Universidad de Valladolid

FACULTAD DE CIENCIAS

TRABAJO FIN DE GRADO

Grado en Estadística

Aplicación de metaheurísticas al problema del p-Centro

Autor: David Morchón Payo

Tutores: Jesús Alberto Tapia García
Jesús Sáez Aguado

Año 2024

Resumen

En este trabajo de fin de grado se emplearán una serie de metaheurísticas para encontrar una solución aproximada al problema del p-centro. Estas soluciones se compararán con la solución óptima. También se compararán las metaheurísticas entre sí. Estas comparaciones se realizarán tanto a nivel de calidad de la solución como a eficiencia computacional.

El objetivo es poder obtener soluciones al problema de localización p-centro cercanas a la óptima en tiempos de ejecución mucho menores que el tiempo empleado por el modelo que usa la formulación clásica del problema.

Se emplean ficheros de datos de tamaños muy diversos para mayor calidad de la comparación. Así se podrá observar cómo se comporta cada algoritmo según vaya creciendo el número de observaciones.

Abstract

In this dissertation, a series of metaheuristics to the p-center problem will be used and compared with the optimal solution and other classical algorithms such as greedy. These comparisons are made both in terms of solution quality and computational efficiency.

The objective is to be able to obtain near-optimal solutions in much shorter execution times than the time used by the model using the classical formulation of the problem.

Data files of very different sizes will be used to improve the quality of the comparison. In this way it will be possible to observe how each algorithm behaves as the number of observations increases.

Índice general

Capítulo 1

Introducción

1.1. Descripción del problema

La Investigación Operativa (IO) es la rama de las matemáticas que estudia la resolución de problemas y la toma de mejores decisiones mediante la aplicación de métodos analíticos avanzados.

Dentro de este campo se encuentra el problema del p-Centro. En 1964, Hakimi [8] introduce el problema del centro absoluto para ubicar una estación de policía o un hospital tal que la distancia máxima de la estación a una serie de localizaciones conectadas por carretera sea mínima. Desde un punto de vista de grafos, el punto solución se conoce como *centro absoluto* del grafo, y la distancia máxima obtenida es el *radio absoluto*. En 1965, Hakimi [9] menciona la generalización del problema del centro absoluto al problema del p-centro. Este problema consiste en ubicar como mucho p instalaciones entre un conjunto de posibles localizaciones y asignar cada cliente a una instalación con el objetivo de minimizar la distancia máxima entre cualquier cliente y la instalación a la que está asignado. Hay muchas variaciones del problema, pero en este trabajo se estudiará el problema con coste de apertura de instalación unitario y arcos entre nodos con distintos pesos.

La característica más destacable de este problema es que trata de optimizar un criterio *minmax*. A diferencia de la p-Mediana, que busca minimizar la suma de tiempos, el problema del p-Centro lo que busca es minimizar el tiempo máximo. Debido a esto, se suele decir que el problema de la p-Mediana es un problema orientado hacia la eficiencia mientras que el p-Centro está orientado hacia la equidad.

Históricamente, la búsqueda de soluciones exactas para el problema del p-Centro ha sido un desafío computacionalmente costoso. La solución exacta del p-Centro implica examinar todas las combinaciones posibles de ubicaciones de instalaciones para determinar la configuración que minimiza la distancia máxima entre las instalaciones y los clientes. Sin embargo, esta búsqueda exhaustiva se vuelve rápidamente inviable a medida que crece el tamaño del problema y el número de ubicaciones potenciales. A lo largo de los años, los investigadores han explorado diversas técnicas para abordar este desafío, incluidas las formulaciones matemáticas exactas y los algoritmos de optimización combinatoria. Sin embargo, debido a la complejidad inherente del

problema y su naturaleza combinatoria, encontrar soluciones exactas para instancias de grandes dimensiones sigue siendo un objetivo difícil de alcanzar en un tiempo razonable. Es por ello que las heurísticas y metaheurísticas desempeñan un papel crucial en la resolución de problemas complejos como el del p-Centro.

A medida que aumenta la complejidad del problema, encontrar soluciones óptimas se vuelve cada vez más difícil, y en muchos casos, es simplemente impracticable debido al tiempo y los recursos necesarios. Es aquí donde entran en juego las heurísticas y metaheurísticas [6]. Estas técnicas proporcionan métodos eficientes y efectivos para encontrar soluciones cercanas a óptimas en un tiempo razonable. Las heurísticas ofrecen enfoques intuitivos que aprovechan la estructura del problema para guiar la búsqueda hacia soluciones aceptables, mientras que las metaheurísticas van más allá, utilizando estrategias de búsqueda más sofisticadas y adaptativas para explorar el espacio de soluciones de manera más exhaustiva y eficiente. En el contexto del problema del p-Centro, donde la búsqueda de la ubicación óptima de instalaciones debe equilibrarse con la eficiencia computacional, las heurísticas y metaheurísticas proporcionan herramientas indispensables para obtener soluciones de alta calidad en un tiempo razonable.

En el marco de este estudio, se lleva a cabo una comparación entre las soluciones exactas y las heurísticas y metaheurísticas. Este enfoque comparativo busca arrojar luz sobre las fortalezas y limitaciones de cada tipo de método de resolución en el contexto específico del problema del p-Centro.

1.2. Estructura del trabajo

Este trabajo aborda del problema del p-centro (también conocido como p-Center o k-Center problem) desde un enfoque de programación entera. En el capítulo 2 se formula el problema y se exponen sus aplicaciones. En el capítulo 3 se explica su relación con el problema del conjunto de cobertura (SCP), la formulación de SCP y la posible solución del p-centro a partir de este. En el capítulo 4 se exponen mejoras al método original explicado en el capítulo 3 con las que se obtienen tiempos de computación mucho más bajos, especialmente en 4.3 y 4.4. Hasta el capítulo 4 se presentan soluciones exactas, que garantizan alcanzar la solución óptima pero, debido a las características del problema, solo lo conseguirán en un tiempo razonable para problemas de dimensiones pequeñas. En la mayoría de ocasiones es necesario recurrir a soluciones aproximadas que, no garantizan la solución óptima, pero obtienen aproximaciones en tiempos razonables. Con este objetivo, en el capítulo 5 se presentan las heurísticas básicas de aproximación voraces o *greedy*. A partir del capítulo 6, con el objetivo de mejorar las soluciones obtenidas mediante heurísticas de construcción, se emplean las heurísticas y metaheurísticas de intercambio, siendo estas las más populares y que mejores resultados brindan para el problema del p-centro.

