b_{ir} denota la presencia del nodo *i* en la ruta *r:* $b_{ir} = 1$ si el nodo *i* es visitado por la ruta *r*, y *0* si no.
- El parámetro a_{ir} denota el tiempo de arribo al nodo *i* en la ruta *r*, si la ruta *r* visita al nodo *i* ($b_{ir} = 1$). Si la ruta *r* no visita al nodo *i* ($b_{ir} = 0$), a_{ir} es igual a *T*, un límite superior en el horizonte temporal.
- Se asume que cada ruta candidata en *R* es realizada como máximo por un vehículo (se muestra que existe una solución óptima en donde cada ruta es realizada como máximo una vez).
- Las variables de decisión binaria x_r determinan si la ruta *r* es seleccionada o no: $x_r = 1$ si es seleccionada, *0* si no.
- Las variables de decisión enteras y_{ir} indican el número de pallets entregados al nodo *i* por la ruta *r*.

Se define a *LMDP(Z_f)* como el modelo LMDP con la función objetivo de minimizar alguna de las métricas anteriormente mencionadas.

El *LMDP(Z_f)* se formula de la siguiente forma:

$$min\ Z_f(x,y) \tag{1}$$

sujeto a:

$$\sum_{r \in R} x_r \leq K \tag{2}$$

$$y_{ir} \leq d_i b_{ir} x_r \qquad \forall\, i \in N,\ \forall\, r \in R \tag{3}$$

$$\sum_{i \in N} y_{ir} \leq C \qquad \forall r \in R \tag{4}$$

$$\sum_{r \in R} y_{ir} = d_i \qquad \forall i \in N \tag{5}$$

y_{ir} *integer*, x_r *binary* (6)

Las restricciones de (2) garantiza que el número de rutas no sobrepasa el número de vehículos. Las restricciones de (3) garantiza que a un nodo se le entregan suministros en una ruta sólo si es visitado en la misma y que la entrega no supera la demanda. Las restricciones de (4) limita la cantidad de suministros entregados en cada ruta a la capacidad del vehículo. Las restricciones de (5) asegura que cada nodo receptor reciba la cantidad de pallets solicitados. Finalmente, las restricciones de (6) especifica que las variables de entrega son enteras y las de ruteo binarias.

La función objetivo de (1) se elige según la métrica que se quiera optimizar:

1. Eficiencia (Z_1). Medida que calcula el tiempo de viaje total para rutas seleccionadas.

   $$Z_1(x,y) = \sum_{r \in R} c_r x_r \tag{7}$$

   Ésta es equivalente al objetivo tradicional del VRP, optimizando el tiempo en vez de la distancia, por lo que LMDP(Z_1) es equivalente al SDVRP.

2. Eficacia (Z_2). Medida en función de la suma de los tiempos de arribo de cada pallet.

   $$Z_2(x,y) = \sum_{i \in N} \sum_{r \in R} a_{ir} y_{ir} \tag{8}$$

   Para cada ruta se suman los tiempos de arribo a cada nodo multiplicado por la cantidad de pallets entregados. El objetivo de minimizar (8) es una extensión del objetivo de minimizar la suma de los tiempos de arribo para cada punto de demanda en Campbell et al. (2008) [9].

3. Equidad (Z_3). Se introducen 3 métricas para medir la equidad. Las primeras dos, Z_3^a y Z_3^b, miden la dispersión del nivel de servicio entre los nodos. Se muestra que no son viables como objetivo para el modelo, por lo que se las utiliza con propósitos de evaluación. La tercer métrica, Z_3, es un balance entre eficacia y equidad. De forma similar a Z_2, minimiza el tiempo de espera y atiende la demanda de forma rápida, pero además también incorpora la equidad mediante la ponderación de las unidades de demanda de manera diferente. Para incorporar la equidad se considera una función de desutilidad convexa *f(w)*, donde *w* es la fracción de demanda insatisfecha. Debido a que la tasa de cambio de la desutilidad es mayor cuando el porcentaje de demanda insatisfecha está más cerca de 1 (la demanda atendida es baja), los primeros pallets entregados reciben un mayor beneficio que el último. Esta métrica minimiza la desutilidad del tiempo de arribo. La función de desutilidad alienta a un

vehículo a no satisfacer necesariamente toda la demanda de un nodo, sino preferiblemente a guardar para abastecer a otro. Contrariamente a Z_2 donde el cambio entre la demanda entregada es accesible desde la información del modelo (y_{ir}), el cambio en la desutilidad requiere un cálculo dependiente de muchas variables. Es más simple formular un equivalente a la misma.

Se discretiza el horizonte de tiempo en horas. Para cada $t \in T = 1,...,T$ y cada nodo $i \in N$ se calcula el porcentaje de demanda insatisfecha del nodo i en el tiempo t, w_{it}, y una penalización igual a $f(w_{it})$ se almacena. La desutilidad ponderada en el tiempo se formula de la siguiente forma:

$$w_{it} = \left(1 - \frac{1}{d_i} \sum_{r:a_{ir}<t} y_{ir} \right) \tag{9}$$

$$Z_3(x,y) = \sum_{t=1}^{T} \sum_{i=1}^{N} f(w_{it}) \tag{10}$$

La ecuación de (10) suma las penalizaciones acumuladas debido al porcentaje de demanda insatisfecha de cada nodo $i \in N$ en cada $t \in T$. En la práctica, $f(w_{it})$ no necesita ser calculada para cada $t \in T$, sino sólo cuando $f(w_{it})$ cambia. Esto se puede encontrar ordenando los tiempos de llegada de los envíos al nodo i y determinando el tiempo entre entregas sucesivas.

Se utiliza una función lineal de desutilidad para los cálculos de (11). La misma se elige para imitar la forma de una función polinómica convexa. A medida que se entregan más pallets, w disminuye. La pendiente creciente nos asegura que los últimos pallets entregados tienen menos peso que los primeros, ver Figura 1.

$$f(w) = \begin{cases} \frac{4w}{13} & si \ \ w < 0.25 \\ \frac{8w-1}{13} & si \ \ 0.25 \leq w < 0.5 \\ \frac{16w-5}{13} & si \ \ 0.5 \leq w < 0.75 \\ \frac{24w-11}{13} & si \ \ w \geq 0.75 \end{cases} \tag{11}$$

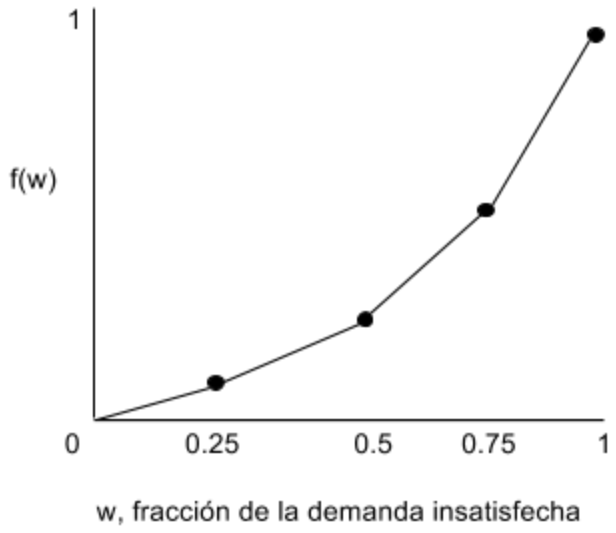

Figura 1 - Función de desutilidad

Se demuestra que para Z_1 y Z_2 existe una solución óptima donde para cada par de nodos, como máximo un vehículo visita a ambos. De esto se desprende que para Z_1 y Z_2, realizar *Split Delivery* es sólo beneficioso por razones relacionadas a la capacidad. Para Z_1, realizarlo puede resultar en el uso de menos vehículos, lo que se traduce por la desigualdad triangular en una reducción de los costos de viaje. En contraste, en Z_2, donde el tiempo de respuesta se minimiza, es deseable utilizar la flota entera de vehículos. Aunque se utilice toda la flota, puede ocurrir *Split Delivery* si un vehículo llega a un nodo antes que los otros pero no es capaz de satisfacer su demanda total. Ahora, lo anterior no se cumple para Z_3, la cual se puede ver altamente beneficiada cuando hay *Split Delivery*. El mismo puede ocurrir para que un nodo reciba su demanda antes a las expensas de que otro nodo las reciba más tarde. Dado lo anterior, se puede introducir las restricciones de (12) a los modelos de Z_1 y Z_2.

$$\sum_{r=1:\,(b_{ir}>0\ \&\ b_{jr}>0)}^{R} x_{ir} \qquad \forall i,j \in N,\ i > j \tag{12}$$

A partir de los resultados computacionales los autores realizan las siguientes observaciones. Cuando la solución tiene como objetivo maximizar la eficiencia, se utiliza la menor cantidad de vehículos posible, salvo en algunas excepciones. Sin embargo, cuando lo importante es la equidad o eficacia las soluciones siempre utilizan la totalidad de los vehículos. Además, cuando se busca optimizar la eficiencia importa el retorno del vehículo al almacén pero esto no tiene importancia cuando se busca optimizar la eficacia o la equidad, por lo que las soluciones forman figuras diferentes en cuanto al recorrido de los vehículos. La figura en caso de optimizar la eficiencia es más bien curva rodeando al almacén y las figuras para las soluciones optimizando la eficacia o equidad forman rayos alejándose cada vez más del mismo. Las soluciones que tienen como objetivo eficacia son influenciadas por la cantidad

de demanda solicitada por los clientes. Se le da prioridad a los clientes con alta demanda, a expensas de los clientes con poca. Por otro lado, las soluciones para equidad permiten que destinos con poca demanda sean atendidos en un tiempo razonable respecto a los demás.

2.2 Objetivo

Como trabajo futuro en Huang et al. (2011) [22] se propone que los 3 objetivos propuestos sean tratados concurrentemente. El problema principal consiste en obtener soluciones para un problema multi-objetivo donde los objetivos entran en conflicto entre sí. Por ejemplo, al mejorar la eficiencia de una solución posiblemente se empeore la equidad y viceversa. Por esta razón no existe una única solución factible y la importancia está en proveer, a un tomador de decisiones, un conjunto de buenas soluciones para que pueda seleccionar la más adecuada para la situación.

La obtención de soluciones de calidad en poco tiempo para problemas de logística humanitaria puede resultar en una gran reducción de daños e incluso salvar numerosas vidas. La importancia de desarrollar un sistema de este tipo es entonces muy alta. Difícilmente se logre un único sistema que resuelva este problema junto a todas sus posibles variables (necesidades de las personas, recursos, objetivos, infraestructura, entre otras), ya que las mismas dependen de cada contexto. Por esta razón es que este trabajo se limita a continuar con el trabajo realizado por Huang et al. (2011) [22].

Como objetivo se plantea desarrollar un sistema que pueda resolver el problema multi-objetivo, donde los objetivos son eficiencia, eficacia y equidad. Es fundamental que la resolución ocurra en un tiempo computacional razonablemente corto, en el orden de minutos, ya que sino se podría utilizar un software de optimización. Estos últimos no se utilizan en la práctica ya que obtener la solución óptima toma un tiempo computacional inaceptable para estas situaciones de emergencia. Por otro lado no se puede dejar de lado que las soluciones obtenidas sean razonablemente buenas, es decir, que estén cerca de las óptimas. Se deberá poder resolver instancias de VRP con *Time Windows* y *Split Delivery*. El sistema deberá presentar el resultado de forma gráfica al usuario final.

3 Definición de requerimientos

3.1 Requerimientos funcionales

- El sistema debe de tener una interfaz gráfica para su configuración y ejecución.
- El sistema debe ser capaz de resolver instancias de SDVRPTW.
- El sistema debe ser capaz de resolver las instancias como un problema multi-objetivo.
- El sistema debe retornar el resultado gráficamente con las rutas que componen la solución obtenida y además mostrar el detalle de las mismas incluyendo la cantidad de demanda entregada a cada cliente y el tiempo en que esto sucede.

3.2 Requerimientos no funcionales

- El tiempo de ejecución del sistema debe ser razonable según la cantidad de nodos que tenga la instancia.
- El resultado debe ser de calidad. Esto significa que las soluciones obtenidas estén razonablemente cerca de las óptimas.
- El sistema deberá ejecutarse correctamente en cualquier plataforma con el ambiente necesario instalado.

4 Solución propuesta

4.1 Generación de rutas

Como se explicó en la Sección 2.1, el modelo propuesto por Huang et al. (2011) [22] asume que un conjunto de rutas válidas *R = {1,....,R}* está dado. Inicialmente se intentó implementar el mismo algoritmo generador de rutas utilizado por ellos, lo cual era lógico si se deseaba comparar resultados. El detalle del mismo se encuentra en una versión extendida la cual está referenciada en el trabajo original. Desafortunadamente, no se pudo acceder a la misma ya que el link está caído. A raíz de esto, se intentó contactar a los tres autores (Michael Huang, Karen Smilowitz y Burcu Balcik), vía email y/o LinkedIn, y al área de Logística Humanitaria de la Universidad Northwestern [12]. Se obtuvo respuesta solamente de Burcu Balcik, pero negativa, ya que le resultaba imposible acceder a la versión solicitada.
Con esto surgió un problema que no se había tenido en cuenta originalmente: crear un algoritmo generador de rutas propio o encontrar uno para luego adaptarlo. Luego de una amplia búsqueda se encontró el siguiente trabajo el cual se explicará en detalle a continuación.

4.1.1 Algoritmo de Alvarenga et al. (2007) para VRPTW

En este trabajo realizado por Alvarenga et al. (2007) [3] se propone una aproximación heurística para el problema de ruteo de vehículos con *Time Windows* (VRPTW) utilizando la distancia recorrida como objetivo principal. El problema se plantea como un *Set Partitioning Problem* (SPP), y se utiliza un algoritmo genético para la generación de rutas.

El modelo de VRPTW se caracteriza por los siguientes puntos:

1. Flota *K* de vehículos idénticos que entregan productos a *N* clientes.
2. Vehículos con capacidad homogénea *C*.
3. Para cada cliente *i = 1...N*, la demanda solicitada c_i, el tiempo de servicio s_i y la *Time Window* $[a_i, b_i]$ son datos conocidos. Siendo a_i el menor tiempo en el cual es posible atender el cliente y b_i el máximo.
4. Si un vehículo llega a un cliente antes que a_i, entonces debe esperar. Se debe empezar el servicio antes que b_i. (*Hard time window*)
5. Todas las rutas comienzan y terminan en el depósito central.
6. Las locaciones del depósito y los clientes, la distancia y el tiempo de viaje entre ellos está dado.
7. Cada cliente debe ser visitado una vez.

El modelo anteriormente descrito se puede formular como un SPP de la siguiente manera:

$$min \sum_{r \in R} c_r x_r \tag{13}$$

$$s.t. \sum_{r \in R} \delta_{ir} x_r = 1 \qquad \forall i \in C \tag{14}$$

$$x_r \in \{0, 1\} \tag{15}$$

Donde *R* es el conjunto de rutas, *C* el conjunto de clientes, c_r el costo de la ruta *r*, x_r la variable de decisión tal que su valor es 1 si la ruta *r* se considera en la solución y 0 si no, δ_{ir} los parámetros auxiliares para indicar el conjunto de clientes presente en cada ruta *r*, 1 si el cliente *i* es atendido por la ruta *r*, y 0 si no.
La búsqueda de la solución entera óptima utilizando el modelo anterior, donde todas las rutas posibles se incluyen en el conjunto *R*, sólo se puede realizar para conjuntos de clientes pequeños. Suponiendo un problema con 50 clientes, en donde las restricciones permitan que las rutas atiendan como máximo a 10 clientes, el número posible de rutas es:

$$\sum_{n=1}^{10} \left(\frac{50!}{(50-n)!} \right) \approx 3.8 \times 10^{16} \tag{16}$$

Por lo tanto, no es posible que el modelo SPP sea resuelto por un algoritmo MIP en un tiempo razonable para instancias con 50 clientes o incluso menos. Entonces, surge la necesidad de generar un conjunto de rutas reducido, pero que sea suficientemente bueno.

La solución propuesta fue motivada por el hecho de que un mínimo local para el VRPTW tiene posibilidades significantes de contener rutas que pertenezcan también al óptimo global. Si se logra producir una cantidad considerable de soluciones óptimas localmente con buena calidad, es posible unir estas rutas en un conjunto *R* que será la entrada del SPP.
Los autores mencionan que muchos métodos heurísticos clásicos, como PFIH [40], han sido propuestos como métodos generadores. Los mismos tienen el siguiente problema: al generar las rutas de forma independiente sin considerar el problema en su totalidad, es difícil que la combinación de las mismas termine siendo una buena solución global. Por lo tanto, las rutas que se desea formen parte del conjunto *R* son aquellas que se evalúen como parte de la solución completa, por ejemplo, aquellas encontradas en un óptimo local. Para producir este conjunto, se implementa un algoritmo genético, el cual es ejecutado de forma independiente varias veces. Cada ejecución es considerada una isla de evolución, ya que no hay influencia ni intercambio de material genético entre ellas. Al ser independientes, se garantiza diversidad entre las diferentes ejecuciones.

Cada ejecución independiente del algoritmo retorna un individuo. Un individuo es una

solución válida que contiene un conjunto de cromosomas. A su vez, un cromosoma se define como una lista la cual representa a los clientes a ser atendidos por un sólo vehículo. Cada cliente tiene un identificador único *i = 0,..,N*, donde *N* es la cantidad de clientes. Ver Figura 2.

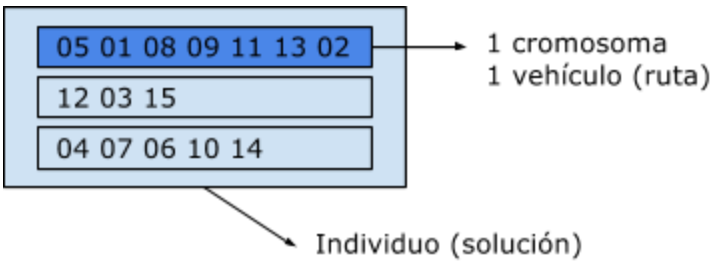

Figura 2 - Solución retornada por una ejecución del algoritmo generador

Pseudocódigo algoritmo genético para producir un mínimo local:
AG() : Individuo

1. Inicializar población
2. Mientras no se supere la cantidad de generaciones
    a. Evaluación
    b. Elitismo
    c. Selección
    d. Crossover
    e. Mutación
    f. Actualizar población
3. Retornar mejor individuo

Además de la utilización de dicho algoritmo, en Alvarenga et al. (2007) [3] se propone el uso de un problema reducido para mejorar el conjunto de rutas previamente creado. El mismo consiste en realizar lo siguiente:
Luego de cada ejecución independiente del algoritmo generador, se utilizan las rutas como entrada de un MIP encargado de encontrar la mejor combinación posible de las mismas. Luego estas rutas son insertadas al conjunto de rutas finales. El procedimiento se repite una cierta cantidad de tiempo. Esto hace que el conjunto final de rutas sea un 30% menor en tamaño sin perder calidad. Se consideró que dicha post optimización no aplica para el presente trabajo ya que al tratarse de un problema multi-objetivo, es conveniente que haya mayor variedad en el conjunto de rutas finales.
En la Figura 13 se puede ver el funcionamiento del algoritmo completo.

A modo de resumen se presentan los siguientes puntos tomados en cuenta para la implementación del algoritmo:

Puntos a favor

1. Al estar enfocado en problemas VRPTW, el modelo tiene varios puntos similares con el propuesto por Huang et al. (2011) [22]. Del punto 1 al 6 del modelo de VRPTW descrito al comienzo de esta Sección.
2. Presentó muy buenos resultados.
3. Se logró entender el algoritmo propuesto en su totalidad, el cual a primera vista resultó ser intuitivo y realizable para el tiempo disponible.

Puntos en contra

1. Hay una diferencia crucial y es que para VRPTW se asume que cada cliente debe ser visitado una vez, y por lo tanto no es necesario contar con *Split Delivery*.
2. El objetivo principal es minimizar la distancia recorrida, lo que difiere de los objetivos propuestos por Huang et al. (2011) [22]. Esto hará que sea necesario modificar ciertas partes del algoritmo para adaptarse a la nueva realidad.

Mencionados los puntos anteriores, se consideró era viable reproducir y modificar el algoritmo. Luego de analizar a fondo el trabajo fue inevitable toparse con ciertas partes que no estaban explicadas en detalle o generaban cierta duda.
Por dicha razón, se contactó a Guilherme Bastos Alvarenga para obtener más información. Se plantearon preguntas principalmente sobre el algoritmo de reproducción, las cuáles contestó casi inmediatamente, razón por la cual se le está muy agradecido.
Además de ésto, Alvarenga brindó el nombre de su tesis de doctorado [2], redactada en portugués, en la cual en una de sus secciones se presenta una variante del algoritmo anteriormente mencionado. Luego de acceder a ella, se pudo ver que contiene partes que están más detalladas, y otras que difieren con Alvarenga et al. (2007) [3]. Una clara diferencia se observa en las mutaciones utilizadas en ambos trabajos, así como en la forma en que se genera la nueva población en cada generación del algoritmo. A raíz de ésto, se decidió implementar el algoritmo tomando como base ambos trabajos.

4.1.2 Propuesta

Como punto de partida, se propone la implementación del algoritmo, realizando en el proceso las modificaciones que se consideran de alta prioridad para cumplir con el modelo propuesto por Huang et al. (2011) [22]. En Alvarenga et al. (2007) [3] el objetivo principal es minimizar la distancia recorrida, mientras que en Huang et al. (2011) [22] se plantea optimizar la eficiencia, eficacia y equidad en función del tiempo utilizado y el nivel de servicio otorgado.
Teniendo en cuenta que el fin principal de este trabajo es poder resolver los tres objetivos como parte de un problema multi-objetivo, se consideró importante generar cuatro subconjuntos de rutas que formen parte del conjunto de rutas final:

1. Orientadas a optimizar eficiencia
2. Orientadas a optimizar eficacia
3. Orientadas a optimizar equidad
4. Orientadas a optimizar eficiencia, eficacia y equidad a la misma vez

Se deben introducir nuevas formas de calcular el *fitness* para cada uno de los subconjuntos mencionados.

Las principales y más complicadas modificaciones a realizar surgen a raíz de poder soportar soluciones con *Split Delivery*. Al estar enfocado para VRPTW, el algoritmo propuesto por Alvarenga et al. (2007) [3] parte de la premisa de que los clientes no se repiten en distintas rutas de un individuo.
Es importante mencionar que aunque en los individuos no haya *Split Delivery*, luego en la solución final, sí puede haber. Esto es así porque el conjunto de rutas de entrada del software seleccionador es la unión de rutas de diferentes individuos. Por esto, puede darse la selección de dos rutas de individuos diferentes que visiten a un mismo cliente.
De todas formas, esto no es suficiente para las instancias que no puedan ser resueltas sin *Split Delivery* porque el algoritmo generador sería incapaz de generar una solución local válida.

En un principio se pensó en modificar el algoritmo propuesto por Alvarenga et al. (2007) [3] para que sea capaz de manejar soluciones con y sin *Split Delivery* a la misma vez. Sin embargo, en Huang et al. (2011) [22] se muestra que para Z_1 y Z_2, realizar *Split Delivery* es sólo beneficioso por razones relacionadas a la capacidad. Además, Archetti et al. (2008) [6] muestran que para objetivos relacionados a eficiencia, el *Split Delivery* es solamente beneficioso cuando la demanda promedio de los clientes se encuentra entre ½ y ¾ de la capacidad de los autos. De esto se desprende que contar con soluciones que contengan *Split Delivery* entre sus rutas no siempre es beneficioso e idealmente el algoritmo debería ser capaz de llegar a una solución sin él cuando no lo necesite.
La pregunta que se planteó fue la siguiente: ¿con qué necesidad generar rutas con *Split Delivery* para situaciones en las que no va a ser beneficioso?. Que el algoritmo no tenga que preocuparse por decisiones acerca de cuánta demanda entregar a cada cliente resulta en una mejor performance general.
Se consideró era mejor implementar una nueva variante que soporte *Split Delivery*. Se trata a dicha variante como un generador de rutas complementario, capaz de agregar mayor diversidad al conjunto R de rutas final. En el caso en que el problema no pueda ser resuelto sin *Split Delivery*, dicho generador toma el papel principal.

4.1.2.1 Población Inicial

Utilizar una heurística rápida y simple para distribuir los clientes entre los vehículos puede reducir significativamente el tiempo utilizado por el algoritmo genético para obtener un mínimo local [3]. Es por esto que el método heurístico propuesto por Solomon [40], llamado

Push Forward Insertion Heuristic (PFIH), ha sido utilizado de manera frecuente por varios investigadores. Para una detallada descripción del método PFIH, ver [44]. En el PFIH original la ecuación de (17), define cuál es el primer cliente de cada nueva ruta. Luego los clientes se eligen uno a uno de forma de minimizar el costo, y por lo tanto se puede observar que es un método determinista. Alvarenga et al. (2007) [3] propone una variante de este algoritmo, el PFIH estocástico, en el cual el primer cliente de la ruta es elegido de forma totalmente aleatoria. Esto es necesario para producir individuos diferentes en la primer generación del AG. Luego de que el primer cliente ha sido seleccionado de forma aleatoria, los siguientes se irán eligiendo de modo que resulten en el menor costo de inserción. Una nueva ruta es creada si no es posible insertar más clientes en la ruta actual.

$$c_i = -\alpha d_{0i} + \beta b_i + \gamma \frac{p_i}{360} d_{0i} \quad (17)$$

Donde $\alpha = 0.7$, $\beta = 0.1$ y $\gamma = 0.2$, calculados empíricamente por Solomon. d_{0i} es la distancia del cliente *i* al depósito central. b_i es la cota superior de la *Time Windows* para llegar al cliente *i*. p_i es el ángulo polar del cliente *i* al depósito central.

Pseudocódigo PFIH estocástico:
PFIHe : Individuo

1. Crear individuo vacío
2. Mientras haya clientes sin atender
    a. Crear nueva ruta
    b. Agregar primer cliente aleatorio
    c. Mientras pueda agregar clientes a la ruta, agregar en posición con menor costo de inserción
    d. Agregar ruta a individuo

Para que sea viable la inserción de un cliente en una ruta, se debe cumplir con las restricciones de *Time Windows* y capacidad. En este caso, al no haber *Split Delivery*, la ruta debe ser capaz de suplir la demanda total del cliente para aceptarlo.

