UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

An Iterated Local Search algorithm for the Facility Localtion Problem

Sergi Uceda santos

16 de Noviembre de 2014

TESIS DE MASTER
Màster en Enginyeria Informàtica
Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

Director: Dr. Àngel A. Juan Pérez
Ponente: Dr. Pau Fonseca i Casas

Resumen

En este proyecto se ha desarrollado un algoritmo de aproximación para el problema de la localización de instalaciones, en inglés *Facility Location Problem (FLP)*. FLP es un problema importante en el campo de *Operations Research* y *Computer Science* el cual puede ser aplicado en diferentes areas como en logística o en redes de computación.

El objetivo principal de este trabajo es solucionar una variante del FLP llamada *uncapacited FLP*, para ello se ha usado un enfoque híbrido que combina un método metaheurístico con técnicas de aleatorización sesgada, integrando un *Iterated Local Search*. Para verificar la efectividad del algoritmo se han utilizado distintos *benchmarks* y comparado los resultados con otros algoritmos que resuelven el mismo problema.

Agradecimientos

A mi familia por por su apoyo incondicional.

A mi pareja, Lilia Garcia, por estar siempre allí cuando se la necesita.

A Àngel A. Juan, por ofrecerme la excelente oportunidad de tenerlo como director, y por su inestimable ayuda, sin la cual este proyecto no hubiese sido posible.

A Pau Fonseca, por resolver todas las dudas y cuestiones surgidas a lo largo del proyecto.

Índice general

Capítulo 1

Introducción

En este proyecto se ha desarrollado una solución para el *Uncapacitated* FLP (UFLP).
El objetivo general del FLP, es el de encontrar la localización óptima a un número indeterminado de facilities para minimizar tanto los costes de instalación como los de atender a los clientes, nodos o demandantes de éstas.

Optimizar los costes a la hora de realizar tareas repetitivas es una cuestión muy importante para las empresas, que intentan optimizar los costes minimizándolos y mejorando su eficiencia.
Algunos ejemplos de FLP son: La instalación de varios centros de distribución; para dar servicio o proveer de recursos a diferentes clientes, la instalación de diferentes puntos de recarga; para vehículos eléctricos en una ciudad, la construcción de redes de transporte, la instalación de servidores; para dar servicio en la nube en una red distribuida, etc. Otro ejemplo podría ser el de una empresa automovilística cuyo objetivo dar a sus clientes acceso a la estación de servicio más cercana. La empresa también podría tener como objetivo minimizar los costes. Una posible solución a este problema sería abrir muchas estaciones de servicio,no obstante, ésta no sería una solución óptima ya que los costes se verían notablemente incrementados. Por lo tanto, lo ideal es que la empresa abra un número determinado de estaciones de servicio donde la distancia media de esta con sus clientes sea mínima.

El objetivo del FLP es dar servicio a un conjunto de puntos demandantes (normalmente llamados clientes o nodos), abriendo un conjunto de puntos de acceso (llamadas facilities) teniendo en cuenta los costes.
Los costes previstos son; los costes de apertura de una facility y costes que un nodo tiene asociados a una facility. De forma típica, una solución para el FLP viene especificada por un conjunto de facilities abiertas y las asignaciones a sus correspondientes nodos que han sido asociados a cada una de estas.

La diferencia entre el *Facility Location Problem* y el *Uncapacitated Facility Location Problem* es que tiene una limitación de capacidad en cuanto a la cantidad de nodos que puede absorber cada una de las facilities, en cambio, en el UFLP no hay ningún límite impuesto. En este proyecto se propone una solución para el *Uncapacitated Facility Location Problem*.

El objetivo actual a la hora de solucionar este problema, es el de obtener unos resultados eficientes y unas soluciones óptimas. El problema es que la mayoría de variantes del FLP son NP-Completas. Por lo tanto, la dificultad reside en la realización de algoritmos eficientes, que puedan encontrar soluciones cercanas a la óptima y con un tiempo adecuado. Para conseguir este propósito, se ha implementado un algoritmo basado en *Iterated Local Search* y el uso de aleatorización sesgada.

1.1. Estructura

Esta memoria está estructurada de la siguiente manera:
Capítulo 1, se hace una introducción al proyecto y al problema a resolver.
Capítulo 2, se presenta la planificación y objetivos del proyecto.
Capítulo 3, se describe el UFLP con el objetivo de situar y comprender el problema base.
Capítulo 4, se revisa la literatura, con el fin de poder explicar los orígenes del problema, cómo éste ha avanzado en el tiempo, y finalmente se ejemplifican otros posibles usos.
Capítulo 5, se explica el *Iterated Local Search*.
Capítulo 6, se expone cómo se ha abordado el problema, detallando el algoritmo, y los métodos usados para solucionarlo.
Capítulo 7, se explican las pruebas computacionales, realizadas para verificar la efectividad del algoritmo, y exponiendo los resultados obtenidos.
Capítulo 8, se exponen las ideas de trabajo futuro.
Capítulo 9, se exponen las conclusiones del proyecto.
Apéndice A, se ilustra el código Java realizado para implementar el algoritmo ILS.
Finalmente, se muestran las referencias bibliográficas utilizadas durante el proyecto.

Capítulo 2

Objetivos y planificación

2.1. objetivos iniciales y planificación

Inicialmente se planteó hacer este proyecto en siete etapas/objetivos diferentes en un tiempo determinado. El listados de tareas esta detallado en la tabla 2.1. Por ejemplo en las tareas T2 y T3, como no se podía empezar a diseñar las metaheurísticas y algoritmos que iban a ser usados sin hacer un estudio previo de la revisión literaria, hay una dependencia entre ellas. Una vez empezada la T3, paralelamente también se podía empezar por la T4. Por Otra dependencia importante, esta entre T4 y T5. Hasta que T4 no estuviese finalizada, no se podía empezar con T5. Una vez implementado (T5), se podía empezar con las pruebas computacionales. La implementación no fue perfecta en un primer momento así que se tenían que ir haciendo retoques y mejoras conforme se hacían las pruebas y se veían los resultados obtenidos. Una vez con las pruebas computacionales realizadas. Se podía empezar a documentar y a realizar la memoria (T7).

Tarea	**Definición**	**Días**
T1	Planteamiento del proyecto y del problema	5
T2	Estudio de la revisión literaria y tecnologías a usar	20
T3	Estudio de las baterías de pruebas adecuadas	4
T4	Diseño de las metaheurísticas y algoritmos	8
T5	Implementación	40
T6	Pruebas computacionales	20
T7	Documentación y memoria	20

Cuadro 2.1: Planificación inicial.

La previsión inicial, fue de 117 días, de Febrero de 2014 a Julio de 2014. Estimando que se finalizaría el proyecto en esos meses. Y asumiendo que durante los fines de semana no se trabajaría.

2.2. Planificación final

Durante el segundo mes de proyecto, paralelamente mientras hacia el proyecto, me puse a trabajar a jornada completa, y por lo tanto, la planificación inicial fue alterada. Por eso motivo, se decidió alargar el proyecto haciendo una extensión de el. Esta extensión fue de 6 meses. Un día de proyecto consistía en 4 horas reales de dedicación a partir de la T3.

La previsión final, fue de 199 días, de Febrero de 2014 a Noviembre de 2014. Y asumiendo que durante los fines de semana no se trabajaría.

Tarea	**Definición**	**Días**
T1	Planteamiento del proyecto y del problema	5
T2	Estudio de la revisión literaria y tecnologías a usar	20
T3	Estudio de las baterías de pruebas adecuadas	8
T4	Diseño de las metaheurísticas y algoritmos	16
T5	Implementación	80
T6	Pruebas computacionales	30
T7	Documentación y memoria	40

Cuadro 2.2: Planificación final.

En la figura 2.1 se puede visualizar el diagrama de GANTT con la duración total.

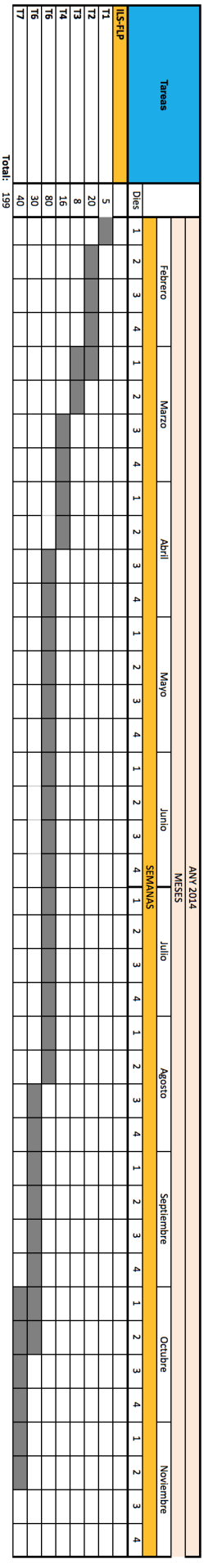

FIGURA 2.1: Diagrama de GANTT final

Capítulo 3

Descripción del FLP

El *Facility Location Problem* fue introducido de forma original por Stollsteimer (1961)[1] y Balinski (1966)[2]. En este problema el objetivo principal es localizar un número indeterminado de facilities, para minimizar la suma del coste de instalación, y de servicio variable de los nodos a dichas facilities. La mayoría de versiones del problema asumen que; en la mayoría de localizaciones donde se puede colocar una facility, están ya predeterminadas, y que las demandas a cada nodo de éstas se conocen por adelantado. Por otra parte, la variante llamada *uncapacitated* FLP, asume que no hay ningún limite en la demanda hacia las facility. Por ejemplo, en el número de nodos que cada una de las facility puede tener asociados o dar servicio. Normalmente, las decisiones ya tomadas respecto a las facility, son muy difíciles de modificar o anular, por el coste fijo asociado a su apertura.

El FLP es muy útil a la hora de modelar problemas de asignación y en una gran diversidad de campos. Por ejemplo, en problemas de logística y gestión de inventario, dónde hay que asignar centros de distribución, o de venta a una cadena de montaje. También, podemos encontrar ejemplos hasta en problemas de redes y telecomunicaciones, dónde hay que asignar servidores de servicio en la nube, en una red distribuida, o cabinas de fibra en una red de fibra óptica.

Referenciando a Verter (2011)[3], en el *Uncapacitated* FLP, la localización de las facility y la posición de los nodos, son considerados puntos discretos en el grafo (figura 3.1). Esta también es considerada una versión simple del FLP. Incluso sin la restricción en la capacidad de la facility, el FLP ha sido probado como NP-Difícil (Cornujols et al. 1990)[4]. Por lo tanto, y para el UFLP, tenemos una entrada de datos que consiste en un conjunto de facilities, un conjunto de nodos, y en la distancia comprendida entre cada uno de ellos. Así pues, el objetivo es abrir facilities con el fin de que el sumatorio del coste de la facility, y el coste de servicio comprendido entre esta y el nodo sea el mínimo.

Un ejemplo de una instancia del UFLP, es el implementado en la figura 3.1.
Esta asume que las distintas alternativas de facilities están predeterminadas. UFLP tiene como objetivo la producción y distribución de una única mercancía, artículo, producto o servicio en un periodo de tiempo determinado. Durante este periodo de tiempo normalmente también se asume que la demanda también es conocida con exactitud.

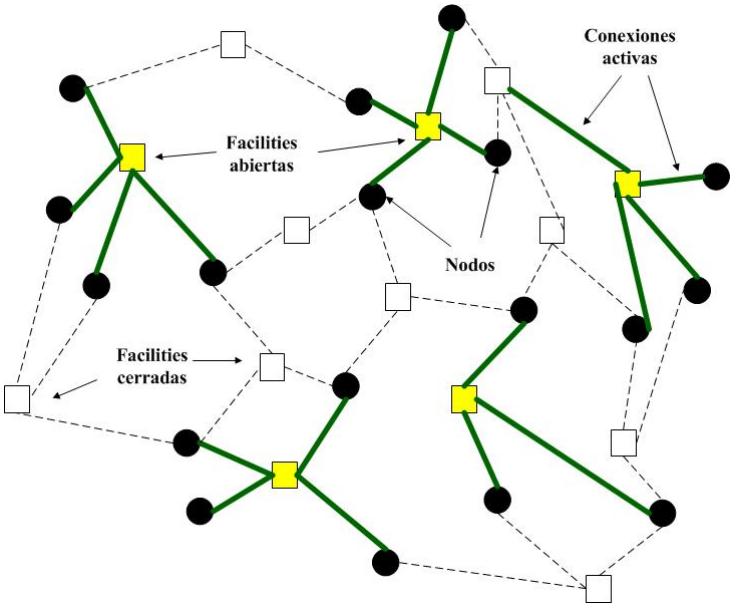

FIGURA 3.1: Ejemplo del Uncapacitated FLP

Explicado de una manera más formal, el *Uncapacitated* FLP es definido sobre un grafo G = (F, C, E), dónde F es un conjunto no vacío de facilities, dónde cada una de ellas tiene una capacidad ilimitada, C es un conjunto no vacío de nodos que tendrán que ser servidos por las facilites seleccionadas, y E es un conjunto de aristas que conectan cada nodo $j \in C$ con alguna de las facility en F (figura 3.1). Conectar un nodo $j \in C$ a una facility $i \in F$ tiene un coste de servicio, $c_{ij} \geq 0$. Además, cada una de las facility $i \in F$ tiene un coste fijo de apertura, $f_i \geq 0$.

