Universidad
de La Laguna

Escuela Superior de
Ingeniería y Tecnología
Sección de Ingeniería Informática

Trabajo de Fin de Máster

Dynamic Routes

Metaheurísticas para la resolución de tareas de Big Data

Martín Chinea, Kevin

La Laguna, 2 de julio de 2018

Dña. **María Belén Melián Batista**, con N.I.F. 44311040E, Pofesora Titular de Universidad adscrita al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor.

D. **José Marcos Moreno Vega** , con N.I.F. 42841047M Profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como cotutor.

C E R T I F I C A N

Que la presente memoria titulada:

"*Dynamic Routes*"

ha sido realizada bajo su dirección por D. **Kevin Martín Chinea**, con N.I.F. 42418683J.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 2 de julio de 2018

Agradecimientos

En primer lugar, agradecer a María Belén Melián Batista y a José Marcos Moreno Vega, por permitirme participar en este proyecto y toda la ayuda y asesoramiento que me han proporcionado.

Y mi familia, ya que sin ellos no podría haber llegado a donde he llegado.

Resumen

En el siguiente proyecto se describirá el estudio de cómo influye el dinamismo en un problema de optimización, en concreto se indagará en cómo el conocimiento completo de un problema puede variar con respecto a que el problema cambie su estructura cada cierto tiempo. En el caso estudiado, el problema de enrutamiento de vehículos con capacidades, se compararán las soluciones obtenidas conociendo todos los clientes con las soluciones alcanzadas al ir añadiendo un cierto número de clientes periódicamente o incluso de manera dinámica. Todo esto con el fin de obtener medidas objetivas de las mejores formas de enfrentarnos a estos problemas, no sólo en la teoría sino en la práctica con casos reales, como lo hacen grandes empresas de transporte que se enfrentan a este tipo de problemas continuamente.

Palabras clave: Problemas dinámicos, CVRP, problema de enrutamiento de vehículos con capacidades, optimización, heurísticas, metaheurísticas.

Abstract

This project describes the study of how tackling the dynamism of a problem as necessary to solve it. This means how the knowledge of the problem can modify it and change the value of the solution. For example, in the case study, the capacitated vehicle routing problem, we consider three different approaches for solving it. The first case corresponds to the static problem, in which all customers are known at the beginning of the planning horizon. The second and the third cases correspond to including a percentage of dynamic customers. In the second case, the customers are added on a regular basis of time, while in the third case, the customers are added in a specific time.

Keywords: *Dynamic problems, CVRP, capacitated vehicle routing problem, optimization, heuristics, metaheuristics.*

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

En este primer capítulo trataremos una breve introducción a los antecedentes que han originado el desarrollo de este proyecto, y de la misma forma se tratará el objetivo final que se pretende obtener, el cuál veremos que se irá logrando a lo largo de cada uno de los siguientes capítulos.

Una breve introducción a lo desarrollado en este trabajo fin de máster es el estudio de un problema de rutas dinámico, definiendo ciertas heurísticas para solucionarlo y aplicando diversos grados de dinamismo al problema seleccionado. Este dinamismo está basado en la aparición de nuevos clientes que tendrán que ser añadidos al problema en tiempo de ejecución. Esto significa que ya los vehículos se encuentran atendiendo a clientes, por lo que se encontrará la situación de que ciertos clientes asignados previamente tendrán su demanda satisfecha mientras que otros seguirán a la espera. Estos últimos serán los que, con los nuevos clientes añadidos, se podrán modificar para obtener una solución global final y así poder analizar los resultados obtenidos.

1.1. Antecedentes

Actualmente y desde hace unos años, se ha vivido una gran revolución en cuanto a tecnología. Cada poco tiempo aparecen nuevas herramientas que agilizan, optimizan y ayudan en ciertas actividades cotidianas, y esto va desde una simple aplicación móvil hasta un nuevo dispositivo electrónico de IoT [1] para el hogar. Esto ha tenido su efecto en la sociedad como puede ser en el incremento de la cantidad de datos que genera una persona. Haciendo que además estos datos sean mucho más variados y complejos, generando otros problemas como podrían ser el almacenamiento, el análisis, la visualización, etc. A través de esto, uno de los aspectos que más se han demandado es el estudio e investigación de estos datos, ya que a partir de estas investigaciones es posible obtener patrones o incluso relaciones entre ellos, que pueden ser de interés a las empresas, ya que si lo conocen les permitirían tener la ventaja sobre sus competidores.

[1] Concepto que se refiere a la interconexión digital de objetos cotidianos con Internet.

Por otro lado, y entrando más en conceptos propios del proyecto, tenemos las heurísticas, metaheurísticas, técnicas, métodos o procedimientos inteligentes con lo que se trata de resolver el problema considerado.

Es por estos aspectos definidos por lo que surge este proyecto. Se tratará el desarrollo y la investigación de un problema de optimización, en concreto, el problema de enrutamiento de vehículos con capacidades. Se desarrollarán diversas heurísticas, aplicándole además un cierto dinamismo en los datos con los que se trabaja, modificando así el enfoque del estudio del problema.

1.2. Objetivo

El objetivo principal de este proyecto es el estudio e investigación del problema de enrutamiento de vehículos con capacidades (CVRP), centrando la investigación en diversos aspectos.

En primer lugar, resolvemos el problema considerado mediante heurísticas y analizamos los resultados obtenidos; tanto desde el punto inicial, donde observaremos los resultados proporcionados por las heurísticas que nos facilitan una solución base inicial, hasta las heurísticas encargadas de mejorar estas soluciones iniciales.

Seguidamente, se aborda el problema desde un punto de vista dinámico. Este planteamiento se enfoca probando distintos grados de dinamismo en las instancias del problema; esto es, se modifica el porcentaje de nuevos clientes dinámicos que se añaden posteriormente a tener una solución construida con clientes estáticos. Finalmente, se comparan los resultados y se analiza cómo afecta este atributo al desarrollo y la calidad de las soluciones. Por lo tanto, la manera de proseguir para lograr este objetivo es la siguiente. Una vez hecho nuestro estudio inicial, tendremos por un lado la mejor heurística que nos proporciona una solución inicial y un orden bien definido de aplicación de todas las heurísticas que modifican la solución. Con una instancia del problema establecido, seleccionaremos un porcentaje de clientes, que serán los estáticos, mientras que el resto quedarán en una lista de dinámicos. Con estas heurísticas resolvemos la parte estática de la instancia. Una vez construida la solución, añadimos elementos de la lista de clientes dinámicos a la solución establecida, teniendo en cuenta que existen clientes ya visitados, los cuales no se pueden modificar en la solución establecida. Se repite la ejecución de las heurísticas encargadas de mejorar la solución. Así hasta finalizar con los clientes en la lista de dinámicos.

Por último, se trata la integración de estas heurísticas con jMetalSP, herramienta desarrollada para trabajar con problemas dinámicos multiobjetivo y Spark [2]. Con ella se contempla el resultado de cada solución cada vez que se le vayan añadiendo nuevos clientes.

[2]Posteriormente se describe en mayor detalle tanto el uso como el funcionamiento de esta herramienta.

Capítulo 2

Problema

En este capítulo trataremos el problema seleccionado, el problema de enrutamiento de vehículos (Vehicle Routing Problem o VRP). En concreto, trabajaremos con una variante del problema la cual está definida por las capacidades del vehículo, Capacitated Vehicle Routing Problem (CVRP).

2.1. Problema

El problema de enrutamiento de vehículos se trata de un problema de optimización, generalizado del conocido problema del viajante de comercio (TSP[1]). Este responde a la pregunta de cuál es el conjunto óptimo de rutas para satisfacer las demandas de un grupo de clientes mediante una flota de vehículos. Como se puede deducir a partir del planteamiento de la pregunta de este problema, está formado por un conjunto de localizaciones las cuales corresponden a clientes, que tienen una determinada demanda de un producto y una localización. A su vez, estos productos demandados estarán guardados en otras localizaciones definidas como los depósitos, y como cabe esperar, existirá un coste desde una localización a otra.

Una forma de definir este problema es mediante un grafo no dirigido

$$G = (V, E),$$

donde tenemos que V corresponde con el conjunto de vértices o nodos

$$V = \{v_0, v_1, ..v_n, \}$$

y E es el conjunto de ejes o aristas

$$E = \{(v_i, v_j)/v_i, v_j \in V, i < j.\}$$

Por lo tanto, en E se define una matriz de costes, distancias o tiempos de viaje entre dos nodos

$$Coste\ entre\ v_i\ y\ v_j : C_{i,j} = (c_{i,j}).$$

[1] https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/

El primer vértice vendrá definido como el nodo de abastecimiento o depósito, m; el número de vehículos. Las ciudades o clientes se representan mediante el conjunto de n vértices. Dentro del nodo se define la demanda de éste y el tiempo de entrega.

$$Abastecimiento : v_0$$

$$Cantidad\ de\ transportes :\ m$$

$$Cantidad\ de\ nodos :\ n$$

$$Demanda : q_i$$

$$Tiempo\ de\ entrega : \partial_i$$

En concreto, nos vamos a centrar en la variante del problema con capacidades, en la que cada vehículo tiene una capacidad Q uniforme del producto básico.

El objetivo de este problema es minimizar la suma del coste de viaje, cumpliendo con la característica de que la capacidad propia del vehículo no supera las demandas de los clientes visitados en cada ruta. Los clientes sólo se pueden visitar una vez. Para obtener esta solución, tendremos que generar las rutas que recorren cada uno de los clientes. Una ruta estará formada por una partición del conjunto V, con un determinado orden en los nodos que la forman, el orden de visitas. Las rutas comienzan y finalizan en el nodo depósito. Por lo tanto, el coste de una ruta

$$R_i = \{v_0, v_1, ..v_k\}\ con\ v_j \in V_i\ si\ 0 <= j <= k\ y\ v_0 = v_{k+1},$$

viene dado por

$$Cost(R_i) = \sum_{j=0}^{k} c_{j,j+1} + \sum_{j=0}^{k} \partial_j.$$

Generando como coste total de la solución la suma de cada uno de los costes de las rutas

$$S = \sum_{i=1}^{m} Cost(R_i).$$

Este problema es NP-duro, por lo que las implementaciones más usadas para resolver el problema están basadas en heurísticas, ya que suelen producir buenos resultados en tiempos computacionales razonables. En casos prácticos se pueden alcanzar ahorros del 5 % en compañías donde el transporte es un componente básico en el desarrollo de sus servicios [7], [9], [4].

2.2. Instancias

A lo largo del proyecto se ha trabajado con diversas instancias de este problema. Las primeras instancias usadas se pueden encontrar en la página de la Universidad de Málaga [2]. Estas tratan los aspectos más simples del problema, suponen que una flota fija de vehículos de reparto debe atender a las demandas conocidas de los clientes de un producto desde un depósito, con la restricción de que cada vehículo tiene una capacidad limitada. Estos ficheros de instancias están formados por el nombre, la descripción, el tipo, la dimensión del problema, el tipo de distancia, la capacidad del vehículo, y finalmente el listado de todas las localizaciones (depósitos y clientes), definidos por las coordenadas (longitud y latitud) y la demanda.

Una vez realizada la primera parte del proyecto, basado en el estudio de heurísticas que hemos establecido, es necesario trabajar con instancias que tengan definidos tiempos de llegada de los clientes, para poder trabajar con dinamismo. De esta manera, sabemos qué nodos no están desde el inicio de la ejecución, y por lo tanto, saber en qué momento estarán disponibles y se podrán añadir a las rutas generadas. Es por ello que se recurre al benchmark de instancias con dinamismo de Solomon. Se trata de instancias de pruebas en las que se definen problemas de enrutamiento de vehículos con capacidades y con ventanas de tiempo. Estos ficheros de instancias están formados por estructuras similares a la definida anteriormente con la diferencia de que se añade si se trata de un nodo presente desde el inicio de la instancia, el tiempo de apertura/cierre del cliente, la fecha del vencimiento y la duración del servicio.

Para trabajar de una forma mucho más sencilla con los datos se ha establecido una estructura genérica con las instancias usadas para el programa desarrollado. En ambos casos, tanto en instancias completamente estáticas como las que tienen un porcentaje de dinámismo, se trabaja con una parte genérica en la que se describe el problema; definiendo aspectos como el nombre de la instancia, el número de nodos y la capacidad. Podemos ver la estructura de este nuevo fichero generado en la Figura 2.1.

Para diferenciar una instancia de otra, tenemos dos tipos de definiciones de nodos. En el caso de que se trate de una instancia completamente estática, un nodo estará formado por un depósito, unas coordenadas y la demanda que requiere satisfacer. La estructura que se formaría en el fichero la podemos ver en la Figura 2.2.

El segundo caso es similar, con la diferencia de que se trabaja con más atributos. Se incluyen algunos nuevos como pueden ser si se añaden de manera dinámica y el tiempo en el que se añadiría. La Figura 2.3 es un ejemplo de la estructura de estos nodos en el fichero generado.

Como se puede ver estos ficheros generados están formados por los mismos

[2] http://neo.lcc.uma.es/vrp/vrp-flavors/capacitated-vrp/

```
<vehicleRouteProblemInstance>
 <name>A-n32-k5</name>
 <comment>
  (Augerat et al, Min no of trucks: 5,
  Optimal value: 784)
 </comment>
 <type>CVRP</type>
 <dimension>32</dimension>
 <edgeWeightType>EUC_2D </edgeWeightType>
 <capacity>100</capacity>
 <graph>
        ....
 </graph>
</vehicleRouteProblemInstance>
```

Figura 2.1: Estructura de datos genérica de las instancias usadas.

```
<graph>
 <node>
  <deposit>true</deposit>
  <lat>82</lat>
  <long>76</long>
  <demand>0</demand>
 </node>
       ....
</graph>
```

Figura 2.2: Ejemplo de nodo para el caso estático.

```
<graph>
 <node>
  <deposit>true</deposit>
  <lat>40</lat>
  <long>50</long>
  <demand>0</demand>
  <serviceCost>0</serviceCost>
  <readyTime>0</readyTime>
 </node>
       ....
</graph>
```

Figura 2.3: Ejemplo de nodo para instancia dínamica.

datos de las instancias originales, siguiendo simplemente una estructura en xml para facilitar la carga de los ficheros a la estructura de datos del programa desarrollado.

Capítulo 3

Herramientas

En este capítulo se definen y describen las herramientas utilizadas a lo largo del desarrollo de este proyecto.

La primera de las elecciones en cuanto a herramientas se trata de jMetalSP. JMetalSP es una plataforma de software para optimizaciones dinámicas multi-objetivo en problemas de Big Data, la cual combina otras dos herramientas:

- jMetal, formado por las infraestructuras para la optimización de problemas de Big Data y las metaheurísticas que lo resuelven.
- Spark, framework de computación que permite administrar las fuentes de datos de transmisión en clústeres de Hadoop para el procesado de grandes cantidades de datos.

A continuación se detallan los requerimientos generales de esta herramienta en su versión actual, la 2.0.

- Java JDK 8
- Apache Maven
- jMetal 5.52

Y en el caso de utilizar Spark, es necesario una versión de esta herramienta:

- Spark 2.3.0, o superior.

La elección de esta herramienta viene determinada principalmente por su enfoque ya que une los temas que se busca tratar en este proyecto, metaheurísticas y Big Data [1]. Además de esto, destaca también por su funcionalidad, ya que se puede combinar con cualquier algoritmo que se defina previamente. Teniendo no sólo la posibilidad del dinamismo, sino la posibilidad de montar clúster de Spark para realizar los cálculos. A parte de esto, y otra de las ventajas que ofrece es su licencia. Actualmente tiene una licencia MIT, lo que permite su uso

comercial, la distribución, el uso privado, y una de las partes más atractivas, la modificación, ya que da la posibilidad de participar en el desarrollo de dicha herramienta con posibles variantes de funcionamiento.