Para la variante que soporte *Split Delivery* se propone una variante del PFIH Estocástico en la que las rutas pueden suplir parte de la demanda solicitada por el cliente. En este caso, una ruta sólo va a dejar de aceptar clientes si su capacidad está completa o no hay ninguno que cumpla con las restricciones de *Time Windows*. A raíz de esto es prácticamente seguro que en los individuos de la población inicial algunos clientes sean atendidos por más de una ruta. Es muy común que se de la siguiente situación: al insertar un cliente tal que la ruta sólo puede cubrir parte de su demanda, el mismo inevitablemente va a tener que luego ser seleccionado por otra.

4.1.2.2 Selección

Para el paso de selección, se seleccionan dos conjuntos de individuos. A un conjunto se le realizará crossover y luego mutación, y al otro solamente mutación. Según Alvarenga et al. (2007) [3] hay muchos métodos propuestos en la literatura para la selección en un AG, siendo los más populares la selección por ruleta giratoria y por torneo.
En el método de selección por ruleta giratoria, la probabilidad que tiene el individuo de ser seleccionado es directamente proporcional a su *fitness*. Este método es bastante sensible a la función de evaluación y casi siempre se necesita algún control adicional, por ejemplo, escalamiento del *fitness*. Se opta por utilizar selección por torneo múltiple *k-way*. En un torneo *k-way*, *k* individuos son seleccionados de forma aleatoria. Luego, el individuo que presente el mejor *fitness* es el ganador. El proceso se repite hasta que el número de individuos seleccionados sea suficiente. En la Figura 3 se muestra gráficamente el proceso de selección.

Figura 3 - Selección mediante torneo múltiplo k-way

4.1.2.3 Fitness

La función de *fitness* para evaluar al individuo se relaciona con la función objetivo, pero no es necesariamente idéntica. Al tener 3 objetivos se tiene una función diferente de *fitness* para cada uno. Además, para la generación del cuarto subconjunto de rutas se incluye un cuarto fitness llamado "todos", el cual consiste en una ponderación normalizada de los 3 *fitness* mencionados.

Fitness Eficiencia: Z_1. Ecuación (7).
Fitness Eficacia: Z_2. Ecuación (8).
Fitness Equidad: Z_3. Ecuación (10).
Fitness Todos: Ponderación equitativa y normalizada de Z_1, Z_2 *y* Z_3

4.1.2.4 Crossover

En el crossover dos individuos diferentes se recombinan para dar lugar a un tercer individuo. En el algoritmo genético propuesto, el espacio de búsqueda se limita a la región factible. Por lo tanto, cada individuo es siempre factible; razón por la cual es necesario tener cuidado con los operadores de crossover y mutación, ya que un simple intercambio entre dos clientes puede violar las restricciones de tiempo y capacidad. En consecuencia, es necesario introducir un operador más complejo sin introducir un sesgo en ninguna dirección particular. En Alvarenga et al. (2007) [3] se propone una nueva estrategia de cruce para abordar las siguientes características:

- Heredar todas las rutas posibles de cada padre.
- Equilibrar el número de rutas recibido de cada padre.
- Evitar individuos con mayor número de vehículos que sus padres.
- Evitar el intercambio excesivo entre clientes para no aumentar el costo de la función.
- Una vez inevitable, se deben crear nuevas rutas que cubran con la demanda de los clientes.

4.1.2.5 Elitismo

Para garantizar que el AG pueda mantener siempre la mejor solución encontrada hasta el momento, se implementa la estrategia de elitismo. El método consiste en que el mejor individuo de la población actual pase a formar parte de la siguiente población. Para que de todas formas se pueda realizar mutaciones sobre el individuo, se deben realizar dos copias del mismo. La primera será sometida a mutaciones, mientras que la segunda permanecerá intacta hasta el próximo ciclo como se puede ver en la Figura 4.

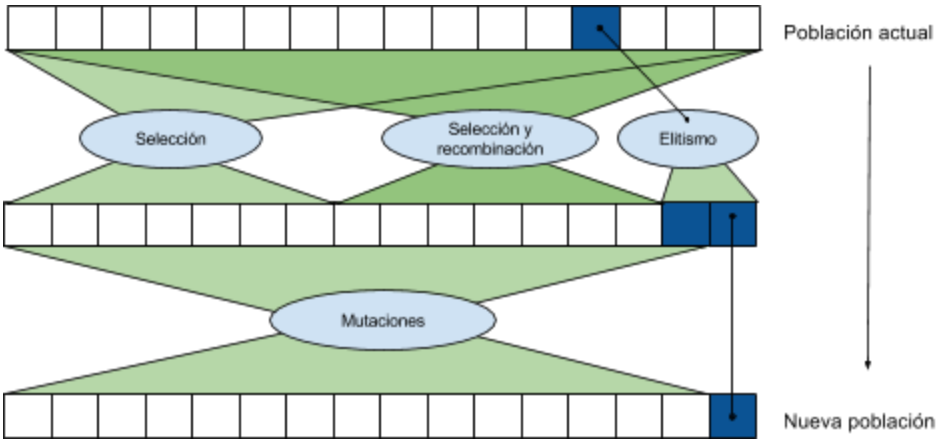

Figura 4. Etapas para generar una nueva población.

4.1.2.6 Mutaciones

El proceso de mutación es necesario para la inserción de nuevas características a la población actual. De no realizarse, la región de búsqueda del algoritmo genético se limita a un área muy pequeña de la región factible.

Se utilizan los operadores especificados en Alvarenga et al. (2007) [3]:

1. Mutación 01 *(Random customer migration)*: Este operador elige un vehículo y un cliente asociado a él de forma aleatoria. Se intenta migrar el cliente a otro vehículo no vacío. Si la inserción resulta en una ruta factible, se acepta independientemente del nuevo costo. Ver Figura 5.
2. Mutación 02 *(Bringing the best customer)*: Este operador elige un vehículo de forma aleatoria y busca e inserta el cliente de los otros vehículos en la posición que resulte en el menor incremento de tiempo de viaje. Ver Figura 6.
3. Mutación 03 *(Re-insertion using stochastic PFIH)*: Este operador elige una ruta de forma aleatoria y lo mantiene intacto. Luego aplica el procedimiento PFIH estocástico con los clientes de todas las rutas restantes. Ver Figura 7.
4. Mutación 04 *(Similar customer exchange)*: Este operador elige un vehículo de forma aleatoria, y busca un cliente con *Time Window* similar en el resto de vehículos para intentar un intercambio. Se considera similar a la *Time Window* que representa la mínima variación respecto al cliente con que se compara. Es decir, la que minimice a_i - a_j, siendo a_i y a_j los tiempos de inicio de la *Time Window* de los clientes *i* y *j*, respectivamente. Ver Figura 8.
5. Mutación 05 *(Customer exchange with positive gain)*: Se eligen dos rutas aleatorias y se analizan todas las posibilidades de intercambio de clientes; se realiza la mejor de todas sólo si resulta en una reducción del tiempo total de viaje. Este operador es costoso en términos de tiempo procesamiento, *O(nm)*, donde *n* y *m* son el número de clientes atendidos por dos vehículos. Sin embargo, es muy importante mejorar la convergencia del algoritmo genético. Ver Figura 9.
6. Mutación 06 *(Merge two routes)*: Este operador elige dos rutas diferentes de forma aleatoria e intenta unirlas de una forma aleatoria. Los clientes restantes se reinsertan en otras rutas o en una nueva. Si de igual forma siguen quedando clientes sin ruta, se aplica PFIH estocástico. Ver Figura 10.
7. Mutación 07 *(Re-inserting customer)*: Este operador elige un cliente de forma aleatoria, de una ruta seleccionada también de forma aleatoria, y luego se inserta en la mejor posición en la misma ruta. Ver Figura 11.
8. Mutación 08 *(Route partitioning)*: Este operador elige una ruta y un cliente asociado a ella de forma aleatoria y luego la divide en otras dos, tomando el cliente elegido como posición de corte. Ver Figura 12.

Figura 5 - Random customer migration

Figura 6 - Bringing the best customer

Figura 7 - Re-insertion using stochastic PFIH

Figura 8 - Similar customer exchange

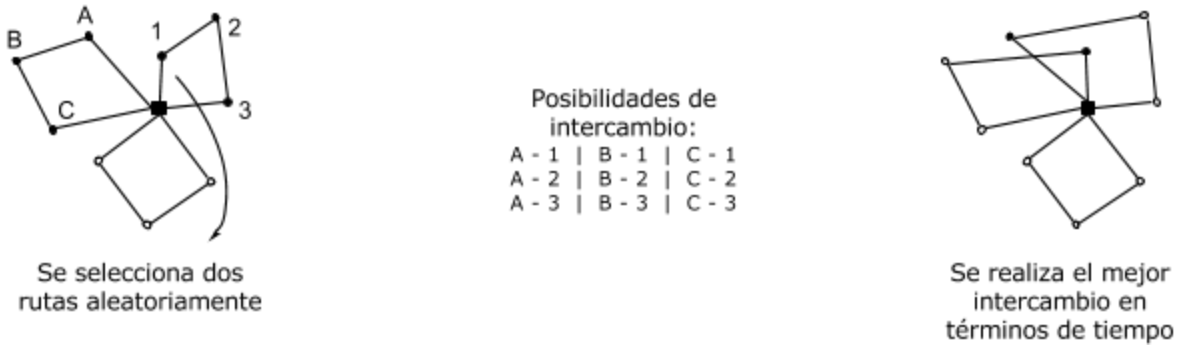

Figura 9 - Customer exchange with positive gain

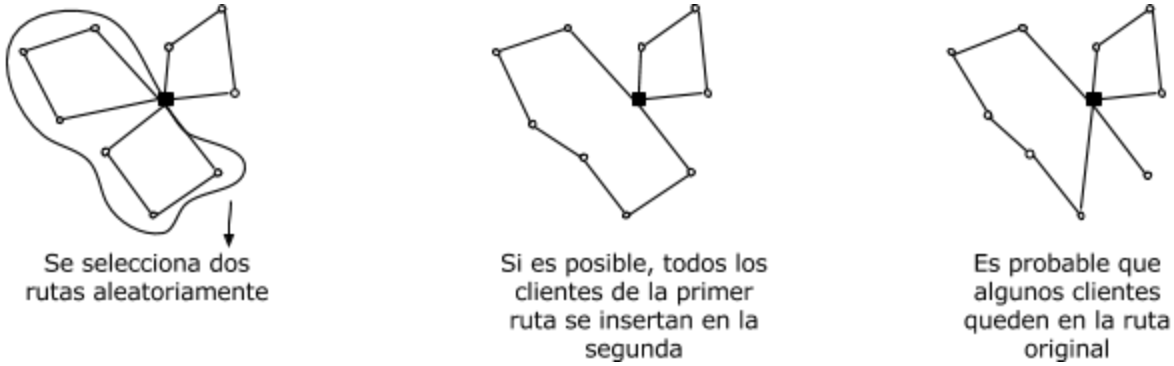

Figura 10 - Merge two routes

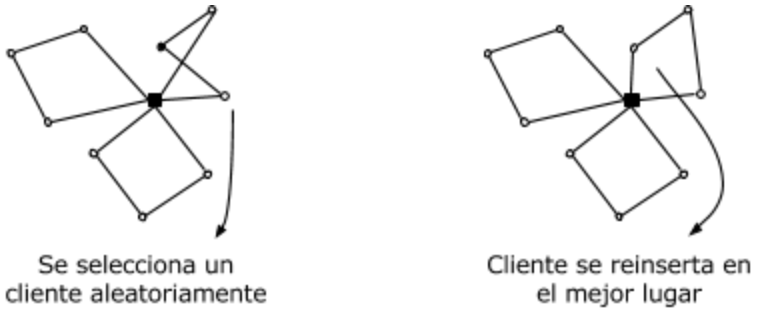

Figura 11 - Reinserting customer

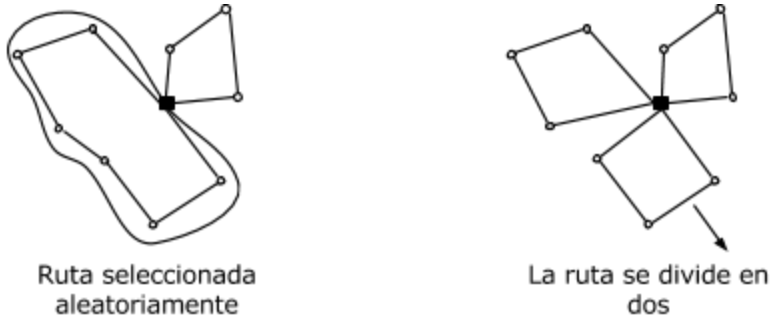

Figura 12 - Route partitioning

Para que las inserciones sean válidas, se debe verificar en todos los casos que se cumpla con las *Time Windows* y que la ruta destino sea capaz de cubrir la demanda total del cliente.

Para la variante que soporta *Split Delivery* se deben modificar los 8 operadores anteriormente descritos. Los principales cambios se detallan a continuación:
En las inserciones, la ruta destino debe ser capaz de cubrir la demanda que la ruta origen le brinda al cliente. Si el cliente a insertar ya existe en la ruta destino, entonces se realiza una unión: la ruta se mantiene igual pero se aumenta la demanda entregada a dicho cliente.

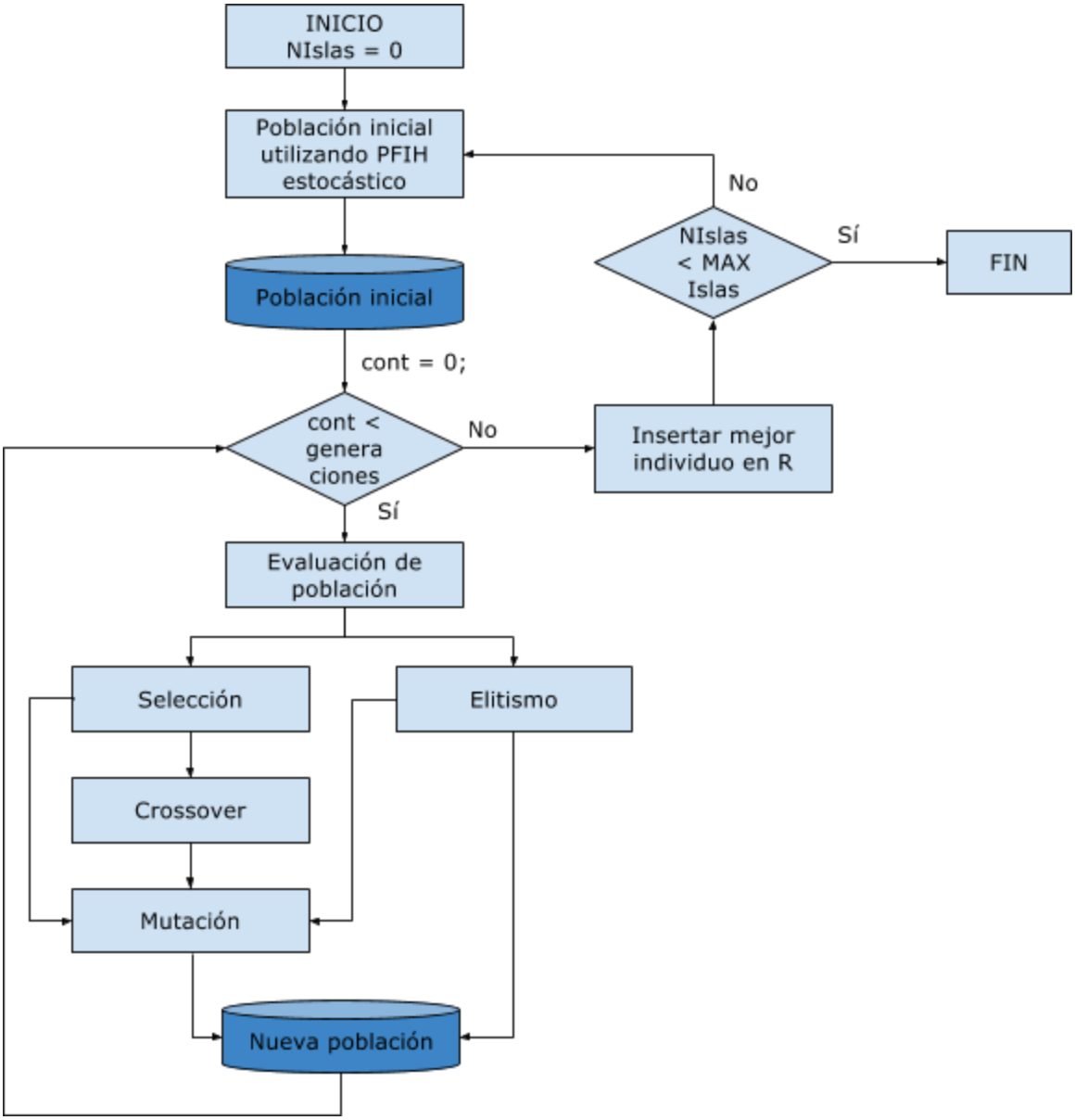

Figura 13 - Flujo propuesto. Se ejecutan varias evoluciones (Max Islas) y todas las rutas del mejor individuo se incluyen en el conjunto de rutas R.

4.2 Resolución del problema multi-objetivo

4.2.1 Consideraciones

La mayoría de los problemas de optimización no lineal en el mundo real requieren la optimización de varios objetivos y por lo general al menos algunos de ellos son contrapuestos. En este tipo de problema se le asigna a las funciones objetivo cierta importancia, y con ello surge el siguiente problema: normalmente no existe una solución que produzca un óptimo de forma simultánea para todos los objetivos que componen el problema. Esto se debe a la existencia de conflictos entre objetivos, que harán que la mejora de uno de ellos dé lugar a un empeoramiento de algún otro.
La solución típica de tal problema es un compromiso. Un buen compromiso es aquel en el que uno de los criterios sólo puede mejorarse empeorando al menos uno de los otros.
Este enfoque se llama optimización de Pareto, y el conjunto de todos los mejores compromisos se llama conjunto óptimo de Pareto o soluciones no dominadas. En la práctica, usualmente sólo se requiere una solución. Por lo tanto, la optimización multi-objetivo basada en el óptimo de Pareto se divide en dos fases: en primer lugar, se determina el conjunto de soluciones óptimas de Pareto, de las cuales una debe ser elegida como el resultado final por una persona de acuerdo a sus preferencias. Esto contrasta con las tareas de optimización de un sólo objetivo, en las que no se requiere ningún segundo paso de selección [25].

Para entender lo anterior es clave dar las siguientes definiciones [32]:

1. Dominancia: Una solución x domina a una solución x' si es igual de buena que x' respecto a todas las funciones objetivo, y mejor en al menos una.
2. Optimalidad de Pareto: Una solución x^* se llama Pareto-óptima cuando no hay otra solución factible que la domine.

En muchos problemas de optimización multi-objetivo, el conocimiento sobre el conjunto óptimo de Pareto ayuda al tomador de decisiones a elegir el mejor compromiso entre las diferentes soluciones. Por ejemplo, al diseñar sistemas informáticos, los ingenieros suelen realizar una exploración del espacio de diseño para aprender más sobre el conjunto Pareto-óptimo. De esta manera, el espacio de diseño se reduce al conjunto de compensaciones óptimas: un primer paso para seleccionar una implementación apropiada. Sin embargo, generar el conjunto óptimo de Pareto puede ser computacionalmente costoso y a menudo imposible, porque la complejidad de la aplicación subyacente impide que los métodos exactos sean aplicables. Los algoritmos evolutivos (EAs) son una alternativa: por lo general no garantizan identificar las compensaciones óptimas, sino tratar de encontrar una buena aproximación, es decir, un conjunto de soluciones cuyos vectores objetivos estén no demasiado lejos de los vectores objetivos óptimos [50].

Uno de los métodos de evaluación más utilizados para la resolución de problemas multi-objetivo, además de la optimalidad de Pareto, es la suma ponderada. La media ponderada es una medida de tendencia central, que es apropiada cuando en un conjunto de datos cada uno de ellos tiene una importancia relativa respecto de los demás. Se obtiene multiplicando cada uno de los datos por su ponderación (peso) para luego sumarlos [25].
Como los objetivos normalmente tienen diferentes escalas, los valores de los mismos deben ser normalizados. Dado cierto objetivo a optimizar, la normalización del mismo se puede realizar utilizando la fórmula (18).

$$O_{Norm} = \frac{O}{O_{Optimo}} \tag{18}$$

Donde O_{Optimo} es el valor óptimo para el objetivo (calculado previamente) y O es el valor del objetivo a optimizar.

Mediante el uso de pesos adecuados, cada punto de un frente convexo de Pareto se puede obtener mediante suma ponderada. En la Figura 14, el punto *P* se puede obtener para los pesos w_1 y w_2. Las flechas muestran la dirección de movimiento de los puntos donde se obtiene la mayor ganancia de calidad [25].

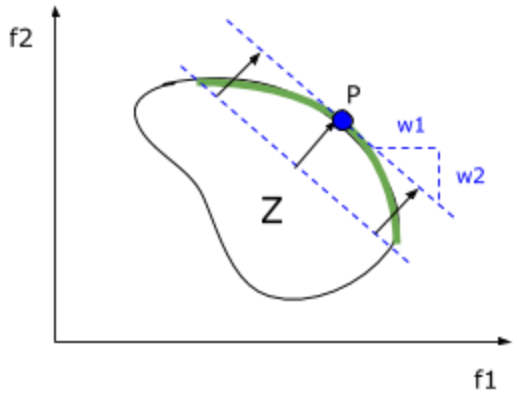

Figura 14 - Frente de Pareto Convexo

Para los frentes de Pareto no convexos, ver Figura 15, es posible que ciertos puntos del frente no puedan obtenerse por la suma ponderada. La región entre los puntos *A* y *B* es un ejemplo de esta falencia de este método de agregación [25].

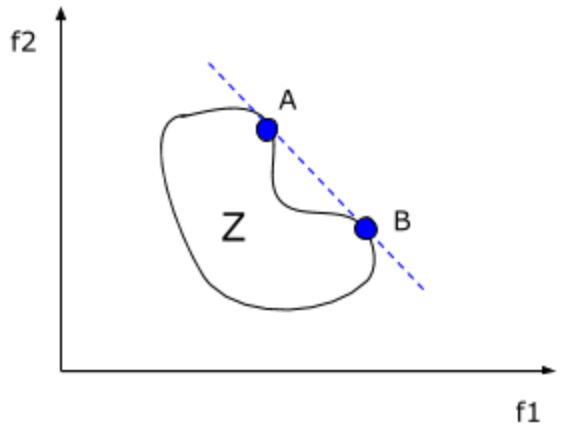

Figura 15 - Frente de Pareto no Convexo

La suma ponderada se utiliza a menudo para aplicaciones prácticas. Las razones son la simplicidad de su aplicación y la manera fácil de integrar las restricciones, que están más allá de las limitaciones propias de la región factible [25].

4.2.2 A fast and elitist multiobjective genetic algorithm: NSGA-II

Para la selección de rutas se decide utilizar la reconocida metaheurística *Nondominated Sorting Genetic Algorithm II* (NSGA-II). Esta decisión se toma principalmente por ser el algoritmo mayormente utilizado para resolver problemas multi-objetivo [37].
Pertenece a la rama de los *Multiobjective Evolutionary Algorithms* (MOEAs), y fue propuesto con el objetivo de atacar las razones por las que otros MOEAs con ordenamiento no dominado y compartición son mayormente criticados. Éstas son [17]:

1. Alta complejidad computacional del ordenamiento no dominado. El algoritmo de ordenamiento no dominado previamente utilizado tiene una complejidad computacional de $O(MN^3)$ donde M es el tamaño de objetivos y N es el tamaño de la población. La complejidad existe por el procedimiento de ordenamiento no dominado en cada generación. NSGA-II presenta un algoritmo de ordenamiento no dominado cuya complejidad computacional es $O(MN^2)$.
2. Enfoque no elitista. Resultados muestran que usar etilismo puede mejorar significativamente el desempeño de un algoritmo genético, además de evitar perder soluciones buenas una vez que ya fueron encontradas. NSGA-II tiene un enfoque elitista ya que selecciona los mejores individuos a partir de la unión de la población inicial de la población evolucionada en esa iteración.
3. La necesidad de especificar un parámetro de compartición para diversificación de la población. Uno de los mecanismos que existe para garantizar la diversidad de la población para poder obtener varias soluciones equivalentes se han basado en el concepto de compartir. El mayor problema de este enfoque es que requiere de la especificación de un parámetro. NSGA-II no utiliza un parámetro de compartición, en

cambio utiliza un comparador que toma en cuenta la dispersión de las soluciones encontradas, para así favorecer la diversidad de la población.

Las pruebas realizadas sobre ciertos complejos problemas de prueba muestran que NSGA-II, en comparación con otros dos MOEAs (*Pareto-Archived Evolution Strategy* y *Strength-Pareto Evolutionary Algorithm*), es capaz de encontrar mayor variedad de soluciones y tiene mayor convergencia respecto al frente óptimo de Pareto.

El *loop* principal de NSGA-II tiene la siguiente forma:

1. Inicializar población
2. Mientras no se supere la cantidad de generaciones
    a. Se genera una nueva población temporal mediante:
        i. Selección con binary tournament usando *crowding-distance* como el comparador
        ii. Crossover
        iii. Mutación
    b. Se combinan las 2 poblaciones obteniendo una población de tamaño *2N*. Esto se hace para garantizar elitismo
    c. Se asigna un ranking a cada uno de los individuos de la población
    d. Se actualiza la población seleccionando los individuos con mejor *ranking* hasta que tenga tamaño *N*. En caso de empate se desempata con el comparador de *crowding-distance*
3. Retornar población final

En la Figura 16 se muestra la ejecución de NSGA-II para la iteración t. Primero se muestra la combinación de la población inicial con la población evolucionada (selección + crossover + mutación). Luego se muestra la población ordenada por el *ranking.* Finalmente se muestra la conformación de la población final.