En donde:

- X especifica el conjunto de facilities abiertas, con $\emptyset \neq X \subseteq F$
- $\sigma : C \to X$ es una función dónde se asigna a cada uno de los nodos $j \in C$ a una facility $\sigma(j) \in X$ que satisface que $c_{\sigma(j)j} = \min_{i \in X} \{c_{ij}\}$.

Por lo tanto, el *uncapacitated* FLP consiste en minimizar el coste total de proveer servicio a todos sus clientes teniendo en cuenta también los costes de abrir una facility.

$$\min \sum_{i \in X} f_i + \sum_{j \in C} c_{\sigma(j)j} \text{ sujeto a } \emptyset \neq X \subseteq F$$

Capítulo 4

Revisión literaria

En esta capítulo, se explican y repasan distintas soluciones propuestas en el tiempo, a la vez que se repasan las distintas variantes del FLP, y algunos de los escenarios que este problema puede abarcar.

El FLP fue introducido por Balinski (1966)[2] y Stollsteimer(1961)[1], al que bautizaron como *Plant Location Problem*. Uno de los primeros trabajos sobre el FLP fue un algoritmo *branch-and-bound* desarrollado por Efroymson y Ray (1966)[5]. Otra de las primeras soluciones propuestas usa el método de búsqueda directa, que fue propuesto por Spielberg (1969)[6]. El autor define dos algoritmos basados en direct search, uno consideró que las facilities estaban abiertas al principio, y el otro consideró que las facility estaban cerradas. Schrage (1975) presentó una solución que difiere a la presentada por Efroymson y Ray (1966) [5], basando la obtención de la solución en programación lineal. Por otro lado, Erlenkotte (1978)[7] presentó una solución basándose en la aportada anteriormente, pero considerando una función con objetivo dual. Más adelante Körkel (1989)[8] presentó una versión mejorada del algoritmo original de Heilenkotter.

Uno de los primeros algoritmos de aproximación propuestos para este problema, vino desarrollado por Hochbaum (1982)[9], que presentó un algoritmo glotón. Nozick et al. (1998), presentó una aproximación lineal para calcular el total del stock de seguridad, teniendo en cuenta el número de facilities. Más tarde Shmoys et al. (1997)[10] dio el primer factor de aproximación para este problema, que fue mejorado por Chudak (1998)[11]. Estos dos algoritmos están basados en *LP-Rounding*. Jain y Vazirani (1999)[12] que desarrollaron un algoritmo *primal-dual* con tiempos de ejecución más rápidos y adaptados, con el fin de poder resolver varios problemas relacionados a este. Este mismo algoritmo, fue mejorado por Jain et al. (2003)[13] con lo que obtuvo unos mejores resultados.

Los algoritmos de aproximación, han sido superados en la práctica por heurísticas más sencillas aunque sin garantías de rendimiento si son usadas con grandes instancias de datos.
Algoritmos y métodos basados en búsqueda local, han sido usados durante décadas, empezando por Kuehn y Hamburger (1963)[14]. Estos autores presentaron uno de los primeros modelos para el problema y un procedimiento heurístico para solucionarlo. Éste, estaba organizado principalmente en dos fases, primero una fase constructiva que era considerada como el programa principal, y luego otra de mejora. Siguiendo este trabajo, se han aplicado metaheurísticas mucho más sofisticadas para solucionar el FLP. Alves y Almeida (1992)[15] propusieron un algoritmo dónde se utilizaban una de las primeras metaheurísticas al problema. Kratica et al. (2001)[16] presentaron un algoritmo genético mejorando todos los trabajos previos. Ghosh (2001)[17] presentó una heurística para este problema, ésta usaba una búsqueda tabú como búsqueda local consiguiendo así soluciones muy competitivas con tiempos de ejecución muy cortos, si lo comparamos con otros algoritmos. Michael y Van Hentenryck (2003)[18] definieron un algoritmo de búsqueda tabu, que, comparado con los trabajos anteriores basados en algoritmos genéticos, éste demostraba ser mas simple, robusto, eficiente y competitivo pues usaba una aproximación lineal en donde movía una única facility en cada una de las iteraciones. Resende y Werneck (2006)[19] propusieron un algoritmo basado en una metaheurística llamada *Greedy Randomized Adaptive Search Procedure* (GRASP). El algoritmo combina una fase de construcción agresiva con un posterior procedimiento basado en búsqueda local obteniendo muy buenos resultados aún comparandolos con la mejor solución conocida y con un gran número de instancias probadas.

Cooper (1963)[20] estudió el problema de como decidir las localizaciones de los almacenes y la asignación de la demanda de los clientes hacia estos, teniendo en cuenta sus localizaciones y demandas. Este problema fue considerado como el más básico de todos los FLP. A partir de aquí, se han ido estudiando muchas variantes a este problema. Una de las primeras, trató de añadir una restricción de capacidad a cada una de las facilities, limitado el numero de nodos que estas podían abarcar. Esta variante ahora es llamada *Capacitated Facility Location Problem* (CFLP). Otra variación del FLP original, es el problema dónde se considera la entrega de distintos productos, a esta variación se le llama *Multi Commodity Facility Location Problem* (MCFLP). Esta fue estudiada inicialmente por Klincewicz y Liss (1987)[21], que estudiaron el problema sin ninguna restricción en el número de productos en cada una de las facility. En el FLP con una función de coste general (FLP-GCF), el coste de la facility está basado en función del número de nodos asignados a esta. Otra de las variantes para el problema original, es la que considera que los nodos o puntos de demanda, van llegando uno a uno, con el objetivo de mantener un conjunto de facilities que sean capaces de dar servicio a estos nodos. Este problema es

llamado *Online Facility Location Problem* (OFLP), por Meyerson (2001)[22]. Carrizosa et al. (2012)[23] presentó una una variación no lineal al problema. Esta variación, modificó la formulación básica basada en programación entera, llamada *Nonlinear Minsum Facility Location Problem* (NMFLP).

Otra modificación también muy importante del problema, es la llamada *under uncertainty* (Snyder 2006)[24], dónde se considera que cualquier parámetro del problema (principalmente el coste, la demanda o la distancia) puede variar de forma considerable. El objetivo en estos problemas, en encontrar una solución que, cambiando de forma aleatoria todos los parámetros, esta se ejecute de forma correcta, esto es denominado una solución robusta. Los parámetros aleatorios pueden ser continuos o discretos. Erlebacher y Meler(2000) desarrollaron un modelo no lineal, teniendo en cuenta la ubicación del inventario para diseñar un sistema con dos niveles de distribución dónde las demandas de los nodos son estocásticas, con el objetivo de decidir el número de facilities, su localización, y las asignaciones de los nodos a las facilities, y así minimizar la suma de; el coste de las facilities abiertas, el coste del mantenimiento de los inventarios, el coste total de abastecer a las facilities abiertas, y el coste de transporte entre facilities y clientes.
También hay que mencionar una extensión del FLP que se combina con otro problema de optimización, el *Steiner tree problem* (STP). Como resultado a esto, Karger y Minkoff (2000)[25] definieron el FLP conectado (conFLP). El ConFLP introduce nuevas restricciones adicionales al problema, dónde el problema de Steiner tiene que conectar todas las facility abiertas. Esta variación del problema es especialmente interesante para aplicaciones reales de redes porque combina problemas de localización y de conectividad.

Actualmente, se pueden encontrar algunos problemas basados en el diseño de redes digitales donde se puede aplicar el FLP. Thouin y Coates (2008)[26], proponen un ejemplo aplicándolo al vídeo bajo demanda (VoD). Éstos servicios son complejos y suelen requerir una cantidad alta de recursos, por lo tanto, las implementaciones de este tipo de servicios suelen implicar unos diseños muy cuidados. Una tarea muy importante en la fase dónde se planifica la red, es la del reparto de recursos, ya que debido al gran crecimiento de las redes peer-to-peer y el uso de dispositivos móviles han vuelto a este problema más complejo. Otro ejemplo se puede encontrar en el artículo de Lee y Murray (2010)[27]. Donde los autores exponen una solución basada en el FLP para solucionar un problema en que se diseña una red inalámbrica para toda una ciudad, donde aparecen dos dificultades principales; la primera es como poner el equipamiento Wi-Fi para maximizar la cobertura, y la segunda como conectar el equipamiento Wi-Fi para asegurarse que la red no se cae.
Maric (2013)[28] aplicó el FLP para abordar un problema dónde se tenían que colocar puntos de atención sanitaria en un conjunto de posibles lugares donde el objetivo era

minimizar el máximo número de pacientes asignados a los puntos sanitarios. Otros ejemplos donde aplicar el FLP, se pueden encontrar en el área de la gestión de cadenas de montaje o distribución.

También se pueden encontrar usos para las variantes del FLP en redes. Por ejemplo, el OFLP puede modelar un problema de diseño de red en donde se tienen que comprar varios servidores y cada uno de los nodos tiene que ser conectado a uno de estos. Una vez la red ha sido construida, puede haber la necesidad de añadir nuevos nodos a esa red. En este caso, se añadirían nuevos costes al problema como el coste de conectar nuevos nodos al cluster o el coste de instalar servidores adicionales.

Capítulo 5

Metaheurísticas usadas

Como introducción, el objetivo general de un algoritmo es el de conseguir buenos tiempos de ejecución y soluciones (normalmente las óptimas). Los algoritmos heurísticos abandonan uno o ambos objetivos. Por ejemplo normalmente pueden encontrar soluciones buenas, pero estas soluciones encontradas no pueden ser probadas como las mejores o sin un error existente. También pueden tener tiempos de ejecución muy rápidos, aunque no se puede asegurar que todas la ejecuciones sean siempre así.

En cambio, en este proyecto se habla de metaheurísticas, una metaheurística se define como un método heurístico para resolver un tipo de problema computacional y general. Las metaheurísticas generalmente se aplican a problemas que no tienen un algoritmo o heurística especifica que dé una solución satisfactoria, o cuando no es posible implementar un método para obtener un resultado óptimo. La mayoría de metaheurísticas tienen como objetivo los problemas de optimización combinatoria.

5.1. Local Search

Local Search es un método metaheurístico usado en una gran variedad de problemas de optimización. Puede ser usado en problemas formulados para encontrar una solución óptima dentro de un conjunto de posibles soluciones. Este algoritmo, va de solución a solución buscando dentro del conjunto de soluciones candidatas y aplicando cambios locales hasta que se encuentra una solución óptima o hasta que el tiempo máximo marcado se acaba. En otras palabras, este se puede ver como un proceso iterativo que empieza con una solución válida para el problema, y la mejora realizando modificaciones locales.

Algoritmo básico de Local Search:

$s \leftarrow$ generar solución inicial;
while *s no es el último local* **do**
    $s' \in N(s) conf(s) < f(s')$;
    (Si la solución mejor esta dentro de la vecindad de s) ;
    $s \leftarrow s'$;
end

Algorithm 1: Pseudocódigo LS.

El algoritmo 1, empieza con una solución inicial y busca entre las facility vecinas una mejor solución a la actual. Si la encuentra, la reemplaza por la nueva y continua con el proceso. El proceso continuará hasta que no se pueda mejorar la solución actual.

La solución final depende fuertemente de la primera solución inicial generada ya que se itera sobre esta, buscando soluciones óptimas en la vecindad. Esta es la principal desventaja de este algoritmo, ya que, por el motivo mencionado anteriormente, se suele quedar atrapado fácilmente en un óptimo local. Un óptimo local es una solución óptima dentro de un subconjunto del espacio de soluciones. En contra, el óptimo global es la solución óptima del espacio de soluciones entero.

5.2. Iterated Local Search

Referenciando a Lourenço et al.(2010)[29], el problema principal del *local search* es que puede quedarse atrapado en óptimos locales, que pueden ser peores que los mínimos globales. El ILS escapa de de los óptimos locales aplicando perturbaciones al mínimo local actual.

El ILS se compone de varias fases, primero se genera una solución inicial de forma aleatoria y se le aplica un local search para mejorarla. Una vez empezado el bucle, cada iteración pasa por tres etapas; la primera etapa se hace un perturbar, que lo que hace es romper la solución actual para generar otra distinta, la segunda etapa pasa por aplicar un *local search* a la solución obtenida de la perturbación, y la última etapa se comprueban los criterios de aceptación.