1.3. Metodología de experimentación

La experimentación se realizo en una computadora con procesador Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz, 8 GB de RAM y sistema operativo Windows 11 x64. Tanto los métodos de resolución exacta como los métodos de resolución aproximada están programados en *mosel* y las experimentaciones se realizaron en

el *Workbench* con la versión del *solver*: FICO Xpress Solver 64bit v9.2.5. Para cada problema a resolver mediante un método no determinista con un fichero de datos dado, el método se ejecuta de 10 a 20 veces en función de la complejidad del mismo. Se guarda el mejor de todos los resultados obtenidos. El *script* encargado del lanzamiento de los métodos está programado en Python.

El enlace al repositorio `GitLab` donde se encuentran todos los ficheros *mosel* desarrollados se encuentra en el Apéndice A.

1.4. Ficheros empleados

Todos los ficheros empleados para la experimentación provienen de la misma fuente: *OR-Library*. Se han utilizado 20 ficheros, desde pmed1.txt (100 puntos de servicio coincidentes con los puntos de demanda) hasta pmed20.txt (400 puntos de servicio coincidentes con los puntos de demanda) para experimentar con algoritmos heurísticos y metaheurísticos. En el caso de los algoritmos más potentes, basados en búsqueda binaria, se han empleado los 40 archivos de *OR-Library*: desde pmed1.txt hasta pmed40.txt. En todos los casos los puntos pueden son tanto de servicio como de demanda, el coste de cubrimiento viene dado por la distancia entre dos puntos. El grafo formado por todos los puntos es conexo, no hay ningún punto aislado, pero no todos los puntos están conectados entre sí, por lo que, para calcular la matriz de distancias se ha aplicado el algoritmo de Floyd-Warshall.

1.5. Exposición de resultados

La exposición de los resultados de cada método se realizará mediante tablas en las que se indicarán las siguientes características:

- Datos: Nombre del fichero de datos.
- n: Número de puntos de servicio/demanda del problema.
- p: Número de instalaciones a abrir.
- Óptimo: Valor objetivo de la solución óptima.
- Método: Nombre del método empleado para obtener una solución.
- Solución: Valor objetivo obtenido mediante el método empleado.
- gap: Diferencia porcentual entre el valor objetivo óptimo y el obtenido mediante el método.

Capítulo 2

El problema del p-Centro

2.1. Formulación

Sea M $= \{1, 2, \ldots, m\}$ el conjunto de los puntos de demanda, N $= \{1, 2, \ldots, n\}$ el conjunto de los puntos donde se puede ubicar una instalación y p el número fijo de puntos de servicio que se deben elegir. Se tiene un problema de localización-asignación que clásicamente se formula con las siguientes variables:

- La variable de localización $x_j \in \{0, 1\}$ donde $x_j = 1$ si y sólo si se sitúa una instalación en el punto j, $x_j = 0$ en caso contrario.
- La variable de asignación $y_{i,j} \in \{0, 1\}$ donde $y_{i,j} = 1$ si y sólo si el punto i de demanda es servido por el punto j.
- El tiempo de respuesta máximo w, que es la variable que se busca minimizar. Se considera $w \geq 0$.

La matriz de distancias $t_{i,j}$ indica la distancia o tiempo entre el punto i y el punto j, $\forall i \in \{1, 2, \ldots, m\}, \forall j \in \{1, 2, \ldots, n\}$

Considerando esta notación, se puede formular el problema:

$$\text{Minimizar} \qquad \max_{\substack{i=1,\ldots,m \\ j=1,\ldots,n}} t_{ij} y_{ij} \tag{1.2.1}$$

$$\text{sujeto a} \qquad \sum_{j=1}^{n} y_{i,j} = 1 \qquad \forall i \in \{1, 2, \ldots, m\} \tag{1.2.2}$$

$$y_{i,j} \leq x_j \qquad \forall i \in \{1, 2, \ldots, m\}, \forall j \in \{1, 2, \ldots, n\} \tag{1.2.3}$$

$$\sum_{j=1}^{n} x_j = p \tag{1.2.4}$$

$$x_j \in \{0,1\} \qquad \forall j \in \{1,2,\ldots,n\} \tag{1.2.5}$$

$$y_{i,j} \in \{0,1\} \quad \forall i \in \{1,2,\ldots,m\}, \forall j \in \{1,2,\ldots,n\} \tag{1.2.6}$$

Se intenta minimizar el tiempo de respuesta máximo entre un punto de demanda y la instalación a la que se le asigna. La función que se intenta minimizar no es lineal al tratarse de un máximo, por lo que para linealizarla, introducimos la variable w anteriormente mencionada, esta representa el tiempo máximo de respuesta.

$$\text{Minimizar} \quad \text{w} \tag{1.2.7}$$

$$\text{sujeto a} \qquad \sum_{j=1}^{n} t_{i,j} y_{i,j} \leq w \qquad \forall i \in \{1,2,\ldots,m\} \tag{1.2.8}$$

(1.2.2),(1.2.3),(1.2.4),(1.2.5),(1.2.6)

Se mantienen el resto de restricciones y se añade la restricción (1.2.8) para así poder minimizar la función objetivo (1.2.7) tratándola como una función lineal. La restricción (1.2.2) obliga a que cada cliente esté servido por un solo punto de servicio, mientras que la restricción (1.2.3) se asegura de que un un cliente pueda estar servido por un punto si y solo si ese punto hay una instalación. La restricción (1.2.4) se encarga de que se abran exactamente p instalaciones de entre los N posibles puntos.