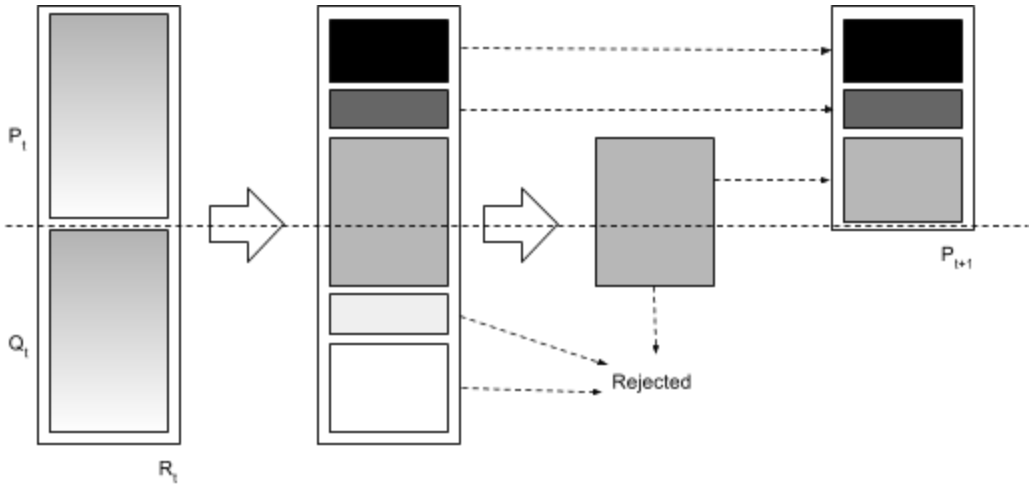

Figura 16 - Iteración t NSGA-II

4.2.3 Propuesta

La propuesta para el algoritmo seleccionador es entonces implementar la metaheurística NSGA-II para el problema específico. Esto requiere definir e implementar:

1. Codificación de la solución
2. Construcción de la población inicial
3. Selección a utilizar
4. Crossover a utilizar
5. Mutación a utilizar

Adicionalmente hay que definir de qué manera se va a soportar *Split Delivery* ya que las rutas no establecen la demanda entregada a los clientes.

4.2.3.1 Codificación

La codificación es la representación de la solución. Para el problema planteado se parte de un conjunto de rutas en donde cada una tiene su propio identificador. La codificación de cada cromosoma es entonces un entero, siendo el individuo solución un conjunto de números enteros que representan rutas. A su vez una solución podrá tener como máximo *N* rutas siendo *N* la cantidad de vehículos disponibles. El caso de que un vehículo no sea utilizado se representará con el elemento vacío.

Figura 17 - Forma de una solución

Se puede observar la forma de una solución en la Figura 17. En ella se tienen ocho vehículos, de los cuales solo se utilizan 5 que reparten suministros a través de las rutas con los siguientes identificadores: 6, 18, 33, 25, 14.

Notar que no se puede tener dos elementos iguales en la solución ya que es precondición del problema de Huang que cada ruta puede ser recorrida por sólo un vehículo. Por otro lado notar que alterar el orden de los elementos que conforman una solución no genera ningún cambio en ella.

4.2.3.2 Población inicial

Por la naturaleza de los algoritmos genéticos, NSGA-II requiere de una población inicial la cual irá evolucionando generación a generación. Se tienen en cuenta dos factores fundamentales para su generación [10]:

1. Partir de una solución base considerada “buena”, ya que agiliza la convergencia a la solución óptima.

2. Tener diversidad en la población, ya que de esta manera se evita quedar estancado en óptimos locales.

Para lograr lo anterior, la población inicial se conforma con:

1. Las mejores soluciones retornadas por el algoritmo generador de rutas.
2. Soluciones generadas de manera aleatoria, tomando rutas del conjunto de rutas factibles hasta que la demanda de cada uno de los clientes se satisfaga.

4.2.3.3 Selección

Se decide utilizar el conocido operador *binary tournament selection* con un nuevo criterio de comparación, *crowded-comparison,* que fue introducido junto con NSGA-II en Deb et al. (2002) [17] y mostró buenos resultados.

Binary tournament selection consiste en tomar aleatoriamente dos individuos de la población y elegir al mejor de ellos según cierto comparador o criterio. Esto se repite *N* veces siendo *N* el tamaño de la población, generando así la base de la nueva población [13].

El operador *crowded-comparison* requiere que cada solución en la población tenga previamente calculado su *ranking* y su *crowding-distance* (una medida de la densidad de las soluciones en el vecindario). En pocas palabras este operador compara soluciones de a pares, donde opta por la solución con mejor *ranking* y en caso de que tengan el mismo opta por la que se encuentre en la región menos poblada [17].

El *ranking* de una solución se obtiene a partir de un ordenamiento de toda la población. La misma se ordena según la dominancia, para la cual se consideran tanto las restricciones como los *fitness* de cada uno de los objetivos de manera que:

- Dadas dos soluciones que cumplan con todas las restricciones, una tendrá mejor ranking que otra si y sólo si la domina.
- Una solución que cumpla toda las restricciones tendrá mejor ranking que una que no cumpla con todas las restricciones.
- Dadas dos soluciones que no cumplan con todas las restricciones, tendrá mejor ranking la que cumpla las restricciones en mayor grado.

Para calcular el crowding-distance de una solución, primero se necesita calcular el promedio de la distancia de las dos soluciones a cada lado para cada uno de los objetivos, como se puede ver en la Figura 18. Luego se suman los promedios calculados obteniendo así el valor que representa el crowding-distance de la solución.

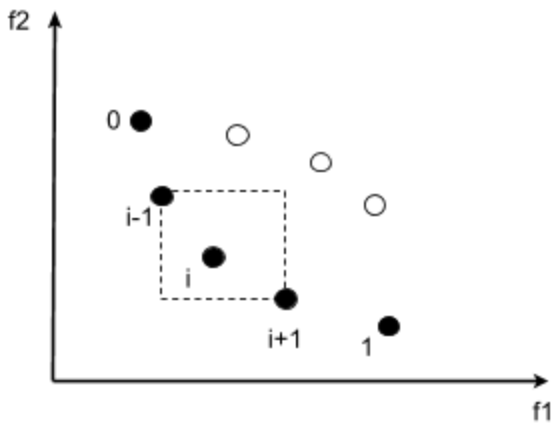

Figura 18 - Cálculo crowding-distance. Los puntos negros son soluciones del mismo frente no dominado.

4.2.3.4 Crossover

Como operador de crossover se decidió implementar el *single-point crossover* ya que es uno de los más utilizados en investigaciones de algoritmos genéticos [42]. En este operador se elige un punto aleatoriamente entre cero y la cantidad de variables de la solución y se generan dos hijos a partir de dos padres de manera que cada hijo tiene una parte de cada padre tal como se muestra en la Figura 19 [38].

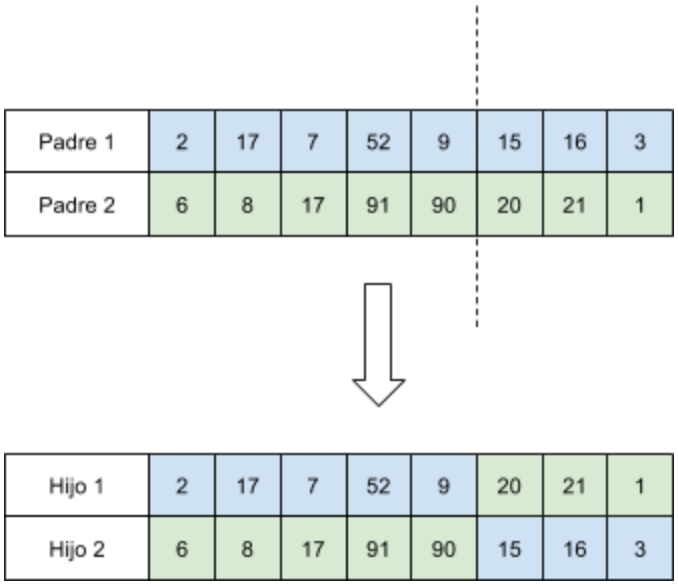

Figura 19 - Single-point Crossover

4.2.3.5 Mutación

Como operador de mutación se elige la *uniform mutation* por ser un operador clásico cuando se trata con codificación de enteros. En esta mutación se intercambia una variable de la solución por un valor aleatorio entre un límite inferior y un límite superior [41].

En la propuesta, esta mutación intercambiará una variable de la solución por una de las siguientes alternativas:

1. Una ruta del conjunto de rutas total, es decir se tomará un número entre *0* y *R* siendo *R* la cantidad de rutas.
2. El valor nulo, que representa que el vehículo no recorre ninguna ruta.

4.2.3.6 Split Delivery

La representación de la solución no indica la entrega para un cliente por una ruta, razón por la cual se debe definir una forma de calcular la distribución de la demanda para cada solución.
Como fue explicado en la Sección 2.1 el *Split Delivery* es solo beneficioso en ciertas situaciones, por lo que se proponen dos estrategias de *Split Delivery*. La primera y más sencilla es simplemente no hacerlo y que el vehículo que arribe primero al cliente entregue toda la demanda requerida por el mismo. Claramente esta solución no podría resolver instancias que requieran realizar *Split Delivery* para satisfacer la demanda de todos los clientes. Por esto, también se debe utilizar la siguiente variante que dada una solución reparta la demanda de los clientes en partes iguales siempre que sea posible. En esta alternativa se decidió tratarlo como una restricción (a la hora de evaluar una potencial solución) en vez de como un objetivo adicional a optimizar. Esto significa que el algoritmo no busca optimizar el *Split Delivery* de las soluciones sino que busca, de alguna manera, que ésta sea válida para así cumplir con las restricciones del problema.

El algoritmo para calcular el *Split Delivery* de una solución es el siguiente:

1. Para cada ruta en la solución
    a. Para cada cliente en la ruta
        i. Se calcula la enésima parte de la demanda a entregar
        ii. Se calcula la capacidad vehicular disponible
        iii. La demanda que se intentará entregar será el mínimo entre la capacidad vehicular disponible y la suma de la enésima parte de la demanda más la demanda sobrante
        iv. Se actualiza la demanda sobrante agregándole lo que no pudo entregar de la enésima parte que le correspondía o restándole el excedente que pudo entregar, según corresponda

En la demanda sobrante se mantiene para cada cliente la demanda que no pudo ser entregada por un vehículo (por cuestiones de capacidad disponible). Esto es para que los siguientes vehículos intenten entregar más que lo que inicialmente les corresponde (que es equitativo entre la cantidad de vehículos que suministran al cliente). Por ejemplo, si se tienen 3 vehículos que reparten al cliente 2, cuya demanda total es 30, cada vehículo intentará entregar inicialmente 10 unidades de demanda. Si sucede que el vehículo 1 no puede entregar todas las unidades estipuladas inicialmente, porque al mismo sólo le quedan 5 unidades disponibles al momento de llegar al cliente, se dejan 5 y se guardan 5 en la estructura. Cuando el vehículo 2 calcule la demanda a entregar será 15 si tiene la capacidad de hacerlo o sino lo máximo posible completando su capacidad y se vuelve a actualizar la estructura. En caso de que al finalizar la función, la estructura quede con demanda sobrante para algunos de los clientes distinta de 0 significa que no se pudo satisfacer la demanda del cliente y eso castigará a la solución cuando se evalúen las restricciones.

Es importante destacar que cuando el algoritmo seleccionador sea invocado, éste realizará dos ejecuciones independientes de NSGA-II, utilizando en cada una de ellas una de las estrategias de *Split Delivery* propuestas. Al finalizar las dos ejecuciones se unen los resultados y se descartan las soluciones dominadas y las repetidas.

5 Arquitectura, Diseño e Implementación

5.1 Arquitectura

En la Figura 20 se presenta la arquitectura básica del sistema implementado. La misma consiste en dos capas:

1. Una capa de presentación encargada de la entrada y salida de datos.
2. Una capa de aplicación que maneja la lógica de negocio y se encarga de la resolución del problema.

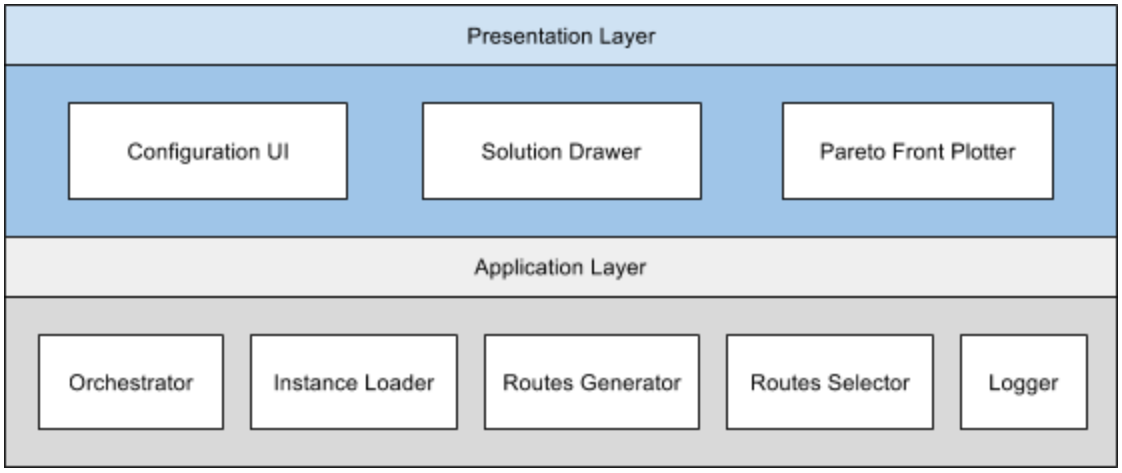

Figura 20. Arquitectura del sistema.

En la capa de presentación se tienen tres módulos: *Configuration UI, Solution Drawer, Pareto Front Plotter*.
Configuration UI, es el módulo que maneja la interacción con el usuario para la configuración y ejecución del sistema. Además, se encarga de la comunicación con los otros dos módulos de la capa de presentación y de invocar a la capa de aplicación.

En la capa de aplicación se tienen cinco módulos. *Orchestrator* es el encargado de orquestar la ejecución del resto tal como se muestra en la Figura 21. Primero invoca al *Instance Loader* para cargar la instancia, luego al *Routes Generator* para la generación de rutas y finalmente al *Routes Selector* para la selección de las mismas y retornar las mejores soluciones encontradas. Toda la información relevante será registrada mediante invocaciones al módulo *Logger*.

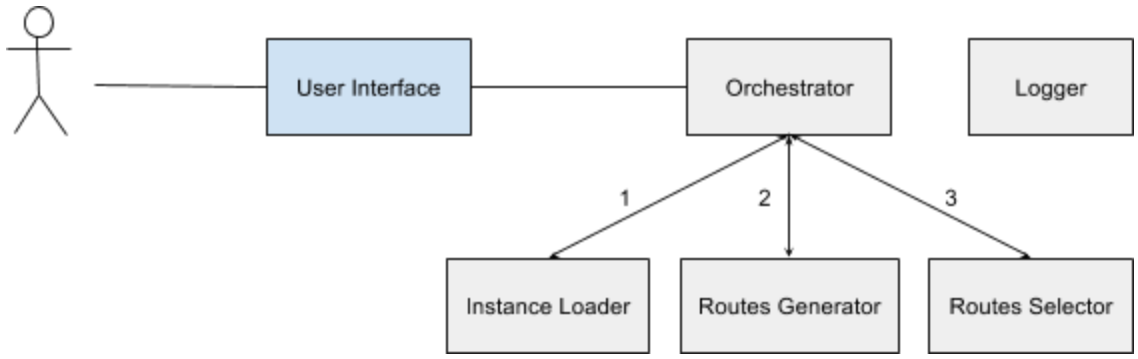

Figura 21. Diagrama de interacción de componentes.

Es importante destacar que la arquitectura se diseñó de manera abstracta (ver Figura 22) para que sea sencillo agregar y reemplazar componentes. Para el módulo *Instance Loader* se podrían desarrollar fácilmente otras implementaciones que carguen instancias con diferentes formatos al de Solomon. También es posible desarrollar nuevos algoritmos de generación de rutas y de selección de rutas y ejecutar cualquier combinación de ellos.

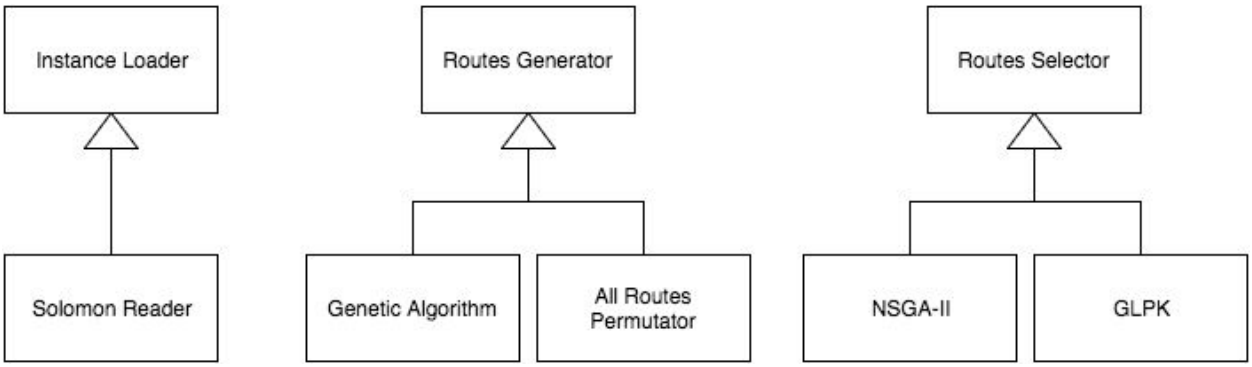

Figura 22 - Módulos principales

5.2 Diseño e Implementación

Se optó por un diseño orientado a objetos ya que facilita la implementación del sistema. Principalmente se consideró el diseño del sistema de forma modular lo cual justifica la utilización de este modelo, ya que permite aislar las responsabilidades de los módulos de su implementación, facilitando el desarrollo y mantenimiento de los mismos.

Se eligió como lenguaje de programación Java (Java SE 8), por múltiples razones:

- Soporte para programación orientada a objetos.
- Multiplataforma. Capacidad de ejecución en cualquier sistema operativo a través de la JVM.
- Compilador JIT (Just in Time compiler). Un JIT tiene acceso a información del hardware en tiempo de ejecución, por lo que puede hacer optimizaciones.

- Vasta cantidad de herramientas y librerías de terceros. Además, muchas de ellas tienen su código fuente abierto por lo que se pueden conocer a fondo e incluso expandirlas.
- Existen varias comunidades dedicadas a otorgar soporte. Además la documentación es muy completa.

5.2.1 Presentation Layer

Para la implementación de los módulos de esta capa se utilizó la librería Swing de Oracle [18].
Mientras se ejecuta el algoritmo en la capa de aplicación, se despliega al usuario una pantalla donde se va registrando en tiempo real información del progreso e información de importancia tal como la cantidad de rutas diferentes generadas.
Al finalizar la ejecución se despliega una pantalla mostrando las diferentes soluciones obtenidas. El usuario podrá seleccionar las soluciones de forma individual para verlas en mayor detalle mediante el uso de *Solution Drawer*. Para la solución seleccionada se desplegará:

- Mapa con la representación gráfica de las rutas
- Detalle de las rutas con los tiempos de arribo a los clientes y la demanda entregada.

A su vez, el tercer módulo *Pareto Front Plotter* permitirá la visualización del frente de Pareto obtenido mediante una Gráfica 3D donde los ejes son cada uno de los objetivos y los puntos son las soluciones encontradas.

5.2.1.1 Configuration UI

Es el módulo encargado de la interacción con el usuario. Se compone de 3 pantallas con funcionalidades específicas que son presentadas secuencialmente a medida que se ejecuta el sistema en el orden detallado en las siguientes subsecciones.

Configurator Screen
Es la pantalla inicial en la cual el usuario debe especificar las diferentes configuraciones para ejecutar el sistema (ver Figura 23):

1. Datos generales. Ubicación de la instancia a resolver, objetivo a optimizar (eficiencia, eficacia, equidad, multi-objetivo), y directorio donde se desea guardar los *logs* de la ejecución del sistema.
2. Generación de rutas. Se debe especificar si se desea generar rutas con el algoritmo genético ó todas las rutas válidas. En caso de utilizar el algoritmo genético, se necesita ingresar la cantidad de ejecuciones del mismo. En caso contrario, se debe especificar el largo máximo deseado de las rutas.
3. Selección de rutas. Se puede realizar con NSGA-II y/o GLPK. En caso que se utilice NSGA-II es necesario especificar la cantidad de evaluaciones deseadas y el tamaño

de la población. En caso de que se utilice GLPK es necesario especificar el tiempo máximo de ejecución con solución encontrada (que no garantiza ser la óptima) y el tiempo máximo de ejecución sin haber encontrado ninguna. Notar que a diferencia de la generación de rutas, se pueden elegir múltiples seleccionadores de rutas simultáneamente para así poder comparar resultados partiendo del mismo conjunto de rutas. La opción GLPK está deshabilitada para el problema multi-objetivo ya que no es capaz de resolver este tipo de problemas.

En caso de cliquear el botón *Limpiar*, todos los valores serán borrados, en caso de cliquear el botón *Ejecutar* se ejecuta en un *thread* separado el sistema con las configuraciones previamente especificadas y en el *thread* que maneja la UI se muestra la pantalla de logging.

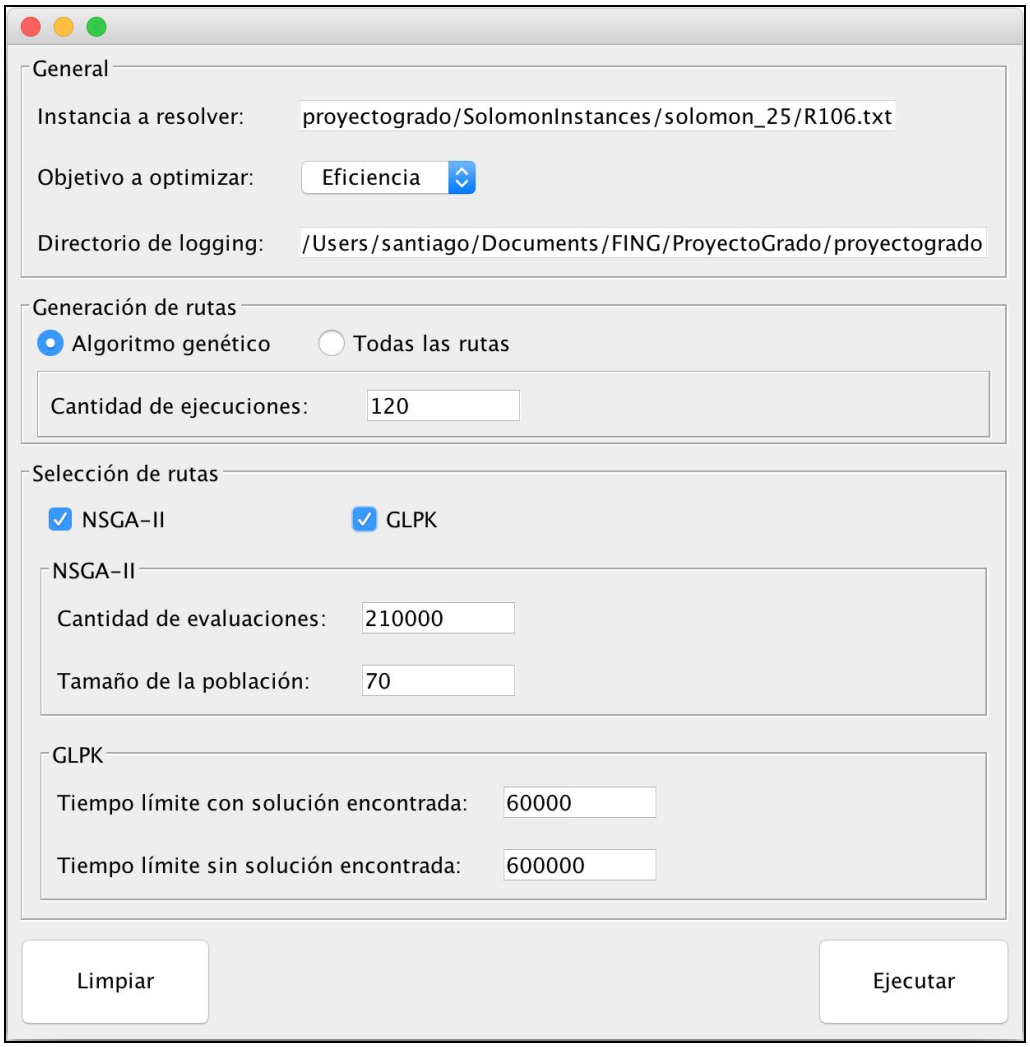

Figura 23 - Pantalla de configuración del sistema

Logger Screen

Esta es la pantalla de *logging*, donde se despliegan los mensajes a medida que se reciben en tiempo real por parte del *thread* de ejecución de los algoritmos.

Contiene un solo botón *Limpiar* que al presionarlo limpia la pantalla y continúa desplegando lo sucedido a partir de ese momento. Ver Figura 24.

Una vez que la ejecución del *thread* de aplicación concluye, se navega a la pantalla donde se muestran los resultados.