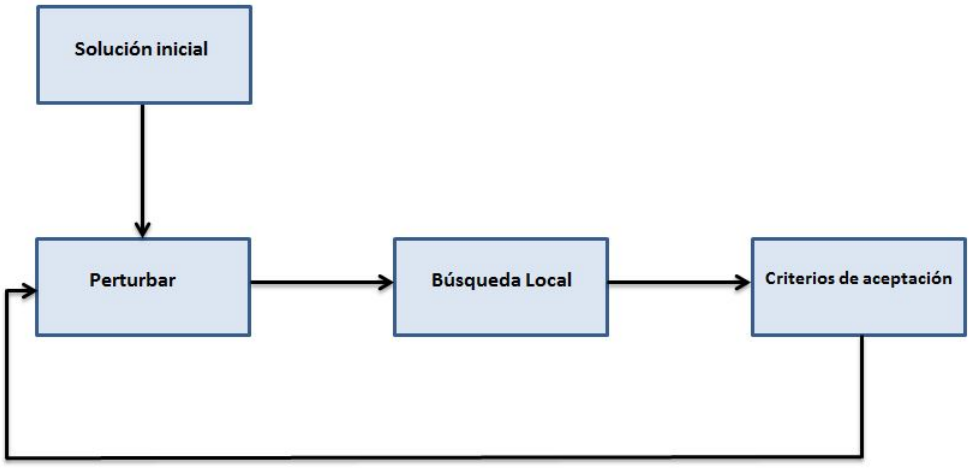

FIGURA 5.1: Flujo básico ILS

Pseudocódigo Iterated Local Search:

$s \leftarrow$ generar solución inicial;
$s* \leftarrow$ Local Search(s);
while *criterio para finalizar* **do**
    $s' \leftarrow$ perturbar(s*, historia);
    $s*' \leftarrow$Local Search(s');
    $s* \leftarrow$ criterio de aceptación (s*, s*', historia);
end

Algorithm 2: Pseudocódigo ILS.

La perturbación tiene que ser suficiente para salir del mínimo local, si es demasiado grande, esta nos puede llevar a un inicio aleatorio, y en cambio, si la perturbación es insuficiente, no será capaz de sacarnos del mínimo local. (figura 5.2).

En la figura 5.2, se puede ver dos ejemplos de perturbación aplicadas a una solución, depende lo fuerte que sea la perturbación (lo mucho o poco que se destruya la solución previa), esta saltará a un espació de solución o a otro espacio de soluciones que pueden ser mejores o no.
Para evitar volver a una misma solución, se deben incluir aspectos aleatorios o adaptativos, sino se transformaría en un proceso determinista.

5.3. Greedy adaptive search procedures (GRASP)

GRASP es un método *multi-start* que fue diseñado para resolver problemas de optimización (Feo y Resende, 1995)[30]. La metodología básica consiste en dos fases:

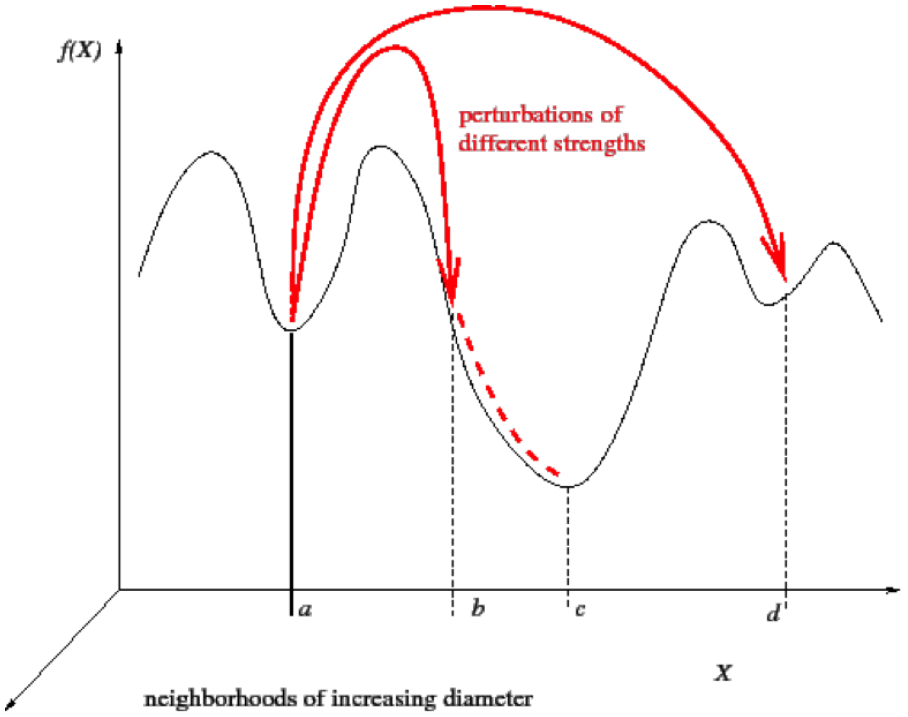

FIGURA 5.2: Perturbación de una solución (Fuente: Wikipedia)

a) Una fase de construcción, que genera una solución valida.

b) Una segunda fase que consiste en un procedimiento local search.

Se repiten las dos fases anteriores hasta que los criterios de aceptación se cumplen.

Pseudocódigo GRASP:

input : iteraciones máximas, semilla
output: Mejor solución obtenida
for *1 hasta número máximo de iteraciones* **do**
    $s \leftarrow$ Greedy Randomized Construction(semilla);
    $s \leftarrow$ Local Search (s);
    actualiza solución (s, mejor solución);
end

Algorithm 3: Pseudocódigo GRASP.

La fase de construcción, crea paso a paso la solución parcial y de manera aleatoria. Esta fase esta sujeta a un elemento de aleatorización, por ese motivo se le pasa una semilla, que se encarga de controlar el proceso.

En la segunda fase, se realiza un *local search* de la fase anterior. Una vez realizado esto, se van guardando los resultados y actualizando el mejor resultado obtenido en cada

una de las iteraciones. El algoritmo se ejecutará hasta que los criterios de aceptación se cumplan.

5.4. Aleatorización sesgada

La aleatorización sesgada se basa en que no todas las posibles facilities tienen el mismo peso probabilístico, dando un cierto tipo de probabilidad a las facilities con un menor coste (Ver figura 5.3).

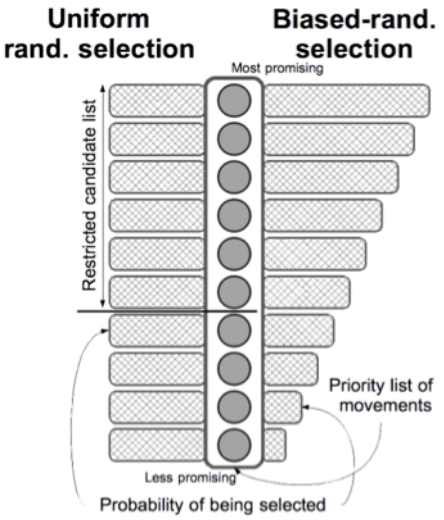

FIGURA 5.3: Aleatorización uniforme vs aleatorización sesgada. Fuente: Juan A. (2014)[31]

Esta técnica usa una distribución geométrica en función de un parámetro beta que oscila entre 0 y 1. Analizando los extremos, si la beta tiene el valor de 0, entonces actuará igual que una aleatorización uniforme. En cambio si el valor de la beta es 1, actuara de forma totalmente determinista.

Capítulo 6

Metodología de resolución

En este proyecto se ha propuesto una solución que combina una metaheurística basada en un Iterated Local Search (ILS) (Lourenço et al 2010)[29], con unas técnicas de aleatorización sesgada (Juan et al. 2011[32], Gonzalez-Martin et al. 2012a[33]) para poder solucionar de forma óptima el UFLP.

6.1. ILS con aleatorización sesgada

En primera instancia, la idea es hallar la solución mas óptima en el mínimo tiempo posible, haciendo aleatorio algunos pasos de la heurística. Por lo tanto cada vez que se ejecute el algoritmo, al usar estos procedimientos aleatorios, se obtendrán, siempre, resultados distintos. Cabe destacar, sin embargo, que dicho procedimiento no es absolutamente aleatorio, pues la aleatorización sesgada del proceso, favorece a los candidatos potencialmente mejores.

El algoritmo propuesto está representado en un diagrama de flujo en la Figura 6.1. Primero, carga con las instancias del problema, generando una primera solución inicial generada de manera aleatoria. Para generar esta solución inicial, el algoritmo elige un número aleatorio entre el numero total de facilities / 2 y el numero total de facilities y, elige aleatoriamente N facilities para abrir dentro de ese rango. Finalmente, calcula el coste total de la solución inicial generada. Esta es usada como punto de inicio.
El motivo por el cual se seleccionan más de la mitad de las facilities para abrir en la primera solución inicial, es debido principalmente, al coste computacional, pues resulta bastante menos costos cerrar una facility que abrirla, y esto sucede por la forma en como son calculados tales costes, así pues, cuando una facility es seleccionada para ser cerrada, a la hora de recalcular el coste total de la solución solo hay que reasignar los

nodos que estaban asignados a la facility que se ha cerrado, por lo que el resto de nodos asignados a otras facilities permanecerán igual. En cambio, cuando se abre una facility, cada uno de los nodos de la instancia que estamos calculando tienen que ser revalidados para asignarlos a una facility, esto es así porque no se puede saber si el nodo tendrá un coste mejor con la nueva facility abierta. Por este motivo, si se empieza con una solución inicial que contenga un gran número de facilities abiertas, será mucho más efectivo computacionalmente.

Una vez se ha generado la primera solución inicial, se le aplica un *Local Search* para refinar-la, en este caso se han propuesto dos *Local Search*. El primero es uno muy simple (al que llamaremos *Local Search* A), y el segundo es un *Local Search* más completo (al que llamaremos *Local Search* B).
El *Local Search* A está basado solo en el cierre de facilities. El funcionamiento es el siguiente: Este empieza por la solución actual y de forma aleatoria cierra una a una cada una de las facilities abiertas. Éstas son seleccionadas de manera aleatoria de entre todas las abiertas. Si la solución al cerrar una facility es mejor (tiene un coste menor), entonces se cierra definitivamente de la solución, sino esta facility se sigue manteniendo abierta.
Por otro lado, el Local Search B, trabaja en una combinación de abrir, cerrar e intercambiar facilities. Primero, este empieza abriendo una a una cada una de las facility cerradas. Este evalúa la solución en cada una de las etapas y solo añade una facility a la solución si esta la mejora. En una segunda etapa hace un cambio, reemplazando un número aleatorio de facilities abiertas en la solución por el mismo número de facilities cerradas. Finalmente, en la tercera etapa, se cierran una a una y también forma aleatoria las facilities abiertas, cerrándolas de forma definitiva siempre que en caso de cerrarla ofrezca una mejora en el coste de la solución.

La solución generada será finalmente el punto de inicio para el ILS y también será considerada como la solución base. El ILS es principalmente un proceso en donde, en cada una de las iteraciones, se genera una solución que mejora la solución base. El ILS propuesto en esta parte del proyecto es:

1) Destrucción/Contrucción de la solución con el perturbar

2) Refinar la solución con un Local Search

3) Calcular los criterios de aceptación de la solución propuesta

En el primer paso, se aplica una perturbación a la solución. Esta perturbación básicamente destruye una parte de la solución eliminando algunas facilities abiertas, y posteriormente reconstruye la solución abriendo otras nuevas. Teniendo en cuenta que siempre se abren más facility que se cierran. Esta solución vuelve a ser otra vez refinada de la

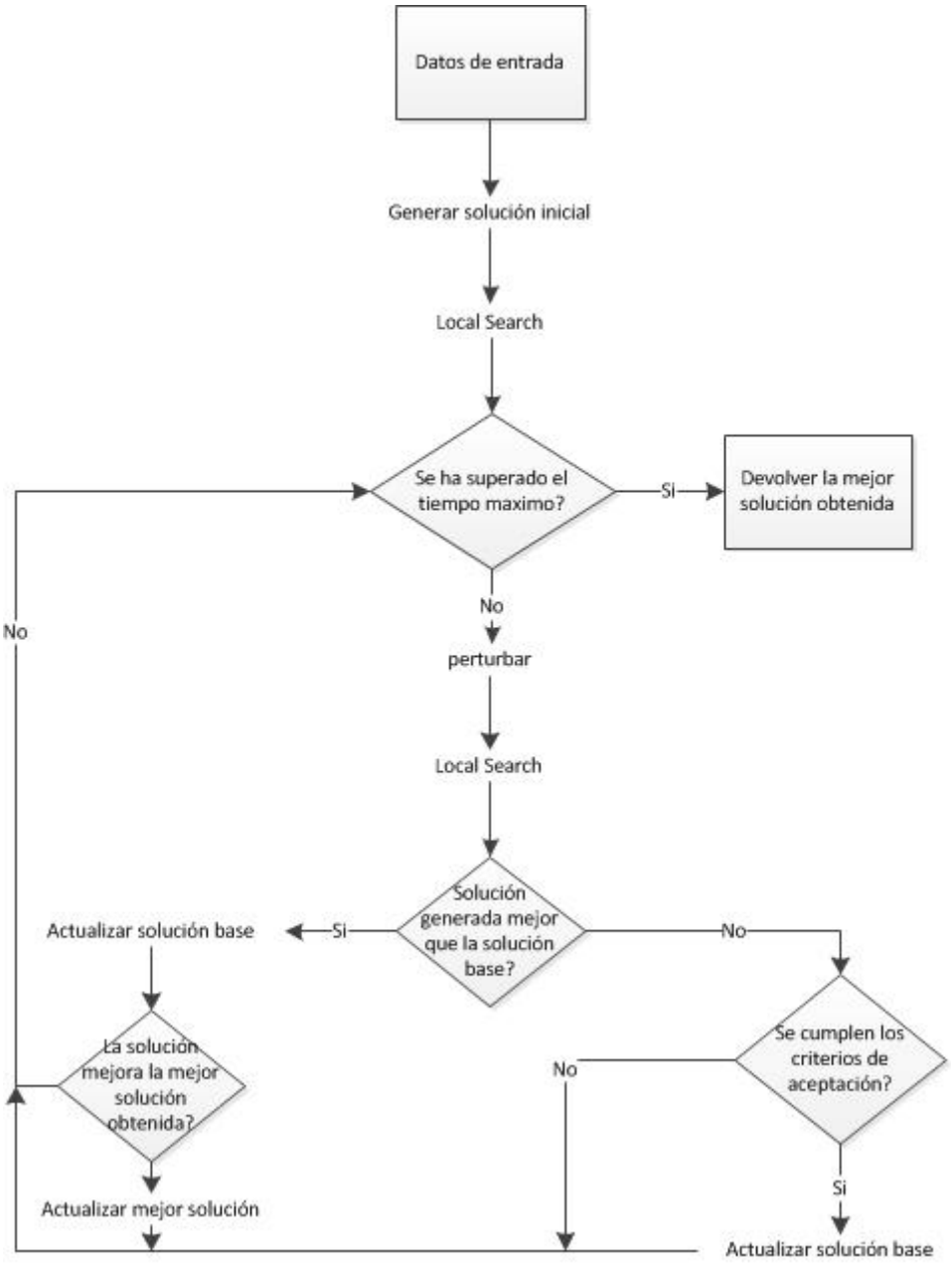

FIGURA 6.1: Diagrama de flujo ILS

misma manera que la solución inicial generada, con un *local search*.