Esto nos lleva a nuestro siguiente punto, el lenguaje de programación. Como es evidente al tener ya la herramienta seleccionada, y ésta está desarrollada en un determinado lenguaje, no sólo por el motivo de contribuir al desarrollo de jMetalSP, sino por la simple integración, el lenguaje tiene que ser el mismo, Java. Java[1] es un lenguaje de programación creado en 1991 y publicado en 1995 por Sun Microsystem, se trata de un lenguaje de propósito general, concurrente, orientado a objeto y multiplataforma. El código se escribe en archivos formato .java para posteriormente compilarlos y generar estos códigos byte en archivos .class que corresponden a las instrucciones para ejecutarse en la JVM, Java Virtual Machine, que se encarga de interpretar dichos códigos.

Figura 3.1: Logo lenguaje de programación Java

Como en cualquier buen desarrollo software, se han usado un cierto número de test con el fin de liberar comprobaciones necesarias en cuanto al código escrito. Para esto se ha recurrido a JUnit, ya que es la principal y más destacada herramienta de pruebas unitarias en Java. JUnit, se trata de un framework, o conjunto de bibliotecas, utilizadas en programación, que como hemos definido trabaja en Java y cuyo objetivo es realizar pruebas unitarias. Permitiendo con esto evaluar el funcionamiento del desarrollo implementado y tener una cobertura del código evaluado. Definiendo así que el comportamiento del programa es el correcto y que futuros cambios no afecten a lo ya desarrollado.

Figura 3.2: Logo JUnit

Como se ha nombrado anteriormente, una de las posibilidades al usar jMetalSP como herramienta es la de colaborar en su desarrollo o desarrollar alguna versión a partir de la usada. Es por esto nuestra elección de la siguiente

[1]Página oficial: https://www.java.com/es/download/

[1]Las pruebas unitarias son fragmentos de código que ejecutan una funcionalidad específica de código.

[1]Porcentaje de código probado mediante pruebas unitarias.

herramienta, Maven. Se trata de una herramienta que usa jMetalSP para gestionar todas sus dependencias externas y además tener el proyecto dividido en submódulos. Se ha seleccionado esta herramienta con el fin de que, en el caso de realizar dicha colaboración sea mucho más simple su integración. Una breve descripción de Maven[2] sería que es una herramienta para la gestión y compresión de proyectos de software en Java, se usa para simplificar las tareas al desarrollador al gestionar todo el proyecto utilizando un Project Object Model (POM); en el que se describen las características del proyecto, cómo podrían ser sus dependencias, los módulos que lo forman, componentes externos, el orden de construcción de los elementos, etc.

Figura 3.3: Logo Maven

La última herramienta usada en este proyecto es el IDE de desarrollo Eclipse. Se trata de una plataforma de software para el trabajo en entornos de desarrollo integrado (IDE). En este caso, se ha usado el del lenguaje seleccionado (Java), el Java Development Toolkit. Dispone de diversas herramientas como por ejemplo, un editor de texto, analizador sintáctico, compilación en tiempo real, pruebas unitarias con JUnit, control de versiones con CVS, etc.

Figura 3.4: Logo herramienta Eclipse

El hecho de usar esta herramienta frente a un editor convencional de texto, es por la sencillez de tener todas las herramientas integradas en una, ya que en el caso de usar un editor de texto habría que trabajar con cada herramienta usada independientemente. Por otro lado, elegir Eclipse frente a otros competidores es por ciertos atributos que presentan como podrían ser la intuitividad al trabajar con este, el fomento del código abierto que representa y que tanto funcionalidades como plugins son gratuitos, no como ocurre en otros IDEs, como por ejemplo IntelliJ Idea.

[2]Página oficial: https://maven.apache.org/

Capítulo 4

Fases y resultado computacional

A lo largo de este capítulo se describen cada una de las fases desarrolladas en el proyecto, describiendo en cada una de ellas los objetivos y las tareas necesarias para alcanzarlos.

4.1. Aproximación al problema

En esta primera fase, tenemos como objetivo una primera aproximación al problema, por lo que la primera tarea trata de la carga y del desarrollo de las estructuras de datos que corresponden con las instancias del problema definido, Capacitated VRP. Para ello se usan inicialmente las instancias publicadas por la Universidad de Málaga[1]. Para esto creamos una clase Instance, donde se guardan cada uno de los atributos que corresponden a la instancia y donde se crea la matriz de costes entre los nodos existentes calculada según la distancia euclidiana[2]. Los nodos estarán representados por una clase propia, Location, en la que se guardan todos los datos específicos de cada nodo. En concreto y definiendo la parte más completa del problema, donde se trabajará con posibles nodos dinámicos. Podemos definir que los atributos que lo formarán son un identificador del nodo, una demanda, un tiempo que emplea el vehículo en el servicio, el tiempo a partir del cual está disponible y las coordenadas, que a su vez están formadas por otra clase, representando tanto la longitud como la latitud. Es posible ver estas estructuras en el apéndice B, Figura B.1.

Previamente a realizar las heurísticas que trabajarán para generar la solución al problema tenemos que definir otras clases. Por ejemplo, una clase propia que defina la solución del problema; se ha definido con el nombre de CVRPSolution. Esta clase estará formada por ciertos atributos como pueden ser los costes

[1]Instancias: http://neo.lcc.uma.es/vrp/vrp-instances/capacitated-vrp-instances/http://neo.lcc.uma.es/vrp/vrp-instances/capacitated-vrp-instances/

[2]Distancia ordinaria entre dos puntos la cual se define a partir del teorema de Pitágoras

```
while thereAreCustomersToVisit {
 newRoute
 newRoute.addADeposit()

 while isTheVehicleHasCapacity(){
  customer = chooseTheClosestCustomers()

  newRoute.addToTheRoute(customer)
  decreaseTheVehicleCapacity()
 }
}
```

Figura 4.1: Pseudocódigo Nearest Neighbor.

totales de la solución, los costes cubiertos en un determinado periodo de tiempo y la lista de rutas que forman dicha solución. Estas rutas a su vez son otra clase, en donde se definirá cada coste específico de la ruta, tanto el total como el cubierto por el vehículo en un determinado instante de tiempo, una lista con el coste de los vehículos de llegada a cada nodo y como es lógico, la lista con todos los nodos clientes representados mediante la clase Location, definida en el párrafo anterior. Es posible ver estas clases en la Figura B.2, del apéndice B.

Una vez realizado esto ya se tiene la estructura en la que se guardarán los datos necesarios, por lo que el siguiente paso es realizar las heurísticas que generen una solución inicial. En este caso se definirán tres heurísticas distintas con las que empezar a trabajar nuestro problema y así generar la solución inicial con la que se trabajará posteriormente con otras heurísticas. Como veremos a continuación, las dos últimas heurísticas se basan en algoritmos GRASP (Greedy Randomized Adaptive Search)[3]. Estas son las heurísticas definidas para generar las soluciones iniciales, relacionadas a continuación.

- Nearest Neighbor (NN).

  Como su nombre indica, el algoritmo del vecino más próximo trata de partir del nodo actual al nodo más cercano. Como se ha definido partiendo inicialmente de un nodo depósito y volviendo a este una vez que el vehículo no pueda satisfacer la demanda de más clientes. Se repite este proceso hasta visitar todos los nodos clientes del problema. En la Figura 4.1 se encuentra el pseudocódigo de este algoritmo.

- GRASP 1 - Closest First (CF)

[3]Heurísticas de construcción que consisten en generar la solución mediante iteraciones compuestas por selecciones aleatorias.

```
while thereAreCustomersToVisit {
 newRoute
 newRoute.addADeposit()
 newRoute.addOneOfTheClosestCustomersFromTheDeposit()

 while isTheVehicleHasCapacity() {
  customer = chooseTheClosestCustomers()

  newRoute.addToTheRoute(customer)
  decreaseTheVehicleCapacity()
 }
}
```

Figura 4.2: Pseudocódigo Closest First.

Esta segunda heurística tiene el mismo objetivo, generar una solución inicial del problema. Trata de proporcionar un poco de aleatoriedad a la obtención de ésta. Para ello tomamos una estructura de trabajo similar a la del algoritmo del vecino más cercano, con la diferencia de que, la primera vez que se selecciona el primer nodo cliente de la ruta, se tratará de un nodo elegido al azar de un grupo de los nodos más cercanos. Una vez se está en ese cliente se procede seleccionando el cliente más cercano hasta que el vehículo no pueda satisfacer más demanda y volver a un depósito. Se reinicia el proceso de crear una ruta hasta visitar todos los clientes. En la Figura 4.2 se puede encontrar el pseudocódigo de este algoritmo.

- GRASP 2 - Furthest First (FF)

  Esta heurística trabaja de forma similar a la definida anteriormente, exceptuando que, en vez de seleccionar al azar un nodo cliente de un grupo de los nodos más cercanos, se escoge de un grupo de los nodos más alejados. El resto del algoritmo funciona igual. Una vez estamos en ese nodo cliente elegido, vamos satisfaciendo la demanda de los clientes más cercanos hasta que el vehículo no tenga más recursos y volvemos al depósito. Se repite hasta que no existan clientes sin visitar. Podemos encontrar el pseudocódigo de este algoritmo en la Figura 4.3.

Una vez tenemos desarrolladas estas heurísticas, que independientemente de cuál ejecutemos nos generará una solución inicial del problema, tenemos que pensar en modificar dicha solución. Esto es debido a que es muy probable que esa solución inicial se pueda mejorar y es por ello que se definen otras heurísticas que a partir de una solución generan una modificación de esta mejorándola. A continuación se describen las cuatro heurísticas definidas para este propósito,

```
repeat thereAreCustomersToVisit {
 newRoute
 newRoute.addADeposit()
 newRoute.addOneOfTheFarthestCustomersFromTheDeposit()

 while isTheVehicleHasCapacity() {
  customer = chooseTheClosestCustomers()

  newRoute.addToTheRoute(customer)
  decreaseTheVehicleCapacity()
 }
}
```

Figura 4.3: Pseudocódigo Furthest First.

las cuales cada una está basada en un objetivo:

- 2-Opt

  Algoritmo de búsqueda local simple, que compara todas las posibles combinaciones al intercambiar dos nodos de la ruta definida y reordenarlos; todo con el fin de eliminar cruces que incrementan innecesariamente el coste de la ruta. En la Figura 4.4 se representa el pseudocódigo de este algoritmo.

- Relocate

  Este algoritmo trata de obtener el mejor movimiento que mejoraría la solución actual sacando un nodo de una ruta y poniéndolo en otra distinta. Para buscar este movimiento se comprueba cómo sería la solución cambiando cada uno de los nodos de cada una de las rutas a una ruta a la que no pertenece, guardando siempre el mejor cambio para finalmente realizarlo. Podemos observar cómo se define el pseudocódigo de este algoritmo en la Figura 4.5.

- SwapIntra

  Algoritmo que busca qué intercambio de nodos mejoraría la ruta, siempre y cuando este cambio se ejecute entre nodos de la misma ruta. Para esto es necesario comprobar todos los cambios posibles entre cada nodo de una misma ruta en cada una de las rutas de la solución, obteniendo el mejor y realizándose. La Figura 4.6 corresponde con el pseudocódigo de este algoritmo.

- SwapInter

```
repeat until no improvement is made {
 startAgain:
 bestDistance = calculateTotalDistance(existingRoute)

 from i = 1 to numberOfNodesEligibleToBeSwapped - 1 {

  from k = i + 1 to numberOfNodesEligibleToBeSwapped {
   newRoute = 2optSwap(existingRoute, i, k)
   newDistance = calculateTotalDistance(newRoute)

   if (newDistance < bestDistance) {
    existingRoute = newRoute
    goto startAgain
   }
  }
 }
}

2optSwap(route, i, k) {
 newRoute

 from j = 0 to i - 1 {
  newRoute.add(route.getCustomer(j))
 }

 from j = k to i {
  newRoute.add(route.getCustomer(j))
 }

 from j = k + to route.size() {
  newRoute.add(route.getCustomer(j))
 }

 return newRoute;
}
```

Figura 4.4: Pseudocódigo 2-opt.

```
relocate(actualSolution) {
 newSolution = actualSolution
 bestSolution = actualSolution

 from i = 0 to numberRoutes {
  from j = 1 to numberNodesInRoute i {
   from k = 0 to numberRoutes {
    if(i != k) {
     move the node j to the best position in route k

     if(bestSolution > newSolution) {
      bestSolution = newSolution
     }

     newSolution.restoreTheChanges()
    }
   }
  }
 }

 return bestSolution
}
```

Figura 4.5: Pseudocódigo Relocate.

```
swapIntra(actualSolution) {
 newSolution = actualSolution
 bestSolution = actualSolution

 from i = 0 to numberRoutes {
  from j = 1 to numberNodesInRoute i {
   from k = 1 to numberNodesInRoute i {
    swapNodes(j, k)

    if(bestSolution > newSolution) {
     bestSolution = newSolution
    }

    newSolution.restoreTheChanges()
   }
  }
 }

 return bestSolution
}
```

Figura 4.6: Pseudocódigo Swap Intra.

```
swapIntra(actualSolution) {
 newSolution = actualSolution
 bestSolution = actualSolution

 from i = 0 to numberRoutes {
  from j = 1 to numberNodesInRoute i {
   from k = 0 to numberRoutes {
    if(i != k) {
     from h = 1 to numberNodesInRoute k {
      swapNodes(j, k)

      if(bestSolution > newSolution) {
       bestSolution = newSolution
      }

      newSolution.restoreTheChanges()
     }
    }
   }
  }
 }
 return bestSolution
}
```

Figura 4.7: Pseudocódigo Swap Inter.

Algoritmo similar al anterior con la característica diferente de que el cambio se tiene que realizar entre nodos localizados en distintas rutas. Se ha definido su pseudocódigo en la Figura 4.7.

Una vez implementados estos algoritmos, se diseñará la estructura en la que serán usados. En un primer paso se usa una de las heurísticas que generan la solución inicial [4]. Y con ésta, se realizarán modificaciones iterativas utilizando estas últimas heurísticas de mejora [5] combinándolas en un cierto orden definido (el cual modificaremos en la parte de experiencia computacional para ver la mejor combinación).

El marco metaheurístico usado en este proyecto es el de Búsqueda por Entornos Variables (Variable Neighborhood Search - VNS) [5], [13]; específicamente usaremos un algoritmo de búsqueda por entornos general (General Variable Neighborhood Search - GVNS). Se definirá en el código como la clase GVNS.

[4]Heurísticas que generan una solución inicial: Nearest Neighbor, Closest First o Furthest First

[5]Heurísticas de modificación: 2-opt, relocate, swapintra y swapinter

El objetivo de este algoritmo es moverse a través de los óptimos locales modificando la solución actual. Con esto lo que se busca es encontrar el óptimo local en un rango lo más amplio posible. Trabaja buscando en diversos entornos de la solución actual, los cuales en cada iteración estarán más distantes ya que varía su conjunto de estructuras de entorno mediante una fase de agitación [6] y otra fase de modificación [7]. El número de ejecuciones vendrá definido por un cierto parámetro, que puede ser un tiempo máximo de CPU, número máximo de iteraciones, número máximo de iteraciones entre mejora y mejora, etc. En este caso se ha seleccionado un número $kMax$ de estructuras de entorno. En la Figura 4.8, se puede apreciar el desarrollo del pseudocódigo de este algoritmo.