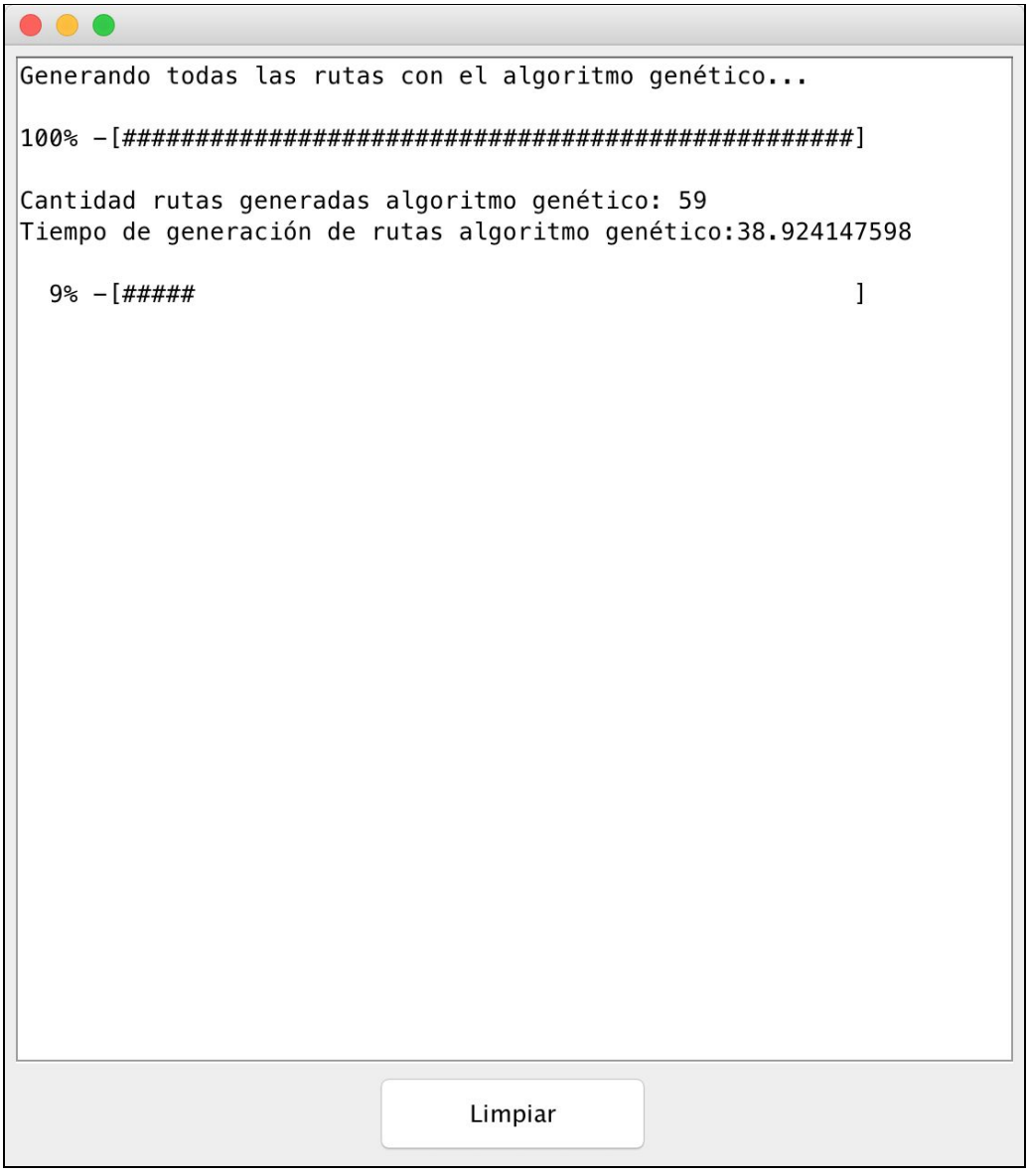

Figura 24 - Pantalla de logging durante la ejecución del sistema

Results Screen

En esta pantalla se presenta una lista con las soluciones encontradas por NSGA-II y/o CPLEX. Para cada solución se muestra con qué seleccionador de rutas fue resuelto, los

identificadores de las rutas elegidas por la solución y los valores para los tres objetivos. Ver Figura 25.

Adicionalmente, esta pantalla tiene los botones *Mostrar resultados* con el cual se invoca al módulo *Solution Drawer* y *Mostrar Gráfica* con el cual se invoca al módulo *Pareto Front Plotter*. En caso de invocarse algunos de estos dos módulos, se abren nuevas ventanas mostrando detalles de la solución de manera que siempre se mantenga abierta la pantalla donde se encuentran todas las soluciones.

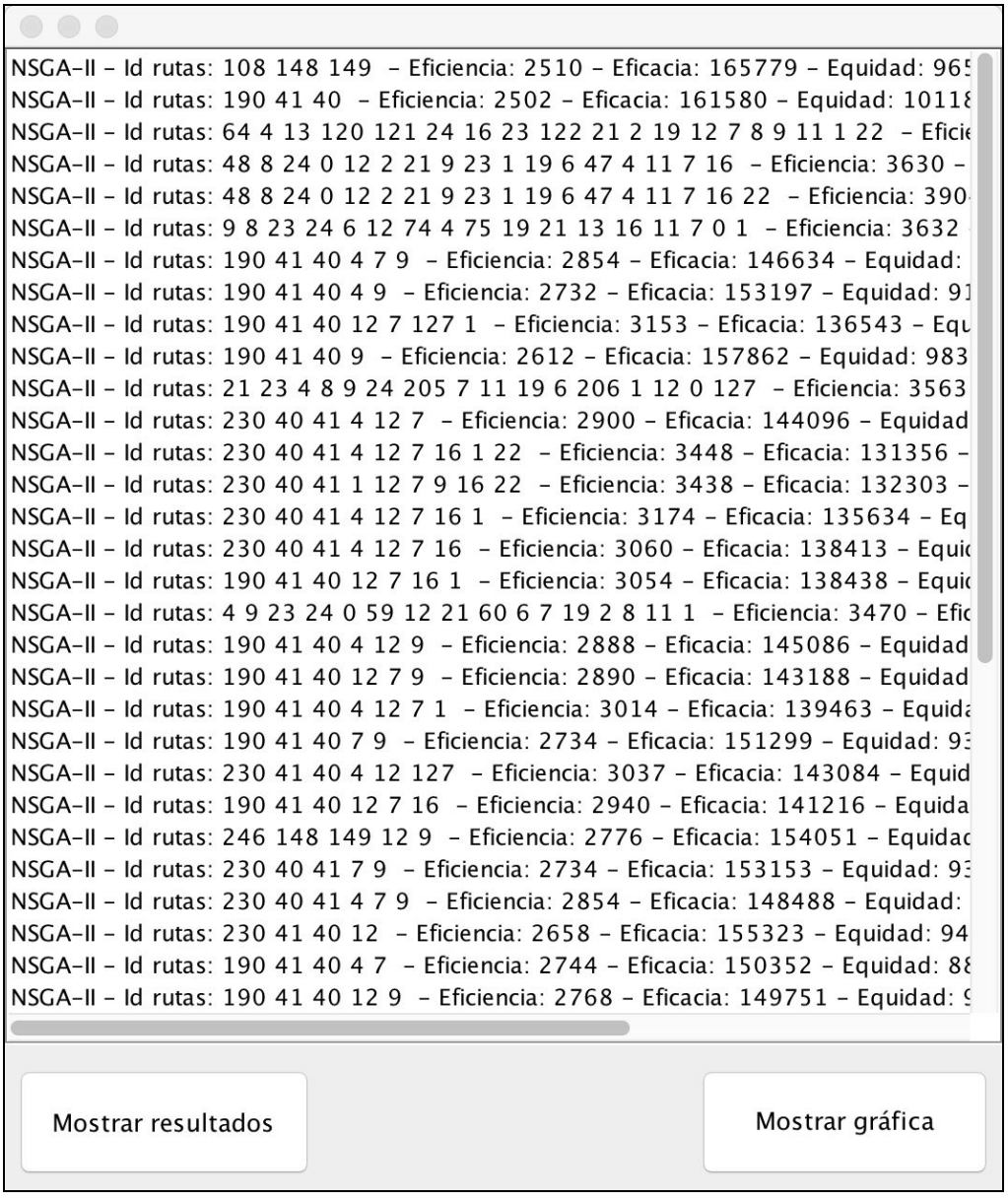

Figura 25 - Pantalla con los resultados de la ejecución del sistema

5.2.1.2 Solution Drawer

Éste módulo es responsable de mostrar el detalle de una solución. Toma como parámetro un individuo con todas sus rutas y nodos y mediante la utilización de la librería JGraphX [23]. Además se despliega otra pantalla detallando de manera escrita las rutas con la información de la entrega a los clientes. En la Figura 26 y Figura 27 se muestra el resultado provisto por el módulo para una solución.

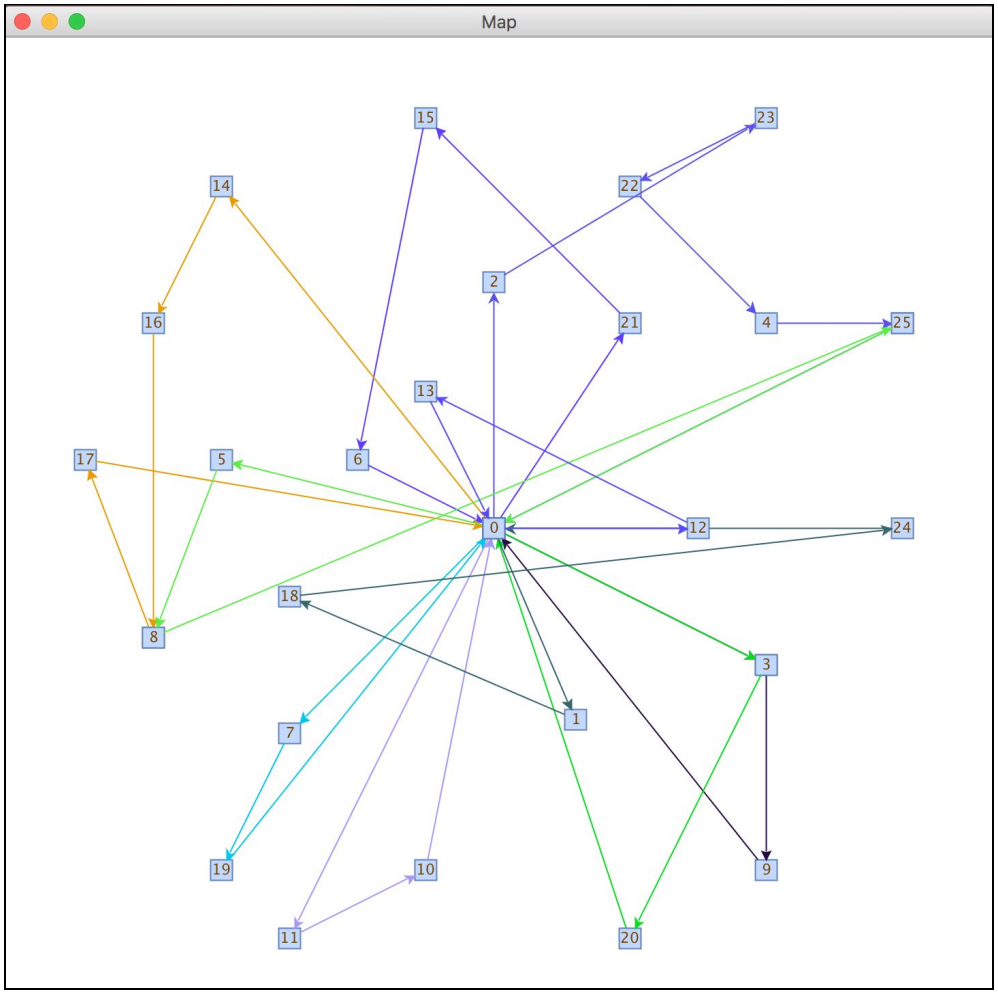

Figura 26 - Mapa de una solución

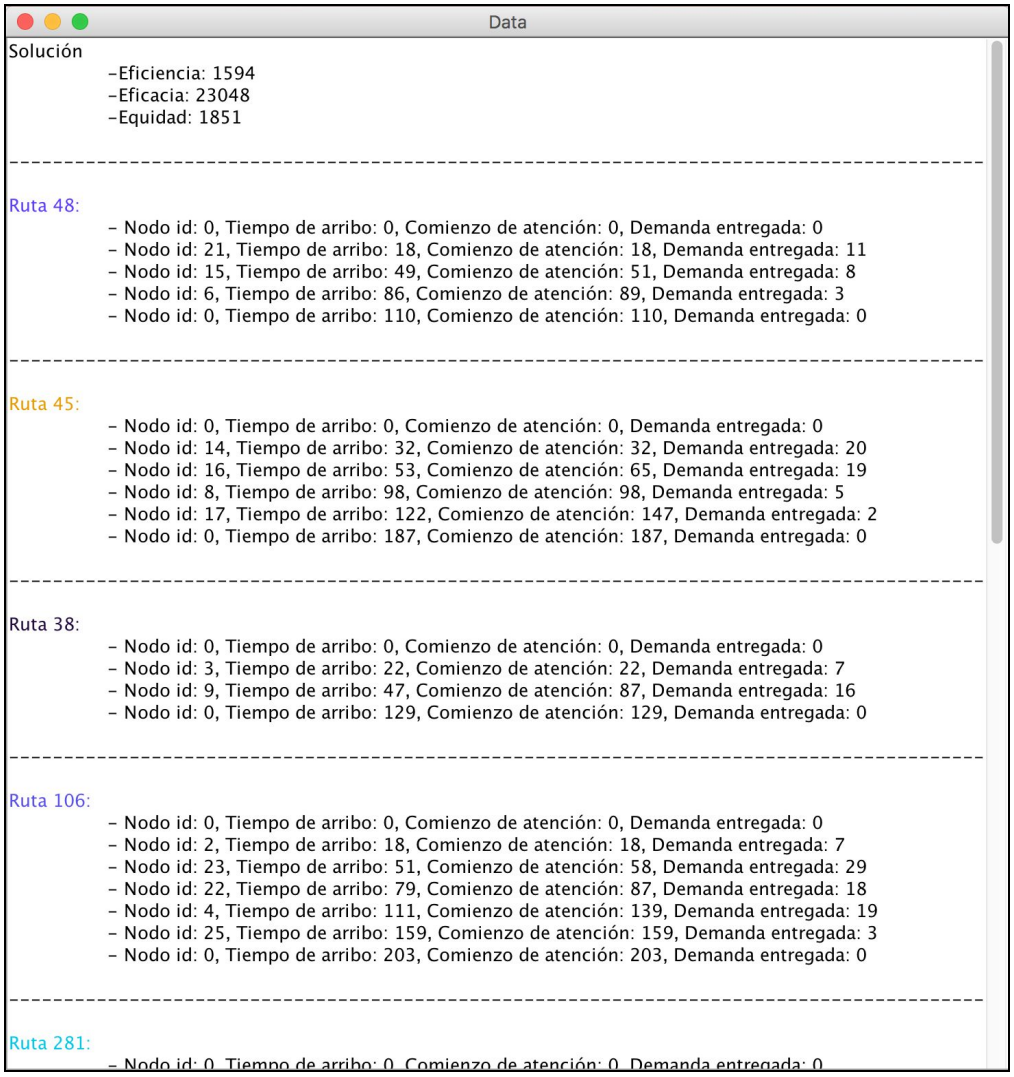

Figura 27 - Datos de una solución

5.2.1.3 Pareto Front Plotter

Es el componente responsable de mostrar gráficamente todas las soluciones multi-objetivo encontradas. Este utiliza la librería JMathPlot [49]. Con esta librería se logró construir una gráfica 3D donde cada eje es uno de los objetivos y por lo tanto las soluciones se representan como puntos. Es una librería muy potente ya que permite acciones sobre la gráfica tales como rotar los ejes, hacer zoom, ver proyección del punto seleccionado, entre otras. En Figura 28 se muestra el resultado provisto por este componente para un conjunto de soluciones.

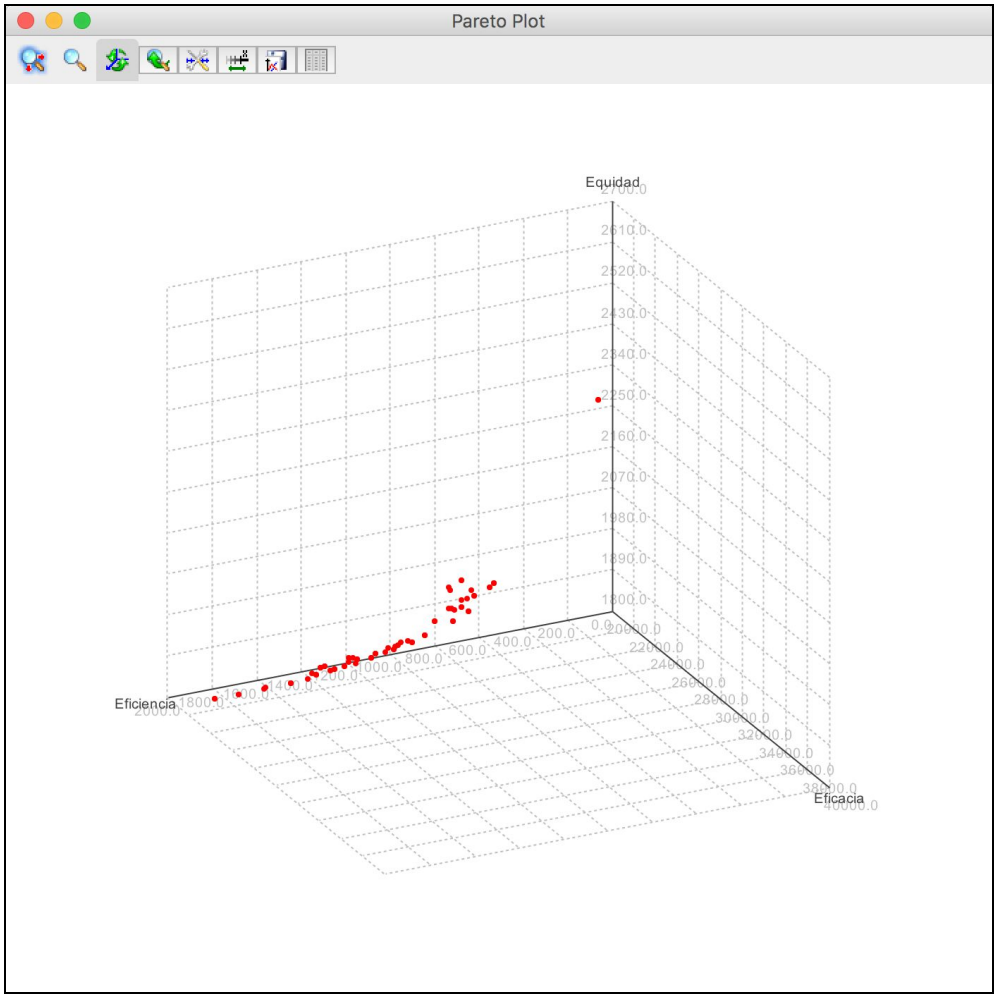

Figura 28 - Gráfica de un conjunto de soluciones

5.2.2 Application Layer

5.2.2.1 Orchestrator

Es el *thread* principal que se encarga de ejecutar cada uno de los componentes necesarios en el orden adecuado. Por la naturaleza del sistema, en que se tienen varios módulos optativos con responsabilidades diferentes, es necesario tener un orquestador que guíe a cada uno de los componentes. Además, actúa como interfaz con la capa de presentación ya que ésta le comunica los parámetros y valores elegidos por el usuario y es el *Orchestrator* quien configura e invoca a cada componente.

5.2.2.2 Instance Loader

Éste módulo abstrae las particularidades de las instancias. Para cada formato de instancia que se desee utilizar se debe implementar un lector que cargue los valores que la interfaz debe proveer:

1. Nodo origen
2. Todos los nodos clientes
3. Capacidad vehicular
4. Cantidad de vehículos

Donde los nodos tienen la siguiente representación:

1. Identificador
2. Tiempo de servicio
3. *Time Window*
4. Demanda
5. Ubicación expresado en coordenadas cartesianas

Luego de cargar toda esta la información, se debe ejecutar la función *buildCostMatrix* la cual calculará a partir de los datos la matriz de costos de viaje para cada par de nodos.

En este caso solo se implementó un lector de instancias de Solomon dado que fue el único formato con el que se trabajó, pero es sencillo de extender para otros.

5.2.2.3 Routes Generator

Éste es el módulo abstracto que se encarga de generar rutas válidas para una instancia. Se realizaron dos implementaciones del mismo:

1. Algoritmo que genera todas las rutas factibles posibles.
2. Algoritmo genético que genera rutas factibles de calidad aplicando el algoritmo detallado en la Sección 4.1.

5.2.2.3.1 Generador de todas las rutas

La única clase de este componente, *RoutePermGenerator,* es la encargada de la generación de todas las rutas factibles para una instancia.
La misma recibe como parámetro de entrada el largo máximo que las rutas pueden tener y se calculan todas las permutaciones para todos los largos posibles. Luego se filtran las rutas que cumplan con las restricciones de *Time Windows* para obtener todas las rutas factibles.
El propósito de este módulo es poder comparar si el conjunto de rutas obtenido por el algoritmo genético generador permite obtener soluciones cercanas a las que se podrían obtener teniendo todas las rutas factibles.

5.2.2.3.2 Algoritmo genético

En la presente sección se describe el algoritmo genético implementado para la generación de rutas. El mismo está a su vez conformado por dos sub-módulos encargados de la generación de rutas con y sin *Split Delivery*. En la Figura 29 se muestra el diseño de la implementación sin *Split Delivery.* El diseño de la variante con *Split Delivery* es igual y será omitida.

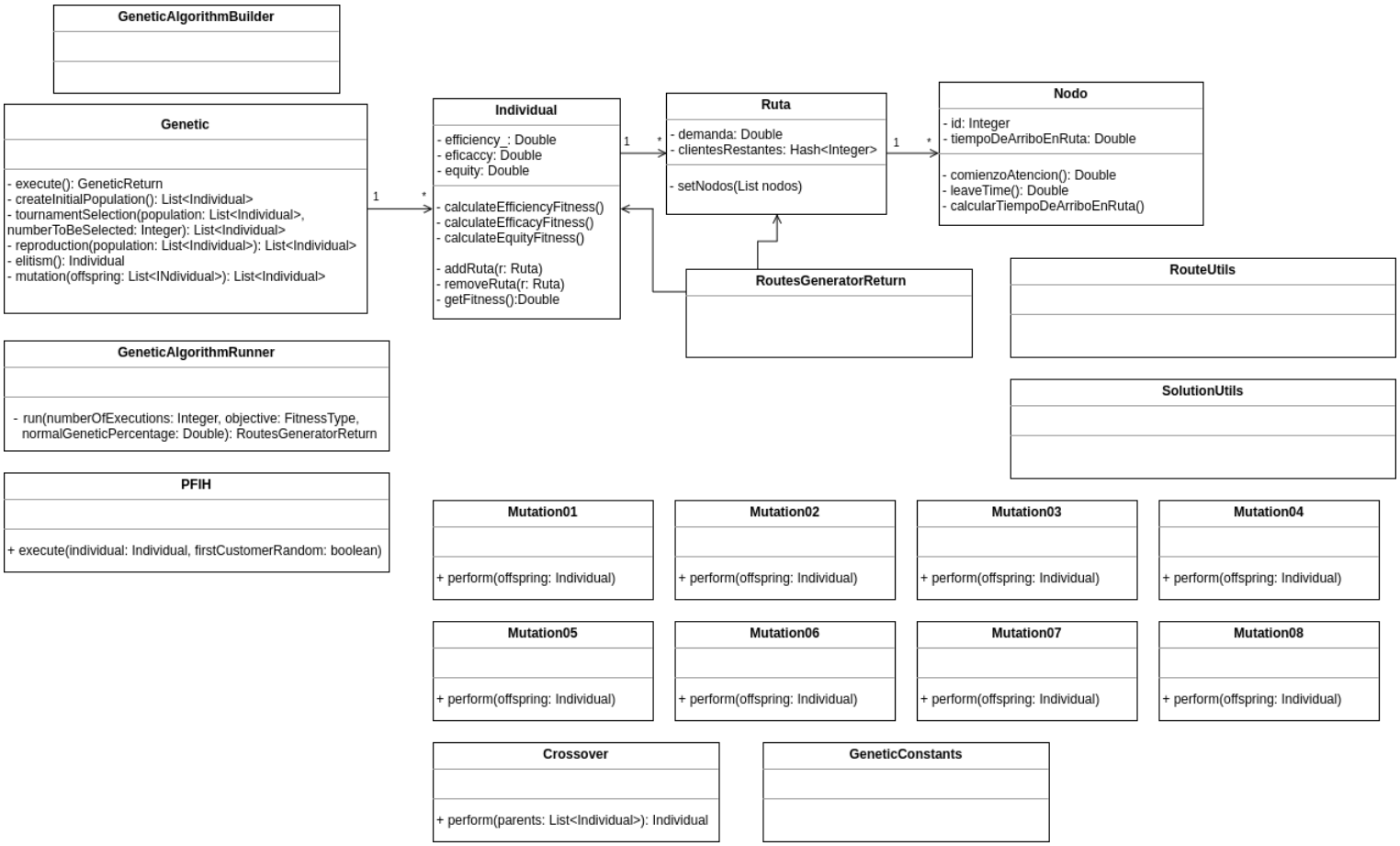

Figura 29 - Diseño de la variante sin Split Delivery. Se observan las clases junto a sus métodos principales.

Genetic Algorithm Runner

Esta clase es el punto de entrada para este módulo. Implementa un único método `run` encargado de dar inicio a la ejecución del algoritmo genético, utilizando los parámetros recibidos como configuración del mismo. Internamente crea y ejecuta dos instancias de `GeneticAlgorithmBuilder`, una para cada una de las variantes. Retorna un objeto de tipo

`RoutesGeneratorReturn` el cual contiene todas las rutas generadas y las soluciones (individuos) encontradas.

`run(Integer numberOfExecutions, FitnessType objective, int minGenerations, int maxGenerations) : RoutesGeneratorReturn`
Recibe como parámetros:

- Cantidad de ejecuciones (islas de evolución).
- Problema a resolver. Aunque el fin principal es generar rutas para el problema multi-objetivo, se permite generar rutas para cada objetivo de forma individual. Para lo anterior se permite seleccionar el modo de trabajo. Según cuál sea el problema a resolver, el algoritmo generará distintos conjuntos de rutas.
  Objetivo eficiencia: se generan sólo rutas para eficiencia.
  Objetivo eficacia: se generan sólo rutas para eficacia.
  Objetivo equidad: se generan sólo rutas para equidad.
  Problema multi-objetivo: se generan de forma equitativa rutas para eficiencia, eficacia, equidad, y "Todos" (los 3 objetivos anteriores ponderados).
- Cantidad mínima y máxima de generaciones. Se ejecuta el `GeneticAlgorithmBuilder` en el modo de generaciones aleatorio utilizando estos valores. Se utiliza 40 y 200 por defecto, respectivamente. Se comprobó empíricamente que dichos valores permiten generar soluciones buenas sin perder variedad.

Si el problema se puede resolver sin la necesidad de utilizar *Split Delivery* de forma obligada:

- Un 70% de las ejecuciones ingresadas son para generar rutas utilizando la variante sin *Split Delivery*.
- Un 30% de las ejecuciones ingresadas para generar rutas utilizando la variante con *Split Delivery*.