Finalmente, en la última etapa del ILS, se comprueban los criterios de aceptación de la solución propuesta, decidiendo cuando se tiene que actualizar la solución base y la mejor solución conocida. En donde la mejor es la obtenida como resultado de todo el proceso anterior y la solución base es la resultante de la solución inicial generada.

Finalmente, si la solución obtenida por la perturbación y el *local search* mejora la mejor solución conocida hasta el momento, entonces la mejor solución y la solución base se actualizan.

6.2. Pseudocódigo

El método principal del código propuesto es el encargado del *Iterated Local Search*, y que se utilizará para resolver el *Uncapacitated Facility Location Problem* (algoritmo 4). Este método recibe como parámetros de entrada las facilities, los nodos que conforman el problema, el parámetro beta que es utilizado durante el proceso de aleatorización, y el parámetro utilizado como parte de los criterios de aceptación para finalizar el bucle del ILS. El formato de entrada de los datos están explicados en el apartado 6.1 del capítulo 6.

Procedimiento Iterated Local Search con aleatorización sesgada:

input : facilitites, nodos, beta, tiempoMax
output: Mejor solución obtenida
$initSol \leftarrow$ generarSolucionInicialAleatoria($facilities, nodos, beta$);
$baseSol \leftarrow$ LocalSearch($baseSol$);
$bestSol \leftarrow BaseSol$;
while $tiempoTranscurrido < tiempoMax$ **do**
    $newSol \leftarrow$ perturbar($baseSol, beta$);
    $newSol \leftarrow$ LocalSearch($newSol$);
    $bestSol \leftarrow$ actaulizarMejorSolucion($newSol$);
    $baseSol \leftarrow$ criteriosAceptacion($baseSol, newSol$);
end

Algorithm 4: Pseudocódigo algoritmo ILS

El algoritmo (4) del ILS se divide en cuatro grandes procesos. Estos cuatros grandes procesos son:

a) Generar solución inicial

b) Local Search

c) Perturbar

d) Criterios de aceptación

Resumiendo, primero en el método generarSolucionInicialAleatoria genera una solución inicial que es mejorada por un procedimiento Local Search. Como se ha mencionado en el capitulo 4, hemos definido dos procedimientos Local Search diferentes: Local Search A y Local Search B. Después de eso, se inicializan los valores que se van a usar para los criterios de aceptación y para finalizar el bucle while. Después, el bucle ILS empieza

usando la solución base refinada previamente con el Local Search. Dentro del bucle, la perturbación es aplicada a la solución base actual (la función se ha llamado perturbate), y la solución obtenida es otra vez mejorada o refinada por el procedimiento Local Search. La nueva solución obtenida se evalúa en base a los criterios de aceptación. A partir de aquí el bucle se ejecuta hasta que los criterios para que finalice se cumplan, el criterio para que el bucle finalice es el tiempo limite o hasta que encuentra una solución categorizada como la mas óptima encontrada hasta el momento. Para finalizar, el algoritmo devuelve la mejor solución encontrada.

6.2.1. Generar solución inicial

La función generarSolucionInicialAleatoria (algoritmo 5), es responsable de generar la solución inicial que sirve como punto de inicio del algoritmo. Este método recibe como parámetros las facilities, los nodos de la instancia del problema, y el parámetro beta para generar un numero aleatorio.
Esta función primero genera un numero aleatorio entre la mitad del numero total de facilities y el total de estas. Seguidamente, se elige un numero aleatorio (dentro del mínimo y máximo definido) de facilites de la instancia para abrir. Finalmente, esta construye la solución, marcando como abiertas las facilities seleccionadas y asignando cada nodo de la instancia a la facility con la que tenga el menor coste de servicio.

Procedimiento generarSolucionInicialAleatoria:

input : facilitites, nodos, beta
output: Solución obtenida
//Dado que cerrar es mas rápido, se abren muchas facilities;
$nFaciltoOpen \leftarrow$ rand$(size(facilities)/2, size(facilities))$;
//La aleatorizació sesgada utiliza una distribución geométrica.;
//Esta se hace en base a un parámetro beta que va entre 0 y 1;
$facilToOpen[] \leftarrow$ biasedRandSelect$(facilities, nFacilToOpen, beta)$;
$sol \leftarrow$constructSol$(facilToOpen, nodos)$;

Algorithm 5: Pseudocódigo método generarSolucionInicialAleatoria

Composición del código:

1) La variable nFacilToOpen, es un entero que indica el número de facilities que se tienen que abrir. Para ello se hace un random entre el número de las faciliites/2 y el número total de las facilities.

2) Se seleccionan de forma aleatorias las facilities que van a ser abiertas, esta selección se hace con el metodo biasedRandSelect, que se encarga de seleccionarlas a partir

del total de las facilities, el numero anteriormente calculado de facilities a abrir y el parámetro beta.

3) Una vez decididas las facilities a abrir, se construye la solución. Para ello se asignan todos los nodos a todas las facilities teniendo en cuenta el coste de asignación de cada nodo con cada facility. Un ejemplo sencillo se puede ver en el capitulo 6, en el apartado 6.1.1.

6.2.2. Perturbar

El objetivo del método perturbar es romper la solución actual destruyéndola y posteriormente construyéndola.
Para conseguir esto, lo que se hace es primero cerrar un número aleatorio de las facilities abiertas. Y luego abrir aleatoriamente un número de las facilities cerradas.

Como parámetros de entrada, recibe una solución base y un parámetro beta para generar el numero aleatorio usando la distribución geométrica.

Procedimiento Perturbar:

input : baseSol, beta
output: Solución obtenida
$sol \leftarrow$ copy($baseSol$);
$openFacilities \leftarrow$ getOpenFacilities(sol);
$nFacilitiesToClose \leftarrow$ rand$(0, size(OpenFacilities))$;
//La aleatorización sesgada utiliza una distribución geométrica.;
$invOpenFacilities \leftarrow$ inverseOrder($openFacilities$);
$facilitiesToClose[] \leftarrow$ biasedRandSelect($invOpenFacilities, nFacilToClose, beta$);
$sol \leftarrow$destructSol($sol, facilToClose$);
$closedFacilities \leftarrow$getClosedFacilities($sol$);
//Abrimos mas facilities dado que cerrar es mas rápido que abrir nuevas;
$faciliesToOpen[] \leftarrow$biasedRandSelect($closedFacilities, nFacilToOpen, beta$);
$sol \leftarrow$constructSol($facilToOpen$);

Algorithm 6: Pseudocódigo método perturbar una solución

Composición del código:

1) Se hace una copia de la baseSol, que es el único parámetro de entrada junto con el número beta. La variable baseSol será una clase que englobará todas las facilities abiertas, cerradas, coste total y por cada facility abiertas los nodos que tiene asignados con los costes.

2) Extrae una lista de las facilities abiertas de la solución copiada en el paso anterior.

3) Selecciona el numero de facilities para cerrar y lo guarda en la variable nFacilities-ToClose.

4) Se reordena la lista de facilities abiertas, esta reordenación lo que hace es invertir el orden actual (que ya venia ordenado) de las facilites. El objetivo de esto es que en el momento en que se haga la selección de facilities para cerrar, y se use la técnica de la aleatorización sesgada, se seleccionen las peores (costes mas altos) para cerrar.

5) Se seleccionan las facility a cerrar utilizando la técnica de aleatorización sesgada.

6) destructSol cierra las facilities que han sido seleccionadas para ser cerradas y genera una nueva solución.

7) Ahora hay que reconstruir la solución, por lo tanto se hace lo inverso a lo que se ha realizado anteriormente. Primero se obtienen de la solución generada en el punto anterior, un listado de todas las facilities cerradas.

8) Se hace una selección de las facilities que tienen que ser abiertas con la técnica de aleatorización sesgada. Siempre se generará un número mas grande de facilities a abrir que las cerradas en los puntos anteriores. Esto es así porque el coste computacional de cerrar facilities es mucho menor que el de abrirlas.

9) En el último paso se reconstruye la nueva solución añadiendo las nuevas facilities abiertas.

6.2.3. Local Search

El método *local Search* A (algoritmo 7), es basado en el cierre de facilities. Este procedimiento recibe como parámetros la solución base y el parámetro beta. Primero este crea una copia de la solución base y extrae la lista de facilities abiertas y ordenandolas de forma aleatoria. Después de eso, todas las facilities son eliminadas una a una de la solución. Si la solución sin la facility que ha sido eliminada tiene un coste total mejor que el anterior, la facility es finalmente eliminada. Sino, es mantenida en el conjunto de facilities abiertas.

Procedimiento LocalSearch A:

```
input  : baseSol, beta
output: Solución obtenida
sol ←copy(baseSol);
openFacilities ←getOpenFacilities(sol);
openFacilitiesSorted ←biasedRandSort(openFacilities);
for openFacility in openFacilities do
    newSol ←deleteFacility(sol, openFacility);
    if cost(newSol) < cost(sol) then
        sol ← newSol;
    else
        newSol ←addFacility(sol, openFacility);
    end
end
```

Algorithm 7: Pseudocódigo método LocalSearch A

El método local search B (algoritmo 8), aplica un *local search* mas completo, que puede ser también usado para refinar las soluciones del UFLP. Este procedimiento recibe como parámetro la solución base y el parámetro beta. Este empieza creando una copia de la solución base. Después, empieza el bucle que esta estructurado en tres bloques distintos.

Procedimiento LocalSearch B:

```
input  : baseSol, beta
output: Solución obtenida
sol ←copy(baseSol);
while improvement do
    improvement ← false;
    closedFacilities ←getClosedFacilities(sol);
    closedFacilitiesSorted ←biasedRandSort(closedFacilities, beta);
    for closedFacility in closedFacilitiesSorted do
        newSol ←addFacility(sol, closedFacility);
        if cost(newSol) < cost(sol) then
            sol ← newSol;
            improvement ← true;
        else
            newSol ←deteleFacility(sol, closedFacility);
        end
    end
    openFacilities ←getOpenFacilities(sol);
    for openFacility in openedFacilities do
        newSol ←deteleFacility(sol, openFacility);
        closedFacilities ←getClosedFacilities(sol);
        for closedFacility in closedFacilities do
            newSol ←addFacility(sol, closedFacility);
            if cost(newSol) < cost(sol) then
                sol ← newSol;
                improvement ← true;
                break;
            else
                newSol ←deleteFacility(sol, closedFacility);
            end
        end
        if cost(newSol) < cost(sol) then
            sol ← newSol;
            improvement ← true;
            break;
        else
            newSol ←addFacility(newSol, closedFacility);
        end
    end
    sol ←LocalSearchA(sol, beta);
end
```

Algorithm 8: Pseudocódigo método LocalSearch B (completo)

En el primer bloque, intentamos abrir facilities que están cerradas. Esta lista de facilities cerradas es obtenida de la solución actual, una vez obtenida, la ordenamos de forma aleatoria. Todas las facilities cerradas se añaden una a una a la solución. Si la solución con la facility que vamos a añadir tiene un coste total menor que sin añadirla, entonces esta facility se añade finalmente a la solución y la variable de improvement se marca como verdadero. Sino no se añade.
El segundo bloque del bucle, lo que hace es intercambiar facilities abiertas por facilities cerradas. Hace los intercambios hasta que se encuentra una mejora en los costes totales de la solución.
Finalmente, en el tercer bloque, eliminamos las facilities abiertas haciendo uso del *local search* A. Este usa el mismo sistema que los bloques anteriores, si el coste total de la solución es menor que el anterior, entonces la facility se cierra de forma definitiva.