Entrando en más detalles sobre el desarrollo del código, es una buena práctica definir dos funciones independientes para esta estructura. Una en la que se ejecute la primera heurística (que genera la solución inicial) y otra que ejecute las heurísticas que la modifican. Y como es lógico, en el caso de la primera tiene que llamar a la segunda una vez genere la solución inicial. Todo esto es por el simple hecho de que, la primera vez que se ejecuta el código genera una solución inicial que posteriormente se intentará mejorar. Una vez finalice y en el caso de que se añada un nodo (o nuevo cliente), es necesario tener de manera independiente las funciones correspondientes a buscar nuevas mejoras en la solución establecida hasta el momento. Porque quizás previamente a este nuevo nodo no podía mejorar pero añadiendolo sí, haciendo que sea necesario tener alguna forma de llamar a esas heurísticas de modificación y buscar esa mejora.

4.2. Simulación de reparto

En este apartado del desarrollo del proyecto se simula cómo una flota de vehículos realiza los recorridos. En este caso se tratará de un número infinito de vehículos para cubrir las rutas de la solución. Todo con el objetivo de ver cómo esta simulación influye en las rutas. En concreto al introducir nuevos nodos como clientes y ya existan ciertos puntos visitados.

En la Figura B.5, del apéndice B está disponible el diagrama de estas clases. Podemos ver que la clase principal estará formada por un objeto que como función realizará la simulación de un número determinado de vehículos, definidos como ya hemos dicho por el número de rutas, y con una capacidad definida según la instancia del problema usada.

La parte más importante de este desarrollo es el cálculo de los costes entre nodos de una ruta, ya que nos dirá en qué posición se encuentra un vehículo en un determinado instante de tiempo. Para este cálculo se establece una variabilidad del +-10/15 por ciento del coste de ir de un nodo cliente inicial al

[6] Heurísticas seleccionada para la fase de sacudida: relocate

[7] Heurísticas que participan en la modificación dentro del GVNS: 2-opt, relocate, swapintra y swapinter

```
currentSolution
k = 1
while k < kMax
 neighbordSolution = Shaking
 Local search
  from l = 1 to lMax
   newSolution = Find a new neighbord with l algorithm

   if newSolution is better than neighbordSolution
    updateNewSolution
   else
    l = l + 1

   if newSolution is better than currentSolution
    updateCurrentSolution
   else
    k = k + 1
}
```

Figura 4.8: Pseudocódigo GVNS.

siguiente, todo esto con el fin de tener un mayor realismo, ya que estos costes son variables. Por ejemplo, en la circulación en una carretera no es igual siempre, en ciertas horas puntuales se incrementa mientras que en otras pasa todo lo contrario y el coste es mucho menor. Por lo tanto, el objetivo de esta clase es definir la posición y el tiempo que tarda en llegar a cada nodo cada uno de los vehículos que recorren cada una de las rutas. Actualizando así estos atributos en cada una de las rutas existentes en la última solución generada por las heurísticas.

Todo esto es para el caso de que se añada un nuevo cliente. Cuando ocurra se comprobará en cada una de las rutas la posición actual de los vehículos; ya que como es comprensible, el nuevo cliente se podrá añadir en esa ruta determinada a partir de esa posición. Para añadirlo buscamos la ruta que empeore menos la solución, y pueda satisfacer la demanda del cliente, ordenando las rutas por sus centroides más cercanos al nuevo punto. En el caso en el que no exista ninguna ruta que pueda satisfacer al nuevo cliente, se añadirá una nueva ruta que pueda hacerlo. Una vez tengamos el nuevo cliente añadido, se realizará de nuevo la ejecución de las heurísticas para mejorar las soluciones, con el fin de mejorar la nueva solución generada de tener un nuevo cliente, con la pequeña diferencia de que los algoritmos se ejecutarán como punto inicial donde se encuentre el vehículo en ese instante de tiempo, y a su vez, las evaluaciones de las soluciones generadas que se hagan estarán basadas en el coste de la ruta que falta por

satisfacer.

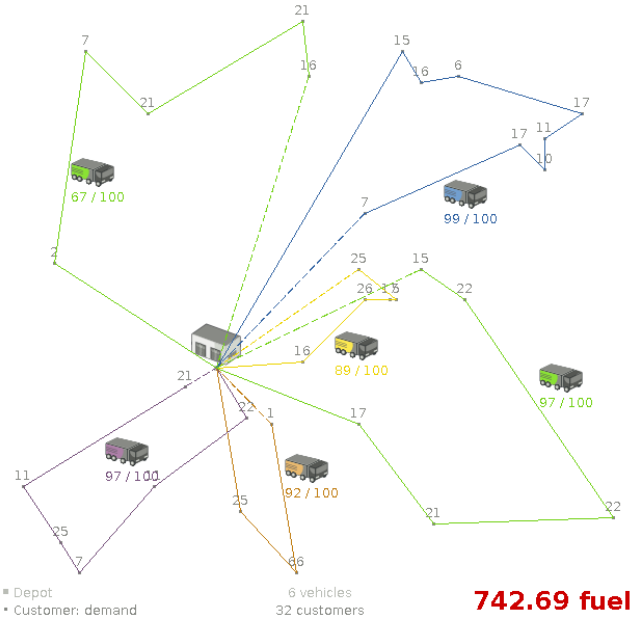

Figura 4.9: Ejemplo representativo de una simulación

Esta simulación se realizará cada vez que se ejecute la parte del GVNS de los modificadores, para así estar preparados para futuros nuevos clientes.

4.3. Integración en jMetalSP

Como se ha definido en el apartado de herramientas, jMetalSP es una plataforma de software para la optimización dinámica de Big Data multiobjetivo formada por jMetal y el sistema de cómputo de Apache Spark. Actualmente es un proyecto en continuo desarrollo teniendo en sus propios repositorios y en los de Maven como última versión la 2.0.

Podemos ver en la Figura 4.9 la arquitectura actual que define esta herramienta.

Como podemos ver, la aplicación jMetalSP está compuesta por:

- Una instancia de la clase DynamicProblem, que se trata del problema a resolver.
- Una instancia de la clase DynamicAlgorithm, definiendo el algoritmo a ejecutar.
- Uno o varios objetos de StreamingDataSource, encargados de procesar datos entrantes y gestionar los cambios en la instancia definida del DynamicProblem.
- Uno o más objetos AlgorithmDataConsumer, que reciben los resultados que genera DynamicAlgorithm.

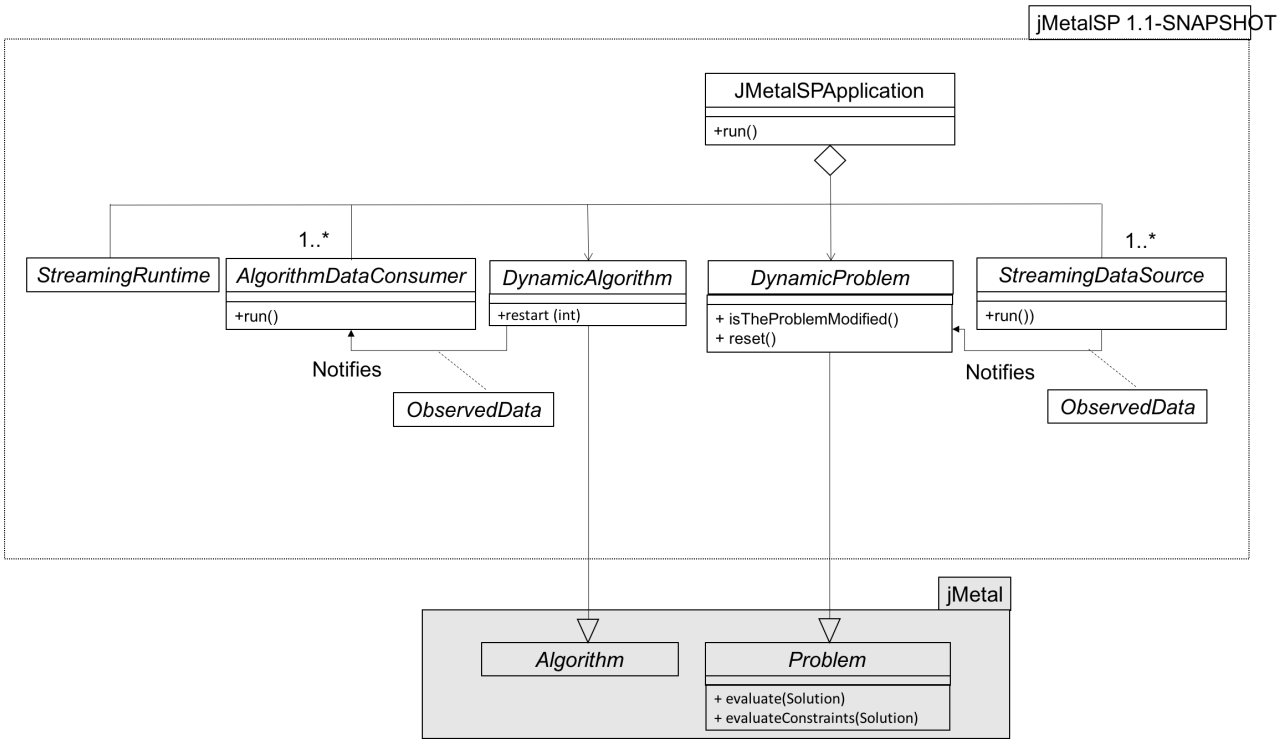

Figura 4.10: Arquitectura jMetalSP

- Un objeto de StreamingRuntime para configurar el motor de transmisión.

Antes de proseguir con el desarrollo de la integración de jMetalSP, se hará un breve paréntesis para explicar el patrón observador, al cual, combinado con hilos, esta herramienta recurre para obtener la funcionalidad que proporciona. El patrón observador (observer), es un patrón de diseño de software de comportamiento, por lo que describe una relación estructural entre objetos y a su vez un esquema de comunicación (es posible que exista una dependencia una a muchos entre los objetos). Y como su nombre indica, el objeto observador creado estará pendiente de cualquier objeto con el fin de notificar a los objetos dependientes si se realiza algún tipo de cambio. En la Figura 4.10 se puede ver la estructura básica de este patrón y las relaciones que genera con respecto a su funcionamiento.

Una vez hecho este paréntesis proseguimos con el desarrollo de las clases de jMetalSP.

Lo primero que se crea se trata de la clase problema, la que se ha definido como DynamicGVNSProblem (visible en el apéndice B, Figura B.2). Ella tendrá el objeto instancia previamente desarrollado, en el cual, como se ha definido anteriormente tiene todos los atributos del problema. Además, esta clase es la encargada de recibir las modificaciones, por lo que una vez el observador definido en ella se active, se actualizará la clase Instance añadiendo un nuevo cliente y se añadirán las relaciones de costes entre los nodos de los clientes ya existentes y el nuevo cliente en la matriz de costes.

Como se acaba de definir es necesario una clase que pase los nuevos nodos

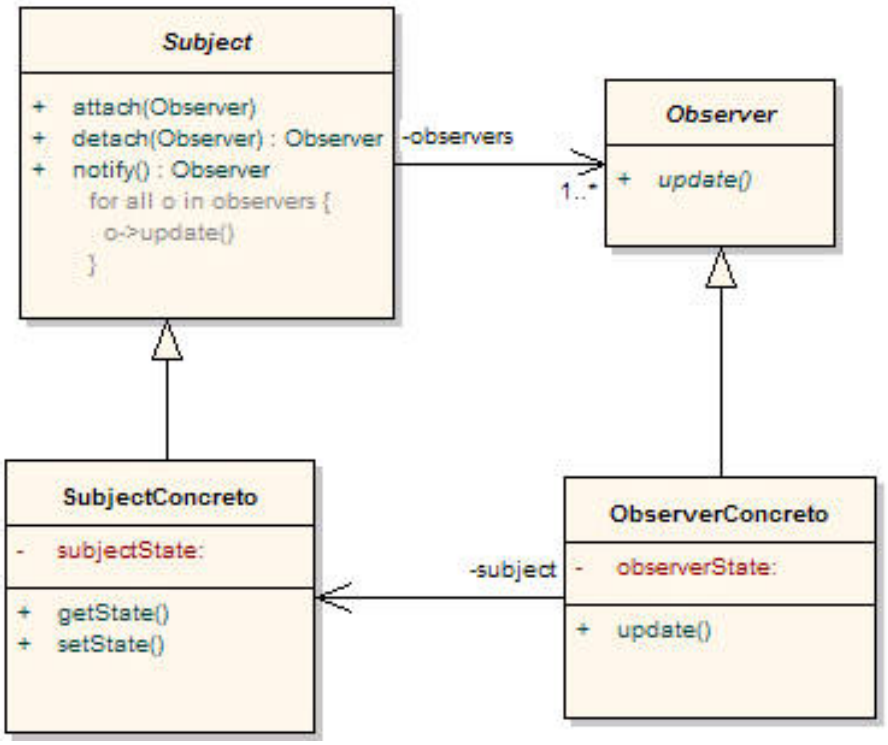

Figura 4.11: Patrón observador

a nuestro problema. Por eso existe una clase que activa esta modificación, se trata de la clase StreamingDataSource. Esta clase está formada en primer lugar por el tiempo que tiene que esperar para actuar y un atributo compuesto por el listado de nodos de las localizaciones que se añadirán dinámicamente. La forma de añadirlos es a través del observador, cada cierto tiempo definido se le pasará el nodo (o nodos) que hay que añadir y se actualizará el problema dinámico. De una forma más específica, se han definido dos clases distintas que parten de la genérica de StreamingDataSource, una que se ejecuta con la forma periódica, cada diez segundos comprueba que nodos tienen un tiempo de apertura inferior al total transcurrido, estos serían los nuevos clientes a añadir, y los añade. Y la forma completamente dinámica, que va añadiendo los clientes en cada instante de tiempo de apertura que tienen definido. A la hora de ejecutar nuestros problemas se experimentará con cada una de las dos clases como podremos ver posteriormente en la experiencia computacional. Es posible visualizar estas relaciones de clase en el apéndice B, Figura B.6.

Otro aspecto a tener en cuenta en el desarrollo de este programa propio que usa jMetalSP, es que hay que añadir a la clase propia que representa la solución la implementación de la clase Solution perteneciente a la librería jMetal. Esta implementación obliga a definir ciertos métodos heredados, los cuales eran abstractos en el padre. Esta clase es tan importante porque dichos métodos serán los llamados posteriormente para realizar la representación de las gráficas. Los métodos específicos y necesarios son obtener el número de objetivos y el número de variables. Que según el número que se haya definido en cada uno se irán ob-

```
public class DynamicGVNS<S extends
        Solution<?>> extends GVNS
    implements DynamicAlgorithm
      <S, AlgorithmObservedData<S>> {
    ...
```

Figura 4.12: Clase DynamicGVNS.

teniendo de la lista de atributos correspondientemente para su representación. En el caso desarrollado, no se llegan a usar las variables, sólo dos atributos que usamos como objetivos, el coste de la solución generada y el tiempo en el que se actualizó el problema con nuevos nodos. Como el problema desde un principio se ha enfocado con un sólo objetivo, minimizar el coste de la solución, se ha usado el tiempo, que obviamente no es un objetivo, pero sirve para realizar la representación gráfica y tener el contraste entre el coste de la solución y tiempo en el que se añadió un nuevo nodo modificando el problema.