Como se verá más adelante, esta formulación clásica, si bien es correcta, es muy poco eficiente a nivel computacional por causa de el elevado número de variables necesarias en algunos casos. Por ejemplo, en caso de tener n = 300 posibles puntos de instalación y m = 300 puntos de demanda, se requerirán $n * m + m = 90300$ variables. En 1979, Kariv y Hakimi [11] demostraron que el problema es NP-hard. Es muy difícil obtener soluciones óptimas, el tamaño del problema excede rápidamente la capacidad de computación de ordenadores actuales. Para conseguir buenas aproximaciones de las soluciones se recurre a las heurísticas, especialmente heurísticas de intercambio en el caso del p-centro.

2.2. Aplicaciones

La equidad es la característica que hace que el p-Centro sea fundamental en problemas de urgencia en los que interesa que se llegue a cualquier punto en el mínimo tiempo posible, sin dejar puntos aislados a mucha distancia de algún punto de servicio.

En general, se suele emplear en problemas de campos muy diversos:

1. Logística y Distribución: En logística, el problema del p-centro se utiliza para determinar la ubicación óptima de centros de distribución o almacenes con

el fin de minimizar los costos de transporte y maximizar la eficiencia en la distribución de bienes.

2. Telecomunicaciones: En telecomunicaciones, el problema del p-centro puede aplicarse para determinar la ubicación óptima de estaciones base o centros de conmutación con el objetivo de maximizar la cobertura de red y minimizar el costo de infraestructura.

3. Servicios de Emergencia: En la planificación de servicios de emergencia, como la ubicación de estaciones de bomberos, ambulancias o centros de atención médica, el problema del p-centro se utiliza para optimizar la distribución de recursos y mejorar los tiempos de respuesta.

4. Planificación Urbana: En la planificación urbana, el problema del p-centro puede ayudar a determinar la ubicación óptima de instalaciones públicas, como escuelas, parques, bibliotecas o centros comunitarios, para satisfacer las necesidades de la población de manera eficiente.

5. Diseño de Redes de Distribución: En el diseño de redes de distribución de agua, gas o electricidad, el problema del p-centro se puede utilizar para optimizar la ubicación de plantas de tratamiento, estaciones de bombeo o subestaciones eléctricas, minimizando las pérdidas y maximizando la eficiencia del sistema.

6. Logística Militar: En aplicaciones militares, como la ubicación de bases militares, depósitos de suministros o centros de comando y control, el problema del p-centro puede ser útil para optimizar la distribución de recursos y mejorar la capacidad de respuesta en situaciones críticas.

Capítulo 3

Resolución del problema del p-Centro mediante el problema de Set Covering

A lo largo de esta sección se considerará que las distancias o tiempos solo pueden tomar valores enteros. Suposición asumible ya que cualquier distancia que tome un número racional se puede multiplicar por un número suficiente grande para convertirlo en entero. En cuanto a los números irracionales, se pueden redondear a un número racional y, posteriormente, convertirlo a entero de la forma que se ha mencionado.

3.1. Relación del problema del p-Centro con el problema de Set Covering

En el problema de Set Covering se tiene también un conjunto de puntos de demanda $\mathrm{M} = \{1, 2, \ldots, m\}$ y $\mathrm{N} = \{1, 2, \ldots, n\}$ el cojunto de los puntos donde se puede situar una instalación. Este problema consiste en decidir en cuántos puntos de servicio p, de todos los posibles, es necesario ubicar con tal de cubrir todos los puntos de demanda y que el coste sea el mínimo posible. Se da una constante d_c llamada *distancia de cubrimiento* que marca la distancia máxima que puede haber entre un punto de demanda y el punto que le sirve. Generalmente, se conoce o se calcula la distancia $d_{i,j}$ entre cada punto de demanda y cada posible punto de servicio. A raíz de las distancias, se puede saber para cada punto de demanda i, qué puntos pueden servirle.

$$N_i = \{j \in N : d_{i,j} \leq d_c\}$$

N_i contiene todos los posibles puntos de servicio que podrían servir al punto de demanda i. Con todos los conjuntos N_i podemos construir la matriz a:

$$a_{i,j} = \begin{cases} 0 & si \quad j \notin N_i \\ \\ 1 & si \quad j \in N_i \end{cases}$$

Es decir, $a_{i,j} = 1$ si el punto de servicio j puede servir al punto de demanda i, $a_{i,j} = 0$ en caso contrario.

Cada punto de instalación j tiene un coste de apertura f_j, pero al estar relacionando el problema de Set Covering con el p-centro, consideramos que el coste es unitario para todos los puntos. De esta manera lo que se minimiza es el número de instalaciones que se abrirán.

Tomaremos las variables x_j para indicar si se ha abierto una instalación en el punto j, de tal forma que $x_j = 1$ si se sitúa una instalación en el punto j, $x_j = 0$ en caso contrario. Ya habiendo definido las variables, podemos formular el problema.

3.2. Formulación

$$\text{Minimizar} \qquad \sum_{j=1}^{n} f_j x_j \tag{2.2.1}$$

$$\text{sujeto a} \qquad \sum_{j=1}^{n} a_{i,j} x_j \geq 1 \qquad \forall i \in \{1, 2, \ldots, m\} \tag{2.2.2}$$

$$x_j \in \{0, 1\} \qquad \forall j \in \{1, 2, \ldots, n\} \tag{2.2.3}$$

Se minimiza el coste total de abrir las instalaciones (2.2.1). La segunda restricción se asegura de que todos los puntos de demanda queden cubiertos por al menos una instalación (2.2.2).

Como hemos mencionado antes, tomamos $f_j = 1$ para todo $j \in N$, se pasa a querer encontrar el número mínimo de instalaciones que cubrirá todos los puntos de demanda.

3.2.1. Formulación clásica: Mejor cota

Durante la ejecución del experimento utilizando el método de la mejor cota, nos encontramos con una limitación significativa. Aunque logramos obtener los tiempos de ejecución de manera exitosa para los primeros 10 archivos del conjunto de datos, nos enfrentamos a serios problemas al intentar procesar los archivos restantes.