6.2.4. Criterios de aceptación

El algoritmo no usa un criterio de aceptación clásico como normalmente se usa en la mayoría de ILS, en este caso, se usa un proceso computacionalemnte mas simple llamado *Demon-like* (Talbi, 2009). Este, complementando el proceso de perturbación, esta diseñado también para evitar los mínimos locales.

Procedimiento para los criterios de aceptación para el ILS:

```
delta ←cost(newSol) − cost(baseSol);
if delta < 0 then
    credit ← −delta;
    baseSol ← newSol;
    if cost(newSol) < cost(bestSol) then
        bestSol ← newSol;
    end
else
    if delta > 0 and credit >= delta then
        credit ← 0;
        baseSol ← newSol;
    end
end
```

Algorithm 9: Pseudocódigo criterios de aceptación usados

El proceso de aceptación funciona de la siguiente manera:

a) Cada vez que una nueva solución generada mejore la solución base, esta se actualiza y la mejora. Además, esta nueva solución es comparada con la mejor solución conocida hasta el momento, para comprobar si esta también puede ser actualizada y mejorada.

b) En el caso de que la nueva solución generada sea peor que la solución base, esta solución base será actualizada (deteriorada) siempre y cuando no se realicen mas de una deterioración consecutiva, y siempre que el nivel de deterioro de la solución no exceda la ultima mejora. Con esta técnica, se consigue reducir las probabilidades de que el algoritmo quede atrapado en un mínimo local.

Capítulo 7

Pruebas computacionales

Para poder evaluar el rendimiento del algoritmo propuesto en este proyecto, se han realizado una serie de experimentos computacionales, de este modo dicho algoritmo ha sido implementado en Java® SE 7. Para evaluar el rendimiento se han realizado todos los test en un PC con un Intel® CoreTM i5-2400 a 3.20 GHz y 4 GB RAM corriendo Ubuntu GNU/Linux 14.04 como sistema operativo.

Java tiene una gran cantidad de API que pueden ser usadas, también es un lenguaje orientado a objetos, pudiendo facilitar y acelerar así el proceso de desarrollo, aun teniendo presente que Java es un lenguaje de programación que se ejecuta en una maquina virtual (JVM), y por lo tanto puede tener un rendimiento inferior a otros lenguajes de programación como C o C++.

El proceso de implementación del algoritmo no es trivial, ya hay que tener en cuenta varios detalles que pueden hacer marcar la diferencia. Por ejemplo:

- El correcto diseño de las diferentes clases, con el fin de obtener un gran nivel de acoplamiento y cohesión.
- El nivel de precisión usado para guardar y operar con valores numéricos, es un factor importante para conseguir un buen nivel de efectividad y rendimiento.

7.1. Datos de entrada

Para testear la eficiencia del algoritmo propuesto, se han usado 4 tipos diferentes de instancias. Se han seleccionado diferentes conjuntos de datos, separándolos por instancias pequeñas, medianas y grandes (en términos del numero de facilities y el número de nodos incluidos en el grafo).

A continuación se describen los diferentes grupos de instancias de datos que han sido usadas, pudiéndose ampliar los detalles que se describen a continuación a partir del articulo de Hoefer (2014)[34].

- *BK*: Conjunto de datos de tamaño pequeño. Propuesto por Bilde y Krarup (1977)[35]. Este incluye un total de 220 instancias divididas en 22 subconjuntos, con un número de facilities que varia de 30 a 50 y un numero de nodos que varia de 80 a 100. Estas instancias fueron creadas de forma artificial por sus creadores, creando los costes de asignación de forma aleatoria dentro de un rango que oscila entre 0 y 1000, y teniendo en cuenta que los costes de abrir una facility son siempre mayores que 1000.
- *GAP*: Conjunto de datos de tamaño medio. Propuesto por Kochetov y Ivanenko (2003)[36]. Este consiste en tres subconjuntos, cada uno con 30 instancias. Los subconjuntos son: GAPA, GAPB, GAPC, siendo GAPC la mas difícil de las 3.
- *FPP*: Conjunto de datos de tamaño medio. Propuesto también por Kochetov y Ivanenko (2003)[36]. Este consiste en dos subconjuntos, cada uno con 40 instancias. Los subconjuntos son FPP11 y FPP 17. Estas instancias están preparadas para tener un gran numero de óptimos locales. Por lo tanto, estas instancias están consideradas como difíciles para algoritmos basados en Local Search.
- *MED*: Conjunto de datos de tamaño grande. Pensado para problemas p-median y propuesto en primera instancia por Ahn et al. (1998), y usado posteriormente por Barahona y Chundak (1999)[37] para el Uncapacitated FLP. Este consiste en un conjunto de instancias, donde cada una de las instancias es un conjunto de n puntos que han sido elegidos de manera aleatoria. Éste, consiste en un total de seis subconjuntos dónde cada uno de estos tiene un número diferente de facilities y de nodos (500, 1000, 1500, 2000, 2500 y 3000).

Para realizar los experimentos, se ha puesto un tiempo limite proporcional para cada una de las instancias (Ver tabla 7.1). Con el fin de garantizar que el algoritmo le de tiempo y pueda encontrar la mejor solución óptima. En la mayoría de los casos, se ha encontrado la solución óptima antes de llegar a este tiempo limite.

7.1.1. Formato de los datos de entrada

Las instancias explicadas en el punto anterior, están guardadas con el formato especificado en [38].

La primera linea consiste en FILE:<nombreDelFichero.txt>.

Conjunto	**Numero de facilities (n)**	**Tiempo(s)**
Bk	80-100	30
FPP11	133	600
FPP17	307	600
GAPA	100	180
GAPB	100	180
GAPC	100	180
MED	500-3000	3600

CUADRO 7.1: Limites de tiempo por conjunto de facilities

La segunda linea indica las dimensiones del problema. Consiste en [N] [M] 0, donde N son el número de facilities y M el numero de nodos.

A partir de aquí, habrá tantas lineas como facilities N, cada una de las lineas es para una facility. El primer numero de cada linea es el identificador de la facility, el segundo número es el coste de abrir esta facility, y a partir de aquí, los M nodos. Esta lista de nodos, indica el coste de asignar el nodo i-esimo con esta facility.

```
Ejemplo: n = 4, m = 3

FILE: Ejemplo.txt
4 3 0
1 300 130 120 80
2 400 140 100 50
3 150 130 90 140
4 200 100 120 150
```

También hay otro fichero .opt que contiene los datos de la solución óptima conocida. Este fichero esta compuesto de una única linea con una lista de números, en donde cada número indica que ID de cada facility se asigna al nodo i-esimo. El último número de esta linea indica el coste total de esta solución.

Por ejemplo, un .opt para el ejemplo anterior sería:

```
ejemplo.opt
3 3 3 3 510
```

Esto indica que la solución óptima al problema del ejemplo es abrir solo la facility con el identificador número 3. Esto es así porque la suma del coste de abrir la facility y el coste de conectar el resto de facilities a esta, es mejor que cualquier otra combinación que se pueda realizar.

7.2. Resultados obtenidos

En este apartado se muestran los resultados obtenidos. Para visualizar correctamente la eficiencia de estos, se han comparado estos resultados con los resultados obtenidos por el algoritmo usado por Resende y Werneck (2006)[19] usando GRASP. Hay que remarcar, que las pruebas computacionales realizadas han sido echas en un PC de escritorio normal, a diferencia de las realizadas por Resende y Werneck, que usaron un superordenador.

En las tablas se compara, los resultados obtenidos por Resende y Werneck, los resultados obtenidos con el *Local Search* A, y con el *Local Search* B. Cada una de estas columnas esta dividida por dos mas, estas muestran primero el error y luego el tiempo empleado.

7.2.1. Conjunto de datos BK

La tabla 7.2 muestra la comparativa con el mas pequeño de los conjuntos de datos, el propuesto por Bilde y Krarup (1977)[35].

			GRASP		**ILS-UFLP A**		**ILS-UFLP B**	
Instancia	**# nodos**	**# facilities**	**Error**	**t(ms)**	**Error**	**t(ms)**	**Error**	**t(ms)**
B	100	50	0.000	310	0.000	794	0.000	0.20
C			0.016	450	0.130	835	0.000	0.05
D01	80	80	0.000	223	0.001	116	0.000	0.02
D02			0.000	211	0.000	317	0.000	0.04
D03			0.000	199	0.000	58	0.000	0.06
D04			0.000	170	0.000	48	0.000	0.08
D05			0.000	162	0.000	41	0.000	0.10
D06			0.000	186	0.000	124	0.000	0.12
D07			0.000	174	0.000	9	0.000	0.15
D08			0.000	166	0.000	20	0.000	0.08
D09			0.000	175	0.000	11	0.000	0.07
D10			0.000	166	0.000	28	0.000	0.21
E01	100	50	0.000	476	0.201	667	0.000	0.03
E02			0.000	588	0.011	1176	0.000	0.07
E03			0.019	512	0.000	286	0.000	0.11
E04			0.000	464	0.000	389	0.000	0.14
E05			0.000	376	0.000	223	0.000	0.18
E06			0.000	408	0.000	247	0.000	0.22
E07			0.000	416	0.000	451	0.000	0.25
E08			0.000	418	0.000	728	0.000	0.28
E09			0.000	352	0.000	35	0.000	0.29
E10			0.000	353	0.000	70	0.000	0.32
MEDIA			0.002	316	0.015	290	0.000	0.14

CUADRO 7.2: Resultados obtenido por los 22 BK subconjuntos de instancias.

En esta prueba (ver tabla 7.2), el algoritmo realizado mejora los resultados obtenidos por GRASP cuando se usa el algoritmo con el método de Local Search B. La mejora

consiste en que obtenemos la solución óptima para todas las instancias en un tiempo de ejecución mucho menor. Por el contrario, con el Local Search A se obtienen resultados peores, tanto por el error conseguido como con el tiempo.

7.2.2. Conjunto de datos GAP

A continuación, se presentan los resultados obtenidos por el siguiente conjunto de datos GAP.

			GRASP		**ILS-UFLP A**		**ILS-UFLP B**	
Instancia	**# nodos**	**# facilities**	**Error**	**t(s)**	**Error**	**t(s)**	**Error**	**t(s)**
GAP A			5.140	1.41	2.470	47.68	1.095	31.25
GAP B	100	100	5.980	1.81	2.811	63.56	1.642	40.72
GAP C			6.740	1.89	2.859	70.17	1.207	59.55
MEDIA			5.953	1.700	3.405	54.43	1.314	43.84

Cuadro 7.3: Resultados obtenido por los subconjuntos de instancias GAP.

En la tabla 7.3, el algoritmo realizado obtiene resultados de mejor calidad en los dos Local Search realizados. Destacando el Local Search B, en donde el error obtenido es mucho menor que el Local Search A y el que GRASP. Por otro lado, los tiempos que ha tardado en encontrar este error son mas altos que los tiempos obtenidos por GRASP. Resumidamente en esta instancia se obtienen mejores resultados pero sacrificando algo de tiempo.

7.2.3. Conjunto de datos FPP

La siguiente prueba ha sido realizada sobre el conjunto de instancias FPP.

			GRASP		**ILS-UFLP A**		**ILS-UFLP B**	
Instancia	**# nodos**	**# facilities**	**Error**	**t(s)**	**Error**	**t(s)**	**Error**	**t(s)**
FPP 11	133	133	8.480	2.58	0.063	127.126	0.000	111.02
FPP 17	307	307	58.270	25.18	70.731	272.94	12.283	253.44
MEDIA			33.375	13.88	35.397	200.03	6.142	182.22

Cuadro 7.4: Resultados obtenido por los subconjuntos de instancias FPP.

Del mismo modo que los resultados obtenidos en las pruebas realizadas con la instancia GAP (tabla 7.3), en la tabla 7.4 el algoritmo realizado mejora el error obtenido, tanto con el Local Search A como con el Local Search B. El Local Search A, obtiene unos resultados bastante malos en comparación con los otros dos algoritmos, esto es debido a su simplicidad para generar nuevas soluciones, teniendo en cuenta la complejidad del problema.

7.2.4. Conjunto de datos MED

Finalmente, se presenta en la tabla 7.5 la última prueba realizada con instancias grandes, llamada MED. La cual está formada por un conjunto de instancias que se van incrementando en tamaño y dificultad.

			GRASP		**ILS-UFLP A**		**ILS-UFLP B**	
Instancia	**# nodos**	**# facilities**	**Error**	**t(s)**	**Error**	**t(s)**	**Error**	**t(s)**
500-10			0.022	33.2	0.022	213	0.022	7.35
500-100	500	500	0.016	32.9	0.093	676	0.014	345
500-1000			0.071	23.6	0.071	3168	0.078	2803
1000-10			0.101	173.9	0.399	2038	0.099	69
1000-100	1000	1000	0.048	148.8	0.333	3566	0.088	2898
1000-1000			0.037	141.7	1.165	3468	0.447	4494
1500-10			0.191	347.8	0.236	2418	0.175	1012
1500-100	1500	1500	0.030	378.7	0.687	3582	0.094	1427
1500-1000			0.034	387.2	3.514	3540	0.320	20934
2000-10			0.052	717.5	0.223	3415	0.052	1276
2000-100	2000	2000	0.036	650.8	1.198	3573	0.299	2611
2000-1000			0.031	760.0	5.468	2838	0.403	57678
2500-10			0.164	1419.5	0.622	2988	0.168	2248
2500-100	2500	2500	0.049	1128.2	1.537	3569	0.349	4369
2500-1000			0.052	1309.4	5.964	3533	0.308	108557
3000-10			0.104	1621.1	0.372	3570	0.102	2362
3000-100	3000	3000	0.124	1977.6	1.707	3538	0.545	4904
3000-1000			0.043	2081.4	6.238	3427	0.319	228691
MEDIA			0.067	740	1.659	2951	0.938	12970

CUADRO 7.5: Resultados obtenido por los subconjuntos de instancias MED.