La siguiente clase a desarrollar, la clase que gestiona el algoritmo. Está clase estará formada por la herencia de la clase que gestiona el GVNS definido previamente, en donde como describimos anteriormente tiene definida la ejecución de la Búsqueda de Vecindad Variable General. Que además, y resaltando las clases necesarias para usar jMetalSP, tienen que ser una implementación de la interface DynamicAlgorithm, Figura 4.12. Esta implementación obliga a definir ciertos métodos necesarios en la clase hija. En concreto las funciones necesarias son la de ejecución del algoritmo, método run, en donde se define la generación de la primera solución, ejecutada con todos los nodos clientes estáticos iniciales y posteriormente un bucle que va comprobando si la clase del problema se ha modificado. En el caso de que se haya modificado se definirá la ejecución de los algoritmos que modifican la solución existente y se establece el problema a un estado no modificado. Siempre que se modifica la solución final se notifican los observadores asociados a esta clase los cuales describiremos a continuación.

Existen dos controladores que realizarán cada uno una determinada tarea al finalizar la ejecución y ser notificados por la clase que implementa de DynamicAlgorithm. El primero de ellos se trata del LocalDirectoryOutputConsumer, al que se le pasa el algoritmo y un nombre de directorio. Esta clase se encarga simplemente de una vez que se notifica los observadores en el algoritmo dinámico, accede a la clase que representa la solución, obtiene todos los valores de los objetivos y de las variables y las guarda en ficheros en el directorio definido. Estos ficheros estarán diferenciados entre objetivos y variables, y el nombre tiene la característica de estar formado por el número de la iteración a la que corresponde. El siguiente controlador se trata del ChartConsumer, éste es el encargado de actualizar la parte gráfica de la aplicación jMetalSP, de similar forma accede a los valores establecidos en la solución generada por el algoritmo dinámico y los plasma en la interfaz gráfica. Ambos no son necesarios modificar-

```
setLocalDirectoryOutputConsumer(
 new LocalDirectoryOutputConsumer<
  DynamicCVRPSolution>(OUTPUT_DIRECTORY));

setChartConsumer(
 new ChartConsumer<DynamicCVRPSolution>(
  getDynamicAlgorithm()));

getDynamicAlgorithm()
 .getObservable()
 .register(getLocalDirectoryOutputConsumer());

getDynamicAlgorithm()
 .getObservable()
 .register(getChartConsumer());
```

Figura 4.13: Controladores del DynamicGVNSAlgorithm.

los de ninguna forma para una correcta ejecución de jMetalSP. Con declararlos, asignarlos a su observador correspondiente (Figura 4.13) y pasarlos a la clase JMetalSPApplication en su ejecución es suficiente.

Finalmente, se instancia la clase de JMetalSPApplication y se le asignan a la ejecución todas estas clases ya nombradas (como podemos ver en la Figura 4.14), que una vez ejecutada trabajará a través de diversas tareas repartidas en hilos, en donde funcionarán simultáneamente los observadores definidos en párrafos anteriores, y poder así ver cómo se va actualizando la gráfica con nuevos valores de soluciones según se van añadiéndo nuevos nodos clientes al problema. La Figura 4.15 es un ejemplo de esta ejecución.

4.4. Unit test

Como cualquier desarrollo de código un aspecto básico es el desarrollo de test, como se conoce el desarrollo de estos aspectos facilita introducir cambios en el código evitando nuevos errores, haciendo así que la integración sea mucho más simple. Además de que al desarrollar test se puede generar un código más desacoplado en donde las funciones están enfocadas a devolver un resultado que se testeará. Y que a parte de estos aspectos técnicos, el desarrollo de test funciona como documentación para otros desarrolladores.

El desarrollo del proyecto no contempla un cubrimiento completo del código pero se ha establecido un mínimo de test para comprobar determinadas funcionalidades. En concreto, los test desarrollados se han enfocado a la correcta comprobación de las soluciones posteriormente al uso de cualquiera de

```
getJMetalSPApplication().setStreamingRuntim(
 new DefaultRuntime())
 .setProblem(getDynamicProblem())
 .setAlgorithm(getDynamicAlgorithm())
 .addStreamingDataSource(getStreamingSource(),
  getDynamicProblem())
 .addAlgorithmDataConsumer(
  getLocalDirectoryOutputConsumer())
 .addAlgorithmDataConsumer(getChartConsumer())
 .run();
```

Figura 4.14: Ejecución de jMetalSP.

las heurísticas definidas. Ya que con esto conseguimos evitar comprobaciones manualmente de que cada uno de los nodos están en alguna ruta, que los costes totales son correctos, que la suma de la demanda no supera la de la capacidad del vehículo y está correctamente calculada, ..

4.5. Experiencia

A continuación, en esta sección y después de ejecutar nuestros algoritmos un cierto número de veces podemos ver los resultados obtenidos en cada paso realizado.

El número de iteraciones varía en cuanto a los algoritmos que se estén ejecutando. Esto se debe a la variación en la complejidad computacional que requieren. En el caso de las heurísticas tanto que generan una solución inicial como las que se encargan de modificarlas mediante el GVNS, se han repetido treinta veces. Mientras que trabajar con el dinamismo, y por lo tanto tener que ejecutar las modificaciones de las soluciones según definidas en el GVNS, ha incrementado su complejidad por lo que su tiempo de cómputo es mucho mayor, por lo que se han definido diez iteraciones en cada experimentación.

El entorno para realizar los cálculos que se han desarrollado ha sido el siguiente:

- Hardware: MacBook Pro (Procesador 2,5 GHz Intel Core i7 y memoria RAM de 16 GB, 1600 MHz).
- Sistema Operativo macOS High Sierra Versión 10.13.5.
- Java 8, JDK 1.8.0_b171-b11.

Como se definió en un comienzo, las instancias utilizadas se dividen en dos partes. Por un lado, problemas completamente estáticos, en donde se ejecutarán

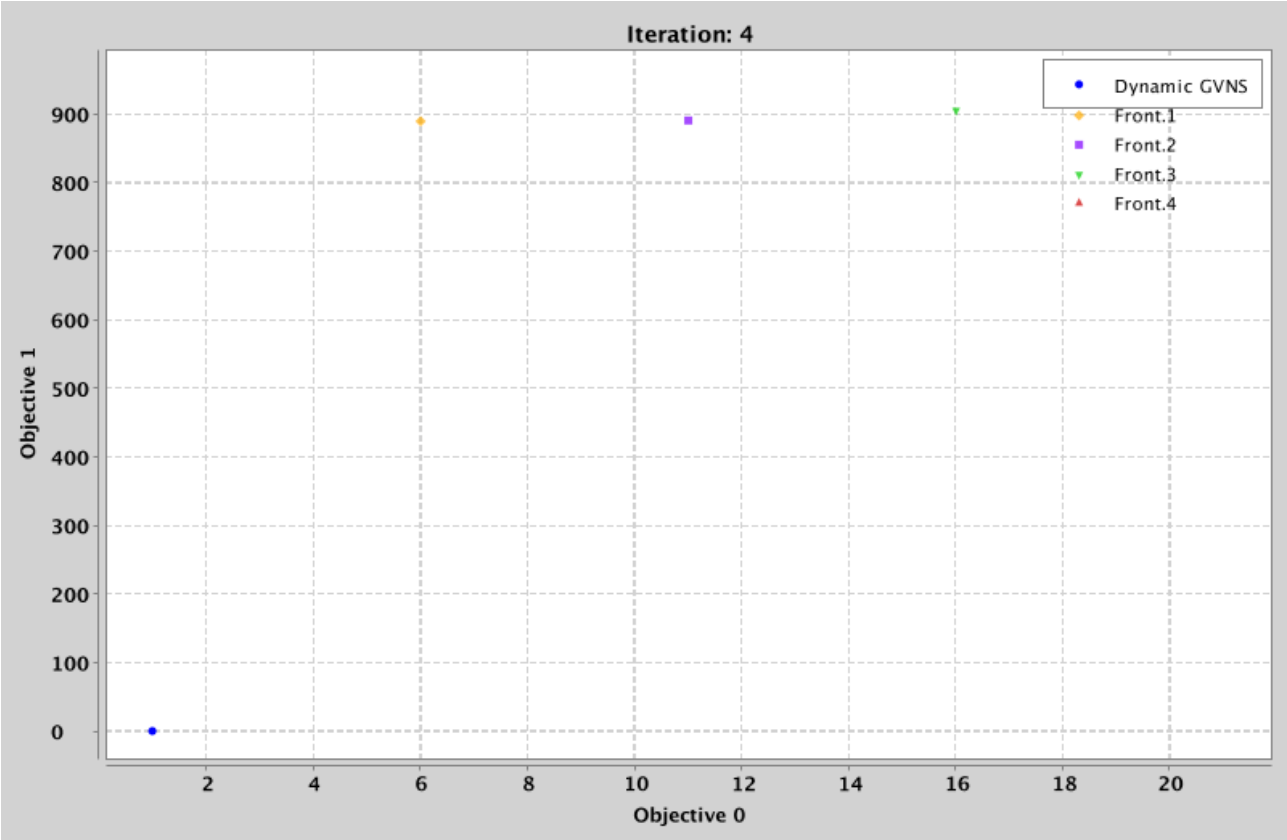

Figura 4.15: Ejemplo de jMetalSP

las heurísticas y se observará la mejor combinación de estas. Mientras que por otro lado, se usarán instancias en dónde no todos los clientes existen desde un comienzo, estudiando así dónde se encuentra el mejor rango de dinamismo en este problema de optimización.

Las instancias usadas en estas primeras tablas son instancias completamente estáticas, A-n32-k5, A-n54-k7 y A-n80-k10.

En esta primera tabla, el Cuadro 4.1, se estudia el comportamiento de las heurísticas encargadas de generar una solución inicial. La ejecución de estas heurísticas se ha hecho durante treinta repeticiones de cada heurística (Nearest Neighbor, Closest First y Furthest First) con cada una de las instancia completamente estáticas, definiendo a partir de esto la peor, la media y la mejor solución de cada una.

Como podemos observar, evidentemente en el caso de la heurística Nearest Neighbor, al no poseer nada de aleatoriedad e ir añadiendo el vecino más cercano, siempre finaliza con el mismo valor dando lógicamente un mismo coste tanto en peor, en media como en el mejor coste de la solución. Entrando un poco más en los algoritmos Closest First y Furthest First vemos como de media existe aproximadamente una diferencia de coste cien entre las soluciones, e igual ocurre en las mejores soluciones obtenida en cada instancia. Lo que sí podemos destacar de estas dos heurísticas son las peores soluciones, ya que existe una mayor diferenciación entre ellas, superior en todos los casos a doscientos, a fa-

	Instance	Peor	Media	Mejor
Nearest Neighbor (NN)	A-n32-k5	1.139	1.139	1.139
	A-n54-k7	1.440	1.440	1.440
	A-n80-k10	2.413	2.413	2.413
Closest Neighbor (CF)	A-n32-k5	2.098	1.986	1.851
	A-n54-k7	3.436	3.214	3.051
	A-n80-k10	5.077	4.965	4.792
Furthest Neighbor (FF)	A-n32-k5	2.253	2.081	1.821
	A-n54-k7	3.620	3.297	3.132
	A-n80-k10	5.405	5.117	4.937

Cuadro 4.1: Resultados de costes solución inicial mediante heurísticas

vor de optar por la heurística Closest First. Ya que esto significa que en el peor de los casos va a tener un resultado que seguro mejora las peores soluciones de la heurística Furthest First. Por lo tanto, de esta primera tabla llegamos a la conclusión de que, a pesar de dar siempre el mismo valor, la heurística que mejor genera una solución inicial se trata de Nearest Neighbor.

En las siguientes tablas, que iremos describiendo una a una, vemos el estudio realizado con respecto al orden de ejecutar las heurísticas que modifican la solución. Se ha realizado el estudio con los mismos órdenes de heurísticas que modifican una solución en cada una de las heurísticas encargadas de generar una solución inicial. Como estudiar todas permutaciones de órdenes posibles consumiría demasiados recursos se ha realizado el estudio con sólo ciertas permutaciones. El estudio llevado se ha repetido el mismo número de veces que el anterior, treinta. Y como vemos en los Cuadros 4.2, 4.3 y 4.4, tenemos para cada heurística que genera una solución inicial todos los órdenes de heurísticas que modifican la solución. Y a su vez, para cada una de estas su peor y mejor solución, acompañada de la media de todas las repeticiones realizadas.

En la primera tabla, Cuadro 4.2, se muestran los resultados de cada una de las ordenaciones usando como solución inicial la generada por la heurística Nearest Neighbor. Como podemos ver en estos resultados, y después de observar los cambios en el orden de los modificadores, vemos como el resultado final siempre es el mismo óptimo local. Generando por lo tanto un mismo resultado tanto en el peor como en el mejor de los casos y por supuesto, en la media. El motivo de que siempre se caiga en el mismo resultado es por un lado debido a que parte de la misma solución, como vimos en el Cuadro 4.1. Y por otro lado, por la forma de trabajar de las heurísticas que modifican está solución. En el GVNS, cuando los modificadores interactúan siempre ejecutan una modificación, la cual no conlleva nada de aleatoriedad. En el caso de que no exista una mejora con esa heurística pasan a la siguiente, que intenta modificarlo. Es por ello que siempre se ejecute la misma secuencia de modificadores dando el mismo resultado, ya

Instancia	Orden modificadores	Peor	Media	Mejor
A-n32-k5	2-opt / relocate / intra / inter	810,3259	810,3259	810,3259
	relocate / intra / inter / 2-opt	810,3259	810,3259	810,3259
	inter / 2-opt / relocate / intra	810,3259	810,3259	810,3259
	intra / inter / relocate / 2-opt	810,3259	810,3259	810,3259
	intra / relocate / 2-opt / inter	810,3259	810,3259	810,3259
A-n54-k7	2-opt / relocate / intra / inter	1333,2028	1333,2028	1333,2028
	relocate / intra / inter / 2-opt	1333,2028	1333,2028	1333,2028
	inter / 2-opt / relocate / intra	1333,2028	1333,2028	1333,2028
	intra / inter / relocate / 2-opt	1333,2028	1333,2028	1333,2028
	intra / relocate / 2-opt / inter	1333,2028	1333,2028	1333,2028
A-n80-k10	2-opt / relocate / intra / inter	2176,8291	2176,8291	2176,8291
	relocate / intra / inter / 2-opt	2176,8291	2176,8291	2176,8291
	inter / 2-opt / relocate / intra	2176,8291	2176,8291	2176,8291
	intra / inter / relocate / 2-opt	2176,8291	2176,8291	2176,8291
	intra / relocate / 2-opt / inter	2176,8291	2176,8291	2176,8291

Cuadro 4.2: Nearest Neighbor

que no encuentra un ciclo de modificaciones distintas si no una específica que lleva al mismo valor.

En este segundo caso, Cuadro 4.3, usando la heurística Closest First para generar la primera solución del problema. Tenemos pequeñas diferencias en cada una de las ejecuciones. Pero mirándolo de un modo más genérico y centrándonos en los mejores valores de cada instancia, tanto en el peor, el mejor y la media, vemos pequeños detalles.

En la primera instancia destacan el segundo orden de heurísticas que modifican la solución. Teniendo el mejor costo dentro de los peores y de los mejores existentes, y el segundo mejor valor con respecto a la media.

En la segunda instancia podemos observar de forma similar que el segundo orden de heurísticas tienen una mejor solución con respecto a la media y a las mejores soluciones. Además de que la última secuencia también destaca por tener un primer, segundo y tercer puesto con respecto a mejores soluciones en peor, mejor y media correspondientemente.

En la tercera y última instancia, cabe destacar como mejor resultados la última ordenación y la segunda. Entrando en el listado de las mejores además la primera, que en esta instancia tiene la mejor de las mejores soluciones y de las mejores medias.

Por lo tanto generando la solución inicial con la heurística Closest First, destacan los siguientes órdenes:

- Relocate, SwapIntra, SwapInter y 2-Opt.