Si el problema no se puede resolver sin *Split Delivery*:

- Un 70% de las ejecuciones ingresadas son para generar rutas utilizando la variante con *Split Delivery*. Se considera que esta cantidad es adecuada dado que esta variante toma más tiempo y genera mayor cantidad de rutas diferentes.

Para determinar si una instancia puede ser resuelta sin la necesidad de realizar *Split Delivery* se ejecuta la variante sin *Split Delivery* del PFIH estocástico 1000 veces. Si en ninguna de las ejecuciones se puede obtener una solución válida entonces se asume que la solución sólo puede ser resuelta mediante el uso de la variante con *Split Delivery*. En este caso, se aumenta la cantidad de ejecuciones para esta variante de un 30% a un 70%.

Genetic Algorithm Builder
Esta clase carga y valida todos los parámetros de configuración ingresados por el usuario vía `GeneticAlgorithmRunner`. Los parámetros se detallan a continuación:

- Cantidad de ejecuciones (islas de evolución).

- Problema a resolver.
- Cantidad de generaciones de cada ejecución. Se tienen dos modos de funcionamiento: exacto y aleatorio. En el exacto se ingresa el número de generaciones que se desean. Si el número de generaciones fuera un número chico, las soluciones no podrían converger al óptimo. Si el número de generaciones fuera un número grande, la mayoría de las soluciones podrían converger a lo mismo. En el aleatorio se ingresa un rango de números (e.g. entre 25 y 170), del cual se seleccionará aleatoriamente la cantidad. El último modo permite que haya mayor variedad entre las rutas de la solución, razón por la cuál es el modo que utiliza el `GeneticAlgorithmRunner`.
- Validación de rutas. Si está activado el algoritmo verificará que las rutas sean válidas luego de cada generación. Por defecto está desactivado. Se utilizó en las instancias tempranas del desarrollo.

Además se cargan ciertos parámetros estáticos. Los mismos son los que obtuvieron mejor resultado en Alvarenga (2005) [2]:

- Porcentaje de la población que será seleccionada para crossover y mutación (120% ya que en crossover cada dos padres se genera un sólo hijo), y solamente para mutación (40%).
- Tamaño de la población: 50.
- Cantidad de soluciones aleatorias obtenidas por el torneo múltiplo k-way. K = 3.

Internamente crea la clase encargada de la ejecución de la variante seleccionada del algoritmo `Genetic`. La cantidad de rutas generadas depende de la cantidad de ejecuciones aisladas que se ingresaron.

Genetic

Esta es la clase que implementa el método principal de la variación sin *Split Delivery* del algoritmo genético. Aquí se ejecuta el ciclo del algoritmo generador y al finalizar, se retorna una estructura de tipo `RoutesGeneratorReturn` la cual contiene dos conjuntos: uno con todas las rutas encontradas y otro con las mejores soluciones (individuos).

- `execute(): RoutesGeneratorReturn`

A grandes rasgos la implementación del método se puede especificar con el siguiente pseudocódigo:

1. Mientras no se supere la cantidad de ejecuciones
    a. Crear población inicial y evaluarla
    b. Mientras no se supere la cantidad de generaciones
        i. Se obtiene el individuo élite *E* y se realiza una copia *Ec*
        ii. Generar dos conjuntos de individuos *C1* y *C2*
        iii. 120% y 40% del tamaño de la población son obtenidos mediante tournament selection y agregados a *C1* y *C2* respectivamente

iv. Se realiza *crossover* sobre *C1*
v. Se realiza mutación sobre *C1*, *C2* y *E*
vi. Se genera la nueva población creada a partir de *C1*, *C2*, *E* y *Ec*
vii. Se evalúa la nueva población

c. Se obtiene el individuo élite. Se agrega al conjunto de mejores individuos. Sus rutas se agregan al conjunto de mejores rutas

2. Se retorna una estructura `RoutesGeneratorReturn` con los mejores individuos y todas las rutas

A continuación se detallan los demas metodos principales de esta clase:

- `createInitialPopulation() : List<Individual>` : Crea una solución inicial utilizando PFIH estocástico. Si no puede crearla luego de 1000 intentos se lanza una excepción `SolutionWithoutSplitNotFeasible`.
- `evaluatePopulation(List<Individual> p) : List<Individual>` : Se ejecuta al final de cada generación, con el fin de calcular los nuevos fitness de cada individuo de la próxima población.
- `tournamentSelection(List<Individual> p, Integer numberOfIndividualsToBeSelected) : List<Individual>` : Retorna `numberOfIndividualsToBeSelected` individuos utilizando el método de selección de torneo múltiple k-way.
- `reproduction(List<Individual> p) : List<Individual>` : Se toman los padres de a pares y se aplica *crossover* para generar un hijo que será agregado a la población.
- `elitism() : Individual` : Retorna el mejor individuo de la población actual.
- `mutation(List<Individual> p) : List<Individual>` : Aplica las mutaciones propuestas.

La variante con *Split Delivery* es análoga.

PFIH

En esta clase se implementa el PFIH estocástico. A su vez, se brinda la posibilidad de ejecución del PFIH determinístico propuesto en Solomon mediante un parámetro. Como se indicó en la Sección 4.1, la única diferencia entre estas dos variantes es que en el caso del PFIH estocástico el elemento inicial se elige de forma aleatoria, contrario al PFIH de Solomon en el cual se elige según una función determinista.

La clase se compone de un único método:

- `execute(Individual individual, Boolean firstCustomerRandom)` :
  Toma como parámetros un individuo recién creado (sin rutas) sobre el cual se ejecutará el PFIH y un parámetro booleano el cual indica la variante del PFIH que se debe ejecutar.

Al finalizar la ejecución del método, el individuo ingresado se convierte en una solución inicial para el problema.

La variante con *Split Delivery* tiene una diferencia crucial. Los vehículos pueden aceptar un cliente incluso si no cubren su capacidad completa. Para lo anterior se tuvo que implementar una estructura auxiliar para saber no sólo que clientes faltan por atender, sino también cuánta demanda falta entregarles. Al momento de asignar un cliente al vehículo, se entrega toda la demanda posible. Si el vehículo no es capaz de suplirla, se dejará al cliente lo máximo posible y el resto será suplido por otros vehículos.

Crossover

El operador de crossover se implementa en el único método que compone esta clase:

- `perform(List<Individual> parents) : Individual`

El método toma como parámetros una lista de individuos correspondiente a los 2 padres a los cuales se les quiere aplicar el operador de *crossover*. Como resultado retorna un hijo compuesto por rutas pertenecientes a los padres, o una copia del primero de los padres en caso de que el hijo generado este compuesto por más cantidad de rutas que los vehículos que se tienen.

Pseudocódigo del algoritmo crossover:

Crossover (Individuo *padre1*, Individuo *padre2*) : Individuo

1. Crear nuevo individuo hijo vacío *INDh*.
2. Inicializar *ningunaRutaPosible* = false;
3. Mientras (hay rutas en *padre1* y *padre2,* y *ningunaRutaPosible* == false)
    a. *ningunaRutaPosible* = true;
    b. Obtener ruta aleatoria de *padre1*;
    c. Si es factible, insertar ruta en *INDh*, retirarla de *padre1* y *ningunaRutaPosible* = false.
    d. Obtener ruta aleatoria de *padre2*;
    e. Si es factible, insertar ruta en *INDh*, retirarla de *padre2* y *ningunaRutaPosible* = false.
4. Eliminar todos los clientes ya atendidos en *INDh* del *padre1*, generando algunas rutas vacías y otras no vacías.
5. Insertar todas las rutas factibles de *padre1* en *INDh*.
6. Si hay clientes sin atender en *INDh*.
    a. Si (*fitness* == Eficiencia) o (*fitness* == "Todos" con una probabilidad del 50%):
    Insertar clientes restantes en rutas existentes en las posiciones de menor costo.
    b. Si hay clientes sin atender en *INDh*. Ejecutar PFIH en *INDh*.

Para que la inserción de una ruta sea factible, ningún cliente de la misma debe estar ya atendido en el individuo hijo. Además se deben cumplir las restricciones de capacidad y

Time Windows. Según lo explicado en la Sección 2, cuando el objetivo es eficacia o equidad, las rutas tienden a ser lo más cortas posible ya que se intenta utilizar todos los vehículos disponibles. Se observó que insertar clientes restantes en las rutas existentes no es entonces beneficioso para ellos. Es por esto que lo mismo se realiza cuando el *fitness* es eficiencia, o la mitad de las veces cuando el *fitness* es "Todos".

La variante con *Split Delivery* es análoga.

Mutaciones

Las 8 mutaciones propuestas se implementan en clases independientes compuestas por un único método el cual aplica dicha mutación a un individuo.

- `perform(Individual offspring)`

Pseudocódigo de cómo se aplican las diferentes mutaciones a la población (offspring) en el paso mutación:

```
Mutation (List<Individuo> población) : List<Individuo>
```

1. Ejecutar Mutación1 CANT_VECES_MUTACION1 sobre individuo aleatorio.
2. Ejecutar Mutación2 CANT_VECES_MUTACION2 sobre individuo aleatorio.
3. Ejecutar Mutación3 CANT_VECES_MUTACION3 sobre individuo aleatorio.
4. Ejecutar Mutación4 CANT_VECES_MUTACION4 sobre individuo aleatorio.
5. Ejecutar Mutación5 CANT_VECES_MUTACION5 sobre individuo aleatorio.
6. Ejecutar Mutación6 CANT_VECES_MUTACION6 sobre individuo aleatorio.
7. Ejecutar Mutación7 CANT_VECES_MUTACION7 sobre individuo aleatorio.
8. Ejecutar Mutación8 CANT_VECES_MUTACION8 sobre individuo aleatorio.
9. Retornar población mutada.

Las mutaciones individuales se ejecutan de forma secuencial, una determinada cantidad de veces. Cada vez, se selecciona un individuo de forma aleatoria de la población, al cual se le aplica la mutación en cuestión. La cantidad de ejecuciones utilizada para cada mutación individual es: {20, 10, 1, 20, 2, 1, 30, 1}. Las mismas fueron calculadas empíricamente en Alvarenga et a. (2007) [3]. Se optó por mantener la misma cantidad de ejecuciones para todos los casos aunque existan varias funciones de *fitness*. Se decidió que sea así luego de observar que todas las mutaciones anteriormente descritas podrían llegar a mejorar todos los *fitness* con los que se trabaja.

Mutacion 01 - Random customer migration

Este operador elige un vehículo y un cliente asociado a él de forma aleatoria. Se intenta migrar el cliente a otro vehículo no vacío. Si la inserción resulta en una ruta factible, se acepta independientemente del nuevo costo.

El pseudocódigo del método `perform` *se detalla a continuación:*

1. Se elige una ruta *R1* de forma aleatoria.
2. Se elige un cliente *C*, perteneciente a *R1,* de forma aleatoria.

3. Se elige una ruta *R2* de forma aleatoria.
4. Mientras haya clientes de *R2* por recorrer y no se encontró un lugar válido:
    a. Si es posible insertar el cliente en el lugar actual manteniendo la ruta válida:
        i. Se guarda la posición y se sale de la iteración.
        ii. Si no, se avanza a la siguiente posición.
5. Si se encontró una posición válida:
    a. Se remueve el cliente de la ruta *R1.* Si era el único cliente en la ruta, ésta se remueve del individuo.
    b. Se inserta el cliente en la posición encontrada en la ruta *R2*.
    c. Se recalculan los tiempos de arribo para cada cliente de las rutas *R1* y *R2*.

Mutacion 02 - Bringing the best customer
Este operador elige un vehículo de forma aleatoria y busca e inserta el cliente de los otros vehículos en la posición que resulte en el menor incremento de tiempo de viaje.

El pseudocódigo del método `perform` *se detalla a continuación:*
1. Se elige una ruta *R1* de forma aleatoria.
2. Mientras haya rutas del individuo por recorrer:
    a. Se obtiene la ruta R_i de la iteración actual.
    b. Si la ruta es igual a la seleccionada, se procede a la siguiente iteración.
    c. Mientras haya clientes de la ruta R_i por recorrer:
        i. Se obtiene el cliente C_i de la iteración actual.
        ii. Mientras haya clientes de la ruta *R1* por recorrer:
            1. Se obtiene el cliente C_j de la iteración actual.
            2. Si es válido insertar el cliente C_i a continuación de C_j:
                a. Si el incremento en el tiempo de viaje es menor al encontrado hasta el momento, se guardan el id de la ruta R_i y la posición actual.
3. Si se encontró alguna ruta y posición válidas:
    a. Se remueve el cliente de la ruta *R1*. Si era el único cliente en la ruta, ésta se remueve del individuo.
    b. Se inserta el cliente en la posición encontrada en la ruta R_i.
    c. Se recalculan los tiempos de arribo para cada cliente de las rutas *R1* y R_i.

Mutacion 03 - Re-insertion using stochastic PFIH
Este operador elige una ruta de forma aleatoria y lo mantiene intacto. Luego aplica el procedimiento PFIH estocástico con los clientes de todas las rutas restantes.

El pseudocódigo del método `perform` *se detalla a continuación:*
1. Se elige una ruta *R1* de forma aleatoria.
2. Todas las rutas distintas a *R1* se vacian dejando sus clientes sin asignar.
3. Sobre el individuo resultante, que ahora solo contiene a *R1*, se aplica el PFIH estocástico descrito en la Sección 4.1.2.1.

4. Si el individuo resultante del PFIH contiene más rutas que los vehículos disponibles, se descartan los cambios realizados sobre el individuo.

Mutacion 04 - Similar customer exchange
Este operador elige un vehículo de forma aleatoria, y busca un cliente con *Time Window* similar en el resto de vehículos para intentar un intercambio. Se considera similar, a la *Time Window* que representa la mínima variación respecto al cliente con que se compara. Es decir, la que minimice a_i - a_j, siendo a_i y a_j los tiempos de inicio de la *Time Window* de los clientes *i* y *j*, respectivamente.

El pseudocódigo del método `perform` *se detalla a continuación:*

1. Se elige una ruta *R1* de forma aleatoria.
2. Mientras haya rutas del individuo por recorrer:
    a. Se obtiene la ruta R_i de la iteración actual.
    b. Si R_i es *R1*, se procede a la siguiente iteración.
    c. Mientras haya clientes de la ruta por recorrer:
        i. Se obtiene el cliente C_i de la iteración actual.
        ii. Mientras haya clientes de la ruta *R1* por recorrer:
            1. Se obtiene el cliente C_j de la iteración actual.
            2. Si la diferencia entre los inicios de las *Time Windows* de C_i y C_j es menor a la encontrada hasta el momento, se guardan el id de la ruta R_i y la posición actual.
3. Si se encontró alguna ruta y posición válidas:
    a. Se intercambian los clientes C_i y C_j.
    b. Se recalculan los tiempos de arribo para cada cliente de las rutas *R1* y R_i.

Mutacion 05 - Customer Exchange with positive gain
Se eligen dos rutas aleatorias y se analizan todas las posibilidades de intercambio de clientes; se realiza la mejor de todas sólo si resulta en una reducción del tiempo total de viaje. Este operador es costoso en términos de tiempo procesamiento, *O(nm)*, donde *n* y *m* son el número de clientes atendidos por dos vehículos. Sin embargo, es muy importante para mejorar la convergencia del algoritmo genético.

El pseudocódigo del método `perform` *se detalla a continuación:*

1. Se elige una ruta *R1* de forma aleatoria.
2. Se elige una ruta *R2* de forma aleatoria.
3. Mientras haya clientes de la ruta *R1* por recorrer:
    a. Se obtiene el cliente C_i de la iteración actual.
    b. Mientras haya clientes de la ruta *R2* por recorrer:
        i. Se obtiene el cliente C_j de la iteración actual.
        ii. Si la variación del tiempo de arribo resultante de intercambiar los clientes C_i y C_j es menor a la encontrada hasta el momento, se guardan las posiciones de ambos clientes.

4. Si se encontraron clientes para un intercambio válido:
    a. Se intercambian los clientes C_i y C_j.
    b. Se recalculan los tiempos de arribo para cada cliente de las rutas *R1* y *R2*.

Mutation 06 - Merge two routes

Este operador elige dos rutas diferentes de forma aleatoria e intenta unirlas de una forma aleatoria. Los clientes restantes se reinsertan en otras rutas o en una nueva. Si de igual forma siguen quedando clientes sin ruta, se aplica PFIH estocástico.

El pseudocódigo del método `perform` *se detalla a continuación:*

1. Se elige una ruta *R1* de forma aleatoria.
2. Se elige una ruta *R2*, diferente a *R1*, de forma aleatoria.
3. Mientras haya clientes de la ruta *R1*
    a. Se elige un cliente *C1* de la ruta *R1* de forma aleatoria.
    b. Mientras haya posiciones posibles para insertar *C1* en la ruta *R2*:
        i. Se obtiene una posición *P1* válida de *R2*.
        ii. Si se cumple que se puede insertar el cliente *C1* en la posición *P1* de la ruta *R2*, se inserta.
4. Si todavía existen clientes de la ruta *R1*:
    a. Si (*fitness* == Eficiencia) o (*fitness* == "Todos" con una probabilidad del 50%):
    Insertar clientes restantes en rutas existentes en las posiciones de menor costo.
    b. Si aún quedan clientes en la ruta *R1*, se ejecuta el PFIH estocástico para generar nuevas rutas a agregar al individuo con estos clientes.
5. Si el individuo resultante contiene más rutas que los vehículos disponibles, se descartan los cambios realizados sobre el individuo.

Mutation 07 - Reinserting customer

Este operador elige un cliente de forma aleatoria, de una ruta seleccionada también de forma aleatoria, y luego se inserta en la mejor posición en la misma ruta.

El pseudocódigo del método `perform` *se detalla a continuación:*

1. Se elige una ruta *R1* de forma aleatoria.
2. Se elige un cliente *C1* de la ruta *R1* de forma aleatoria.
3. Se remueve el cliente *C1* de la ruta *R1*.
4. Se recalculan los tiempos de arribo para cada cliente de la ruta *R1*.
5. Mientras haya clientes en la ruta *R1*:
    a. Si el incremento en el tiempo de viaje resultante de insertar el cliente *C1* en la posición actual es menor al encontrado hasta el momento, se guarda la posición actual.
6. Se inserta el cliente *C1* en la posición guardada.

Mutation 08 - Route partitioning
Este operador elige una ruta y un cliente asociado a ella de forma aleatoria y luego la divide en otras dos, tomando el cliente elegido como posición de corte.

El pseudocódigo del método `perform` *se detalla a continuación:*

1. Se elige una ruta *R1*, que atienda al menos dos clientes, de forma aleatoria.
2. Se elige un cliente *C1* de la ruta *R1* de forma aleatoria.
3. Mientras haya clientes de la ruta *R1* partiendo de la posición en que se encuentra *C1*:
    a. Se obtiene el cliente C_j de la iteración actual.
    b. Se inserta el cliente C_j en la nueva ruta *R2*.
    c. Se remueve el cliente C_j de la ruta *R1*.
4. Se recalculan los tiempos de arribo para los clientes de la rutas *R1* y *R2*.
5. Se inserta la ruta *R2* en el individuo.

Variante con Split Delivery
Dado que al momento de realizar inserciones la ruta destino puede no cubrir la demanda completa del cliente, se deben implementar variantes con *Split Delivery* para todas las mutaciones teniendo en cuenta lo siguiente:

- La ruta destino debe ser capaz de suplir la demanda que la ruta origen le otorga al cliente al momento del intercambio.
- Si el cliente a insertar ya existe en la ruta destino, entonces solamente se aumenta la demanda que la misma le entrega.

5.2.2.4 Routes Selector

Éste es el módulo abstracto que se encarga de seleccionar un conjunto de rutas, partiendo de rutas obtenidas a través de alguno de los algoritmos del módulo generador de rutas, que conforme una solución de la instancia. Se realizaron dos implementaciones del mismo:

1. GLPK. La implementación consiste en una encapsulación de la librería GLPK para Java que es utilizada para resolver el problema.
2. NSGA-II. Corresponde a la implementación del algoritmo genético NSGA-II descrito en la Sección 4.2.

5.2.2.4.1 GLPK

El GNU Linear Programming Kit (GLPK) es una librería de software concebida para resolver a gran escala problemas de programación lineal (LP), programación entera mixta (MIP), así como otros relacionados. El paquete forma parte del GNU Project y se publica bajo la licencia General Public Licence (GPL) [20].

La librería está escrita en el lenguaje C pero se dispone de interfaces para su utilización desde otros lenguajes. En el caso de Java, la interfaz es provista por el proyecto independiente GLPK Java.
GLPK utiliza el método *simplex* y el *método de punto interior primal-dual* para problemas no enteros y el algoritmo de *branch and bound* junto con los cortes enteros mixtos de Gomory para problemas de enteros mixtos.
Los problemas se pueden modelar en el lenguaje GNU MathProg (antes conocido como GMPL), el cual es un subconjunto de AMPL, cuyos modelos se pueden resolver con el solver independiente *GLPSOL*.

Modelo AMPL
Para poder resolver el problema de selección con un software de optimización se requirió implementar el mismo en el lenguaje de modelado AMPL [4]. Este lenguaje permite especificar los datos, variables, objetivos y restricciones del problema a optimizar.

A partir del modelo propuesto por Huang et al [22] se implementan cuatro modelos similares cuya diferencia más sustancial es la función objetivo. A raíz de lo anterior se implementan cuatro archivos AMPL: para eficiencia, eficacia, equidad y multi-objetivo. Los datos se cargan a partir de otro archivo cuya ubicación se debe indicar a la hora de resolver la instancia.
En la Figura 30 se muestra como ejemplo el modelado de Z_2, donde la función objetivo busca optimizar la eficacia.

```
param n, integer, >= 3; /* number of nodes */
set V := 0..n; /* set of nodes */
param f, integer, >= 0; /* number of routes - 1 */
set R := 0..f; /* set of routes */
param t, integer >= 0; /* upper bound for time */
set T := 1..t; /* set of hours */
param k, integer, >= 1; /* number of vehicles */
param C, integer >= 0; /* capacity of vehicles */
param c{r in R}; /* cost of route r */
param b{(i,r) in V cross R}; /* b[i,r] = 1 means that the node i is present in the route R */
param a{(i,r) in V cross R}; /* a[i,r] = 6 means that the node i is delivered at the time 6 in the route R */
var x{r in R}, binary; /* x[r] = 1 means that route R is selected */
var y{(i,r) in V cross R}, integer, >= 0; /* determines number of pallets delivered to node i by route r */
param d{i in V}; /* d[i] is the demand of node i */
minimize objective: sum{i in V, r in R} a[i,r] * y[i,r]; /* the objective function */
s.t. first: sum{r in R}(x[r]) <= k; /* number of routes must be at most the number of vehicles */
s.t. second{i in V, r in R}: y[i,r] <= d[i] * b[i,r] * x[r]; /* number of delivered pallets to a node in a route can be at most the demand of the
node */
s.t. third{r in R}: sum{i in V}(y[i,r]) <= C; /* number of delivered pallets within a route can't be higher than the vehicles capacity */
s.t. fourth{i in V}: sum{r in R}(y[i,r]) = d[i]; /* the demand of all node is satisfied */
s.t. fifth{(i,j) in V cross V: 0 < i and i < j}: sum{r in R}(x[r] * b[i,r] * b[j,r]) <= 1; /* the same pair of nodes can't be in more than one route
(optimization just for efficiency and efficacy)*/
solve;/* solve model */
```

Figura 30 - Modelo de Z2 en AMPL

Cabe destacar que para el caso multi-objetivo (mediante suma ponderada) se agregaron dos parámetros adicionales en los datos. Estos son:

1. Normalizadores. Para cada objetivo se obtiene el valor óptimo con los mismos datos y éste será utilizado para normalizar al objetivo en la suma ponderada. Esto es necesario porque los objetivos tienen diferentes magnitudes y, si no se normaliza, uno podría tener mayor peso que los demás.
2. Ponderaciones. Dado que interesa correr las instancias con diferentes ponderaciones de los objetivos, es necesario indicarle en los datos los diferentes valores.

La ecuación (19) es la función objetivo para el caso multi-objetivo.

$$min \ \frac{p_1 \times Z_1}{o_1} + \frac{p_2 \times Z_2}{o_2} + \frac{p_3 \times Z_3}{o_3} \tag{19}$$

Siendo p_i la ponderación del objetivo *i* , Z_i la función para el objetivo *i* y o_i el valor óptimo de la instancia para el objetivo *i*.

Todos los modelos retornan el valor de los tres objetivos en cuestión, ya que cuando se optimiza uno de ellos es importante saber cuánto se penaliza al resto. Además se imprime la información de las rutas utilizadas y la cantidad de demanda entregada a cada cliente por cada ruta, lo cual es particularmente necesario por la existencia de *Split Delivery*.

Ejecución
Con el fin de poder realizar la selección de rutas luego de la generación de forma automática se utilizó la librería de GLPK para Java [21].
La misma permite comunicarse con GLPK mediante el uso de la *Java Native Interface* (JNI).
Luego de generar las rutas, se ejecuta un método que partiendo de ellas, genera un archivo con los datos en el formato que el modelo necesita.

El método encargado de la resolución es el siguiente:
`solve(String modelPath, String dataPath, Integer timeLimit, Integer max_time_without_sol) : GlpkSolution`
Recibe como parámetros :

- Ubicación del archivo con el modelo a resolver.
- Ubicación del archivo con los datos.
- Tiempo límite si luego de su transcurso se ha encontrado al menos una solución válida.
- Tiempo límite sin encontrar ninguna solución.

Retorna un objeto del tipo `GlpkSolution` que contiene las rutas elegidas y la cantidad de demanda que cada una le entrega a cada cliente. El fin principal de este seleccionador es poder encontrar los óptimos para instancias pequeñas para tomar como punto de referencia.

5.2.2.4.2 NSGA-II

La implementación de este módulo se basa en una librería realizada por terceros llamada jMetal. Dicha librería provee una API Java orientada a optimización multi-objetivo, proveyendo implementaciones de problemas y algoritmos clásicos, pero también brindando la posibilidad de implementar propios.
A continuación se detallan las clases principales de la librería y su funcionamiento.

jMetal
La arquitectura de jMetal se basa en 4 interfaces. En el diagrama de la Figura 31 se muestra el funcionamiento básico de jMetal: un algoritmo `Algorithm` resuelve un problema `Problem` manipulando un conjunto de objetos de potenciales soluciones `Solution` haciendo uso de múltiples operadores `Operator`. La interfaz `Solution` representa los individuos en el caso de algoritmos evolutivos, como lo es NSGA-II.