Los resultados obtenidos en la tabla MED (ver 7.5), muestra como el algoritmo propuesto da mejores resultados en los casos donde el coste de instalación de una facility es mayor. En las instancias pequeñas, el algoritmo es bastante competitivo comparándolo con los resultados de Resende. En cambio, en las instancias mas grandes, el algoritmo propuesto obtiene, en comparación, peores resultados, debido la gran cantidad de facilities abiertas en estas soluciones. Aun así los resultados obtenidos son muy buenos y no están lejos del mejor resultado óptimo conocido.

Cabe remarcar la posibilidad que en algunos resultados obtenidos, el tiempo dedicado sea mayor al máximo marcado en la tabla 1, esto es debido a que, dado a la gran cantidad facilities y nodos que hay en las instancias, el tiempo máximo ha sido superado mientras se estaba realizando una iteración.

Capítulo 8

Ideas de trabajo futuro

Con el objetivo de mejorar los resultados obtenidos, se ha planteado un nuevo algoritmo basado en GRASP y no en un ILS. El objetivo ideal es desarrollar un algoritmo mas sencillo, rápido y eficiente.

Se presentan a continuación lo que podría ser una solución a los objetivos anteriormente planteados. La primera versión planteada es la mas sencilla, e idealmente cuando funcionase se implementaría la mejora.

La primera idea fundamental sería implementar un algoritmo multi-start que hiciera lo siguiente:

1. Generar una solución inicial, abriendo todas las facilities. El motivo es el mismo que el explicado durante el proyecto, es mucho menos costoso computacionalmente cerrar facilities que abrirlas.
2. Hacer un *shuffle*(ordenación aleatoria) del array de las facilities.
3. Siguiendo el orden del *shuffle*, cerrar una a una las facilities siempre y cuando al cerrarla se produzca una mejora en los costes.
4. Aplicar un Local Search (abrir + cerrar) de la solución obtenida en (3).
5. Una vez probadas todas las facilities, se comprobaría si la solución ha mejorado la mejor solución conocida hasta el momento y, si es el caso, la actualiza.
6. Repetir des del paso (1) hasta que quede tiempo.

La anterior sería una versión muy parecida a GRASP. Una vez funcionase y estudiados los resultados obtenidos, se implementarían una serie de mejoras:

a) Se definiría el "ahorro" asociado a una facility A siendo este la diferencia entre el coste de la solución con todas las facilities abiertas y el coste de la solución con la facility A cerrada. Este concepto de "ahorro" es "global", se calcularía solo una vez antes de empezar el multi-start, y nos indicaría lo mucho o lo poco que conviene cerrar una facility (podría haber facilities con "ahorro" positivos y otras con "ahorro" negativo).

b) Adaptar el multi-start descrito en la sección anterior para que utilice la técnica de aleatorización sesgada (con la distribución geométrica) en vez del shuffle. Se ordenarían las facilities en base al concepto de "ahorro" explicado en el apartado (a), de modo que cuando se buscase una facility para cerrar, se seleccionaría de la lista ordenada de facilities usando la aleatorización sesgada, y del mismo modo cuando se buscase una facility para abrir se haría a partir de una lista ordenada en base al "ahorro" y se seleccione con la aleatorización sesgada.

Una vez aplicadas estas dos mejoras, obtendríamos un algoritmo muy parecido al GRASP, pero con dos factores nuevos, que sería el concepto del ahorro y la técnica de aleatorización sesgada.

Aparte de la propuesta detallada anteriormente, también se plantea estudiar otras variantes para solucionar problemas similares.

Capítulo 9

Conclusiones

Con la realización de este proyecto, se ha logrado diseñar e implementar un nuevo algoritmo basado en un *Iterated Local Search*, con un añadido muy potente de aleatorización sesgada, el cual, ha sido el factor clave para conseguir muy buenos resultados en las instancias de datos más grandes.

Otro valor añadido, es el ILS ya que es relativamente sencillo y adaptable a muchos tipos de problemas y variantes del FLP. También es muy eficiente, tal como se ha visto en las pruebas computacionales explicadas en el capítulo 7. Otra ventaja a destacar del ILS es que, como se ha demostrado y gracias a su funcionamiento basado en su estructura eficiente entre las fases del *Local Search*, perturbación y criterios de aceptación, necesita pocos parámetros de entrada. El FLP y sus variantes, soy muy útiles y relativamente fáciles de implementar y adaptar a problemas reales, tal y como se ha visto y comentado en el capítulo dos dónde se habla de la revisión literaria.

Los resultados obtenidos han sido muy prometedores, tal como se muestran y comentan en el capítulo 7. En la mayoría de instancias, se han obtenido resultados muy competitivos, excepto en las más grandes donde se ha obtenido de media un error mas grande respecto al óptimo conocido en contra de los resultados obtenidos por Resende con GRASP. No obstante, son muy buenos y con errores muy bajos.

Tal como se ha propuesto en el capítulo 8, modificando las estructuras de datos, y haciendo pequeñas modificaciones en la metaheurística, se podrían conseguir resultados más competitivos con las instancias de datos grandes.