Instancia	Orden modificadores	Peor	Media	Mejor
A-n32-k5	2-opt / relocate / intra / inter	1348,3457	995,9537	845,6038
	relocate / intra / inter / 2-opt	1127,2451	985,6456	850,5195
	inter / 2-opt / relocate / intra	1339,9730	974,2741	861,7370
	intra / inter / relocate / 2-opt	1183,3658	987,7183	861,7670
	intra / relocate / 2-opt / inter	1203,9543	1006,2400	890,1847
A-n54-k7	2-opt / relocate / intra / inter	2001,7863	1784,7069	1496,8386
	relocate / intra / inter / 2-opt	2097,0839	1736,0922	1441,5771
	inter / 2-opt / relocate / intra	2372,6959	1851,4851	1549,4196
	intra / inter / relocate / 2-opt	2090,6174	1825,7623	1537,3101
	intra / relocate / 2-opt / inter	1981,3724	1756,5596	1522,0614
A-n80-k10	2-opt / relocate / intra / inter	2471,9721	2236,2777	2078,2019
	relocate / intra / inter / 2-opt	2467,8087	2273,1654	2089,4436
	inter / 2-opt / relocate / intra	2461,4220	2286,6083	2096,8183
	intra / inter / relocate / 2-opt	2457,0255	2289,2521	2169,6043
	intra / relocate / 2-opt / inter	2599,1686	2331,6621	2174,4203

Cuadro 4.3: Closest First

- SwapIntra, Relocate, 2-Opt y SwapInter.
- 2-Opt, Relocate, SwapIntra, y SwapInter.

Este tercer caso, Cuadro 4.4, usa como generador de la solución inicial la heurística Furthest First. De forma similar al caso anterior tenemos valores que se asemejan bastante entre ellos, pero se pueden observar ciertos patrones en los datos.

En la primera instancia vemos como destaca el primer orden de heurísticas que modifican la solución con respecto a la mejor media y al mejor de los mejores resultados. Además del segundo orden, ya que de los peores casos entre todos los órdenes obtiene una mejor solución.

En la segunda instancia, el segundo orden tiene el coste más bajo en cuanto a la categoría de peor mientras que lo siguen el primer orden de heurísticas. Curiosamente, en el caso de los mejores resultados ocurre al revés, el mejor resultado de los mejores obtenidos pertenece al primer orden mientras que le sigue el segundo orden, en el caso de la media el segundo orden tiene los mejores resultados seguido del primero.

Y por último, la tercera instancia, que destaca por tener las mejores puntuaciones en cada ámbito en el segundo orden, destacando también este orden por tener la segunda posición con respecto a las medias.

Por lo tanto, de está tabla, que usa Furthest First para generar la solución inicial destacamos dos ordenes:

- Relocate, SwapIntra, SwapInter y 2-Opt.

Instancia	Orden modificadores	Peor	Media	Mejor
A-n32-k5	2-opt / relocate / intra / inter	1074,9699	948,1234	820,4066
	relocate / intra / inter / 2-opt	1054,4361	962,5146	841,2563
	inter / 2-opt / relocate / intra	1184,6805	975,2362	851,8690
	intra / inter / relocate / 2-opt	1273,0762	1015,2205	868,5181
	intra / relocate / 2-opt / inter	1266,5072	953,4893	864,5173
A-n54-k7	2-opt / relocate / intra / inter	1995,1438	1781,1648	1493,9318
	relocate / intra / inter / 2-opt	1991,2951	1758,2832	1501,3539
	inter / 2-opt / relocate / intra	2134,0036	1830,7271	1588,1526
	intra / inter / relocate / 2-opt	2154,1086	1851,7970	1580,2967
	intra / relocate / 2-opt / inter	2082,3001	1784,8247	1570,9259
A-n80-k10	2-opt / relocate / intra / inter	2538,7212	2272,4357	2136,0309
	relocate / intra / inter / 2-opt	2403,7623	2256,4749	2093,7137
	inter / 2-opt / relocate / intra	2725,9716	2304,9405	2100,4381
	intra / inter / relocate / 2-opt	2763,5762	2346,6187	2174,2824
	intra / relocate / 2-opt / inter	2432,0649	2274,4489	2157,9717

Cuadro 4.4: Furthest First

- 2-Opt, Relocate, SwapIntra, y SwapInter.

Observando por lo tanto las soluciones obtenidas a partir tanto en el peor de los casos, como en el mejor de los casos y por supuesto, el rango de medias entre todas las calculadas, observamos que el mejor orden (que será el que se usará en las siguientes pruebas) es el: Relocate, SwapIntra, SwapInter y 2-Opt.

A partir de aquí ya hemos definido por un lado la mejor heurística para generar la solución inicial, Nearest Neighbor. Y por otro, el mejor orden de heurísticas que se encargará de la modificación dentro del GVNS, Relocate, SwapIntra, SwapInter y 2-Opt. Ahora es el momento de comprobar cómo influye el dinamismo en el problema con estas heurísticas. Destacar de la heurística Nearest Neighbor, usada para generar la solución inicial, que además de dar los mejores resultados no desarrolla ningún tipo de aleatoriedad en su código, por lo que los resultados que se expondrán al resto de heurísticas y a la simulación estarán basados completamente en cálculos. Haciendo que la comparación de resultados tenga un mayor peso que si hubiéramos usado cualquiera de las otras dos heurísticas para generar la solución inicial. Ya que estas al tener aleatoriedad en su base podría generar resultados mucho más distantes entre ellos. Permitiendo así que las comparaciones realizadas a continuación se basen mucho más en cambios en el dinamismo.

Para realizar esta nueva experimentación lo primero que realizamos es cambiar nuestras instancias de experimentación, ya que las usadas hasta ahora solo contemplan casos completamente estáticos. Para ello seleccionamos la instancia C101 de Solomon, con diferentes tipo de dinamismo aplicando dos variantes de

	Peor	Media	Mejor
NN	981,8948	981,8948	981,8948

Cuadro 4.5: Instancia C101, ejecución sin dinamismo.

este. Por un lado cada cierto tiempo (diez segundos) se busca los nuevos clientes para añadir a la instancia y por otro lado se van añadiendo estos clientes en el momento justo de su apertura. En estas siguientes tablas se ejecutarán por cada forma de añadir los clientes, periódicamente o completamente dinámica, todas las instancias con sus diferentes dinamismos. Se puede ver en columna de dinamismo dos valores, el primero de ellos el porcentaje de nodos clientes estáticos y el segundo, el porcentaje restante, los nodos que se añadirán posteriormente de manera dinámica. Repitiéndose cada una diez veces[8].

Previamente a las ejecuciones con dinamismo comprobamos los resultados de la instancia de forma estática, Cuadro 4.5.

Antes de proceder con el análisis de las tablas, destacar que el estudio desarrollado no sólo se ha enfocado en la ejecución repetitiva de los algoritmos comprobando sus resultados. Además de esto, se ha aplicado modificaciones de los costes entre los nodos clientes con respecto a las simulaciones, haciendo así que las posiciones de los vehículos varíen en mayor medida y que a la hora de incluir un nuevo cliente se note mucho más la posición a partir de la que se puede incluir y ejecutar todas las heurísticas que interactúan en la modificación de la solución. Ya que como es evidente no se puede modificar una solución desde el punto en el que se encuentra un vehículo ni al que ya se dirige. En la primera de las ejecuciones no se ha modificado el coste, pero en las siguientes se han modificado haciendo que esta distancia sea cinco veces menor, y por otro lado, haciendo que sea dos veces mayor. Esto ha dado los siguientes resultados, que podemos ver en los Cuadros 4.6, 4.7 y 4.8, correspondientemente.

Como podemos ver, no existe una gran diferencia entre añadir los clientes de forma periódica o en el instante exacto en el que aparecen en el problema, esto se debe al tiempo de observación que usa la forma periódica. Al ser un tiempo relativamente bajo (10 segundos) es posible que ciertos nodos se añadan de golpe, pero al añadirse juntos y modificarse lleva a una solución razonable dentro del rango de las mejores soluciones del dinamismo determinado. En el caso de que este tiempo se alargue, simplemente al tener un tiempo mayor, dependiendo claro está de los tiempos en los que entran los nodos, existiría un grupo mayor de nodos a introducir causando una pequeña desventaja de la que hablaremos a continuación.

La desventaja de aumentar el tiempo es la influencia de la simulación de la situación de los vehículos. Al amplificar/disminuir los costes entre nodos

[8]Se ha disminuido el tiempo de repeticiones debido a que el consumo computacional de ejecutar una y otra vez la secuencia de heurísticas que modifican la solución en el GVNS es muy elevado.

	Dinamismo	Peor	Media	Mejor
Periódico	10 - 90	1437,8820	1437,8820	1437,8820
	30 - 70	1539,6466	1516,3184	1505,5208
	50 - 50	1388,1505	1388,1505	1388,1505
	70 - 30	1393,3391	1393,3391	1393,3391
	90 - 10	1163,1295	1163,1295	1163,1295
Dinámico	10 - 90	1524,2238	1478,4830	1409,8717
	30 - 70	1498,4080	1484,6648	1455,0748
	50 - 50	1300,0253	1300,0253	1300,0253
	70 - 30	1346,2029	1346,2029	1346,2029
	90 - 10	1112,7211	1112,7211	1112,7211

Cuadro 4.6: Instancia C101, Nearest Neighbor, sin modificación de rutas.

	Dinamismo	Peor	Media	Peor
Periodic	10 - 90	2924,2138	2924,2138	2924,2138
	30 - 70	2725,8937	2725,8937	2725,8937
	50 - 50	2145,6944	2145,6944	2145,6944
	70 - 30	1566,9875	1566,9875	1566,9875
	90 - 10	1414,0849	1414,0849	1414,0849
Dynamic	10 - 90	2754,1443	2754,1443	2754,1443
	30 - 70	2492,6501	2492,6501	2492,6501
	50 - 50	2093,1564	2093,1564	2093,1564
	70 - 30	1733,6385	1733,6385	1733,6385
	90 - 10	1412,8607	1412,8607	1412,8607

Cuadro 4.7: Instancia C101, Nearest Neighbor, rutas 5 veces más pequeñas.

	Dinamismo	Peor	Media	Peor
Periódico	10 - 90	1088,1646	1088,1646	1088,1646
	30 - 70	1130,4275	1130,4275	1130,4275
	50 - 50	1163,6360	1163,6360	1163,6360
	70 - 30	1225,9825	1225,9825	1225,9825
	90 - 10	1146,9682	1146,9682	1146,9682
Dinámico	10 - 90	1204,0411	1204,0411	1204,0411
	30 - 70	1131,8018	1131,8018	1131,8018
	50 - 50	1057,8639	1057,8639	1057,8639
	70 - 30	1213,4108	1213,4108	1213,4108
	90 - 10	1096,1300	1096,1300	1096,1300

Cuadro 4.8: Instancia C101, Nearest Neighbor, rutas 2 veces más grandes.

las soluciones cambian bruscamente. En el caso de que el coste entre nodos sea menor el coste de la solución aumentará considerablemente, como vemos en el Cuadro 4.5, esto es debido a que los vehículos finalizan las rutas muy rápido. Esto significa que cuando se añade un nuevo nodo cliente, en el mejor de los casos la ruta estará a punto de acabar, por lo que este nuevo nodo se asignará en las últimas posiciones, en el peor de los casos se asignará en la última, y como hemos definido en secciones anteriores, las modificaciones en las soluciones se aplican a partir de los nodos no visitados. Por lo tanto, cuanto más nodos añadamos a una solución prácticamente finaliza, tendrá una posibilidad muy baja de mejorar con el GVNS generando que los costes sean mayores cuando mayor sea el dinamismo fijado en el problema. En el caso contrario, que los costes entre las rutas sean grandes, el tiempo que pasa entre ir de un nodo a otro es mayor, y como se ve en el Cuadro 4.7, independientemente del dinamismo que se tenga en el problema genera resultados mucho más cercanos a la solución sin dinamismo. Simplemente lo que pasa es el caso contrario al anterior, cuando se realiza la simulación al tener distancias largas entre nodos los vehículos se sitúan en posiciones iniciales de cada ruta, dando más juego a los nodos restantes, incluidos los nuevos, a modificarse mediante el GVNS y generar estas mejores soluciones.

Por lo tanto llegamos a una simple conclusión, cuanto más tiempo se espere a añadir nuevos nodos en una solución aumentará la posibilidad de que sus resultados sean peores. De forma general lo podemos ver en cada una de estas tablas, ya que cuanto menor son los nodos clientes añadidos posteriormente a tener una solución, las soluciones tienden a ser mejores. Aunque no siempre es así dando resultados tanto mejores como peores en casos específicos de dinamismo. Y es aquí donde entra añadir los nodos nada más están disponibles, con esto se actualiza la solución nada más conocer dicho cambio, haciendo que la búsqueda de mejoras se realice lo antes posible.

Si queremos definir que es mejor en un problema dinámico, tenemos que valorar cómo influyen estos apartados y ver qué prioridades existen, sobretodo en un negocio. La necesidad de tener las mejores soluciones a costa de un mayor tiempo de cómputo, ejecutando los algoritmos cada vez que se modifique el problema. O alentar la velocidad de las soluciones ejecutando de manera periódica (en el caso de que existan modificaciones) y así ahorrando tiempo de ejecución a costa de peores soluciones.

Por la parte de jMetalSP, podemos ver las correspondientes gráficas a la evolución de los dinamismos en donde se puede comprobar como evidentemente al añadir nuevos nodos se incrementa el coste total del problema, disminuyendo en ciertos casos concretos, debido a la aleatoriedad aplicada en los algoritmos. Que es el caso de la simulación que se aplica una modificación aleatoria en los costes entre nodos. A partir de la Figura 4.12 hasta la 4.16, se pueden ver las distintas gráficas de ejecutar la instancia C101 de Solomon con jMetalSP con distintos grados de dinamismo y las heurísticas definidas previamente. Desde un

noventa porciento de dinamismo decrementandolo este valor un veinte porciento en cada una hasta llegar a un diez porciento [9].