Específicamente, al llegar al archivo número 11, el uso de memoria del ordenador comenzó a incrementarse de manera desproporcionada, lo que eventualmente llevó a un desbordamiento de la memoria. Este problema de manejo de memoria no solo impidió la continuación del procesamiento, sino que además provocó que el software Xpress, utilizado para llevar a cabo los cálculos, se quedara colgado de forma reiterada.

A pesar de varios intentos para mitigar este problema, incluyendo el ajuste de los parámetros de memoria y la optimización de los recursos disponibles, el desbordamiento persistió, haciendo imposible la obtención de resultados para los archivos restantes. Este obstáculo subraya la necesidad de encontrar alternativas que per-

mitan manejar de manera más eficiente el uso de memoria o considerar métodos distintos que sean menos intensivos en términos de recursos computacionales.

				Mejor cota
Datos	n	p	Óptimo	Tiempo CPU
pmed1	100	5	127	20.01
pmed2	100	10	98	16.19
pmed3	100	10	93	14.34
pmed4	100	20	74	6.96
pmed5	100	33	48	11.13
pmed6	200	5	84	119.10
pmed7	200	10	64	139.40
pmed8	200	20	55	163.92
pmed9	200	40	37	142.56
pmed10	200	67	20	173.68

Tabla 3.1: Tiempo de ejecución obtenido con la mejor cota de la formulación clásica considerando diferentes valores de n y p.

Podemos ver en la Tabla 3.1 que, a pesar de que el tamaño de los problemas sea pequeño, la resolución del problema es costosa computacionalmente.

3.3. Solución mediante Set Covering

Si resolvemos un problema del p-centro, obtenemos el valor óptimo w (tiempo máximo de respuesta) para un valor de p dado. Desde la perspectiva de Set Covering (con coste unitario), obtenemos el número mínimo de instalaciones dado d_c. Se puede decir que el problema de Set Covering, es el inverso del problema del p-centro. Se establece un paralelismo entre w y d_c y entre p y el número de instalaciones abiertas en Set Covering.

Estableciendo esta relación, se puede resolver el problema del p-centro a partir del problema de Set Covering. Si resolvemos el problema de Set Covering para todos los valores enteros d_c posibles obtendremos, para cada valor de d_c, el número mínimo de instalaciones necesarias. Entonces con la solución óptima del problema de Set Covering $\#instalaciones = p$ con menor valor de d_c, obtenemos la solución óptima del problema del p-centro: $w = d_c$.

Capítulo 4

Mejoras a la aplicación de Set Covering en el problema del p-Centro: La búsqueda dicotómica

4.1. Enfoque inicial

Minieka [13] en 1970 fue el primero que propuso resolver el problema del p-centro desde un enfoque de Set Covering. Este enfoque está basado en resolver el problema de Set Covering asociado para cada valor posible de la distancia de cubrimiento. La distancia de cubrimiento d_c se va reduciendo paso a paso. El resultado de interés para el problema del p-centro es aquel en el que se abran p instalaciones con el mínimo d_c posible. El valor de d_c es la solución óptima (tiempo máximo de respuesta) del problema del p-centro. El pseudocódigo se puede definir de la siguiente manera:

Algoritmo 1 Resolución p-Centro mediante Set Covering

1: $d_c \leftarrow \text{máx}_{i,j}\, d_{i,j}, \quad \forall i \in \{1, 2, \ldots, m\} \forall j \in \{1, 2, \ldots, n\}$
2: $d_{max} = \infty$
3: **mientras** $n \geq p$ **hacer**
4: Resolver problema **Set Covering** con distancia de cubrimiento d_c
5: $n = \#instalaciones\ abiertas$
6: **si** $n = p$ y $d_c < d_{max}$ **entonces**
7: $d_{max} = d_c$
8: **fin si**
9: $d_c = d_c - 1$
10: **fin mientras**
11: d_{max} **es la solución óptima.**

Daskin [1] mejoró el enfoque propuesto por Minieka usando una técnica de búsqueda binaria para la selección de la distancia de cubrimiento para reducir el espacio de búsqueda.

4.2. Búsqueda dicotómica

La solución del problema del p-centro mediante el problema del Set Covering anteriormente propuesta es mucho más eficiente que el modelo clásico del p-centro. Sin embargo, es mejorable ya que se necesita resolver una gran cantidad de problemas de Set Covering.

Aquí es cuando entra la *búsqueda dicotómica o búsqueda binaria* propuesta por Daskin[1]. Esta búsqueda nos permite resolver un número mucho menor de problemas de Set Covering. La búsqueda se realizará en el rango de posibles valores de la distancia de cubrimiento d_c.

En rasgos generales, el algoritmo funciona así: Se seleccionan unos límites inferior y superior para la distancia de cubrimiento y se resuelve el problema de cubrimiento total con la media de los dos límites, redondeando hacia el entero más próxima por debajo. En función del resultado obtenido, hay 4 casos posibles:

1. Si se abren menos de p instalaciones, el nuevo límite superior pasa a ser igual a la media que se acaba de calcular
2. Si se abren más de p instalaciones, el nuevo límite inferior pasa a ser igual a la media que se acaba de calcular
3. Si el límite inferior y superior coincide, se termina el algoritmo y el resultado final es igual a estos límites.
4. Si el límite inferior y superior no coinciden, se vuelve a calcular una nueva media de la misma forma que antes y se repiten los pasos.

Este algoritmo está tanteando continuamente distintos valores del valor óptimo del problema del p-centro. A partir de ahora llamaremos $p^*(x)$ a la solución óptima del problema de Set Covering con $d_c = x$. También nombramos a las cotas del valor óptimo: denominamos d_c^L al límite inferior y d_c^H al límite superior. Con estas definiciones, podemos declarar formalmente el algoritmo de la búsqueda dicotómica.