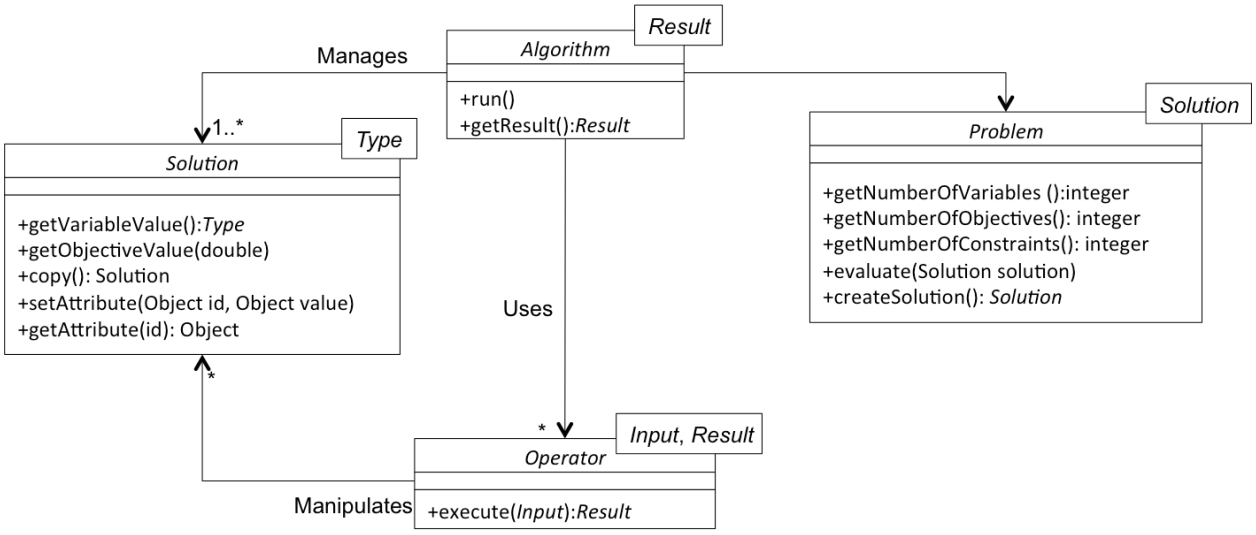

Figura 31 - Funcionamiento jMetal

Para el caso del algoritmo genético elegido, NSGA-II, jMetal brinda una estructura genérica base y requiere únicamente de la implementación de módulos específicos al problema.

En el diagrama de la Figura 32 se muestra la clase que implementa el algoritmo junto con las relaciones con otros componentes y los métodos más importantes. Como se puede ver, consiste en una herencia de la clase `AbstractGeneticAlgorithm` y a su vez ésta hereda de la clase `AbstractEvolutionaryAlgorithm`.

En la Figura 33 se puede apreciar la aplicación de la arquitectura base para NSGA-II. En la misma, se presenta la estructura básica del diseño implementado para la selección de rutas con NSGA-II, cuyos componentes se describen a continuación. Los componentes marcados de color amarillo corresponden a componentes pertenecientes a la librería jMetal.

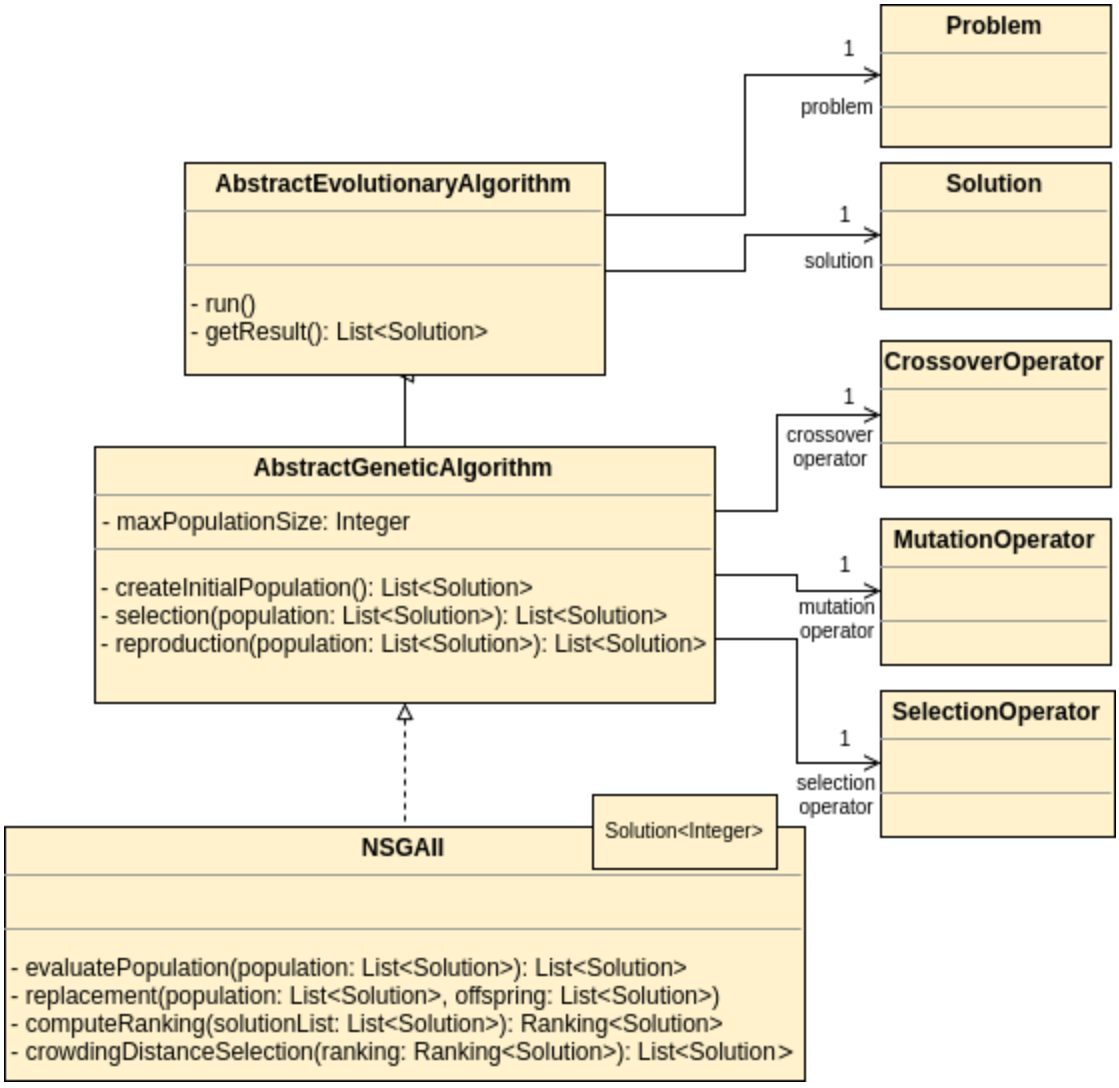

Figura 32 - Diseño NSGA-II provisto por jMetal

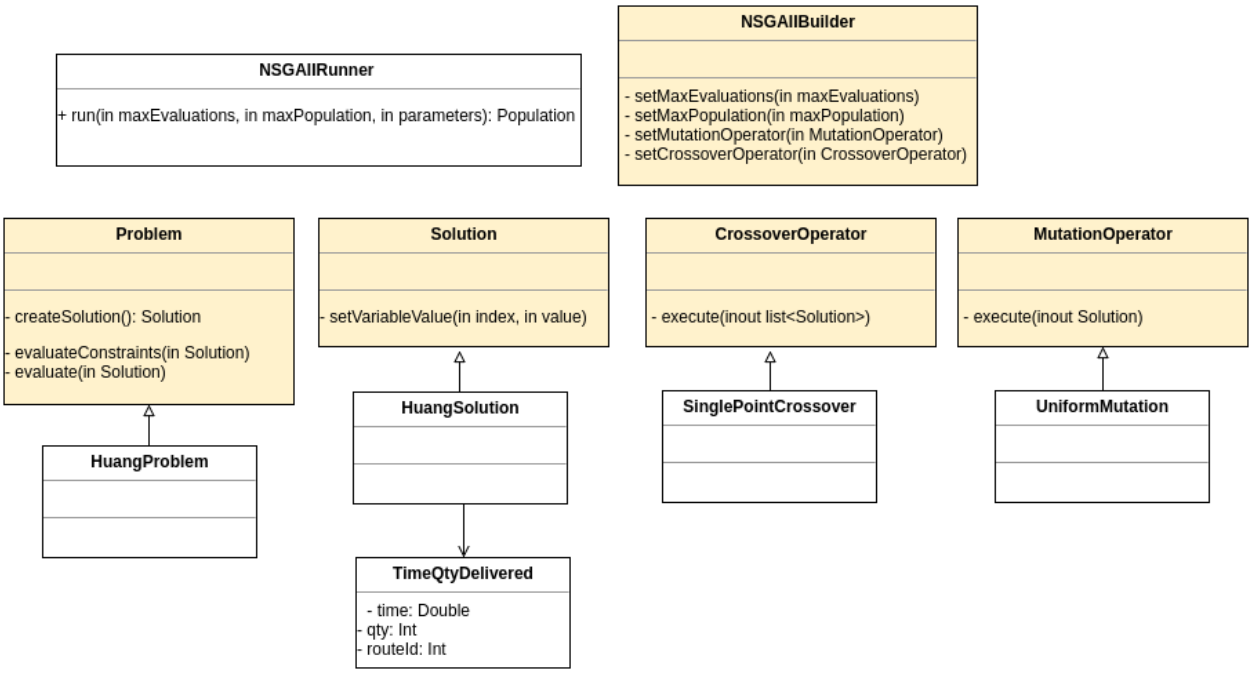

Figura 33 - Diseño de la utilización de NSGA-II de jMetal

NSGAIIRunner

Esta clase es el punto de entrada al módulo NSGA-II. Se define el siguiente método para la configuración y ejecución del mismo:

`run(GeneticReturn gr, List<FitnessType> objectives, Integer maxEvaluations, Integer populationSize): List<HuangSolution>`

Recibe como parámetros :

- Una estructura que contiene: las rutas posibles y un conjunto con los mejores individuos obtenidos en la etapa de generación de rutas.
- Los objetivos a minimizar, esto es, una combinación de eficiencia, eficacia y equidad.
- El tamaño de la población.
- La cantidad de evaluaciones que realiza algoritmo genético. Este valor indica la cantidad de veces que se evalúan las soluciones y cuando se llega a la cantidad especificada termina la ejecución del mismo.

Este método actúa como una encapsulación de la librería jMetal, encargándose de configurar las instancias de NSGA-II, comenzar su ejecución y luego retornar los resultados obtenidos. Como fue mencionado en la Sección 4.2.3.6, se tiene dos estrategias completamente diferentes del *Split Delivery*, por lo que `NSGAIIRunner` es el encargado de ejecutar el algoritmo una vez para cada una de ellas.

Para crear la instancia de NSGA-II se hace uso de la clase `NSGAIIBuilder`, incluida en jMetal, la cual haciendo uso del patrón de diseño *Builder* retorna una instancia de `Algorithm` correspondiente a NSGA-II. Se especifican las siguientes configuraciones:

- El problema a resolver es implementado en la clase `HuangProblem`.
- Las soluciones son del tipo `HuangSolution`.
- El operador de *crossover* implementado corresponde a la clase `SinglePointCrossover`, configurado con su probabilidad.
- El operador de mutación implementado corresponde a la clase `UniformMutation`, configurado con su probabilidad.
- El tamaño de la población.
- La cantidad de evaluaciones de las solución durante toda la ejecución del algoritmo.
- Estrategia de *Split Delivery*.

Cabe destacar que en el caso de NSGA-II, la cantidad de generaciones del algoritmo se calculan en base al tamaño de la población y la cantidad de evaluaciones como se muestra en (20).

$$generaciones = \frac{maxEvaluaciones}{maxPoblacion} \tag{20}$$

En cuanto a la estrategia de *Split Delivery*, el algoritmo genético tiene dos modos de "ejecución" en los cuales varía la forma de asignar demandas a los clientes de cada ruta: el primer modo sólo permite que un cliente sea abastecido de la totalidad de su demanda por una única ruta, mientras que el segundo aplica la estrategia de *Split Delivery* detallada en la Sección 4.2. Internamente, al ejecutar el algoritmo, se realizan dos ejecuciones consecutivas, una con cada modo.

Al finalizar la ejecución del algoritmo seleccionador, se obtiene una lista de soluciones `HuangSolution`. Esta lista corresponde a la unión de las soluciones obtenidas por las ejecuciones de NSGA-II utilizando los dos modos disponibles. Los resultados de las ejecuciones se unen y se eliminan los resultados repetidos y dominados. En la Figura 34 se muestra un ejemplo de la ejecución del algoritmo seleccionador.

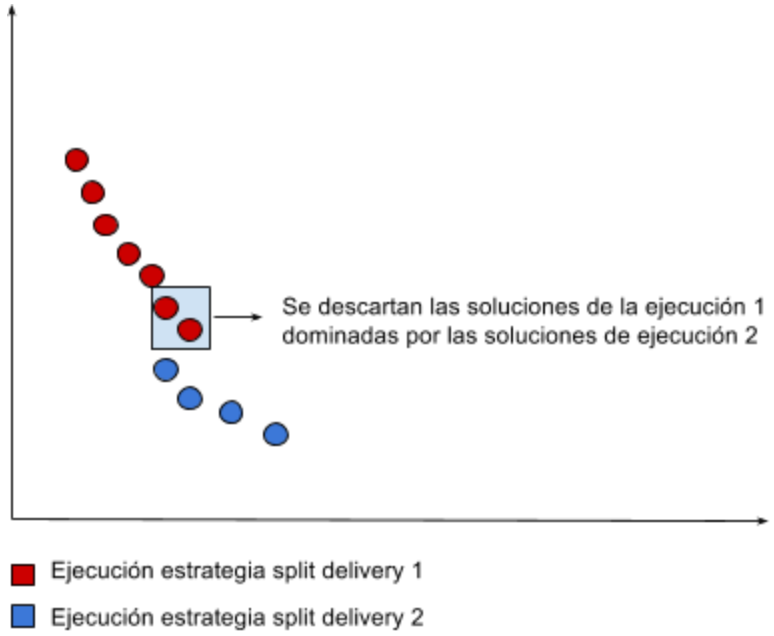

Figura 34 - Unión de las soluciones obtenidas por las dos ejecuciones NSGA-II

HuangProblem

Esta clase representa el problema que se quiere resolver con el algoritmo genético y, como se indicó anteriormente, implementa la interfaz `Problem` definida en jMetal. Se debe indicar la clase correspondiente a la codificación de la soluciones del problema, en este caso: `HuangSolution`. A su vez, también implementa la interfaz `ConstrainedProblem` ya que el problema tiene restricciones.

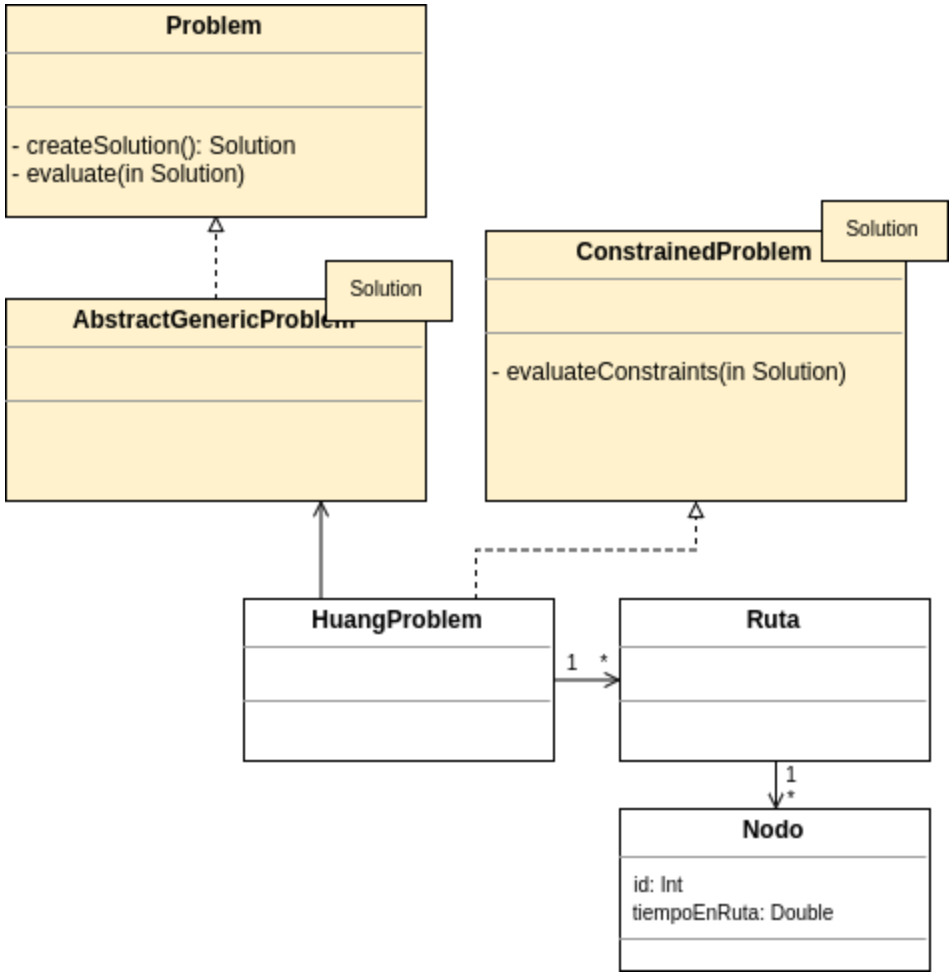

Figura 35 - Diseño de Problema en NSGA-II

Todo problema `Problem` se caracteriza por la cantidad de variables de decisión, la cantidad de funciones objetivo y la cantidad de restricciones del problema. En la Figura 35 se muestra el diseño del problema en NSGA-II.

`HuangProblem` es modelado de manera que las variables de decisión son un conjunto de *N* variables enteras, donde *N* equivale a la cantidad de vehículos disponibles. A su vez, como está especificado en Huang et al. (2011) [22], los vehículos podrán recorrer a lo sumo una ruta o ninguna. Esto se representa con el identificador de la ruta o con el valor nulo respectivamente.

La cantidad de objetivos corresponderá a los indicados al inicializar el módulo en `NSGAIIRunner`.

Además se deben indicar la cantidad de restricciones, las cuales se desprenden del problema de Huang et al. (2011) [22]:

1. La demanda entregada por cada uno de los vehículos no supere la capacidad de los mismos.
2. Cada uno de los clientes reciban la totalidad de su demanda.

A continuación se detallan los métodos más importantes:

- `determineInitialSolutionsDistribution(Integer populationSize)` : Se ejecuta una única vez durante la inicialización de la instancia del problema. Se encarga de pre-computar los mejores individuos obtenidos por el algoritmo generador en una estructura auxiliar. Estos se utilizarán durante la inicialización de las soluciones.
- `createSolution(): HuangSolution` : Definido en la interfaz `Problem`. Retorna una solución inicial del problema. Algunas de ellas son obtenidas de la estructura auxiliar y otras se generan seleccionando rutas aleatoriamente. Es ejecutado *N* veces por el algoritmo genético en su etapa de inicialización para crear las soluciones correspondientes a la población inicial.
- `evaluate(HuangSolution solution)` : Toma como parámetro una solución y se encarga de evaluarla para cada uno de los objetivos que se quieren optimizar.
- `evaluateConstraints(HuangSolution solution)` : Toma como parámetro una solución y se encarga de evaluar las dos restricciones mencionadas anteriormente.

En cada iteración del algoritmo genético, y luego de la ejecución de los operadores de selección, *crossover* y mutación, se evalúan todas las soluciones haciendo uso de los métodos `evaluate` y `evaluateConstraints`. Posteriormente, teniendo todas las soluciones evaluadas para cada objetivo, estas se ordenan en un *ranking* de dominancia y se selecciona la población inicial de la siguiente iteración. En caso de tener que seleccionar entre soluciones con mismo *ranking* se utiliza el comparador *crowding-distance*.

HuangSolution

Esta clase modela una solución del algoritmo genético como una lista de enteros que corresponde a la identificación de las rutas obtenidas en el proceso de generación.
Se implementa como una herencia de la clase `AbstractGenericSolution` donde se especifican `HuangProblem` (el problema) e `Integer` (identificación de las rutas) como los tipos de las variables de la solución. A su vez, implementa la interfaz `Solution`, lo cual agrega a la clase el comportamiento que espera jMetal de una solución, ver Figura 36.

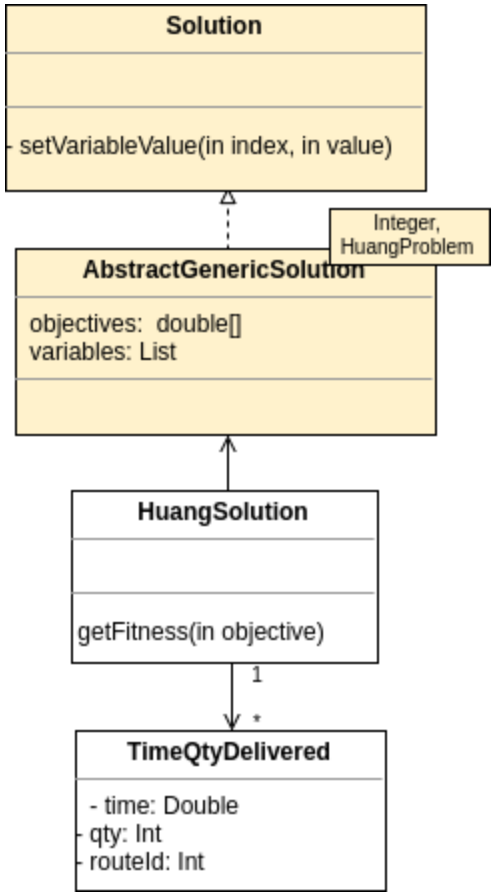

Figura 36 - Diseño de una solución utilizando NSGA-II de jMetal

Los métodos principales se describen a continuación:

- `setVariableValue(int index, Integer newValue)` : Se encarga de asignar el identificador de ruta *newValue* al vehículo representado por el índice *index*. También se encarga de actualizar la demanda satisfecha por cada vehículo.
- `getFitness(FitnessType objective)` : Calcula el *fitness* de la solución para el objetivo especificado por el parametro *objective*.

Para que el algoritmo soporte *Split Delivery* se requiere una estructura auxiliar en cada solución, en donde se mantienen los tiempos de entrega y cantidad entregada para cada cliente en cada ruta.
Esta estructura se actualiza cada vez que el valor de alguna de las variables de la solución cambia (por medio del método `setVariableValue`) como corresponda. De no tener la estructura auxiliar con los datos calculados, cada vez que se necesite evaluar los objetivos o las restricciones de una solución, habría que recalcular todo volviendo ineficiente el algoritmo genético.

SinglePointCrossover

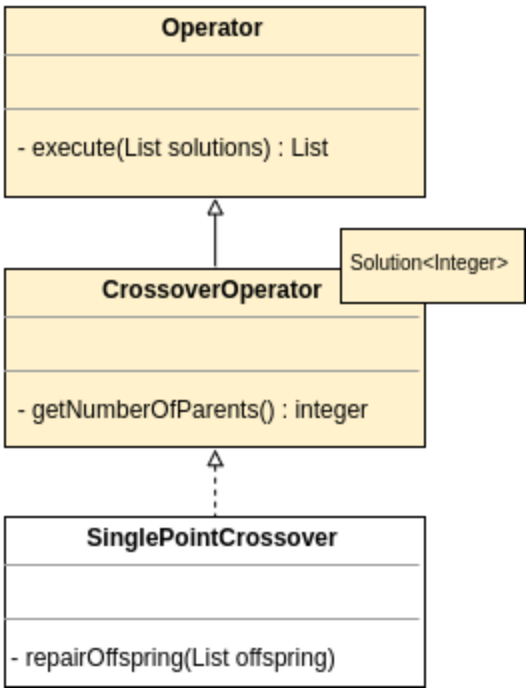

Figura 37 - Diseño del operador de crossover utilizando NSGA-II de jMetal

El operador de crossover consiste en implementar la interfaz `CrossoverOperator` de jMetal, como se puede ver en el diagrama de la Figura 37.

Los métodos que se deben implementar son los siguientes:

- `List<Solution<Integer>> execute(List<Solution<Integer>> solutions)` : Es el método principal de esta clase. Es ejecutado por el algoritmo genético $\frac{maxPoblacion}{2}$ veces en cada iteración para obtener hijos que agregar a la población general.
- `int getNumberOfParents()` : Retorna la cantidad de padres con que se quiere realizar el crossover, en este caso siempre equivale a 2.

El método `execute` realiza el crossover, dependiendo de una probabilidad $P_c = 0.8$ que es elegida empíricamente basándose en [30]. En caso de que la probabilidad se cumpla, se crean dos soluciones nuevas y en caso contrario se retornan copias de los padres.
En caso de ocurrir el primer escenario, y al ser el método de crossover un *Single Point Crossover* que consiste en unir una parte de cada padre, se debe verificar que las nuevas soluciones sean válidas dentro del alcance de este método. Una solución es considerada válida si se cumple que no existen rutas repetidas. Para validar esto último, se implementa el método `repairOffspring(List<Solution<Integer>> offspring)` que se encarga de buscar rutas repetidas en la solución y, en caso de encontrar, reemplazar el valor de la segunda ocurrencia por el valor nulo.