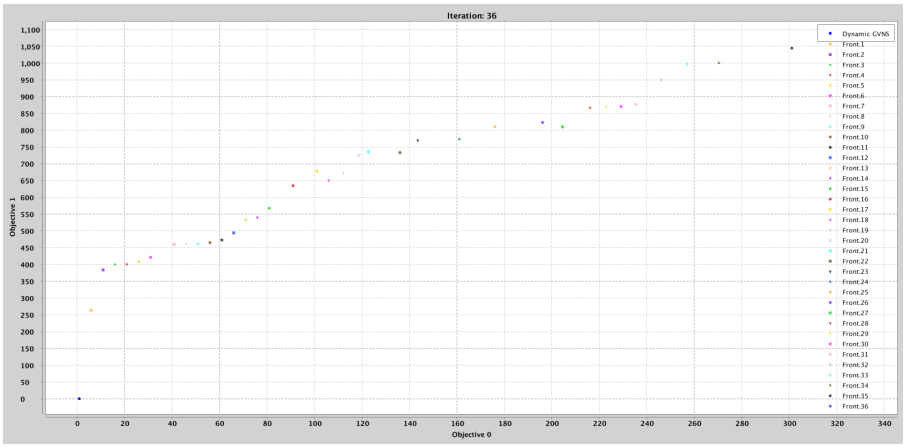

Figura 4.16: Evolución en jMetalSP de la instancia C101 con 90 % de dinamismo

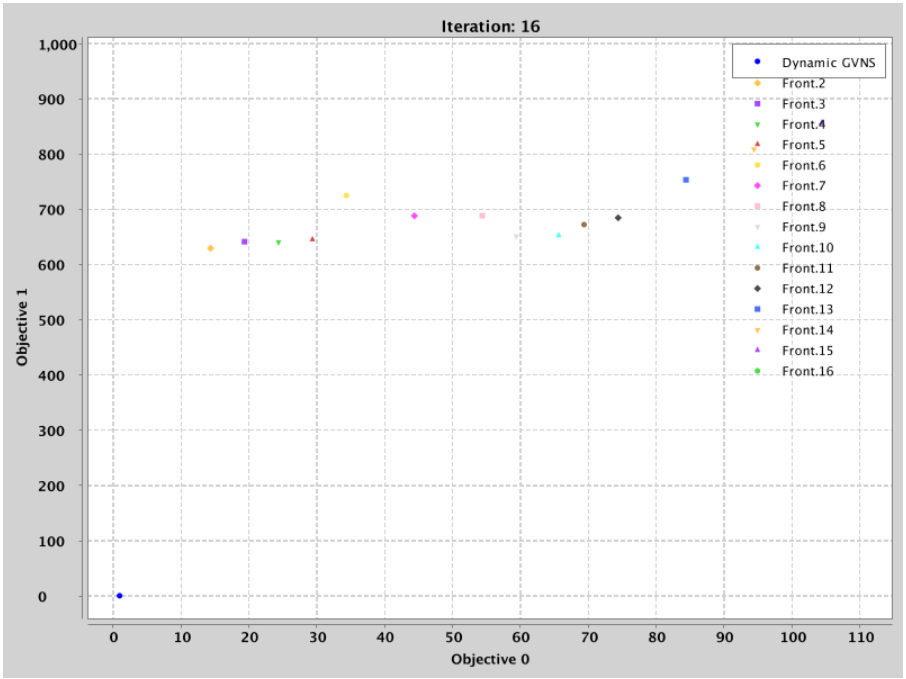

Figura 4.17: Evolución en jMetalSP de la instancia C101 con 70 % de dinamismo

[9]Los grados de dinámismo usados son: 90, 70, 50, 30 y 10 porciento.

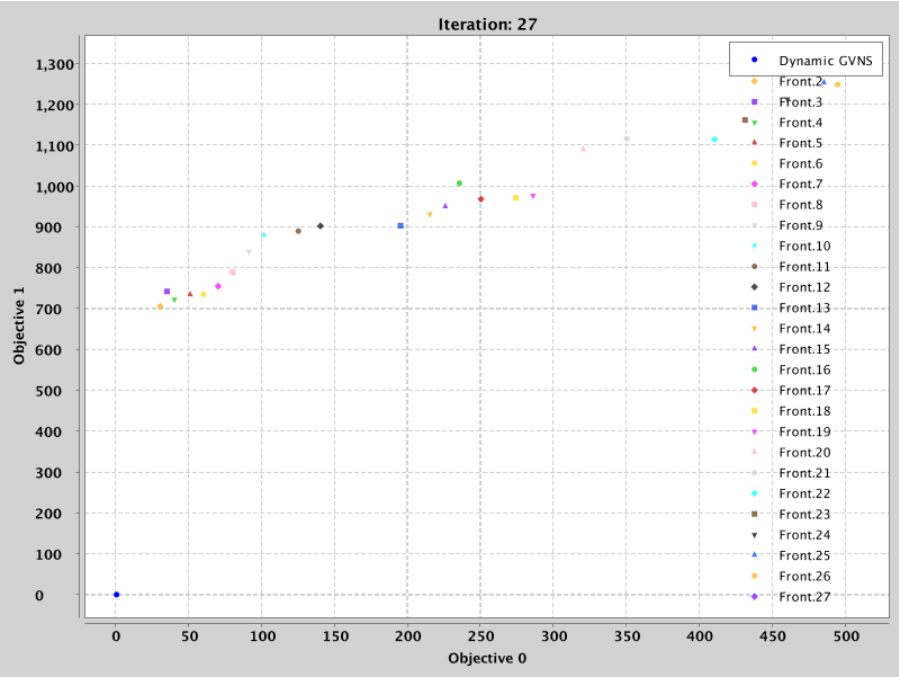

Figura 4.18: Evolución en jMetalSP de la instancia C101 con 50 % de dinamismo

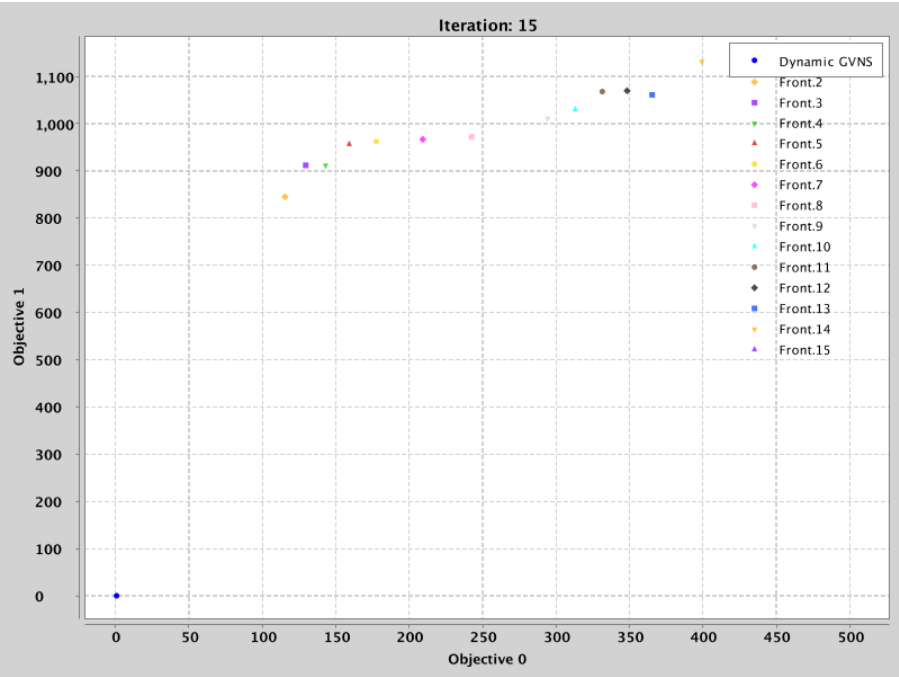

Figura 4.19: Evolución en jMetalSP de la instancia C101 con 30 % de dinamismo

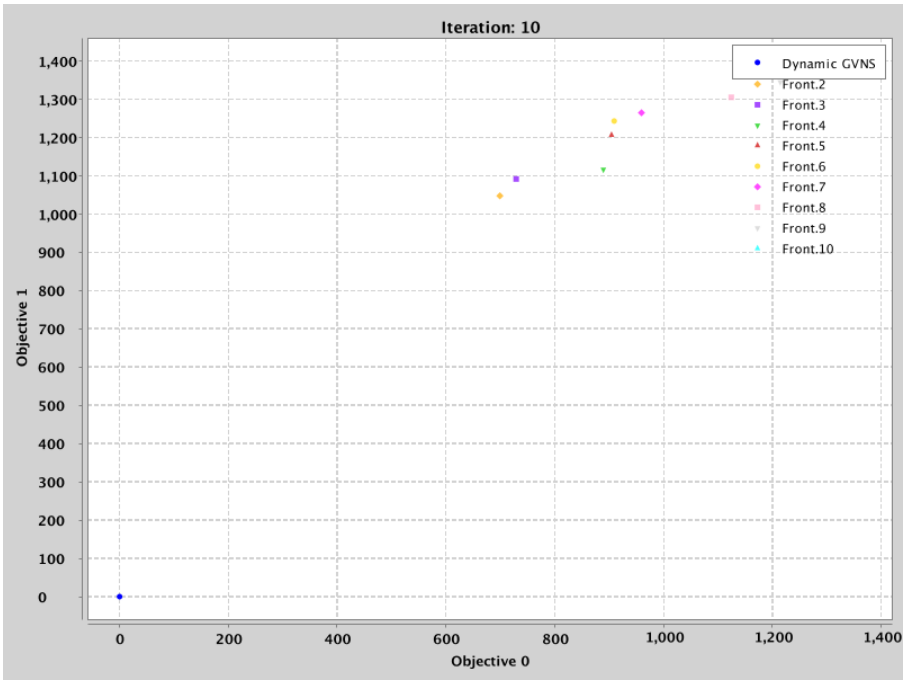

Figura 4.20: Evolución en jMetalSP de la instancia C101 con 10 % de dinamismo

Capítulo 5

Conclusiones y líneas futuras

Como conclusiones finales obtenemos ciertas ideas claras. La principal es que se han trabajado cierto número de heurísticas para resolver el problema considerado, algunas de ellas nos han llevado a mejores soluciones que otras. Todo dependerá de las heurísticas usadas, el desarrollo que se aplique y la forma de aplicarlas al problema. El segundo punto, el cual es el principal a desarrollar en este proyecto, trata de cuánto conocimiento es lo idóneo para resolver un problema de este tipo. Como se ha visto, lo ideal para poder optar a una solución óptima, es tener el mayor conocimiento posible del problema, ya que independientemente de si se añade nueva información en su momento justo o de forma periódica, no es seguro que se puedan añadir estos datos en la posición ideal según el algoritmo que se haya desarrollado, o el propio software ya desplegado en producción. Como hemos definido en nuestra experiencia, no es seguro que esté disponible, ya que esto puede depender de muchos valores; como es el caso del coste entre clientes que puede acarrear que las soluciones sean mucho peores.

Para un siguiente paso en este proyecto, existen varios caminos. El primero de ellos y más sencillo, seguir variando los parámetros usados, nombrando por ejemplo el número de vehículos en la simulación, que actualmente es el ideal para cubrir todas las rutas, o la relación de costes entre clientes, que como hemos visto influye bastante en la resolución del problema. Y así, jugando con estos valores, ver cómo afectaría lógicamente en el código para la obtención de una solución y a la calidad de ésta.

La segunda posibilidad es seguir indagando en diversas metaheurísticas, que se puedan aplicar tanto al desarrollo simple de solucionar nuestro problema (o cualquier variante), con el fin de obtener nuevos estudios y quizás nuevas formas de llegar a mejores soluciones.

El tercer enfoque que se puede aplicar es combinarlo con más herramientas. Por ejemplo, este proyecto se ha enfocado a jMetalSP, una herramienta que se

desarrolló pensando en la modificación de los costes entre nodos, pero en este caso se centró en la modificación del número de nodos clientes existentes. Es por ello que este aspecto de coste no se llega a aplicar en esta herramienta, pero sí se usa en la simulación idealizada del sistema de vehículos, en donde simulamos un coste que suponemos real en nuestro marco teórico. Para un funcionamiento más enfocado al uso practico, este apartado se podría combinar con alguna herramienta como podría ser Traffic layer de google en la que poder acceder a datos lo más exactos posibles del tráfico, haciendo la simulación de las rutas más reales y gestionando mejor el añadir nuevos clientes posteriormente a tener una solución. Incluso continuando el enfoque de una manera más práctica se podría gestionar la posibilidad de salir del margen de instancias tan teóricas y enfocar puntos en donde se contemple que el camino de un nodo a otro puede estar formado por una o varias aristas, en donde se podría aplicar mucho más juego. Como por ejemplo, que la arista que se use sea la que tenga un menor coste, y que el coste de las aristas vengan determinadas por diversos parámetros como el tráfico, los kilómetros totales, la inclinación, el estado de la carretera, etc. Haciendo que no sólo el tráfico influya pudiendo aplicar otras técnica de minería de datos o inteligencia artificial con el fin de elegir esta mejor arista.

Capítulo 6

Summary and Conclusions

As a conclusion we get certain ideas. The main one is that a number of heuristics has been worked on to solve this problem. Some of them have led us to better solutions depend on the heuristics used, the development carried on and the way in which apply them to the problem.

The second one would be how much knowledge is ideal to solve a problem. As we have seen, it is necessary to have the greatest possible knowledge of a problem to achieve an optimal solution. Regardless of new information is added in real time or periodically, it is not certain that this data can be added in the ideal position according to the algorithm that has been developed, or the software already in production. It has defined in our computing experience, storing the data in the best position could depend on many values, such as the cost between nodes and it could mean that the solutions are much worse.

For the next step there is several ways. The first one is to continue changing the parameters used. For example the number of vehicles in the simulation, which is currently the ideal to cover all routes, or the cost between nodes. Which, as we have obtain on the data it has great influence on the resolution of the problem. Therefore, changing these values, affect the code to obtain a solution and its quality.

The second possibility is to continue researching on various heuristics, which can be applied to develop solutions for our problem (or any variant) in order to obtain new studies and perhaps, new ways of reaching better solutions.

The third approach that can be applied, is to combine more tools. For example, this project has focused on jMetalSP, a tool that was developed with the aim of changing costs between nodes but in this case is used on modifying the number of existing client nodes. This is the reason because the cost is not applied in this tool, but it is used in the idealized simulation of the vehicle system, where we simulate a cost that we assume real in our theoretical framework. For a practical use, this section could be combined with a tool such as Google Traffic layer where you can access to the most accurate traffic data possible, simulating the most realistic routes and managing the addition of new clients later. Even continuing the approach, it could be managed the possibility of let the margin

of theoretical instances and focus points where it is contemplated that the path from one node to another can be formed by one or several edges. In this one could be applied much more methods to solve it, for example the edge used is the lowest cost, and the cost of the edges are determined by various parameters such as traffic, total kilometers, inclination, the state of the road, etc. This, could be used for other techniques of data mining or artificial intelligence in order to select the best edge.

Apéndice A

Tablas de cálculos

A continuación se definen diversas tablas en donde se puede ver los cálculos desarrollados con respecto a la trabajar con distintos porcentajes de dinamismo en las instancias seleccionadas.

	Peor	Media	Mejor
NN	981,8948	981,8948	981,8948

Cuadro A.1: Instancia C101, ejecución sin dinamismo con Nearest Neighbor.

	Dinamismo	Peor	Media	Peor
Periodic	10 - 90	2924,2138	2924,2138	2924,2138
	30 - 70	2725,8937	2725,8937	2725,8937
	50 - 50	2145,6944	2145,6944	2145,6944
	70 - 30	1566,9875	1566,9875	1566,9875
	90 - 10	1414,0849	1414,0849	1414,0849
Dynamic	10 - 90	2754,1443	2754,1443	2754,1443
	30 - 70	2492,6501	2492,6501	2492,6501
	50 - 50	2093,1564	2093,1564	2093,1564
	70 - 30	1733,6385	1733,6385	1733,6385
	90 - 10	1412,8607	1412,8607	1412,8607

Cuadro A.2: Instancia C101, Nearest Neighbor, rutas 5 veces más pequeñas.

	Dinamismo	Peor	Media	Mejor
Periódico	10 - 90	3131,2586	2967,8472	2699,2977
	30 - 70	2706,0685	2565,8818	2354,8190
	50 - 50	2280,8100	2158,2221	2051,7722
	70 - 30	2017,2833	1785,4977	1530,8873
	90 - 10	1538,4611	1397,4775	1289,9775
Dinámico	10 - 90	2832,3358	2650,8848	2515,8340
	30 - 70	2638,0216	2332,9292	2130,1495
	50 - 50	2524,0820	2244,2710	2065,5966
	70 - 30	1972,5792	1800,6256	1653,2975
	90 - 10	1604,1086	1420,8587	1277,7360

Cuadro A.3: Instancia C101, GRASP Closest First, rutas 5 veces más pequeñas.