Algoritmo 2 Búsqueda dicotómica

$d_c^L \leftarrow 0$
$d_c^H \leftarrow (n-1) \cdot \text{máx}_{i,j}\, d_{i,j}$
repetir
    $p^*(d_c) \leftarrow$ Solución problema de **cubrimiento total** con $d_c = \lfloor (d_c^H + d_c^L)/2 \rfloor$.
    si $p^*(d_c) \leq p$ **entonces**
        $d_c^H \leftarrow d_c$
    si no
        $d_c^L \leftarrow d_c + 1$
    fin si
    $d_c \leftarrow d_c - 1$
hasta que $d_c^L = d_c^H$
d_{max} **es la solución óptima.**

Se toma $(n-1)\,\text{máx}_{i,j}\, d_{i,j}$ como valor inicial para la cota superior ya que un camino

entre dos nodos tendrá como mucho $n-1$ nodos.

Cuando termina el algoritmo, se toma d_c^L como la solución óptima del problema del p-centro. Los puntos en los que se ubican las instalaciones son los óptimos para el problema del p-centro.

4.3. Mejora a la búsqueda dicotómica

Ilhan y Pinar [10] proponen una extensión del algoritmo de búsqueda dicotómica que se divide en dos fases. En la primera fase se comprueban si son factibles las relajaciones lineales (PL) del problema considerando la cota superior de instalaciones abiertas para obtener una cota inferior adecuada para el valor óptimo. En la segunda fase, empezando por la cota inferior anteriomente calculada, se comprueba si el problema de programación entera (PE) es factible (si hay una solución al problema con $z = \epsilon$ o no) con distintos valores ϵ para la distancia máxima. PE puede formularse de la siguiente manera:

$$\text{Minimizar} \qquad \sum_{j=1}^{n} w_j \tag{3.2.1}$$

$$\text{sujeto a} \qquad \sum_{j \in N} b_{i,j} w_j \geq 1 \qquad \forall i \in \{1, 2, \ldots, m\} \tag{3.2.2}$$

$$w_j \in \{0, 1\} \qquad \forall j \in \{1, 2, \ldots, n\} \tag{3.2.3}$$

Donde $w_j = 1$ si y solo si se abre una instalación en el punto j, $w_j = 0$ en caso contrario y los parámetros $b_{i,j}$ se calculan:

$$b_{i,j} = \begin{cases} 1 & si \quad d_{i,j} \leq \epsilon \\ \\ 0 & si \quad d_{i,j} > \epsilon \end{cases}$$

El parámetro $b_{i,j}$ indica si el posible punto de servicio j está lo suficientemente cerca (en función de la *distancia de cubrimiento* ϵ) del punto de demanda i como para poder servirlo.

Si esta formulación (PE) no es factible, significa que para la distancia máxima ϵ, no hay una solución factible para el problema del p-centro, por lo que se deduce que ϵ es demasiado pequeño. Cuando se alcance el valor de ϵ para el que el problema de vuelva factible, se obtendrá la solución óptima.

El algoritmo se puede dividir en los siguientes pasos:

Paso 1.- Se encuentra el valor mínimo (l) y máximo (u) de todas las distancias/tiempos.
$l = \text{mín}\{d_{i,j} : \forall i \in M, \forall j \in N\}$
$u = \text{máx}\{d_{i,j} : \forall i \in M, \forall j \in N\}$

Paso 2.- Se calcula $dif = \lfloor (u-l)/2 \rfloor$ y se actualiza $\epsilon = l + dif$.

Paso 2.1 Si $d_{i,j} \leq \epsilon$ entonces $b_{i,j} = 1 \quad \forall i \in M, \forall j \in N$.

Paso 2.2 En caso contrario $b_{i,j} = 0$.

Paso 3.- Se resuelve la relajación lineal de PE con distancia de cubrimiento ϵ sustituyendo la restricción (3.2.3) por:

$$0 \leq w_j \leq 1 \quad \forall j \in N$$

Paso 3.1 Si la relajación lineal (PL) no es factible entonces actualizamos $l = \epsilon$.

Paso 3.2 Si es factible, $u = \epsilon$.

Paso 4.- Se calcula $(u - l)$.

Paso 4.1 Si $(u - l) \leq 1$ entonces ir al Paso 5.

Paso 4.2 En caso contrario, volver al Paso 2.

Paso 5.- Se comprueba si la formulación PL es factible.

Paso 5.1 Si es factible, se actualiza $\epsilon = l$

Paso 5.2 En caso contrario, se actualiza $\epsilon = u$

Paso 6.- Se actualizan los valores de $b_{i,j}$ para el nuevo valor de ϵ de la misma forma que en los pasos 2.1 y 2.2.

Paso 7.- Se resuelve el problema PE.

Paso 7.1 Si no es factible, se disminuye el valor de ϵ:
$\epsilon\prime = \text{mín}\{d_{i,j} : d_{i,j} > \epsilon, \forall i \in M, \forall j \in N\}$
Se actualiza $\epsilon = \epsilon\prime$ y se vuelve al Paso 6.

Paso 7.2 En caso contrario, (PE factible) se termina el algoritmo y ϵ es la solución óptima.

En este algoritmo, se van resolviendo problemas de programación lineal y problemas de programación entera sucesivamente para distintos valores de ϵ. En la primera fase del algoritmo (Pasos 1-5), se busca la mejor cota inferior que se usará como ϵ inicial en la segunda fase. La primera fase termina cuando las cotas inferior y superior convergen. En la segunda fase, se va aumentando continuamente la cota inferior (calculada en la anterior fase) a la distancia $d_{i,j}$ superior más cercana a ϵ. Se repetirá este paso hasta que se llegue al valor de ϵ mínimo para el que el problema PE sea factible. Cuando se llega a que PE es factible, ϵ es la solución óptima al problema del p-centro.