UniformMutation

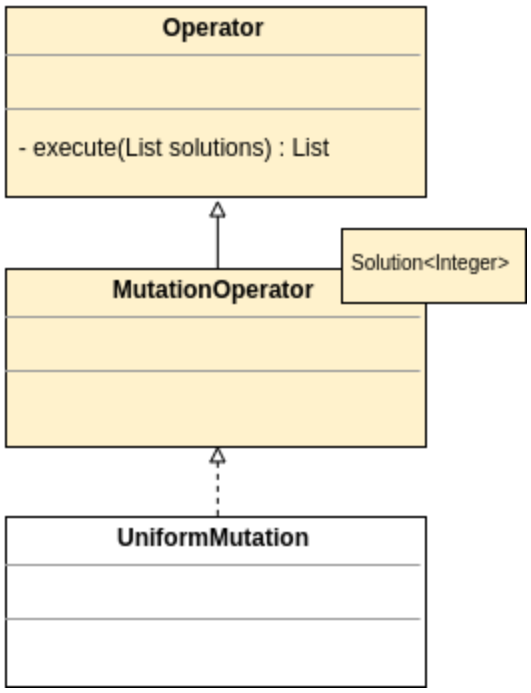

Figura 38 - Diseño del operador de mutación utilizando NSGA-II de jMetal

Por otro lado, el operador de mutación consiste en implementar la interfaz `MutationOperator` de la librería jMetal, como se observa en la Figura 38.

El método principal de esta clase es la implementación del método `execute` de la interfaz `Operator`. El mismo toma como parámetros una solución a la cual se le va a aplicar (o no) la mutación. En caso de aplicarla, se crea una copia de la solución ingresada como parámetro y se retorna ésta mutada.
Como se explicita en la Sección 4.2, el operador de mutación implementado corresponde a un `UniformMutation`. Se deben definir los límites inferior y superior entre los cuales elegir el valor utilizado para sustituir; en este caso corresponden a 0 y el número de rutas generadas durante el proceso de generación, respectivamente.

Es de importancia definir la probabilidad que tendrá la mutación de ejecutarse sobre cada elemento de la solución. Las mutaciones ayudan a escapar al algoritmo de óptimos locales pero una probabilidad muy alta convertiría al algoritmo en una búsqueda aleatoria. Por esta razón el número debe ser considerablemente bajo y se elige $P_m = 0.1$ [38].

En el bloque principal de este método se itera sobre las variables de la solución ingresada como parámetro y para cada una de ellas se genera un número aleatorio entre 0 y 1. Pueden darse 2 situaciones:

1. El número es menor a P_m (10%). En este caso se bifurcan dos opciones:

a. El número es menor que $\frac{P_m}{6}$ (~1.6%). En esta situación se intercambia el valor de la variable actual por el valor nulo. La elección de tomar la probabilidad dividido 6 fue empírica y la explicación se debe a que la mayor cantidad de las veces se prefiere cambiar la ruta que recorre un vehículo, pero en algunas situaciones puede ser beneficioso que el vehículo no recorra ninguna ruta, especialmente para eficiencia.
b. El número es mayor que $\frac{P_m}{6}$ (~8.3%). En esta situación se intercambia el valor de la variable actual por una ruta elegida de forma aleatoria que no sea parte de la solución actual.

2. En caso que el número sea mayor a P_m (90%), no se realiza mutación sobre el elemento de la solución.

5.2.2.5 Logging

El módulo de *Logging* brinda un método el cual es invocado pasándole como parámetro el mensaje u objeto a imprimir. Es el único responsable de imprimir en todas las salidas especificadas tal como se muestra en la Figura 39:

- Interfaz gráfica
- Consola
- Archivo de logging

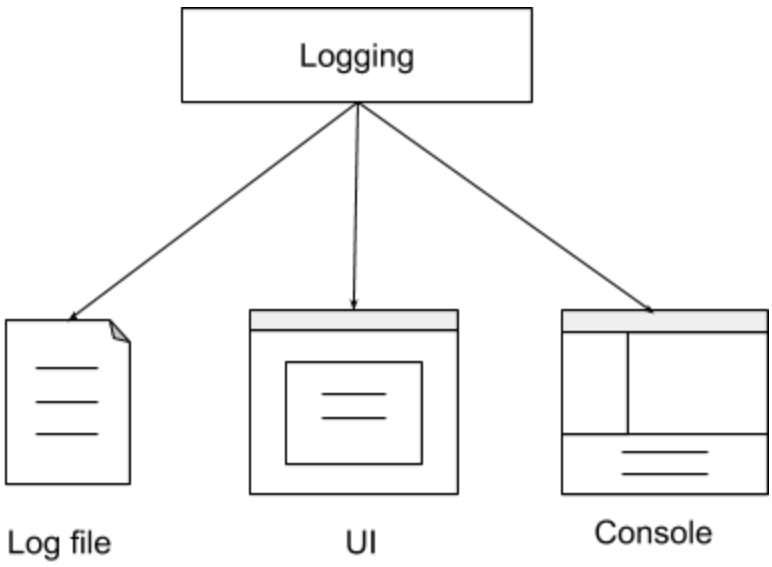

Figura 39 - Diseño del módulo de Logging

6 Pruebas

6.1 Planificación

6.1.1 Métricas

Como se explicó en la Sección 4.1, el sistema cuenta con dos fases fundamentales que deben ser probadas: la generación y la selección de rutas. Para esto se definen las siguientes métricas:

Generación de rutas

Métrica 1
Teniendo en cuenta que el algoritmo generador es capaz de generar conjuntos de rutas orientadas a optimizar cada objetivo de forma individual, resulta razonable probar por separado cada uno de ellos.

El objetivo de las pruebas del algoritmo genético encargado de la generación de rutas es verificar que el conjunto de rutas retornado por el mismo permita construir una solución cercana a la óptima. Esto se logra comparando los siguientes puntos:

1. La solución óptima del problema. Para esto se crean todas las rutas factibles (hay rutas no factibles por las *Time Windows*) y se resuelve con un software de optimización.
2. La mejor solución considerando únicamente las rutas generadas por el algoritmo genético desarrollado. Para esto se ejecuta el algoritmo generador de rutas y se utiliza el software de optimización para encontrar la mejor solución para el conjunto de rutas obtenido.

Selección de rutas

Por otro lado se debe probar el algoritmo seleccionador de rutas. El fin principal es probar el seleccionador para el problema multi-objetivo, pero además se lo prueba para cada objetivo de forma individual.

Métrica 2
El objetivo de las pruebas del algoritmo seleccionador de rutas, para cada objetivo de forma individual, es verificar que dado un conjunto de rutas la solución obtenida sea cercana a la óptima.

Dado un conjunto de rutas, esto se logra comparando los siguientes puntos:

1. La resolución mediante un software de optimización.
2. La resolución mediante NSGA-II.

Métrica 3

Para probar el problema multi-objetivo, la dificultad se encuentra en la forma de evaluar la calidad de los resultados. En general, el rendimiento de un EA se evalúa mediante pruebas experimentales y, como consecuencia, se han definido varias métricas de rendimiento para este propósito. Las métricas consideran principalmente tres aspectos del conjunto de soluciones [36]:

- La convergencia, es decir, la cercanía al frente óptimo de Pareto.
- La diversidad: distribución y dispersión.
- El número de soluciones.

Se decidió el uso de los siguientes indicadores, incluyendo algunos de los más utilizados en la actualidad [36]:

1) *Generational Distance* (GD). Es una métrica que representa que tan "lejos" se encuentra el conjunto de referencia (PFc) del frente óptimo de Pareto (PFt).
   Se define como (21)[46]:

$$G \widehat{=} \frac{\left(\sum_{i=1}^{n} d_i^p\right)^{\frac{1}{p}}}{n} \tag{21}$$

   Donde n es el número de vectores en PFc, $p = 2$, y d_i es la distancia euclidiana (en el espacio objetivo) entre cada vector y el miembro más cercano de PFt. Un resultado igual a 0 indica PFt = PFc. Cualquier otro valor indica que PFc se desvía de PFt.
2) *Inverted Generational Distance* (IGD). Es una variante invertida de GD, la cual presenta diferencias significativas:
   a) Calcula la distancia euclidiana mínima (en lugar de la distancia media), entre el conjunto de referencia y el frente óptimo de Pareto.
   b) Utiliza como referencia las soluciones en PFt en vez de PFc.
   c) Si se conocen suficientes miembros de PFt, IGD puede medir tanto la diversidad como la convergencia de PFc.
3) *Epsilon* (ε). Es un indicador que da un factor por el cual un conjunto de aproximación es peor que otro teniendo en cuenta todos los objetivos. Formalmente, sean A y B dos conjuntos de aproximación, entonces $\varepsilon(A,B)$ es igual al factor mínimo tal que para cualquier solución en B hay al menos una solución en A que no es peor por un factor ε considerando todos los objetivos.
4) *Cantidad de Soluciones* (PFS). Métrica que indica la cantidad de soluciones diferentes obtenidas.

5) *Cantidad de soluciones no dominadas* (C). Sean *A* y *B* dos conjuntos de aproximación. *C(A,B)* es la cantidad de puntos no dominados de *B* por *A*.

6.1.2 Instancias

Para las pruebas, se utilizaron instancias de Solomon para VRPTW [40], las cuáles son ampliamente utilizadas como referencia en trabajos de ruteo de vehículos [3][22][45]. Las instancias están divididas en seis conjuntos que resaltan distintas variables que afectan a los problemas de ruteo de vehículos:

- Ubicación geográfica de los clientes.
- Cantidad de clientes que puede atender un único vehículo.
- Porcentaje de clientes con restricciones de tiempo.
- Amplitud y posicionamiento de las *Time Windows*.

Los clientes de las instancias *R1* y *R2* están distribuidos en el mapa de manera aleatoria, mientras que en las instancias *C1* y *C2* se distribuyen de tal manera que formen un *cluster*. Las instancias *RC1* y *RC2* son una combinación de las anteriores teniendo varios *clusters* de clientes dispersos aleatoriamente.
Los conjuntos de problemas *R1*, *C1* y *RC1* comparten la característica de tener *Time Windows* muy restrictivas, lo que resulta en pocos clientes atendidos (de 5 a 10) por vehículo. Por el contrario, para los conjuntos *R2*, *C2* y *RC2* las *Time Windows* son menos restrictivas y los vehículos tienen la posibilidad de atender una mayor cantidad de clientes (más de 30). Las instancias fueron creadas con 100 nodos, pero existen versiones reducidas de 25 y 50 nodos [48].
Cabe destacar que los *benchmark* de las instancias utilizadas de Solomon se encuentran en función de la distancia recorrida. Por esta razón no fue tomada como métrica ya que ninguno de los tres objetivos propuestos es comparable.

Con el fin de asegurar probar *Split Delivery*, se crearon nuevas instancias a partir de las de Solomon, con capacidades más restrictivas, como fue sugerido por Vacca y Salani (2009) [45]. Se eligió una instancia de cada conjunto (*R1*, *R2*, *C1*, *C2*, *RC1*, *RC2*) y se crearon las instancias *SR110*, *SR201*, *SC105*, *SC201*, *SRC101*, *SRC201*. La única variación respecto a las instancias originales fue aumentar la demanda de todos los clientes de modo que la suma de las mismas se aproxime a la suma de las capacidades de los vehículos.
En total se tienen 62 instancias diferentes: 56 originales y 6 creadas para probar *Split Delivery*.

Partiendo del conjunto de todas las rutas factibles de una instancia, y utilizando un software de optimización, es posible obtener el óptimo para el objetivo deseado. Pero como la cantidad de rutas factibles incrementa de forma exponencial con la cantidad de nodos de la instancia, no es posible encontrar la solución óptima para instancias con muchos nodos en un tiempo razonable. Por esta razón, algunas pruebas sólo se pueden realizar con instancias

pequeñas. Para las mismas se decide utilizar solamente los 10 y 8 primeros nodos para las instancias originales de Solomon y las modificadas para *Split Delivery*, respectivamente.
Las instancias originales de 100 nodos cuentan con 25 vehículos. Para mantener una relación, se bajó la cantidad de vehículos a 4 para las instancias de 10 y 8 nodos mencionadas. Con 9 clientes y el almacén se podrían tener como máximo 986409 rutas factibles, aunque con las restricciones de *Time Windows* la mayoría no serán factibles y serán descartadas.
La fórmula (22) es la utilizada para obtener la cantidad de rutas posibles con *n* nodos.

$$\sum_{i=1}^{n-1} P(n-1,\ i) \tag{22}$$

Donde *P(a,b)* son las permutaciones de *a* elementos tomados de a *b*. Se utiliza *n-1* en vez de *n* porque el almacén (nodo 0) siempre estará al inicio y al final de todas las rutas.

Para la resolución de instancias grandes se partirá del conjunto de rutas generadas por el algoritmo generador, ya que es inviable el uso de todas las rutas factibles por lo explicado anteriormente. Sólo se realizarán pruebas para la métrica 3 (selección de rutas multi-objetivo).

Instancias pequeñas
Para todas las instancias pequeñas de 10 y 8 nodos, utilizando 4 vehículos, se calcula para cada objetivo de forma individual lo siguiente:

1. Todas las rutas existentes resueltas con GLPK.
2. Todas las rutas existentes resueltas con NSGA-II.
3. Rutas generadas por algoritmo generador resueltas con GLPK.
4. Rutas generadas por algoritmo generador resueltas con NSGA-II.

Comparando:

- (1) con (3) se mide la métrica 1.
- (1) con (2) se mide la métrica 2.
- (1) con (4) se mide la unión de las métricas 1 y 2. Se puede ver como funciona la unión del generador de rutas junto al seleccionador como una unidad. Tiene sentido probarlo de este modo ya que se prueba el sistema en su totalidad.

Instancias grandes
Como se explicó anteriormente, para instancias grandes sólo se va a medir la métrica 3.
Lo ideal sería compararse contra el frente óptimo de Pareto, pero esto no es posible dado que el problema tratado es NP-Hard y generarlo es computacionalmente inviable [50]. Por lo general, la forma más común para compararse es hacerlo contra conjuntos obtenidos mediante la utilización de otros algoritmos [50], pero al no haber *benchmarks* de este problema, es imposible. Por lo tanto, se consideró que utilizar la técnica de suma ponderada

para obtener al menos algunos puntos del frente óptimo de Pareto con los cuales compararse es una buena alternativa.
Como fue explicado en la Sección 4.2, los puntos obtenidos a partir de la resolución del problema mediante la suma ponderada pertenecen al frente de Pareto. Utilizando pesos lo suficientemente diferentes para los objetivos, se podría obtener una aproximación del frente óptimo de Pareto. Estos resultados se consideran un límite inferior ya que no será posible encontrar una solución mejor, lo que en términos multi-objetivo sería una solución que los domine.

Para todas las instancias de 100 nodos y 25 vehículos se realiza y compara lo siguiente:

1. Rutas generadas por algoritmo generador. Se calculan 16 ponderaciones diferentes de los tres objetivos normalizados utilizando CPLEX. Lo anterior brinda algunos puntos del frente de Pareto.
2. Mismas rutas generadas en (1) resueltas con NSGA-II para los tres objetivos simultáneamente. Se obtiene un conjunto de Pareto.

Comparando:

- (1) con (2) se mide la métrica 3.

6.2 Resultados esperados

En cuanto a la generación de rutas, se espera obtener un conjunto de rutas de tamaño ampliamente inferior al conjunto total de rutas factibles, que permita obtener una solución cercana a la óptima. Para instancias grandes (100 nodos) el tiempo de obtención del conjunto debe ser del orden de minutos.

En cuanto a la selección de rutas, se espera sea capaz de encontrar soluciones ya sea con un único objetivo o multi-objetivo cercanas a las óptimas. El tiempo de ejecución debe ser del orden de segundos.

6.3 Realización

Instancias pequeñas

1. Se generan todas las rutas factibles. Se resuelve el problema para cada objetivo individual con GLPK y NSGA-II.
2. Para cada objetivo individual se generan rutas con el algoritmo generador y luego se resuelve el problema con GLPK y NSGA-II.

Parámetros de configuración utilizados

- Cantidad de ejecuciones algoritmo generador: 120. Para 10 nodos, se comprobó empíricamente que 120 ejecuciones es suficiente para obtener la mayor variedad posible de soluciones.
- Cantidad de generaciones algoritmo generador: rango (25,190). Se comprobó que dicho rango de generaciones brinda variedad al conjunto de rutas sin perder calidad.
- Población NSGA-II: 70. Se elige empíricamente por permitir al algoritmo explorar el espacio de soluciones sin sacrificar el tiempo de ejecución.
- Generaciones NSGA-II: 3000. Se elige empíricamente para permitir al algoritmo converger hacia el frente óptimo de Pareto sin sacrificar el tiempo de ejecución.

Para las instancias que deben ser resueltas mediante GLPK se especifican dos límites de tiempo:

- 30 minutos si se ha encontrado al menos una solución válida. Al llegar a dicho tope se toma la mejor solución encontrada hasta el momento.
- 45 minutos sin encontrar ninguna solución. Si no se pudo obtener ninguna solución válida, dicha instancia no se toma en cuenta para el promedio.

Las instancias para las cuales se generaron más de 4500 rutas no pueden resolverse para el objetivo equidad debido a insuficiencia de memoria RAM para generar el modelo.

Como resultado de la ejecución de estas pruebas se producen dos archivos en formato .csv cuyas estructuras se describen en el Anexo A.

Hardware utilizado
Intel Core i7 4720HQ 2.60GHz Base - 3.60GHz Turbo 4 cores 8 threads 6MB Cache.
16GB RAM (DDR3L - 1600 MHz)
Windows 10

Instancias grandes
1. Se generan las rutas para el problema multi-objetivo. Como se mencionó en la Sección 4.1.2 dichas rutas son el resultado de la unión de las rutas para eficiencia, eficacia, equidad, y todos los anteriores ponderados.

Parámetros de configuración utilizados

- Cantidad de ejecuciones algoritmo generador: 200 para las instancias originales de Solomon. Al haber más clientes, se aumenta la cantidad de ejecuciones para obtener mayor variedad de rutas. Para las instancias enfocadas a probar *Split Delivery* se realizan 80 ejecuciones para reducir la cantidad de rutas y permitir a CPLEX encontrar soluciones en un tiempo estipulado.
- Cantidad de generaciones algoritmo generador: rango (40,200). Al haber más clientes, se aumenta el número mínimo de generaciones para escapar de soluciones

iniciales malas. Al haber más clientes se necesitan más generaciones para converger al óptimo.

2. Dado que la selección de rutas para instancias grandes mediante GLPK toma demasiado tiempo, se decidió utilizar CPLEX en una computadora más potente de la facultad. Utilizando los conjuntos de rutas generados en (1) se generó un *script bash* que automatiza la ejecución de todos los pasos necesarios para poder obtener un conjunto de puntos del frente de Pareto para cada instancia. El *script* se compone de 2 pasos:

a. Para cada instancia se obtienen los óptimos de eficiencia, eficacia y equidad por separado.
b. Para cada instancia se corre 16 veces el problema multi-objetivo con diferentes ponderaciones. Para normalizar los objetivos se utilizan los valores obtenidos en el punto (a).

Para cada ejecución de CPLEX se especifica un tope de 15 minutos. Esto es así porque al haber 62 instancias, en el peor caso, la ejecución podría llegar a demorar 12,2 días (62 instancias * 19 ejecuciones * 15 minutos = 17670 minutos = 294 horas). Por otro lado, para las instancias enfocadas a asegurar la realización de *Split Delivery* se debió aumentar el límite de tiempo a 1 hora, ya que con 15 minutos no era suficiente para garantizar la optimalidad de las soluciones.

Los pesos fueron seleccionados empíricamente para obtener un conjunto reducido de ponderaciones pero que brinden suficiente variedad en las soluciones obtenidas. Es necesario obtener la mayor cantidad diferente de soluciones posibles a los efectos de poder obtener mayor cantidad de puntos del frente de Pareto. Estos son:

{0.01, 0.495, 0.495}
{0.04, 0.48, 0.48}
{0.1, 0.45, 0.45}
{0.15,0.425,0.425}
{0.2,0.4,0.4}
{0.25,0.375,0.375}
{0.3, 0.35, 0.35}
{0.334, 0.333, 0.333}
{0.4, 0.3, 0.3}
{0.5,0.25,0.25}
{0.6,0.2,0.2}
{0.7,0.15,0.15}
{0.8, 0.1, 0.1}
{0.9, 0.05, 0.05}
{0.95, 0.025, 0.025}
{0.98, 0.01, 0.01}

Donde las tuplas tienen la siguiente estructura: {eficiencia, eficacia, equidad}.

Notar que los pesos coinciden para eficacia y equidad. Esto es así ya que, cuando no es necesario realizar *Split Delivery*, son objetivos afines y por lo general cuando uno mejora el otro también. Por ejemplo, es muy probable que las siguientes tuplas: {0.4, 0.2, 0.4} y {0.4, 0.4, 0.2} resulten en la misma solución lo cual es redundante.
Las soluciones de las ponderaciones calculadas que tengan un GAP mayor al 5% serán descartadas, ya que no se puede saber si dicho punto pertenece al frente de Pareto.

3. Utilizando los conjuntos de rutas generados en (1), se ejecuta NSGA-II para el problema multi-objetivo con el fin de obtener un conjunto de Pareto para cada instancia.

Parámetros de configuración utilizados

- Población NSGA-II: 70. Se elige empíricamente por permitir al algoritmo explorar el espacio de soluciones sin sacrificar el tiempo de ejecución.
- Generaciones NSGA-II: 3000. Se elige empíricamente para permitir al algoritmo converger hacia el frente óptimo de Pareto sin sacrificar el tiempo de ejecución.

4. Para probar la calidad del algoritmo seleccionador se compara el conjunto de Pareto obtenido por NSGA-II (paso 3) con la aproximación al frente de Pareto obtenido por CPLEX (paso 2).
Como no se cuenta con todo el frente de Pareto, es probable que al utilizar ciertos indicadores el resultado no sea del todo acertado. Por ejemplo, si NSGA-II obtiene un punto del frente llamado *A*, pero no se obtiene el mismo mediante suma ponderada, la distancia euclidiana de *A* con el punto más cercano de la aproximación del frente de Pareto no será 0, cuando sí lo sería si se tuviera el frente exacto. Para solucionar este problema, se agregan todos los puntos no dominados obtenidos por NSGA-II (respecto a los puntos obtenidos por CPLEX) al conjunto aproximación del frente de Pareto.
Para que esto no degrade el conjunto de aproximación, se pide que el conjunto de puntos del frente de Pareto calculados por CPLEX tenga como mínimo 5 soluciones distintas, de forma de garantizar 5 puntos que pertenezcan al frente de Pareto real. Si se tienen menos, es probable que el conjunto de aproximación esté conformado en su mayoría por puntos obtenidos por NSGA-II quitando valor a la comparación.

A modo de resumen, dado que no es factible obtener el frente de Pareto real, se combinan los puntos del frente de Pareto real obtenidos con CPLEX con los no dominados de NSGA-II, para obtener una aproximación del mismo que se considera razonablemente buena. En la Figura 40 se muestra la conformación de la aproximación del frente de Pareto:

- Puntos obtenidos por CPLEX utilizando suma ponderada.
- Puntos obtenidos por NSGA-II no dominados por los anteriores.

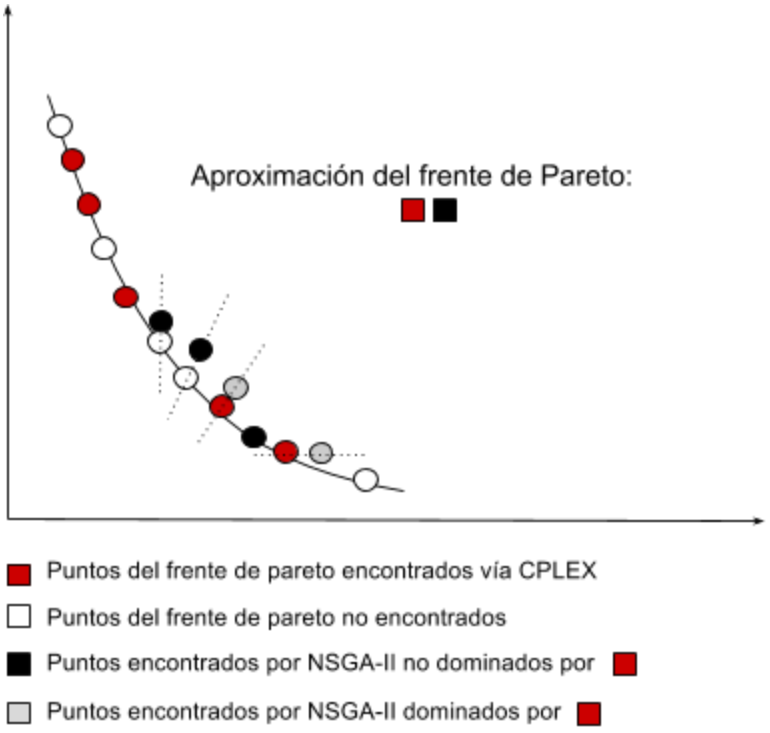

Figura 40 - Aproximación del frente de Pareto

Se compara la calidad de los puntos obtenidos por NSGA-II respecto al conjunto anterior, mediante la utilización de los indicadores de calidad detallados en la Sección 6.1.1. Para esto se utiliza jMetal que provee la implementación de los mismos.

Pseudocódigo:

1. Se crea un normalizador de frentes de Pareto con la aproximación del frente óptimo de Pareto *OP* (CPLEX + Puntos no dominados NSGA-II)*:*
    a. Para cada objetivo *o:*
        i. Se calcula $mayorValorObjetivo = max(v_o \in OP)$.
        ii. Se calcula $menorValorObjetivo = min(v_o \in OP)$.
2. Se normaliza *OP* generando *OPN* y el frente de Pareto obtenido con NSGA-II generando *NN* por separado:
    a. Para cada punto en el frente:
        i. Para cada objetivo:
            1. $valorObjetivo_{Norm} = \frac{valorObjetivo - menorValorObjetivo}{mayorValorObjetivo - menorValorObjetivo}$.
3. Se evalúa la calidad de *NN* respecto a *OPN* utilizando cada uno de los indicadores de calidad.

Cabe mencionar que durante la implementación para utilizar los indicadores de calidad de jMetal se encontró un *bug* en la normalización de los frentes de Pareto. Se hizo el arreglo correspondiente y se abrió un *pull request* el cual fue aceptado por uno de los autores de jMetal, Antonio J. Nebro e introducido al código fuente de la librería [24].