	Dinamismo	Peor	Media	Peor
Periódico	10 - 90	3131,2586	2895,0877	2597,1315
	30 - 70	2742,4020	2547,1891	2248,0468
	50 - 50	2249,2657	2105,1838	1957,8089
	70 - 30	1853,6495	1731,6080	1374,9931
	90 - 10	1524,5625	1452,8606	1343,7584
Dinámico	10 - 90	2881,9706	2667,6681	2448,0523
	30 - 70	2605,9778	2339,9037	2204,3796
	50 - 50	2595,9192	2340,9349	2117,6562
	70 - 30	2096,1249	1817,5718	1622,2768
	90 - 10	1637,1719	1471,9862	1278,3496

Cuadro A.4: Instancia C101, GRASP Furthest First, rutas 5 veces más pequeñas.

	Dinamismo	Peor	Media	Mejor
Periódico	10 - 90	1437,8820	1437,8820	1437,8820
	30 - 70	1539,6466	1516,3184	1505,5208
	50 - 50	1388,1505	1388,1505	1388,1505
	70 - 30	1393,3391	1393,3391	1393,3391
	90 - 10	1163,1295	1163,1295	1163,1295
Dinámico	10 - 90	1524,2238	1478,4830	1409,8717
	30 - 70	1498,4080	1484,6648	1455,0748
	50 - 50	1300,0253	1300,0253	1300,0253
	70 - 30	1346,2029	1346,2029	1346,2029
	90 - 10	1112,7211	1112,7211	1112,7211

Cuadro A.5: Instancia C101, Nearest Neighbor, sin modificación de rutas.

	Dinamismo	Peor	Media	Mejor
Periódico	10 - 90	1694,3459	1538,3532	1412,1704
	30 - 70	1748,8704	1504,3813	1262,9188
	50 - 50	1558,5323	1447,3288	1284,9812
	70 - 30	1659,9927	1435,0954	1166,8248
	90 - 10	1345,7599	1249,8984	1093,7747
	100 - 0	1298,2016	1173,1416	1046,4848
Dinámico	10 - 90	1604,0691	1541,3963	1452,2189
	30 - 70	1703,7924	1451,7171	1237,2557
	50 - 50	1627,1901	1414,3823	1153,6471
	70 - 30	1540,8079	1358,4777	1248,1646
	90 - 10	1284,9147	1238,5346	1192,8526
	100 - 0	1355,8797	1167,1091	1008,0347

Cuadro A.6: Instancia C101, GRASP Closest First, sin modificación de rutas.

	Dinamismo	Peor	Media	Mejor
Periódico	10 - 90	1732,5218	1564,2144	1473,6355
	30 - 70	1575,9742	1427,1648	1186,1770
	50 - 50	1549,7678	1439,2056	1339,6570
	70 - 30	1504,1642	1421,9228	1351,6976
	90 - 10	1399,5309	1244,3697	1145,5670
	100 - 0	1358,9762	1203,5693	1070,9813
Dinámico	10 - 90	1714,4649	1527,2491	1331,5660
	30 - 70	1515,6801	1426,9325	1262,4698
	50 - 50	1635,0277	1428,5169	1263,1597
	70 - 30	1512,1857	1294,9069	1202,9061
	90 - 10	1375,4659	1255,9246	1148,3625
	100 - 0	1336,0030	1231,7529	1090,0689

Cuadro A.7: Instancia C101, GRASP Furthest First, sin modificación de rutas.

	Dinamismo	Peor	Media	Peor
Periódico	10 - 90	1088,1646	1088,1646	1088,1646
	30 - 70	1130,4275	1130,4275	1130,4275
	50 - 50	1163,6360	1163,6360	1163,6360
	70 - 30	1225,9825	1225,9825	1225,9825
	90 - 10	1146,9682	1146,9682	1146,9682
Dinámico	10 - 90	1204,0411	1204,0411	1204,0411
	30 - 70	1131,8018	1131,8018	1131,8018
	50 - 50	1057,8639	1057,8639	1057,8639
	70 - 30	1213,4108	1213,4108	1213,4108
	90 - 10	1096,1300	1096,1300	1096,1300

Cuadro A.8: Instancia C101, Nearest Neighbor, rutas 2 veces más grandes.

	Dinamismo	Peor	Media	Mejor
Periódico	10 - 90	1276,6132	1209,6223	1132,0475
	30 - 70	1283,3353	1165,5143	1048,8196
	50 - 50	1229,5197	1142,4942	1023,8113
	70 - 30	1227,2606	1154,2579	1099,5694
	90 - 10	1263,6761	1154,4216	1020,5877
Dinámico	10 - 90	1203,1924	1166,9568	1122,4829
	30 - 70	1214,4298	1153,7093	1079,7365
	50 - 50	1264,1524	1199,3460	1070,8189
	70 - 30	1289,9005	1144,2675	1035,2837
	90 - 10	1222,7954	1138,8993	1067,1147

Cuadro A.9: Instancia C101, GRASP Closest First, rutas 2 veces más grandes.

	Dinamismo	Peor	Media	Mejor
Periódico	10 - 90	1211,3809	1157,9382	1087,2181
	30 - 70	1313,5387	1142,9333	1074,5151
	50 - 50	1244,4971	1121,1437	1054,7143
	70 - 30	1235,5832	1129,6572	1066,8786
	90 - 10	1171,7963	1125,3116	996,9858
	100 - 0	1361,2555	1220,8205	1045,9236
Dinámico	10 - 90	1250,6568	1209,4598	1154,0005
	30 - 70	1245,2257	1140,9519	1063,4388
	50 - 50	1172,0555	1104,0345	1055,0028
	70 - 30	1327,6894	1124,3251	1041,0372
	90 - 10	1320,7769	1150,8367	1076,2155
	100 - 0	1486,5174	1279,3981	1126,9459

Cuadro A.10:

	Peor	Media	Mejor
NN	777,9315	777,9315	777,9315

Cuadro A.11: Instancia RC208, ejecución sin dinamismo con Nearest Neighbor.

	Dinamismo	Peor	Media	Mejor
Periódico	10 - 90	947,7176	947,7176	947,7176
	30 - 70	921,5743	912,9071	905,9332
	50 - 50	951,5801	947,4746	943,3691
	70 - 30	922,9675	922,9675	922,9675
	90 - 10	727,0353	727,0353	727,0353
Dinámico	10 - 90	964,0746	901,2402	885,5316
	30 - 70	746,8707	716,3989	655,4045
	50 - 50	990,2188	983,8903	983,1871
	70 - 30	880,3436	880,3436	880,3436
	90 - 10	727,0353	727,0353	727,0353

Cuadro A.12: Instancia RC208, Nearest Neighbor, sin modificación de rutas.

	Dinamismo	Peor	Media	Mejor
Periódico	10 - 90	2265,9293	2213,6435	2131,1839
	30 - 70	2013,1786	2004,4450	1939,6574
	50 - 50	1995,4952	1989,4928	1984,2970
	70 - 30	1507,9059	1507,9059	1507,9059
	90 - 10	1036,0197	1036,0197	1036,0197
Dinámico	10 - 90	2168,4604	2127,9585	2123,4583
	30 - 70	1990,2000	1976,6476	1870,0914
	50 - 50	2014,6135	1989,4398	1775,1455
	70 - 30	1425,7894	1425,7894	1425,7894
	90 - 10	967,2341	967,2341	967,2341

Cuadro A.13: Instancia RC208, Nearest Neighbor, rutas 5 veces más pequeñas.

	Dinamismo	Peor	Media	Mejor
Periódico	10 - 90	928,6874	928,6874	928,6874
	30 - 70	804,8188	804,8188	804,8188
	50 - 50	818,1315	818,1315	818,1315
	70 - 30	818,4113	818,4113	818,4113
	90 - 10	760,9902	760,9902	760,9902
Dinámico	10 - 90	767,5006	767,5006	767,5006
	30 - 70	796,1579	796,1579	796,1579
	50 - 50	859,4256	859,4256	859,4256
	70 - 30	795,8446	795,8446	795,8446
	90 - 10	760,9902	760,9902	760,9902

Cuadro A.14: Instancia RC208, Nearest Neighbor, rutas 2 veces más grandes.

	Dinamismo	Peor	Media	Mejor
Periódico	10 - 90	2288,2872	2166,6727	2090,3595
	30 - 70	2182,5520	2055,8572	1794,1836
	50 - 50	2099,3558	1926,4593	1664,2068
	70 - 30	1651,1096	1495,5683	1354,8812
	90 - 10	1280,7260	1082,1982	935,9133
	100 - 0	1035,4375	961,0889	855,3178
Dinámico	10 - 90	2177,8831	2101,5140	2033,4568
	30 - 70	2131,0494	2036,2350	1899,0803
	50 - 50	2084,4983	1771,4102	1601,1418
	70 - 30	1601,9784	1484,0398	1235,4579
	90 - 10	1334,8481	1097,9893	968,6388
	100 - 0	976,3311	938,1726	885,9287

Cuadro A.15: Instancia RC208, GRASP Closest First, rutas 5 veces más pequeñas.

Apéndice B

Diagrama de clases

En este apartado podemos ver diversas clases en UML implementadas para el desarrollo del código de este proyecto.

Figura B.1: Estructura de las instancias del problema.

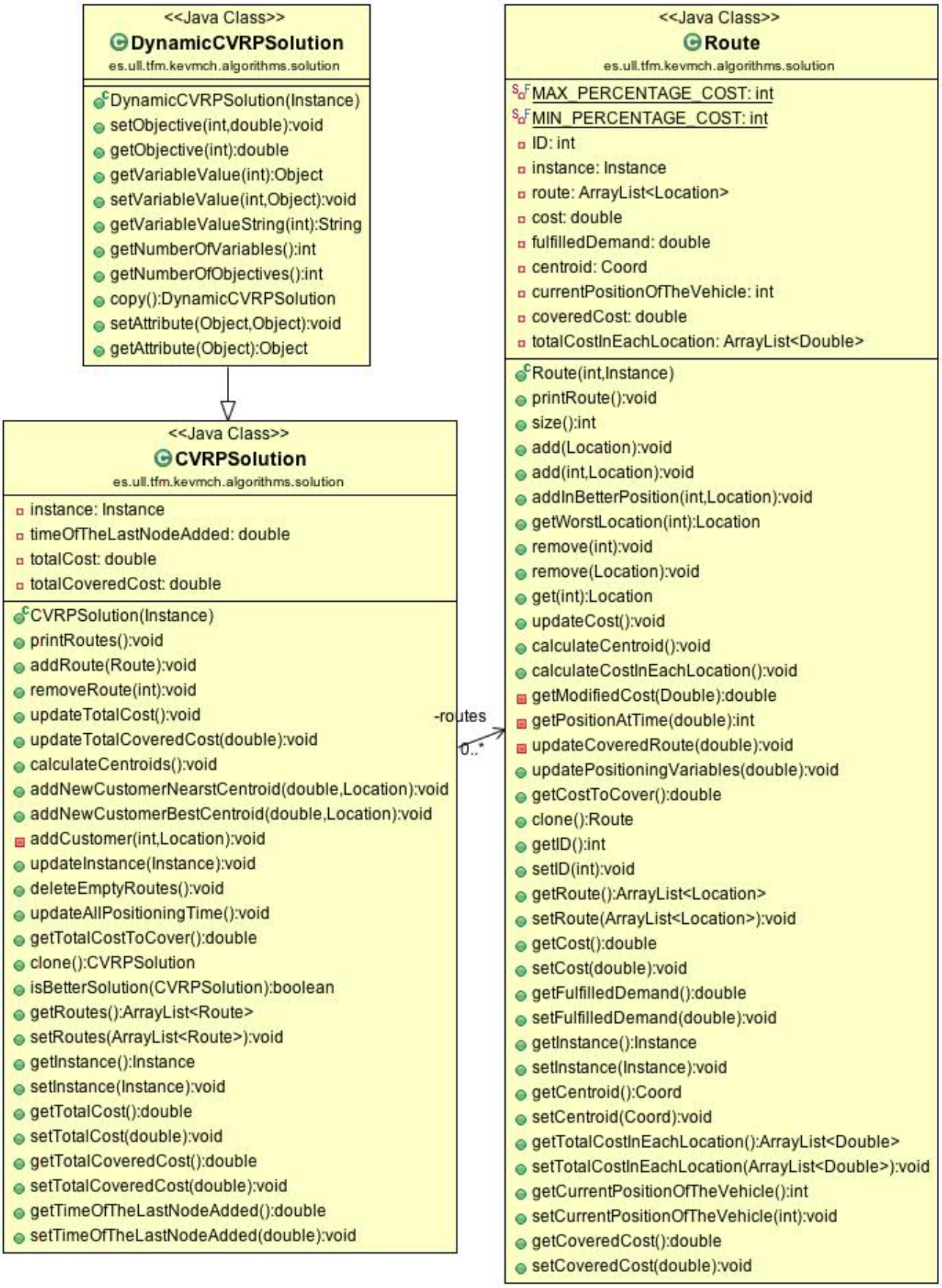

Figura B.2: Estructura de la solución.

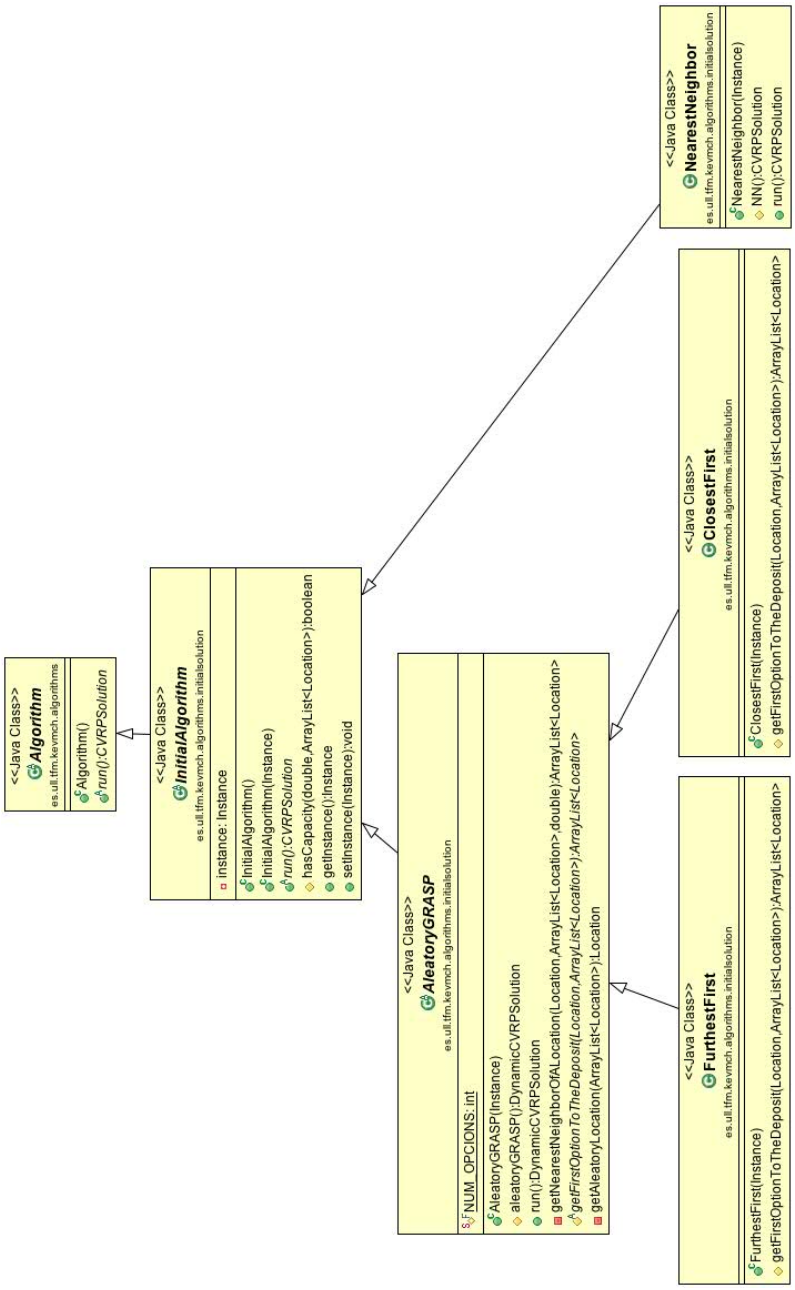

Figura B.3: Algoritmos para generar la solución inicial.