El algoritmo se puede definir formalmente de la siguiente forma:

Algoritmo 3 Búsqueda dicotómica Ilhan-Pinar

$l \leftarrow \min\{d_{i,j} : \forall i \in M, \forall j \in N\}$
$u \leftarrow \max\{d_{i,j} : \forall i \in M, \forall j \in N\}$
repetir
    $\Delta \leftarrow \lfloor (u-l)/2 \rfloor$
    $\epsilon \leftarrow l + \Delta$
    para $i \in 1, \ldots, m$, $j \in 1, \ldots, n$ **hacer**
        si $d(i,j) \leq \epsilon$ **entonces**
            $b(i,j) \leftarrow 1$
        si no
            $b(i,j) \leftarrow 0$
        fin si
    fin para
    Se resuelve la relajación lineal de PE con $d_c = \epsilon$
    si PE es factible **entonces**
        $l \leftarrow \epsilon$
    si no
        $u \leftarrow \epsilon$
    fin si
    $d_c \leftarrow d_c - 1$
hasta que $\lfloor (u-l) \rfloor \leq 1$
si PL es factible **entonces**
    $\epsilon \leftarrow u$
si no
    $\epsilon \leftarrow l$
fin si
repetir
    para $i \in 1, \ldots, m$, $j \in 1, \ldots, n$ **hacer**
        si $d(i,j) \leq \epsilon$ **entonces**
            $b(i,j) \leftarrow 1$
        si no
            $b(i,j) \leftarrow 0$
        fin si
    fin para
    Se reformula PE con los nuevos $b_{i,j}$ calculados
    si PE no factible **entonces**
        $\epsilon \leftarrow \min\{d_{i,j} : d_{i,j} > \epsilon, \forall i \in M, \forall j \in N\}$
    fin si
hasta que PE es factible
ϵ **es el valor objetivo óptimo.**

Es una formulación muy potente, obtiene el menor tiempo medio de cómputo de la solución óptima en comparación al resto de métodos exactos. Especialmente en problemas de grandes dimensiones. El número de iteraciones necesarias es de carácter logarítmico y, el número de restricciones impuestas por la formulación, es solo $m+n$ por lo que el coste computacional es bajo.

4.3.1. Resultados Ilhan-Pinar

				Ilhan-Pinar
Datos	n	p	Óptimo	Tiempo CPU
pmed1	100	5	127	0.92
pmed2	100	10	98	0.17
pmed3	100	10	93	0.12
pmed4	100	20	74	0.13
pmed5	100	33	48	0.10
pmed6	200	5	84	0.41
pmed7	200	10	64	0.41
¡pmed8	200	20	55	0.23
pmed9	200	40	37	0.32
pmed10	200	67	20	0.21
pmed11	300	5	59	0.54
pmed12	300	10	51	0.56
pmed13	300	30	36	0.46
pmed14	300	60	26	0.39
pmed15	300	100	18	0.36
pmed16	400	5	47	0.90
pmed17	400	10	39	0.85
pmed18	400	40	28	0.50
pmed19	400	80	18	0.37
pmed20	400	133	13	0.68
pmed21	500	5	40	2.40
pmed22	500	10	38	2.82
pmed23	500	50	22	2.96
pmed24	500	100	15	2.06
pmed25	500	167	11	1.09
pmed26	600	5	38	3.74
pmed27	600	10	32	2.30
pmed28	600	60	18	0.91
pmed29	600	120	13	0.75
pmed30	600	200	9	2.34
pmed31	700	5	30	2.14
pmed32	700	10	29	5.14
pmed33	700	70	15	2.01
pmed34	700	140	11	1.31
pmed35	800	5	30	4.00
pmed36	800	10	27	3.44
pmed37	800	80	15	1.99
pmed38	900	5	29	7.99
pmed39	900	10	23	5.47
pmed40	900	90	13	3.32

Tabla 4.1: Tiempo de ejecución obtenido con el método exacto Ilhan-Pinar considerando diferentes valores de n y p.

En la Tabla 4.1, Ilhan-Pinar demuestra proporcionar los mejores resultados gracias a ser un método que combina el hecho de ser un método exacto con un coste computacional relativamente bajo. Esta notable capacidad para obtener soluciones exactas con un uso moderado de recursos computacionales lo convierte en una opción destacada.

Debido a estas características ventajosas, decidimos evaluar el desempeño del método Ilhan-Pinar con archivos de datos de gran tamaño, empleando los archivos pmed21-pmed40.txt a mayores de los empleados con el resto de algoritmos: pmed1-pmed20.txt. El objetivo era explorar si su eficiencia podía mantenerse cuando se aplicara a conjuntos de datos más extensos y complejos, abriendo la posibilidad de utilizarlo en aplicaciones prácticas que requieren manejar grandes volúmenes de información.

En efecto, se puede observar como la cantidad de tiempo empleado para la obtención de la solución es muy bajo en comparación a la complejidad del problema. Este método es el que mejor resultados ha brindado en este trabajo.

4.4. Formulación de Elloumi-Labbé-Pochet

La búsqueda binaria no es la única técnica para resolver el problema del p-centro desde el punto de vista del problema del Set Covering. Elloumi [3] propuso una nueva formulación de programación entera también basada en el subproblema de Set Covering.

Dada la matriz de distancias $d_{i,j}$ de dimensión $m \times n$ se calcula el vector de distancias D de dimensión $K = m \cdot n$ ordenado ascendentemente (tal que $D_k \leq D_{k+1}, k = 1, \ldots, K$). Las variables que se emplean en la formulación son $y_j, j = 1, \ldots, m$, y z^k, $k = 1, \ldots, K$.