Hardware utilizado

Para la generación de rutas y para la resolución mediante NSGA-II se utiliza el siguiente equipo:

Intel Core i7 4720HQ 2.60GHz Base - 3.60GHz Turbo 4 cores 8 threads 6MB Cache.
16GB RAM (DDR3L - 1600 MHz)
Windows 10

Para el cálculo de los óptimos y las ponderaciones mediante CPLEX se utiliza el siguiente equipo:

Intel Core i7 5960X 3.00GHz Base - 3.50GHz Turbo 8 cores 16 threads 20MB Cache.
64GB RAM (DDR4 - 2133 MHz)
CentOS 7

7 Resultados obtenidos

7.1 Instancias pequeñas

Los resultados se exponen en las Tablas 1, 2 y 3. En cada una de ella se evalúan las comparaciones propuestas en la Sección 6.1.1 para cada uno de los objetivos. A continuación se analizan los resultados obtenidos para cada comparación.

Comparación 1:

- Todas las rutas existentes resueltas con GLPK.
- Rutas generadas por algoritmo generador resueltas con GLPK.

En este caso se mide la métrica 1.
El conjunto de rutas generadas por instancia para cada objetivo es sumamente menor al conjunto de todas las rutas posibles. Se puede observar que a partir de las rutas generadas fué posible encontrar la solución óptima o muy cercana a la óptima en todos los casos. Esto demuestra la buena calidad de las rutas generadas.

Comparación 2:

- Todas las rutas existentes resueltas con GLPK.
- Todas las rutas existentes resueltas con NSGA-II.

En este caso se mide la métrica 2.
Notar que en este caso, la solución inicial de NSGA-II sólo se genera a partir de la selección aleatoria de rutas del conjunto de todas las rutas. No se puede partir de soluciones buenas ya que para ello es necesario ejecutar el algoritmo generador de rutas. Por lo anterior es entendible que al algoritmo le resulte más difícil converger al óptimo. De todas formas, demostró muy buenos resultados.

Comparación 3:

- Todas las rutas existentes resueltas con GLPK.
- Rutas generadas por algoritmo generador resueltas con NSGA-II.

En este caso se pueden medir las métricas 1 y 2 a la misma vez.
En este caso el conjunto de rutas generado es mucho menor al de todas las rutas, pero con muy buena calidad. Esto sumado a que la población inicial estará en parte formada por buenas soluciones obtenidas por el algoritmo generador, hace que la convergencia de NSGA-II hacia el óptimo sea mayor.

Resultados

Las tablas con resultados obtenidos se separan por objetivo. De esta forma se pueden observar las tres comparaciones realizadas en una misma tabla, con el fin de facilitar la comparación.

La desviación, en porcentajes, se calcula según la ecuación (23).

$$\frac{valor - óptimo}{óptimo} \times 100 \tag{23}$$

#IR: es el número de instancias en las que se pudo calcular el óptimo con todas las rutas. Por ejemplo, para RC2 y objetivo Eficiencia, sólo se pudo obtener el óptimo en 3 de las 8 instancias existentes. Esto quiere decir que a la hora de calcular la desviación de los valores obtenidos, sólo se pudo comparar y promediar una parte del total.

Eficiencia

	#IR	Comparación 1			Comparación 2			Comparación 3		
		MIN	AVG	MAX	MIN	AVG	MAX	MIN	AVG	MAX
C1	6/9	0	0	0	0	0	0	0	0	0
C2	5/8	0	0	0	0	0	0	0	0	0
R1	12/12	0	0	0	0	2.57	6.41	0	0	0
R2	3/11	0	0	0	0	0	0	0	0	0
RC1	6/8	0	0	0	0	5.59	12.7	0	0	0
RC2	3/8	0	0	0	0	0	0	0	0	0
S	6/6	0	0	0	0	1.98	4.86	0	0	0

Tabla 1 - Resultados para objetivo Eficiencia

Eficacia

	#IR	Comparación 1			Comparación 2			Comparación 3		
		MIN	AVG	MAX	MIN	AVG	MAX	MIN	AVG	MAX
C1	8/9	0	0	0	0.001	0.04	0.22	0.001	0.02	0.08
C2	7/8	0	0	0	0	0.41	1.54	0	0.02	0.036
R1	12/12	0	0	0	0.01	1	2.73	0.01	0.28	0.42
R2	3/11	0	0	0	0	0.04	0.09	0	0.03	0.05
RC1	8/8	0	0	0	0	1.56	4.28	0	0.21	0.38
RC2	4/8	0	0	0	0	0.06	0.22	0	0.02	0.04
S	6/6	0	0	0	0	0.03	0.16	0	0.03	0.16

Tabla 2 - Resultados para objetivo Eficacia

Equidad

	#IR	Comparación 1			Comparación 2			Comparación 3		
		MIN	AVG	MAX	MIN	AVG	MAX	MIN	AVG	MAX
C1	5/9	0.12	0.19	0.55	0	0	0	0.12	0.19	0.55
C2	4/8	0	0	0	0	0	0	0	0	0
R1	10/12	0	0.31	1.17	0	1.34	3.97	0	0.33	1.17
R2	1/11	0.49	0.49	0.49	0	0	0	0.49	0.49	0.49
RC1	2/8	0	0	0	0	0.69	1.39	0	0	0
RC2	1/8	0	0	0	0	0	0	0	0	0
S	6/6	0	0	0	0	0	0	0	0	0

Tabla 3 - Resultados para objetivo Equidad

7.2 Instancias grandes

En la Tabla 4 se muestran los resultados agrupados por tipo de instancia.
#IR es el número de instancias en las que se pudo obtener un mínimo de 5 puntos distintos válidos del frente de Pareto mediante CPLEX. El resto de valores calculados es un promedio de los resultados obtenidos para estas instancias.
$\#P_{CPLEX}$ es la cantidad de puntos distintos válidos del frente de Pareto que se encontraron mediante CPLEX.
#Rutas es la cantidad de rutas diferentes generadas por el algoritmo genético generador de rutas y $T_{GenRutas}$ el tiempo de generación de las mismas en segundos.
T_{CPLEX} y $T_{NSGA-II}$ son el tiempo en segundos de ejecución de la selección de rutas para CPLEX y NSGA-II respectivamente.
El resto de columnas son los indicadores mencionados en la Métrica 3: *Generational Distance, Inverted Generational Distance, Epsilon,* cantidad de soluciones diferentes encontradas y cantidad de soluciones no dominadas.

	#IR	$\#P_{CPLEX}$	$T_{GenRutas}$	#Rutas	T_{CPLEX}	$T_{NSGA-II}$	GD	IGD	Epsilon	PFS	C
C1	7/9	12	135	2917	8325	68	0.021	0.025	0.110	62	7
C2	2/8	7	357	1918	11610	63	0.005	0.003	0.020	61	46
R1	11/12	7.45	132	3664	13163	70	0.022	0.027	0.171	65	22
R2	2/11	6	660	2119	13934	64	0.011	0.004	0.094	69	39
RC1	8/8	11.5	133	3459	13962	70	0.024	0.038	0.179	68	5
RC2	1/8	5	476	2131	12713	64	0.005	0.003	0.087	68	52
S	3/6	6	257	1320	57841	101	0.052	0.027	0.182	20	9

Tabla 4 - Resultados para problema multi-objetivo

Se observa que cuando la cantidad de puntos válidos del frente obtenidos por CPLEX es mayor, el número de soluciones no dominadas (*C*) obtenidas por NSGA-II es menor. Al contar con más puntos que pertenecen al frente de Pareto, es más difícil que NSGA-II obtenga un punto no dominado. De todas formas, es importante remarcar que NSGA-II fue capaz de encontrar soluciones no dominadas en todos los casos.
Se puede afirmar que cuantos más puntos se obtienen vía CPLEX, mejor es la aproximación del frente, y por lo tanto los indicadores son más acertados en estos casos.

Para las instancias que tienen *Time Windows* menos restrictivas, en pocos casos pudieron obtenerse al menos cinco puntos del frente óptimo de Pareto mediante suma ponderada (CPLEX). Sin embargo, para las instancias con *Time Windows* más restrictivas casi siempre se pudo obtener al menos 5 puntos.
En cuanto al tiempo de generación de rutas, las instancias con *Time Windows* menos restrictivas tomaron más tiempo que el resto.
En cuanto al tiempo de la selección de soluciones, CPLEX pudo obtener algunos pocos puntos en un tiempo varios órdenes superior al que le tomó a NSGA-II obtener varios puntos.

A pesar de que los indicadores de calidad son principalmente utilizados para comparar los resultados entre distintos algoritmos, se puede observar que los valores están muy cercanos al 0, lo que es muy bueno. Si fueran 0 significaría que los puntos obtenidos son los de la aproximación del frente óptimo de Pareto. Específicamente hablando de GD e IGD los resultados se encuentran entre un 0.003 y 0.038. Por otro lado el indicador de calidad Epsilon es más desafiante y los resultados se encuentran entre 0.020 y 0.179.

8 Conclusiones y trabajo futuro

En este proyecto se abordó y extendió el problema propuesto por Huang et al. (2011) [22], el cual consistió en tratar simultáneamente los objetivos eficiencia, eficacia y equidad como parte de un problema multi-objetivo de ruteo de vehículos en logística humanitaria. Esto es muy importante ya que permite dar un uso práctico y aplicable a la realidad del modelo propuesto, para que sea utilizado por distintas organizaciones que deban tratar dichos aspectos de forma concurrente.
Se logró desarrollar un sistema que permite obtener soluciones multi-objetivo de calidad para problemas SDVRPTW mediante el uso de dos algoritmos genéticos. El primero de ellos se encarga de la generación de rutas y adapta el trabajo realizado por Alvarenga (2005)[2] y Alvarenga et al. (2007)[3] para logística humanitaria. El otro, implementa la metaheurística NSGA-II [17] para poder resolver el problema multi-objetivo. Para instancias de 100 nodos el tiempo de resolución es de pocos minutos, lo que es muy satisfactorio para este tipo de problemas.

La arquitectura del sistema permite la implementación de algoritmos alternativos, ya sea para la generación y/o la selección de rutas, para poder fácilmente compararse contra los que fueron desarrollados. La visualización de los resultados permite al usuario el entendimiento de la solución obtenida por el sistema y la posibilidad de verificarla manualmente.

Se realizaron pruebas para la generación y selección de rutas para cada objetivo de forma individual en instancias chicas, obteniendo el resultado óptimo o muy cercano a él en todos los casos. Adicionalmente, partiendo de las rutas generadas por el algoritmo generador, se prueba la selección de rutas para el problema multi-objetivo para instancias grandes, obteniendo resultados muy satisfactorios. Los resultados obtenidos por los indicadores de calidad que se aplicaron para medir el desempeño de NSGA-II al resolver el problema multi-objetivo no son exactos. Esto es así porque no se cuenta con el frente de Pareto completo. Para que futuros trabajos tengan con qué compararse, se entregan las aproximaciones de los conjuntos de Pareto obtenidas para cada instancia utilizada.

Como trabajo futuro se plantea mejorar los algoritmos genéticos implementados. Respecto a la generación de rutas, actualmente para las mutaciones del algoritmo genético se utilizan los 8 operadores planteados por Alvarenga et al. (2007) [3] independientemente de a qué objetivo esté orientado el conjunto de rutas a generar. Se podrían implementar nuevas mutaciones que sean específicas para cada caso y analizar diferentes combinaciones de las mismas así como cantidad de ejecuciones. En cuanto a la selección de rutas mediante la utilización de NSGA-II, sería interesante probar empíricamente si otros operadores de mutación y crossover se logran comportar mejor que los implementados. Por otro lado, se

podrían probar distintas alternativas para realizar el *Split Delivery* con el objetivo de mejorar aún más los resultados.

Glosario

API: Application Program Interface. Son interfaces que proveen las aplicaciones para que terceros puedan utilizarla de manera adecuada.
CPLEX: Es un paquete de software comercial que resuelve problemas de optimización incluyendo problemas LP y MIP [8].
Escalamiento de Fitness: Consiste en aplicar cierta función sobre el fitness del individuo y utilizar dicho resultado en su lugar. Esto se puede utilizar para limitar la capacidad de las soluciones más fuertes de dominar a las más débiles [27].
GLPK: El paquete GNU Linear Programming Kit (GLPK) suministra un solver no comercial para programación lineal a gran escala (LP) y programación entera mixta (MIP) [20].
Greedy: Son algoritmos que siempre toman decisiones localmente óptimas en cada iteración esperando que esto lleve a una solución globalmente óptima [16].
Heurística: Algoritmos que generalmente no retornan el resultado exacto pero que pueden dar un resultado aproximado para problemas en un tiempo computacionalmente razonable [26].
LMDP: Last Mile Distribution Problem. Es la última etapa en la logística, que se enfoca en la distribución de suplementos desde centros de distribución hacia los afectados por desastres [7].
LP: Linear Programming. Son problemas que constan de una función objetivo lineal y de restricciones que son ecuaciones o inecuaciones lineales [29].
Metaheurística: Es un framework algorítmico de alto nivel, independiente del problema que provee una guía o estrategia para desarrollar algoritmos heurísticos de optimización [43].
MIP: Mixed Integer Programming. Es un LP con la restricción de que algunas variables, aunque no necesariamente todas, son enteras [39].
NP: Es la clase de problemas computacionales para los cuales se puede verificar que una solución dada es solución en un tiempo de orden polinomial por una máquina de Turing determinista [33].
NP-Hard: Es una clase de problemas que cumple que es al menos tan difícil como los problemas más difíciles en NP [34].
Optimización multi-objetivo: Trata problemas matemáticos de optimización en los que se tiene multiples funciones objetivo que se quieren optimizar simultáneamente [11].
SDVRPTW: Split Delivery Vehicle Routing Problem with Time Windows. Problema de ruteo de vehículos donde los clientes pueden recibir la demanda en partes y las entregas deben realizarse dentro de cierta ventana de tiempo.

Anexo A

Archivo 1

Contiene los resultados de:

1. Todas las rutas existentes resueltas con GLPK.
2. Todas las rutas existentes resueltas con NSGA-II.

Datos

1. Instancia
2. Cantidad de nodos
3. Cantidad de vehículos
4. Capacidad de vehículos
5. Tiempo de generación de todas las rutas
6. Cantidad de rutas generadas

7. Tiempo de resolución para eficiencia por GLPK
8. Óptimo eficiencia obtenido por GLPK
9. Tiempo de resolución para eficiencia por NSGA-II
10. Óptimo eficiencia obtenido por NSGA-II
11. GAP entre 8 y 10

12. Tiempo de resolución para eficacia por GLPK
13. Óptimo eficacia obtenido por GLPK
14. Tiempo de resolución para eficacia por NSGA-II
15. Óptimo eficacia obtenido por NSGA-II
16. GAP entre 13 y 15

17. Tiempo de resolución para equidad por GLPK
18. Óptimo equidad obtenido por GLPK
19. Tiempo de resolución para equidad por NSGA-II
20. Óptimo equidad obtenido por NSGA-II
21. GAP entre 18 y 20

Archivo 2

Contiene los resultados de:

1. Rutas generadas por algoritmo generador resueltas con GLPK.
2. Rutas generadas por algoritmo generador resueltas con NSGA-II.

Datos

1. Instancia
2. Cantidad de nodos
3. Cantidad de vehículos
4. Capacidad de vehículos

5. Tiempo de generación de rutas para eficiencia
6. Cantidad de rutas generadas para eficiencia
7. Tiempo de resolución para eficiencia por GLPK
8. Óptimo eficiencia obtenido por GLPK
9. GAP entre 8 y óptimo eficiencia para todas las rutas
10. Tiempo de resolución para eficiencia por NSGA-II
11. Óptimo eficiencia obtenido por NSGA-II
12. GAP entre 11 y óptimo eficiencia para todas las rutas

13. Tiempo de generación de rutas para eficacia
14. Cantidad de rutas generadas para eficacia
15. Tiempo de resolución para eficacia por GLPK
16. Óptimo eficacia obtenido por GLPK
17. GAP entre 16 y óptimo eficacia para todas las rutas
18. Tiempo de resolución para eficacia por NSGA-II
19. Óptimo eficacia obtenido por NSGA-II
20. GAP entre 19 y óptimo eficacia para todas las rutas

21. Tiempo de generación de rutas para equidad
22. Cantidad de rutas generadas para equidad
23. Tiempo de resolución para equidad por GLPK
24. Óptimo equidad obtenido por GLPK
25. GAP entre 24 y óptimo equidad para todas las rutas
26. Tiempo de resolución para equidad por NSGA-II
27. Óptimo equidad obtenido por NSGA-II
28. GAP entre 27 y óptimo equidad para todas las rutas

Bibliografía

[1] Alonso, B., González, D., Giráldez, J.R. and Marco, A.I., 1998. Logística comercial. Madrid: Centro de Publicaciones del Ministerio de Educación y Cultura.

[2] Alvarenga, G.B., 2005. Um algoritmo híbrido para os problemas de Roteamento de Veículos Estático e Dinâmico com Janela de Tempo. PhD Thesis. Department of Science Computer. Federal University of Minas Gerais - Brazil.

[3] Alvarenga, G.B., Mateus, G.R. and De Tomi, G., 2007. A genetic and set partitioning two-phase approach for the vehicle routing problem with time windows. Computers & Operations Research, 34(6), pp.1561-1584.

[4] Ampl.com. (2017). AMPL | Streamlined modeling for real optimization. [online] Available at: http://ampl.com/ [Accessed 12 Mar. 2017].

[5] Apte, A., 2010. Humanitarian logistics: A new field of research and action. Foundations and Trends in Technology, Information and Operations Management, 3(1), pp.1-100.

[6] Archetti, C., Savelsbergh, M.W. and Speranza, M.G., 2008. To split or not to split: That is the question. Transportation Research Part E: Logistics and Transportation Review, 44(1), pp.114-123.

[7] Balcik, B., Beamon, B.M. y Smilowitz, K., 2008. Last mile distribution in humanitarian relief. Journal of Intelligent Transportation Systems, 12(2), pp.51-63.

[8] Bansal, M. and Kianfar, K. (2013). CPLEX Concert Technology using C++, A tutorial. 1st ed. [ebook] Available at: http://people.tamu.edu/~bans1571/CPLEXConcertTutorial [Accessed 21 Feb. 2017].

[9] Campbell, A.M., Vandenbussche, D. and Hermann, W., 2008. Routing for relief efforts. Transportation Science, 42(2), pp.127-145..

[10] Campos, G.G.D., Yoshizaki, H.T.Y. and Belfiore, P.P., 2006. Genetic algorithms and parallel computing for a vehicle routing problem with time windows and split deliveries. Gestão & Produção, 13(2), pp.271-281.

[11] Caramia, M. y Dell´ Olmo, P., 2008. Multi-objective optimization. Multi-objective Management in Freight Logistics: Increasing Capacity, Service Level and Safety with Optimization Algorithms, 1, pp.11-36.

[12] Center for Engineering and Health, McCormick School of Engineering, Northwestern University. (n.d.). [online] Health & Humanitarian Logistics | Center for Engineering and Health | Northwestern's McCormick School of Engineering. Available at: http://www.mccormick.northwestern.edu/research/engineering-and-health-center/research/research-areas/health-humanitarian-logistics.html [Accessed 31 Jan. 2017].

[13] Chakraborty, M. and Chakraborty, U.K., 1997, September. An analysis of linear ranking and binary tournament selection in genetic algorithms. In Information, Communications and Signal Processing, 1997. ICICS., Proceedings of 1997 International Conference on (Vol. 1,

pp. 407-411). IEEE.

[14] Cho, Y.H., 2010. An efficient solving the travelling salesman problem: Global optimization of neural networks by using hybrid method. Traveling Salesman Problem, Theory and Applications, 1, pp.155-176.

[15] Clarke, G.U. and Wright, J.W., 1964. Scheduling of vehicles from a central depot to a number of delivery points. Operations Research, 12(4), pp.568-581.

[16] Cormen, T., Leiserson, C. y Rivest. R., 2001. Greedy algorithms. Introduction to Algorithms Chapter 16, 1, pp.370.

[17] Deb, K., Pratap, A., Agarwal, S. y Meyarivan, T.A.M.T., 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE transactions on evolutionary computation, 6(2), pp.182-197.

[18] Docs.oracle.com. (2017). Java SE 7 Swing APIs and Developer Guides. [online] Available at: http://docs.oracle.com/javase/7/docs/technotes/guides/swing/ [Accessed 12 Mar. 2017].

[19] Gallego, R. A., Escobar, A., Toro, E. M. and Romero, R., 2013. Técnicas heurísticas y metaheurísticas de optimización. Universidad Tecnológica de Pereira, Pereira - Risaralda - Colombia.

[20] Gnu.org. (2017). GLPK - GNU Project - Free Software Foundation (FSF). [online] Available at: https://www.gnu.org/software/glpk/ [Accessed 12 Mar. 2017].

[21] H. (2017). GLPK for Java – About. [online] GLPK for Java – About. Available at: http://glpk-java.sourceforge.net/ [Accessed 18 Mar. 2017].

[22] Huang, M., Smilowitz, K. and Balcik, B., 2012. Models for relief routing: Equity, efficiency and efficacy. Transportation Research Part E: Logistics and Transportation Review, 48(1), pp.2-18.

[23] J. (2017). jgraph/jgraphx. [online] GitHub. Available at: https://github.com/jgraph/jgraphx [Accessed 21 Feb. 2017].

[24] J. (2017). Fix CommandLineIndicatorRunner when normalizing fronts by santib · Pull Request #182 · jMetal/jMetal. [online] Available at: https://github.com/jMetal/jMetal/pull/182 [Accessed 12 Mar. 2017].

[25] Jakob, W. and Blume, C., 2014. Pareto Optimization or Cascaded Weighted Sum: A Comparison of Concepts. Algorithms, 7(1), pp.166-185.

[26] Kokash, N., 2005. An introduction to heuristic algorithms. Department of Informatics and Telecommunications, 1, pp.1-8.

[27] Kreinovich, V., Quintana, C. and Fuentes, O., 1993. Genetic algorithms: what fitness scaling is optimal?. Cybernetics and Systems, 24(1), pp.9-26.

[28] Lawler, E. L., Lenstra, J. K., Rinnooy Kan, A. H. G. y Shmoys, D. B., 1985. The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization. Wiley Interscience Series in Discrete Mathematics.

[29] Luenberger, D.G. y Ye, Y., 1984. Linear and Nonlinear Programming, 2, Reading, MA: Addison-wesley.

[30] Man, K.F., Tang, K.S. and Kwong, S., 1996. Genetic algorithms: concepts and applications [in engineering design]. IEEE transactions on Industrial Electronics, 43(5), pp.519-534.

[31] McClean, D., 2010. World disasters report 2010: Focus on urban risk. World Disasters Report 2010: Focus on Urban Risk. International Federation of Red Cross and Red Crescent Societies (IFRC).

[32] Nolz, P.C., Doerner, K.F., Gutjahr, W.J. and Hartl, R.F., 2010. A bi-objective metaheuristic for disaster relief operation planning. Advances in Multi-objective Nature Inspired Computing, pp.167-187. Springer Berlin Heidelberg.

[33] NP-Problem. (n.d.). [online] NP-Problem -- from Wolfram MathWorld. Available at: http://mathworld.wolfram.com/NP-Problem.html [Accessed 21 Feb. 2017].

[34] NP-Hard Problem. (n.d.). [online] NP-Hard Problem -- from Wolfram MathWorld. Available at: http://mathworld.wolfram.com/NP-HardProblem.html [Accessed 21 Feb. 2017].

[35] Rand, G.K., 2009. The life and times of the savings method for vehicle routing problems. ORiON, 25(2).

[36] Riquelme, N., Von Lücken, C. and Baran, B., 2015, October. Performance metrics in multi-objective optimization. In Computing Conference (CLEI), 2015 Latin American (pp. 1-11). IEEE.

[37] Sayyad, A.S. and Ammar, H., 2013, May. Pareto-optimal search-based software engineering (POSBSE): A literature survey. In Realizing Artificial Intelligence Synergies in Software Engineering (RAISE), 2013 2nd International Workshop on (pp. 21-27). IEEE.

[38] Sivanandam, S.N. and Deepa, S.N., 2007. Introduction to genetic algorithms. Springer Science & Business Media.

[39] Smith, J.C. y Taskin, Z.C., 2008. A tutorial guide to mixed-integer programming models and solution techniques. Optimization in Medicine and Biology, pp.521-548.

[40] Solomon, M.M., 1987. Algorithms for the vehicle routing and scheduling problems with time window constraints. Operations research, 35(2), pp.254-265.

[41] Soni, N. and Kumar, T., 2014. Study of various mutation operators in genetic algorithms. IJCSIT) International Journal of Computer Science and Information Technologies, 5(3), pp.4519-4521.

[42] Soni, N. and Kumar, T., 2014. Study of various mutation operators in genetic algorithms. IJCSIT) International Journal of Computer Science and Information Technologies, 5(6), pp.7235-7238.

[43] Sörensen, K. y Glover, F.W., 2013. Metaheuristics. Encyclopedia of Operations Research and Management Science, 1, pp.960-970. Springer US.

[44] Thangiah, S.R., Osman, I.H. and Sun, T., 1994. Hybrid genetic algorithm, simulated

annealing and tabu search methods for vehicle routing problems with time windows. Computer Science Department, Slippery Rock University, Technical Report SRU CpSc-TR-94-27, 69.

[45] Vacca, I. and Salani, M., 2009. The vehicle routing problem with discrete split delivery and time windows. In 9th Swiss Transport Research Conference (No. EPFL-CONF-152355).

[46] Van Veldhuizen, D.A. and Lamont, G.B., 2000. On measuring multiobjective evolutionary algorithm performance. In Evolutionary Computation, 2000. Proceedings of the 2000 Congress on (Vol. 1, pp. 204-211). IEEE.

[47] Viera, O., Moscatelli, S. and Tansini, L., 2012. Logística humanitaria y su aplicación en Uruguay. Gerencia Tecnologica Informatica, 11(30), pp.47-56.

[48] Web.cba.neu.edu. (2017). Benchmarking Problems. [online] Available at: http://web.cba.neu.edu/~msolomon/problems.htm [Accessed 17 Mar. 2017].

[49] Y. (2016). jmathplot. [online] GitHub - yannrichet/jmathplot: Java interactive 2D and 3D plots (no OpenGL). Available at: https://github.com/yannrichet/jmathplot [Accessed 21 Feb. 2017].

[50] Zitzler, E., Thiele, L., Laumanns, M., Fonseca, C.M. and Da Fonseca, V.G., 2003. Performance assessment of multiobjective optimizers: An analysis and review. IEEE Transactions on evolutionary computation, 7(2), pp.117-132.