Apéndice A

Código ILS

```
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.logging.Level;
import java.util.logging.Logger;

public class ILSTester {

    private final static Logger LOGGER = Logger.getLogger("");
    private final static int NUMBER_OF_TEST_PER_INSTANCE = 1;
    private final static String INPUT_FOLDER = "inputs";
    private final static String OUTPUT_FOLDER = "outputs";
    private final static String TEST_FOLDER = "tests";
    private final static String FILE_NAME_EXT = ".txt";
    private final static String FILE_NAME_STATS = "stats";

    private final static String[] TEST_TO_RUN_LIST_SMALL_1 = new String[]{
        // small size instances
        "test2run_BildeKrarup_B", "test2run_BildeKrarup_C"
    };
    private final static String[] TEST_TO_RUN_LIST_SMALL_2 = new String[]{
        // small size instances
        "test2run_BildeKrarup_Dq_1", "test2run_BildeKrarup_Dq_2",
        "test2run_BildeKrarup_Dq_3", "test2run_BildeKrarup_Dq_4", "test2run_BildeKrarup_Dq_5",
        "test2run_BildeKrarup_Dq_6", "test2run_BildeKrarup_Dq_7", "test2run_BildeKrarup_Dq_8",
        "test2run_BildeKrarup_Dq_9", "test2run_BildeKrarup_Dq_10", "test2run_BildeKrarup_Eq_1",
        "test2run_BildeKrarup_Eq_2", "test2run_BildeKrarup_Eq_3", "test2run_BildeKrarup_Eq_4",
        "test2run_BildeKrarup_Eq_5", "test2run_BildeKrarup_Eq_6", "test2run_BildeKrarup_Eq_7",
        "test2run_BildeKrarup_Eq_8", "test2run_BildeKrarup_Eq_9", "test2run_BildeKrarup_Eq_10"
    };

    private final static String[] TEST_TO_RUN_LIST_MEDIUM_1 = new String[]{
        // medium size instances
        "test2run_GapA", "test2run_GapB", "test2run_GapC"
    };

    private final static String[] TEST_TO_RUN_LIST_MEDIUM_2 = new String[]{
        // medium size instances
        "test2run_Fpp11", "test2run_Fpp17"
    };

    private final static String[] TEST_TO_RUN_LIST_BIG = new String[]{
        // big size instances
        "test2run_kmedian"
    };

    private final static String[] TEST_TO_RUN_LIST = new String[]{
        "test2run_BildeKrarup_Dq_10"//"test2run_BildeKrarup_B", "test2run_BildeKrarup_C"
    // "test2run_kmedian"
    // "test2run_GapB"
    };
```

```
    /**
     * @param args the command line arguments
     * @throws java.io.IOException
     */
    public static void main(String[] args) throws IOException {

        final long programStart = ElapsedTime.systemTime();
        LOGGER.setLevel(Level.ALL);

        /* 1. GET THE LIST OF TESTS TO RUN */
        final ArrayList<Test> testsList = TestsManager.getTestsList(TEST_FOLDER + File.separator,
    TEST_TO_RUN_LIST_BIG, FILE_NAME_EXT);
        final StatsWriter stats = new StatsWriter(OUTPUT_FOLDER + File.separator + FILE_NAME_STATS);

        /* 2. FOR EACH TEST IN THE LIST... */
        for (int k = 0; k < testsList.size(); k++) {
            for (int i = 0; i < NUMBER_OF_TEST_PER_INSTANCE; i++) {
                final Test aTest = testsList.get(k);

                LOGGER.log(Level.INFO, "[{0}] Test for instance {1} prepared ({2}/{3})", new
    Object[]{ElapsedTime.calcElapsed(programStart, ElapsedTime.systemTime()), aTest.getInstanceName(), k + 1,
    testsList.size()});

                // 2.0 RESET SOLUTION IDs at every iteration
                Solution.resetIds();

                // 2.1 GET THE INSTANCE INPUTS
                final Input someInputs = InputsManager.getInputs(INPUT_FOLDER + File.separator +
    aTest.getInstanceFullPath());

                System.out.println("Expected optimal solution cost: " + someInputs.getOptimalSolutionCost());
                System.out.println("Expected num of Open Facilities: " +
    someInputs.getOptimalSolutionOpenFacilities().length);

                // 2.2 USE THE METAHEURISTIC TO SOLVE THE INSTANCE
                final ILS ms = new ILS(aTest, someInputs, stats);
                final Solution sol = ms.solve();

                final String outputsFilePath = OUTPUT_FOLDER + File.separator + aTest.getInstanceName() + "_" +
    aTest.getSeed() + FILE_NAME_EXT;
                sol.sendToFile(outputsFilePath);
            }
        }

        /* 3. END OF PROGRAM */
        LOGGER.log(Level.INFO, "[{0}] All tests ended correctly, saving data", ElapsedTime.calcElapsed(programStart,
    ElapsedTime.systemTime()));
        LOGGER.log(Level.WARNING, "[{0}] Full program execution ended correctly " + "in {1}", new
    Object[]{ElapsedTime.calcElapsed(programStart, ElapsedTime.systemTime()),
    ElapsedTime.calcElapsedHMS(programStart, ElapsedTime.systemTime())});
    }
}
```

```
import java.util.List;

public class ILS {

    private final Test test; // The test to execute.
    private final Input inputs; // The input data of the problem instance.
    private final StatsWriter stats; // The stats writer.
    private long startTime; // The start time.
    private final Randomness randomness;

    public ILS(final Test aTest, final Input inputs, final StatsWriter stats) {
        this.test = aTest;
        this.inputs = inputs;
        this.stats = stats;
        this.randomness = new Randomness(aTest);
        this.startTime = ElapsedTime.systemTime();
    }

    public Solution solve() {
```

```
        // 0. Create a base solution and set initial time variables
        this.startTime = ElapsedTime.systemTime();

        // 1. CONSTRUCT A RANDOMIZED STARTING POINT
        final Solution initSol = getInitialRandSol(inputs.getFacilities());
        Solution baseSol = localSearch(initSol.clone(), test.getMaxIter());
        baseSol.setTime(ElapsedTime.calcElapsed(startTime, ElapsedTime.systemTime()));

        Solution bestSol = baseSol.clone();
        // 2. ITERATIVE DISRUPTION AND LOCAL SEARCH
        int delta = 0;
        int credit = 0;
        int nIter = 0;

        while (ElapsedTime.calcSeconds(ElapsedTime.calcElapsed(startTime,
                ElapsedTime.systemTime())) < test.getMaxTime()
                && bestSol.getTotalCosts() > inputs.getOptimalSolutionCost()) {

            Solution newSol = baseSol.clone();

            // 2.1 Perturbate base solution
            // 2.2 Local search
            newSol = localSearch(perturbate(newSol), test.getMaxIter());

            // 2.3 Update time
            newSol.setTime(ElapsedTime.calcElapsed(startTime, ElapsedTime.systemTime()));

            // 2.4 aceptance criteria
            delta = newSol.getTotalCosts() - baseSol.getTotalCosts();

            if (delta < 0) {
                credit = -delta;

                baseSol = newSol;

                if (newSol.getTotalCosts() < bestSol.getTotalCosts()) {
                    bestSol = newSol;
                }
            } else {
                if (delta > 0 && credit >= delta) {
                    credit = 0;
                    baseSol = newSol;
                }
            }
        }

        nIter++;

        bestSol.updateCosts();
        initSol.updateCosts();

        // 3. Set output
        printResults(inputs, bestSol, nIter);

        //4. Save results in a file
        writeResultsToFile(bestSol, baseSol, nIter, test.getMaxIter() > 0 ? "A" : "B");

        return bestSol;
    }

    private Solution getInitialRandSol(Facility[] candidates) {

        // Since closing is fast, start by opening a lot of facilities
        // Biased-randomized selection process using a Geometric(beta)
        // promotes the random selection of those facilities with HIGHEST
        // density levels
        return new Solution(inputs,
                this.randomness.calcPositionsArrayLimited(
                        randomness.getRandomPositionFromInit((int) (candidates.length / 2),
                                candidates.length), candidates.length),
                candidates);

    }
```

```
public Solution localSearch(Solution baseSol, int type) {
    return localSearch(baseSol, type, test.getMaxTime());
}

private Solution localSearch(Solution baseSol, int type, int maxTime) {
    long startTime = ElapsedTime.systemTime();
    if (maxTime == 0) {
        maxTime = 20;
    }
    if (type > 0) {
        return localSearchA(baseSol);
    }
    return localSearchB(baseSol, startTime, maxTime);
}

private Solution localSearchA(Solution baseSol) {
    Solution sol = baseSol.clone();

    final Facility [] openFacilities = sol.getOpenFacilitiesSortedArray();
    final int [] posArray = randomness.calcPositionsArray(openFacilities.length);

    Solution newSol = sol.clone();
    int minCost = sol.getTotalCosts();
    for (int i = 0; i < openFacilities.length; i++) {
        // We cannot obtain a solution with 0 open facilities .
        if (newSol.getOpenFacilities().size() > 1) {

            newSol.removeOpenFacility(openFacilities[posArray[i]]);

            if (newSol.getTotalCosts() < minCost) {
                minCost = newSol.getTotalCosts();
            } else {
                newSol.addOpenFacilityNotUpdatingCosts(openFacilities[posArray[i]]);
            }
        }
    }
    sol = newSol;
    sol.updateCosts();

    return sol;
}

private Solution localSearchB(Solution baseSol, long maxTime, long startTime) {
    Solution sol = baseSol;
    boolean improvement = true;

    do {

        if (ElapsedTime.calcSeconds(ElapsedTime.calcElapsed(startTime,
                ElapsedTime.systemTime())) >= maxTime) {
            break;
        }

        improvement = false;

        // Open one single facility until an improvement is reached
        Facility [] closedFacilities = sol.getClosedFacilitiesSortedArray();
        int [] posArray = randomness.calcPositionsArray(closedFacilities.length);

        Solution newSol = sol.clone();
        int minCost = sol.getTotalCosts();
        int initialCost = minCost;
        int noImprovements = 0;

        for (int i = 0; i < closedFacilities.length; i++) {

            if (ElapsedTime.calcSeconds(ElapsedTime.calcElapsed(
                    startTime, ElapsedTime.systemTime())) >= maxTime) {
                break;
            }

            newSol.addOpenFacilityNotUpdatingCosts(closedFacilities[posArray[i]]);
            newSol.updateCosts();
            if (newSol.getTotalCosts() < minCost) {
                minCost = newSol.getTotalCosts();
```

```
                noImprovements = 0;
            } else {
                noImprovements++;
                if (i == (closedFacilities.length - 1) || noImprovements >= 10) {
                    newSol.removeOpenFacility(closedFacilities[posArray[i]]);
                    break;
                } else {
                    newSol.removeOpenFacilityNotUpdatingCosts(closedFacilities[posArray[i]]);
                }
            }
        }
        sol = newSol;

        // Swap an open facility with a closed one until improvement is reached
        Facility[] openFacilities = sol.getOpenFacilitiesSortedArray();
        posArray = randomness.calcPositionsArray(openFacilities.length);
        for (int i = 0; i < openFacilities.length; i++) {

            if (ElapsedTime.calcSeconds(ElapsedTime.calcElapsed(
                    startTime, ElapsedTime.systemTime())) >= maxTime) {
                break;
            }

            final Facility oFacility = openFacilities[posArray[i]];
            newSol = sol.clone();
            int minCost2 = newSol.getTotalCosts();

            newSol.removeOpenFacility(oFacility);

            for (Facility cFacility : sol.getClosedFacilities()) {

                newSol.addOpenFacility(cFacility);

                if (newSol.compareTo(sol) < 0) {
                    sol = newSol;
                    break;
                } else {
                    newSol.removeOpenFacilityNotUpdatingCosts(cFacility);
                }
            }
            if (newSol.getTotalCosts() < minCost2) {
                minCost2 = newSol.getTotalCosts();
                break;
            } else {
                newSol.addOpenFacilityNotUpdatingCosts(oFacility);
            }
        }

        // Close one single facility until an improvement is reached
        openFacilities = sol.getOpenFacilitiesSortedArray();
        posArray = randomness.calcPositionsArray(openFacilities.length);
        newSol = sol.clone();
        minCost = sol.getTotalCosts();
        noImprovements = 0;

        for (int i = 0; i < openFacilities.length; i++) {
            final double secondsElapsed = ElapsedTime.calcSeconds(ElapsedTime.calcElapsed(startTime,
ElapsedTime.systemTime()));
            if (secondsElapsed >= maxTime) {
                break;
            }

            // We cannot obtain a facility with 0 open facilities.
            if (newSol.getOpenFacilities().size() > 1) {
                newSol.removeOpenFacility(openFacilities[posArray[i]]);

                if (newSol.getTotalCosts() < minCost) {
                    minCost = newSol.getTotalCosts();
                    noImprovements = 0;
                } else {
                    noImprovements++;
                    if (i == (openFacilities.length - 1) || noImprovements >= 10) {
                        newSol.addOpenFacility(openFacilities[posArray[i]]);
                        break;
                    } else {
```

```
                        newSol.addOpenFacilityNotUpdatingCosts(openFacilities[posArray[i]]);
                    }
                }
            }
        }
        sol = newSol;

        if (sol.getTotalCosts() < initialCost) {
            improvement = true;
        }
    } while (improvement);

    return sol;
}

public Solution perturbate(final Solution sol) {
    final List<Facility> openFacilities = sol.getOpenFacilities();
    final int nFacilToClose = randomness.getRandomPosition(openFacilities.size());

    // Biased-randomized selection process using a Geometric(beta)
    int [] posArray = randomness.calcPositionsArrayLimited(nFacilToClose, openFacilities.size());
    Facility [] facilToClose = new Facility[nFacilToClose];
    for (int i = 0; i < nFacilToClose; i++) {
        facilToClose[i] = openFacilities.get(openFacilities.size() - posArray[i] - 1);
    }
    sol.removeMultipleOpenFacilityNotUpdatingCosts(facilToClose);
    final List<Facility> closedFacilities = sol.getClosedFacilities();

    // Re-construct more than destructed
    int nFacilToOpen = randomness.getRandomPositionUniformFromInit(
            nFacilToClose, closedFacilities.size());

    // Biased-randomized selection process using a Geometric(beta)
    posArray = randomness.calcPositionsArrayLimited(nFacilToOpen, closedFacilities.size());
    Facility [] facilToOpen = new Facility[nFacilToOpen];
    for (int i = 0; i < nFacilToOpen; i++) {
        facilToOpen[i] = closedFacilities.get(posArray[i]);
    }
    sol.addMultipleOpenFacilityNotUpdatingCosts(facilToOpen);

    sol.updateCosts();

    return sol;
}

private void printResults(final Input inputs, final Solution bestSolution, final int numIter) {
    final double error = (inputs.getOptimalSolutionCost() - bestSolution.getTotalCosts()) * 100 / (double)
  inputs.getOptimalSolutionCost();

    System.out.println("GAP = " + error + " %");
    System.out.println("OPTIMAL SOLUTION COST = " + inputs.getOptimalSolutionCost());
    System.out.println("OBTAINED BEST SOLUTION = " + bestSolution.getTotalCosts());
    System.out.println("NUMBER OF ITERATIONS = " + numIter);

    System.out.println("TIME EXPENDED = " + bestSolution.getTime() + "s");
}

private void writeResultsToFile(final Solution bestSol, final Solution initSol, final int numIter, final String
  AlgName) {
    stats.WriteTestResult(this.test.getInstanceName(),
            AlgName,
            this.inputs.getOptimalSolutionCost(),
            this.inputs.getOptimalSolutionOpenFacilities(),
            bestSol.getTime(), initSol.getAssignmentCosts(),
            initSol.getOpenningCosts(), initSol.getTotalCosts(),
            bestSol.getAssignmentCosts(), bestSol.getOpenningCosts(),
            bestSol.getTotalCosts(), bestSol.getOpenFacilitiesByIds(),
            numIter);
}
}
```

```
import java.util.Random;
```

```
public class Randomness {

    private final Test test;
    private final float firstParam;
    private final float secondParam;

    public Randomness(Test test) {
        this.test = test;
        this.firstParam = test.getFirstParam();
        this.secondParam = test.getSecondParam();
    }

    public int[] calcPositionsArrayLimited(int limit, int length) {
        int[] posArray = new int[limit];
        int[] auxArray = new int[length]; // pointers to edges in savingsList

        // Reset auxArray
        for (int i = 0; i < length; i++) {
            auxArray[i] = i;
        }

        // Assign new random positions
        for (int i = 0; i < limit; i++) {
            int pos;

            pos = getRandomPositionUniform(length - i);

            posArray[i] = auxArray[pos];
            for (int j = pos; j < length - i - 1; j++) {
                auxArray[j] = auxArray[j + 1];
            }
        }
        return posArray;
    }

    public int[] calcPositionsArray(int length) {

        int[] posArray = new int[length];
        int[] auxArray = new int[length]; // pointers to edges in savingsList

        // Reset auxArray
        for (int i = 0; i < length; i++) {
            auxArray[i] = i;
        }

        // Assign new random positions
        for (int i = 0; i < length; i++) {
            int pos;

            pos = getRandomPositionUniform(length - i);

            posArray[i] = auxArray[pos];
            for (int j = pos; j < length - i - 1; j++) {
                auxArray[j] = auxArray[j + 1];
            }
        }
        return posArray;
    }

    //geometrical distribution.
    public int getRandomPosition(double beta, int n, Random r) {
        return ((int) (Math.log(r.nextDouble()) / Math.log(1 - beta))) % n;
    }

    public int getRandomPosition(int n) {

        return getRandomPosition((float) firstParam
                + test.getRand().nextDouble() * ((float) secondParam - (float) firstParam), n,
                test.getRand());
    }

    public int getRandomPositionFromInit(int init, int n) {
        return getRandomPosition(n - init) + init;
    }
```

```
    public int getRandomPositionUniform(int n) {
        return test.getRand().nextInt(n);
    }

    public int getRandomPositionUniformFromInit(int init, int n) {
        return getRandomPosition(n - init) + init;
    }

}
```

```
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Comparator;
import java.util.HashSet;
import java.util.List;

public final class Solution implements Comparable<Solution> {

    private static int nInstances = 0; // Number of solutions generated.
    private final int id = nInstances++; // inctremental identifier.
    private int openningCosts; // Solution total openning costs.
    private int assignmentCosts; // Solution assignment costs.
    private List<Facility> openFacilities; // Array of the facilities in the solution.
    private List<Facility> closedFacilities; // Array of the closed facilites.
    private long time; // Elapsed computational time (in seconds)
    private Input input;
    private HashSet<Integer> checkList;

    public Solution() {
        checkList = new HashSet<>();
    }

    public Solution(Input input) {
        this();
        this.openningCosts = 0;
        this.assignmentCosts = 0;
        this.closedFacilities = new ArrayList<>(Arrays.asList(input.getFacilities()));
        this.openFacilities = new ArrayList<>();
        this.time = 0;
        this.input = input;
    }

    public Solution(Input input, List<Facility> subPartA, List<Facility> subPartB) {
        this();
        this.openningCosts = 0;
        this.assignmentCosts = 0;
        this.closedFacilities = new ArrayList<>();

        for (Facility f : subPartA) {
            closedFacilities.add(f);
        }
        for (Facility f : subPartB) {
            closedFacilities.add(f);
        }

        this.openFacilities = new ArrayList<>();
        this.time = 0;
        this.input = input;
    }

    public void addSkippedCustomer(int customer) {
        checkList.add(customer);
    }

    public Solution(Input input, int[] toOpen, Facility[] candidates) {
        checkList = new HashSet<>();
        openningCosts = 0;
        assignmentCosts = 0;
        closedFacilities = new ArrayList<>();
        closedFacilities.addAll(Arrays.asList(candidates));
```

```
        openFacilities = new ArrayList<>();
        time = 0;
        this.input = input;

        for (int i = 0; i < toOpen.length; i++) {
            Facility facToAdd = candidates[toOpen[i]];
            openFacilities.add(facToAdd);
            closedFacilities.remove(facToAdd);
        }
        updateCosts();
    }

    public Solution(Input input, int[] toOpen) {
        this(input);

        for (int i = 0; i < toOpen.length; i++) {
            Facility facToAdd = this.input.getFacilities()[toOpen[i]];
            openFacilities.add(facToAdd);
            closedFacilities.remove(facToAdd);
        }
        this.updateCosts();
    }

    /* SET METHODS */
    public void setAssignmentCost(int cost) {
        assignmentCosts = cost;
    }

    public void setOpenningCost(int cost) {
        openningCosts = cost;
    }

    public void setTime(long t) {
        time = t;
    }

    /* GET METHODS */
    public int getTotalCosts() {
        return assignmentCosts + openningCosts;
    }

    public int getAssignmentCosts() {
        return assignmentCosts;
    }

    public int getOpenningCosts() {
        return openningCosts;
    }

    public int getId() {
        return id;
    }

    public List<Facility> getOpenFacilities() {
        return openFacilities;
    }

    public List<Facility> getClosedFacilities() {
        return closedFacilities;
    }

    public long getTime() {
        return time;
    }

    // Returns the sorted open facilities on the solution.
    public Facility[] getOpenFacilitiesSortedArray() {
        Facility[] facilityArray = new Facility[this.openFacilities.size()];
        return this.openFacilities.toArray(facilityArray);
    }

    // Returns the sorted closed facilites on the solution.
    public Facility[] getClosedFacilitiesSortedArray() {
        Facility[] facilityArray = new Facility[this.closedFacilities.size()];
        return this.closedFacilities.toArray(facilityArray);
```

```
}

public final static Comparator<Facility> SORT_FACILITY_BY_ID = new Comparator<Facility>() {
    @Override
    public int compare(Facility o1, Facility o2) {
        return o1.facilityId - o2.facilityId;
    }
};

// Returns the open facilities sorted by id.
public Facility [] getOpenFacilitiesByIds() {
    Facility [] result = new Facility[this.openFacilities.size()];
    this.openFacilities.toArray(result);
    Arrays.sort(result, SORT_FACILITY_BY_ID);
    return result;
}

// Updates the solution costs.
public void updateCosts() {
    if (checkList.isEmpty()) {
        this.assignmentCosts = 0;
        this.openningCosts = 0;
        int [][] costMatrix = new int[this.openFacilities.size()][];
        int i = 0;
        for (Facility f : this.openFacilities) {
            this.openningCosts += f.openCost;
            costMatrix[i] = f.assignmentCosts;
            i++;
        }

        for (i = 0; i < costMatrix[0].length; i++) {
            int minCost = costMatrix[0][i];
            for (int j = 1; j < this.openFacilities.size(); j++) {
                if (costMatrix[j][i] < minCost) {
                    minCost = costMatrix[j][i];
                    if (minCost == this.input.minAssignmentCosts[i]) {
                        break;
                    }
                }
            }
            this.assignmentCosts += minCost;
        }
    } else {
        updateCostForSubProblem();
    }
}

public void updateCostForSubProblem() {

    this.assignmentCosts = 0;
    this.openningCosts = 0;
    int [][] costMatrix = new int[this.openFacilities.size()][];
    int i = 0;
    for (Facility f : this.openFacilities) {
        this.openningCosts += f.openCost;
        costMatrix[i] = f.assignmentCosts;
        i++;
    }

    if (i > 0) {
        for (int ii : checkList) {
            int minCost = costMatrix[0][ii];
            for (int j = 1; j < this.openFacilities.size(); j++) {
                if (costMatrix[j][ii] < minCost) {
                    minCost = costMatrix[j][ii];
                    if (minCost == this.input.minAssignmentCosts[ii]) {
                        break;
                    }
                }
            }
            this.assignmentCosts += minCost;

        }
    }
}
```

```
// Adds a open facility to the solution.
public void addOpenFacility(Facility f) {
    addOpenFacilityNotUpdatingCosts(f);
    updateCosts();
}

// Adds a open facility to the solution.
public void addOpenFacilityNotUpdatingCosts(Facility f) {
    openFacilities.add(f);
    closedFacilities.remove(f);
}

// Adds multiple open facilities to the solution.
public void addMultipleOpenFacilityNotUpdatingCosts(Facility[] facilities) {
    for (Facility f : facilities) {
        this.addOpenFacilityNotUpdatingCosts(f);
    }
}

// Adds multiple open facilities to the solution.
private void cloneSolution(List<Facility> facilities, int assignmnetCost, int openningCost) {
    // The list is already ordered, so we can directly add.
    for (Facility f : facilities) {
        this.openFacilities.add(f);
        this.closedFacilities.remove(f);
    }

    this.assignmentCosts = assignmnetCost;
    this.openningCosts = openningCost;
}

// Removes a open facility from the solution.
public void removeOpenFacility(Facility f) {
    removeOpenFacilityNotUpdatingCosts(f);
    updateCosts();
}

// Removes a open facility from the solution.
public void removeOpenFacilityNotUpdatingCosts(Facility f) {
    openFacilities.remove(f);
    closedFacilities.add(f);
}

// Removes multiple open facilities.
public void removeMultipleOpenFacilityNotUpdatingCosts(Facility[] facilities) {
    for (Facility f : facilities) {
        removeOpenFacilityNotUpdatingCosts(f);
    }
}

public static void resetIds() {
    nInstances = 0;
}

@Override
public Solution clone() {
    Solution cloneSol = new Solution(input, getOpenFacilities(), getClosedFacilities());
    cloneSol.cloneSolution(openFacilities, assignmentCosts, openningCosts);
    cloneSol.setTime(getTime());

    for (int ele : checkList) {
        cloneSol.checkList.add(ele);
    }

    return cloneSol;
}

@Override
public String toString() {

    String s = "\r\n";
    s += "Sol ID : " + this.getId() + "\r\n";
    s += "Sol costs: " + this.getTotalCosts() + "\r\n";
    for (Facility f : this.openFacilities) {
```

```
            s += f.toString();
        }
        return s;
    }

    @Override
    public int compareTo(Solution other) {
        return this.getTotalCosts() - other.getTotalCosts();
    }

    public void sendToFile(String outFile) {
        try {
            try (PrintWriter out = new PrintWriter(outFile)) {
                out.println("*************************************************");
                out.println("*                    OUTPUTS                    *");
                out.println("*************************************************");
                out.println("\r\n");
                out.println("-------------------------------------------------");
                out.println("Solution");
                out.println("-------------------------------------------------");
                out.println(this.toString() + "\r\n");
                out.println("\n\nFound in " + time + "s.");
            }
        } catch (IOException exception) {
            System.out.println("Error processing output file: " + exception);
        }
    }
}
```

```
import java.util.Arrays;

public class Facility implements Comparable<Facility> {

    public final int facilityId; // The facility identifier.
    public final int openCost; // The facility openning costs.
    public final int[] assignmentCosts; // The assignment costs of every customer to the facility.
    public final int facilityCost; // The total facility costs.

    public Facility(int id, int openCost, int[] assignmentCosts) {
        this.facilityId = id;
        this.openCost = openCost;
        this.assignmentCosts = assignmentCosts;

        //Total facility cost = openCost + sum of assignment cost to every customer.
        int tmpCost = openCost;

        final int[] temp = (int[]) assignmentCosts.clone();
        Arrays.sort(temp);

        for (int i = 0; i < temp.length / 4; i++) {
            tmpCost += temp[i];
        }
        this.facilityCost = tmpCost;
    }

    // The compareTo method for sorting purposes.
    @Override
    public int compareTo(Facility other) {
        return this.facilityCost - other.facilityCost;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
```

```
        final Facility other = (Facility) obj;

        if (!Arrays.equals(assignmentCosts, other.assignmentCosts)) {
            return false;
        }
        if (facilityCost != other.facilityCost) {
            return false;
        }
        if (facilityId != other.facilityId) {
            return false;
        }
        if (openCost != other.openCost) {
            return false;
        }
        return true;
    }
}
```