En esta formulación, las dos variables que forman la solución (y, z) son binarias:

$$y_j = \begin{cases} 0 & \text{si no hay instalación en j} \\ \\ 1 & \text{si hay instalación en j} \end{cases}$$

$z^k = 0$ si es posible cubrir a todos los clientes situando p instalaciones con una distancia de cubrimiento igual a D^{k-1}. En caso contrario, $z^k = 1$, esto implica que $z^{k+1} = \ldots = z^K = 0$ por lo que el valor óptimo será estrictamente menor que D^k. Habiendo definido las variables, se puede formular el problema (PC - SC):

$$\text{Minimizar} \qquad D^0 + \sum_{k=1}^{K} (D^k - D^{k-1}) z^k \tag{3.3.1}$$

$$\text{sujeto a} \qquad \sum_{j=1}^{m} y_j \geq 1 \tag{3.3.2}$$

$$\sum_{j=1}^{m} y_j \leq p \tag{3.3.3}$$

$$z^k + \sum_{j:d_{i,j}<D^k} y_j \geq 1 \qquad i = 1, \ldots, n,\ k = 1, \ldots, m \tag{3.3.4}$$

$$y_j \in \{0, 1\} \qquad j = 1, \ldots, m \tag{3.3.5}$$

$$z^k \in \{0, 1\} \qquad k = 1, \ldots, K \tag{3.3.6}$$

La resitricción (3.3.2) descarta soluciones en las que no haya ninguna instalación abierta, mientras que la (3.3.3) se asegura de que no se abran más de p instalaciones. Las restricciones (3.3.4) se encargan de que, dado un valor de k, $z^k = 0$ si y solo si todos los clientes pueden ser servidos con un tiempo de respuesta máximo estricamente menor que D^k. Cuando $z^k = 1$, no se puede cubrir a todos los clientes con D^k, por lo que aumentamos el valor del radio al valor superior más cercano. Esto se consigue sumando el valor $(D^k - D^{k-1})$ a la distancia máxima que teníamos hasta ahora en la función objetivo.

La formulación de Elloumi-Labbé-Pochet para el problema del p-centro, aunque precisa y robusta, presenta ciertos desafíos cuando se utiliza junto con el método de la mejor cota. Específicamente, al aplicar el método de la mejor cota a esta formulación, los resultados se obtienen de manera menos eficiente que en la formulación clásica. Este menor rendimiento en eficiencia se debe a la complejidad inherente de la formulación y a la naturaleza del problema de optimización que se intenta resolver:

				Mejor cota
Datos	n	p	Óptimo	Tiempo CPU
pmed1	100	5	127	141.71
pmed2	100	10	98	74.43
pmed3	100	10	93	85.32
pmed4	100	20	74	85.32
pmed5	100	33	48	70.09

Tabla 4.2: Resultados obtenidos mediante la mejor cota de la formulación Elloumi-Labbé-Pochet.

Al intentar resolver los problemas utilizando la formulación de Elloumi-Labbé-Pochet mediante el método de la mejor cota, se observó una limitación significativa en la capacidad de la memoria del ordenador. Específicamente, la memoria disponible solo permitió resolver los primeros 5 problemas del conjunto de datos como se muestra en la Tabla 4.2. Al intentar procesar problemas adicionales, la memoria se desbordaba, lo que imposibilitaba la obtención de resultados y hacía que el software se colgara.

Reconociendo las limitaciones del método de la mejor cota en esta formulación específica, Elloumi, Labbé y Pochet desarrollaron el algoritmo BSearch, que representa una mejora significativa en términos de eficiencia. El algoritmo BSearch es un enfoque especializado que optimiza el proceso de búsqueda y asignación de centros, logrando resultados óptimos con una utilización mucho más eficiente de los recursos computacionales.

Al igual que el método de Ilhan-Pinar, consiste en una búsqueda binaria en la que se emplean relajaciones lineares y se resuelve el problema del *Set Covering*:

Primero hay que definir varios términos:

- $PC - SC$: Problema del p-centro formulado por Elloumi, Labbé y Pochet.
- SC_d: Problema de Set Covering con distancia de cubrimiento d.
- LSC_d: Relajación lineal de SC_d.
- $D_{min} = D^0 < D^1 < \ldots < D^K < D_{max}$: Distancias $d_{i,j}$ ordenadas de menor a mayor.
- $v(SC_d)$: Valor óptimo de la función objetivo del problema SC_d con $v(SC_d) = \infty$ si no hay solución factible. Es trivial que $v(SC_{D^0}) > v(SC_{D^1}) > \ldots > v(SC_{D^K})$

Algoritmo 4 BSearch

1: $head \leftarrow -1$
2: $tail \leftarrow K$
3: $UB* \leftarrow D^K$
4: **repetir**
5: $h \leftarrow [(head + tail)/2]$
6: $y^G \leftarrow$ solución del problema SC_{D^h}
7: **si** $\sum_{j=1}^{M} y_j^G \leq p$ **entonces**
8: $tail \leftarrow h$
9: $UB^* \leftarrow D^h$
10: Volver a 3.
11: **fin si**
12: $t^G \leftarrow$ solución *greedy* de $PC - SC$ con $p = y^G$
13: **si** $t^G < UB^*$ **entonces**
14: $UB^* \leftarrow t^G$
15: **fin si**
16: $\bar{y} \leftarrow$ valor óptimo del problema LSC_{D^h}
17: **si** $v(LSC_{D^h}) = \sum_{j=1}^{M} \bar{y}_j < p$ **entonces**
18: $head \leftarrow h$
19: Volver a 3.
20: **fin si**
21: $y^V =$ solución *greedy* para SC_{D^h} obtenida a partir de $\bar{y}$
22: $t^{G'} \leftarrow$ solución *greedy* de $PC - SC$ con $p = y^V$.
23: **si** $t^{G'} < UB^*$ **entonces**
24: $UB^* \leftarrow t^{G'}$
25: **fin si**
26: **mientras** $UB^* < D^{tail}$ **hacer**
27: $tail \leftarrow tail - 1$
28: **fin mientras**
29: **hasta que** $head \geq tail - 1$
30: $LB^* \leftarrow D^{tail}$
31: **Se toma** UB^* **como valor objetivo de la solución**

Al tratarse de una búsqueda binaria, el número máximo de problemas LSC_d a resolver es $\log_2(K+1)$. El algoritmo recibe como entradas un problema del p-centro y el valor de la solución UB_1 obtenida con el algoritmo *greedy*. En el algoritmo no se busca en las $N * M$ distancias ordenadas, sino en las $K + 1$ distancias acotadas por $D^K = UB_1$. El algoritmo calcula LB^* e intenta buscar una mejor solución UB^*. Esta cota UB^* se inicializa al valor de la solución *greedy* y se actualiza siempre que se encuentre una mejor solución para el problema del p-Centro. En cada iteración de la búsqueda binaria, dado el índice h y su respectiva distancia D^h, el objetivo principal para calcular LB^* es comparar $v(LSC_{D^h})$ con p. El segundo objetivo es conseguir un mejor valor para UB^* par el problema del p-Centro. En todo momento se conoce que el índice del valor óptimo h^* se encuentra en el rango de índices $\{head + 1, \ldots, tail\}$.