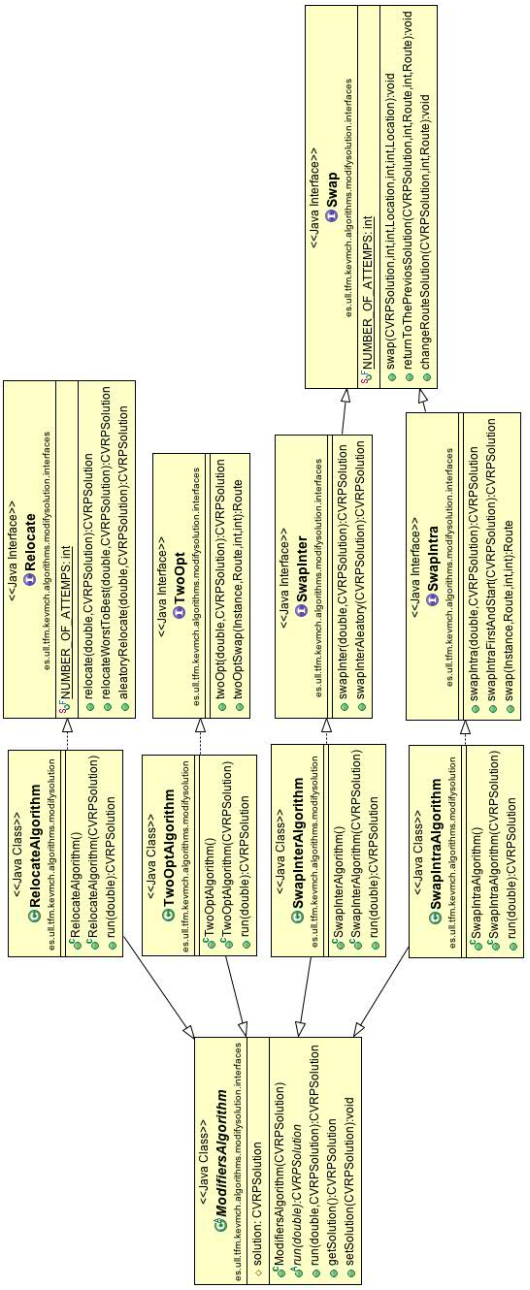

Figura B.4: Algoritmos para modificar la solución.

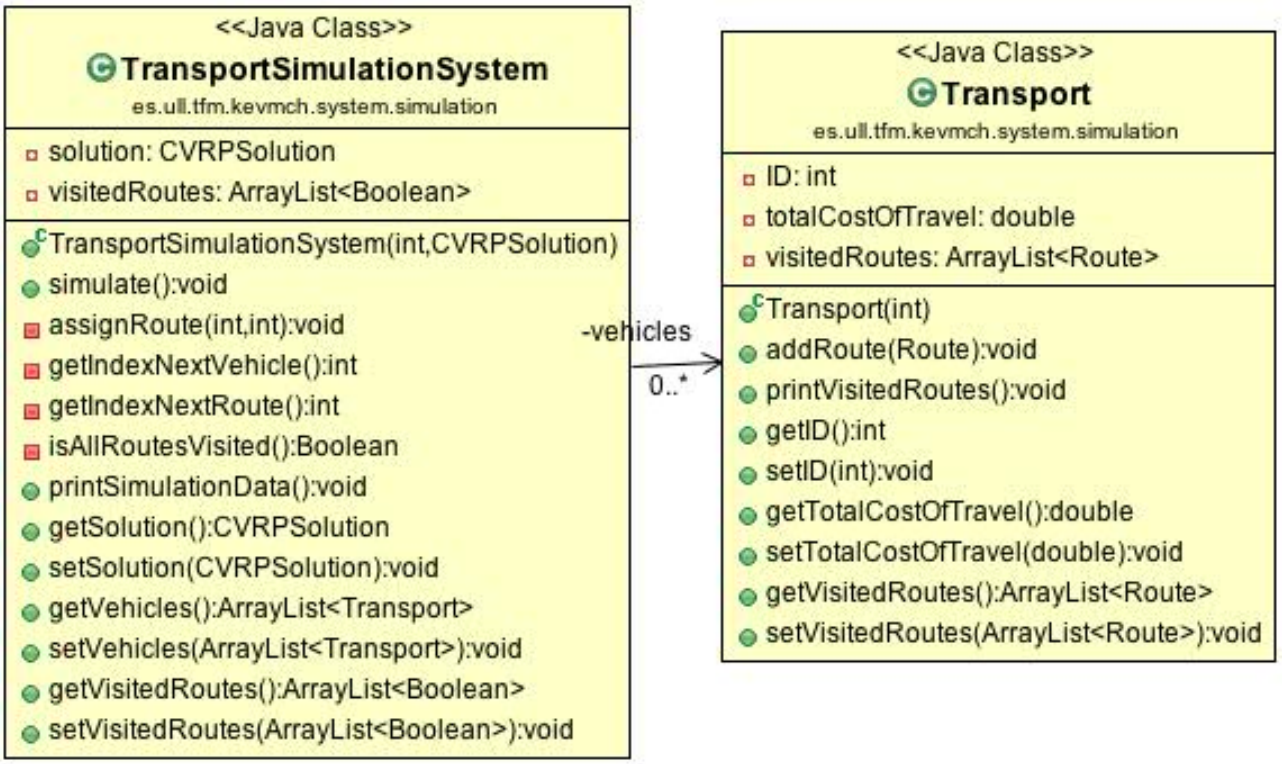

Figura B.5: Sistema de simulación.

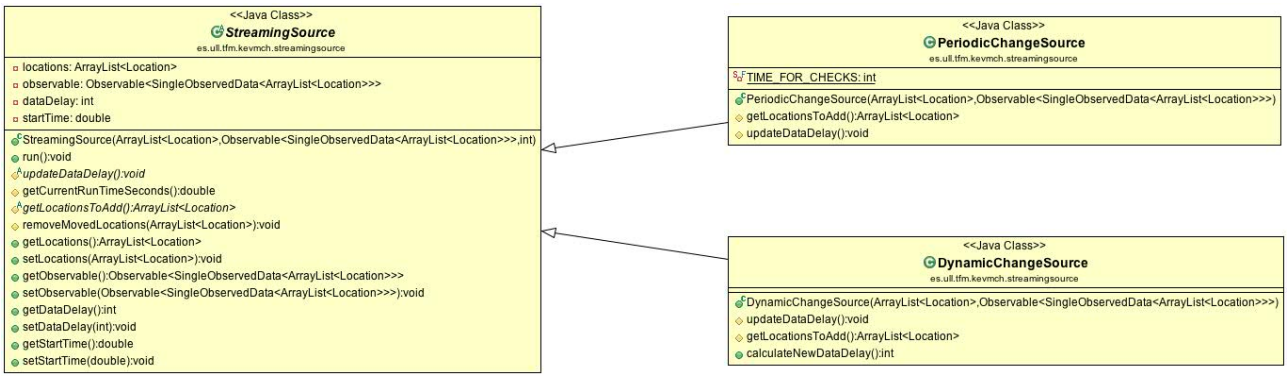

Figura B.6: Streaming source definidos para jMetalSP.

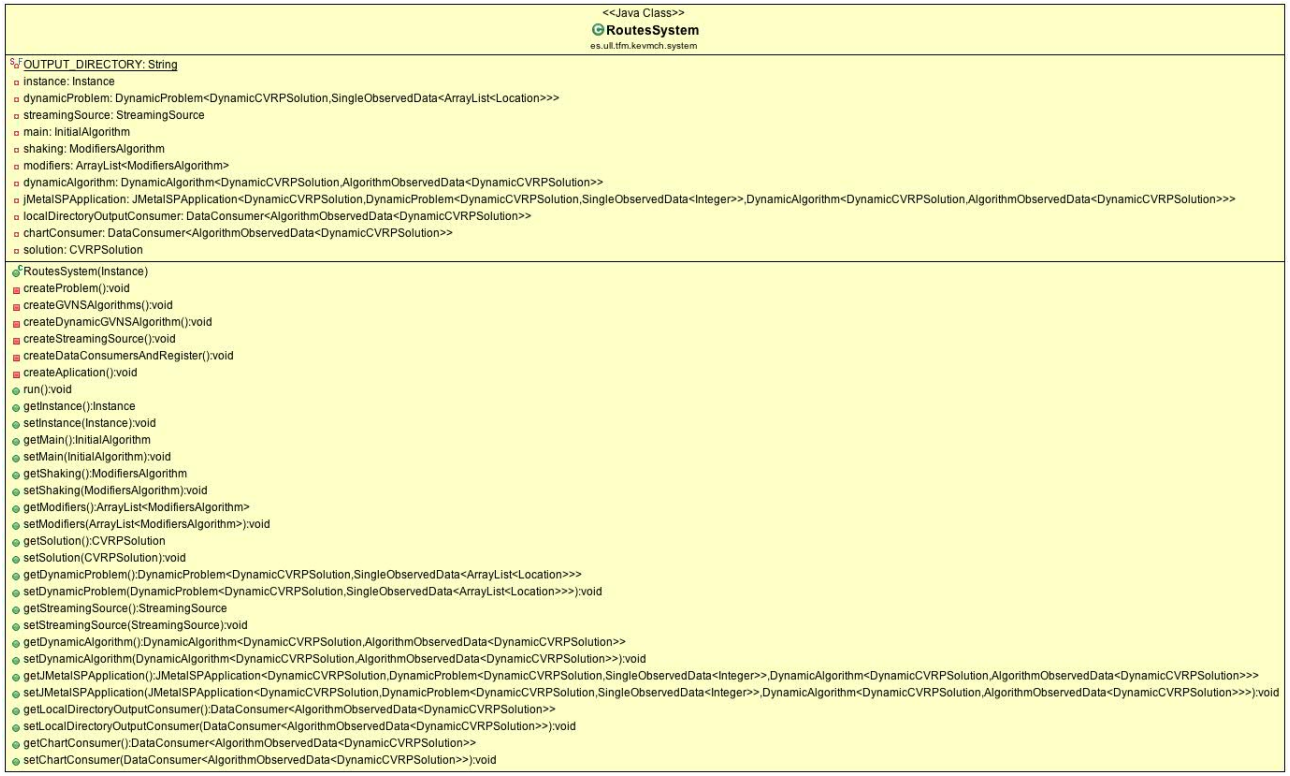

Figura B.7: Sistema con jMetalSP.

Bibliografía

[1] Cristóbal Barba-González, José García-Nieto, Antonio J. Nebro, José A. Cordero, Juan J. Durillo, Ismael Navas-Delgado, and José F. Aldana-Montes. jmetalsp: A framework for dynamic multi-objective big data optimization. *Applied Soft Computing*, 2017.

[2] Sandra Milena Castro Basto. *Revisión del estado del arte del problema de enrutamiento de vehículos con restricción de capacidad, CVRP, involucrado en la administración de la cadena de suministro.* PhD thesis, Universidad Tecnológica de Pereira. Facultad de Ingeniería Industrial. Ingeniería Industrial, 2014.

[3] H. Do, G. Rothermel, and A. Kinneer. Empirical studies of test case prioritization in a junit testing environment. In *15th International Symposium on Software Reliability Engineering*, pages 113–124, Nov 2004.

[4] Laporte Gilbert. What you should know about the vehicle routing problem. *Naval Research Logistics (NRL)*, 54(8):811–819.

[5] P. Hansen, N. Mladenović, and J.A. Moreno Pérez. Variable neighbourhood search: Methods and applications. *Annals of Operations Research*, 175(1):367–407, 2010. cited By 349.

[6] Pierre Hansen and Nenad Mladenović. Variable neighborhood search: Principles and applications. *European Journal of Operational Research*, 130(3):449 – 467, 2001.

[7] Franklin T. Hanshar and Beatrice M. Ombuki-Berman. Dynamic vehicle routing using genetic algorithms. *Applied Intelligence*, 27(1):89–99, Aug 2007.

[8] Ioachim Irina, Gélinas Sylvie, Soumis François, and Desrosiers Jacques. A dynamic programming algorithm for the shortest path problem with time windows and linear node costs. *Networks*, 31(3):193–204.

[9] Mostepha R. Khouadjia, Briseida Sarasola, Enrique Alba, Laetitia Jourdan, and El-Ghazali Talbi. A comparative study between dynamic adapted

pso and vns for the vehicle routing problem with dynamic requests. *Applied Soft Computing*, 12(4):1426 – 1439, 2012.

[10] Philip Kilby, Patrick Prosser, and Paul Shaw. Dynamic vrps: A study of scenarios. *University of Strathclyde Technical Report*, pages 1–11, 1998.

[11] Philip Kilby, Patrick Prosser, and Paul Shaw. Dynamic vrps: A study of scenarios. 04 2002.

[12] Matti Luukkainen, Arto Vihavainen, and Thomas Vikberg. A software craftsman's approach to data structures. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 439–444, New York, NY, USA, 2012. ACM.

[13] Nenad Mladenović, Milan Dražić, Vera Kovačevic-Vujčić, and Mirjana Čangalović. General variable neighborhood search for the continuous optimization. *European Journal of Operational Research*, 191(3):753 – 770, 2008.

[14] Psaraftis Harilaos N., Wen Min, and Kontovas Christos A. Dynamic vehicle routing problems: Three decades and counting. *Networks*, 67(1):3–31.

[15] Michael Olan. Unit testing: Test early, test often. *J. Comput. Sci. Coll.*, 19(2):319–328, December 2003.

[16] M. Pavone, N. Bisnik, E. Frazzoli, and V. Isler. A stochastic and dynamic vehicle routing problem with time windows and customer impatience. *Mobile Networks and Applications*, 14(3):350, Oct 2008.

[17] Esteban Perez-Wohlfeil, Francisco Chicano, and Enrique Alba. An intelligent data analysis of the structure ofnp problems for efficient solution: The vehicle routing case. In Pavel Krömer, Enrique Alba, Jeng-Shyang Pan, and Václav Snášel, editors, *Proceedings of the Fourth Euro-China Conference on Intelligent Data Analysis and Applications*, pages 368–378, Cham, 2018. Springer International Publishing.

[18] S. Sagiroglu and D. Sinanc. Big data: A review. In *2013 International Conference on Collaboration Technologies and Systems (CTS)*, pages 42–47, May 2013.

[19] Briseida Sarasola, Mostepha Khouadjia, Enrique Alba, Laetitia Jourdan, and El-Ghazali Talbi. Flexible variable neighborhood search in dynamic vehicle routing, 04 2011.

[20] W. F. Tichy. A catalogue of general-purpose software design patterns. In *Proceedings of TOOLS USA 97. International Conference on Technology of Object Oriented Systems and Languages*, pages 330–339, Jul 1997.

[21] T. Van Woensel, L. Kerbache, H. Peremans, and N. Vandaele. Vehicle routing with dynamic travel times: A queueing approach. *European Journal of Operational Research*, 186(3):990 – 1007, 2008.

[22] Zhiwei Yang, Jan-Paul van Osta, Barry van Veen, Rick van Krevelen, Richard van Klaveren, Andries Stam, Joost Kok, Thomas Bäck, and Michael Emmerich. Dynamic vehicle routing with time windows in theory and practice. *Natural Computing*, 16(1):119–134, Mar 2017.