Bibliografía

[1] J.F. Stollsteimer. The effect of technical change and output expansion on the optimum number, size and location of pear marketing facilities in a california pear producing region. *Ph.D. dissertation. University of California at Berkeley.*, 1961.

[2] M.L. Balinski. On finding integer solutions to linear programs. *Proceedings of the IBM Scientific Computing Symposium on Combinatorial Problems*, 1966.

[3] V. Verter. Uncapacitated and capacitated facility location problems. *Principles of Location Science, Eiselt H.A., and Marianov V.*, 2011.

[4] Nemhauser G.L. Cornuejols, G. and L.A. Wolsey. The uncapacitated facility location problem. *Mirchandani, P.B., and Francis, R.L.*, 1990.

[5] M.A. Efroymson and T.L. Ray. A branch and bound algorithm for plant location. *Operation Research*, 1966.

[6] K. Spielberg. Algorithms for the simple plant location problem with some side considerations. *Operations Research*, 1969.

[7] D. Erlenkotter. A dual-based procedure for uncapacitated location. *Operation Research*, 1978.

[8] M. Körkel. On the exact solution of large-scale simple plant location problems. *European Journal of Operational Research*, 1989.

[9] D. Hochbaum. Approximation algorithms for the weighted set covering and node cover problems. *SIAM Journal of Computing*, 1982.

[10] Tardos E. Shmoys, D.B. and K.I. Aardal. Approximation algorithms for facility location problems. *Proceedings of the 29th Annual ACM Sumposium on Theory of Computing*, 1997.

[11] F.A Chudak. Improved approximation algorithms for uncapacitated facility location. *Integer Programming and Combinatorial Optimization*, 1998.

[12] K. Jain and V.V. Vazirani. Primal-dual approximation algorithms for metric facility location and k-median problems. *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamos, California*, 1999.

[13] Mahdina M. Markakis-E. Saberi A. Jain, K. and V. Vazirani. Greedy facility location algorithms analyzed using dual fitting with factor-revealing lp. *Journal of the ACM,*, 2003.

[14] A.A. Kuehn and M.J. Hamburger. A heuristic program for locating warehouses. *Management Science*, 1963.

[15] M.L. Alves and M.T. Almeida. Simulated annealing algorithm for the simple plant location problem. *Revista Investigação Operacional*, 1992.

[16] Tosic D. Filipovic-V. Kratica, J. and I. Ljubic. Solving the simple plant location problem by genetic algorithm. *RAIRO Operations Research*, 2001.

[17] D. Ghosh. Neighborhood search heuristics for the uncapacitated facility location problem. *European Journal of Operational Research*, 2003.

[18] L. Michel and P. Van Hentenryck. A simple tabu search for warehouse location. *European Journal on Operations Research*, 2003.

[19] M.G. Resende and R.F. (Werneck. A hybrid multistart heuristic for the uncapacitated facility location problem. *European Journal of Operational Research*, 2006.

[20] L. Cooper. Location-allocation problems. *Operations Research*, 1963.

[21] J.G. Klincewicz and H. Luss. A dual based algorithm for multiproduct uncapacitated facility location. *Transportation Science*, 1987.

[22] A. Meyerson. Online facility location. *FOCS '01 Proceedings of the 42nd IEEE symposium on Foundations of Computer Science*, 2001.

[23] Ushakov A. Carrizosa, E. and I. Vasilyev. A computational study of nonlinear minsum facility location problem. *Computers Operations Research*, 2012.

[24] L.V. Snyder. Facility location under uncertainty. *IIE Transactions*, 2006.

[25] D. Karger and M. Minkoff. Building steiner trees with incomplete global knowledge. *Foundations of Computer Science*, 2000.

[26] F. Thouin and M. Coates. Equipment allocation in video on demand network deployments. *ACM Transaction on Multimedia Computing, Communications and Applications*, 2008.

[27] Murray A. T. Lee, G. Maximal covering with network survivability requirements in wireless mesh networks. *Environment and Urban Systems*, 2010.

[28] Stanimirovic Z. Maric, M. and S. Bozovic. Hybrid metaheuristic method for determining locations for long-term health care facilities. *Annals of Operations Research*, 2013.

[29] Martin O. Lourenço, H.R. and T. Stützle. Iterated local search: framework and applications. *Handbook of Metaheuristics*, 2010.

[30] Resende M.G.C. Feo, T.A. Greedy randomized adaptive search procedures. *Journal of Global Optimization s*, 1995.

[31] Sergio González-Martín Daniel Riera Barry B. Barrios Angel A. Juan, José Cáceres-Cruz. Biased randomization of classical heuristics. 2014.

[32] Faulin J. Jorba-J. Riera D.-Masip D. Juan, A.A. and B. Barrios. On the use of monte carlo simulation, cache and splitting techniques to improve the clarke and wright savings heuristic. *Journal of the Operational Research Society*, 2011.

[33] Juan A.A.-Riera D. Castella-Q. Perez-Bonilla A. Gonzalez-Martin, S. and R. Muñoz. Development and assesment of the sharp and randsharp algorithms for the arc routing problem. *AI Communications*, 2012a.

[34] M Hoefer. Ufllib benchmarks, Aug 2014. URL http://www.mpi-inf.mpg.de/departments/d1/projects/benchmarks/UflLib/.

[35] O. Bilde and J.krarup. Sharp lower bound and efficient algorithms for the simple plant location problem. *Annals of Discrete Mathematics*, I, 1977.

[36] Y. Kochetov and D. Ivanenko. Computationally difficult instances for the uncapacitated facility location problem. *Proceedings of the 5th Metaheuristics International Conference (MIC)*, 2003.

[37] F. Barahona and F. Chudak. Near-optimal solutions to large scale facility location problem. *Technical Report RC21606, IBM, Yorktown Heights, NY, USA.*, 1999.

[38] mpi inf.mpg.de. Ufllib benchmarks data format, Aug 2014. URL http://resources.mpi-inf.mpg.de/departments/d1/projects/benchmarks/UflLib/data-format.html.