4.4.1. Resultados *BSearch*

				BSearch		
Datos	n	p	Óptimo	LB^*	UB^*	Tiempo CPU
pmed1	100	5	127	122	127	0.24
pmed2	100	10	98	98	98	<0.1
pmed3	100	10	93	93	93	<0.1
pmed4	100	20	74	74	74	<0.1
pmed5	100	33	48	48	48	<0.1
pmed6	200	5	84	83	84	0.66
pmed7	200	10	64	64	64	0.41
pmed8	200	20	55	55	55	0.32
pmed9	200	40	37	37	37	0.59
pmed10	200	67	20	20	20	<0.1
pmed11	300	5	59	59	63	1.25
pmed12	300	10	51	51	51	1.54
pmed13	300	30	36	36	36	0.46
pmed14	300	60	26	26	26	0.77
pmed15	300	100	18	18	18	1.09
pmed16	400	5	47	47	48	1.55
pmed17	400	10	39	39	39	2.79
pmed18	400	40	28	28	28	1.73
pmed19	400	80	18	18	19	0.70
pmed20	400	133	13	13	13	1.86
pmed21	500	5	40	40	40	4.51
pmed22	500	10	38	37	38	8.54
pmed23	500	50	22	22	23	1.40
pmed24	500	100	15	15	15	3.50
pmed25	500	167	11	11	11	4.48
pmed26	600	5	38	38	39	6.11
pmed27	600	10	32	32	32	7.92
pmed28	600	60	18	18	18	1.91
pmed29	600	120	13	13	13	3.50
pmed30	600	200	9	9	9	6.56
pmed31	700	5	30	29	30	2.14
pmed32	700	10	29	28	30	13.91
pmed33	700	70	15	15	15	5.14
pmed34	700	140	11	11	11	6.99
pmed35	800	5	30	30	30	11.85
pmed36	800	10	27	27	28	18.15
pmed37	800	80	15	15	15	23.61
pmed38	900	5	29	29	29	17.99
pmed39	900	10	23	23	24	25.47
pmed40	900	90	13	13	13	9.92

Tabla 4.3: Comparación del resultado óptimo con la metaheurística *BSearch* y su tiempo de ejecución considerando diferentes valores de n y p.

Se puede observar como no siempre las cotas convergen, aunque en todos los casos el intervalo $[LB^*, UB^*]$ es pequeño. De todas formas, es recomendable tomar como valor óptimo de la función objetivo la cota superior UB^* para no dar una solución optimista.

4.4.2. Mejora a *BSearch: BSearchEx*

A pesar de los buenos resultados que ofrece BSearch, existe una mejora, llamada *BSearchEx* que aporta la solución exacta. El objetivo de *BSearchEx* es encontrar k^* tal que D^{k^*} es el radio óptimo (valor óptimo de la función objetivo). Se aplica una búsqueda binaria en el espacio de las distancias $LB^* = D^{h^*} \leq D^{h^*+1} \leq \ldots \leq D^{K'} = UB^*$. Se puede obtener el algoritmo aplicando unas pequeñas modificaciones al algoritmo *BSearch*:.

1. En las líneas 1 y 2 se modifican las inicializaciones de *head* y *tail*: $head \leftarrow h^* - 1$; $tail \leftarrow K'$
2. En la línea 21, en vez de calcularse la solución *greedy* para SC_{D^h}, se calcula la solución exacta y^{01}.
3. En la línea 22, en vez de calcular la solución *greedy* para $PC - SC$:
   - Si $v(SC_{D^h}) = \sum_{j=1}^{M} y_j^{01} > p$ entonces $head = h$ sino $tail = h$. Se actualiza UB^* si es necesario.

4.4.3. Resultados *BSearchEx*

Se ha aplicado *BSearchEx* a los 40 ficheros de prueba para evaluar su eficiencia:

				BSearchEx
Datos	n	p	Óptimo	Tiempo CPU
pmed1	100	5	127	1.04
pmed2	100	10	98	0.57
pmed3	100	10	93	0.82
pmed4	100	20	74	0.31
pmed5	100	33	48	0.60
pmed6	200	5	84	0.88
pmed7	200	10	64	0.54
pmed8	200	20	55	0.81
pmed9	200	40	37	0.21
pmed10	200	67	20	0.46
pmed11	300	5	59	2.0
pmed12	300	10	51	1.25
pmed13	300	30	36	0.46
pmed14	300	60	26	1.39
pmed15	300	100	18	0.97
pmed16	400	5	47	1.63
pmed17	400	10	39	2.1
pmed18	400	40	28	1.26
pmed19	400	80	18	1.70
pmed20	400	133	13	0.68
pmed21	500	5	40	4.40
pmed22	500	10	38	6.28
pmed23	500	50	22	1.36
pmed24	500	100	15	3.60
pmed25	500	167	11	3.18
pmed26	600	5	38	4.47
pmed27	600	10	32	5.27
pmed28	600	60	18	2.32
pmed29	600	120	13	3.45
pmed30	600	200	9	5.06
pmed31	700	5	30	2.41
pmed32	700	10	29	64.65
pmed33	700	70	15	4.81
pmed34	700	140	11	1.31
pmed35	800	5	30	10.66
pmed36	800	10	27	43.13
pmed37	800	80	15	2.69
pmed38	900	5	29	20.34
pmed39	900	10	23	38.72
pmed40	900	90	13	9.32

Tabla 4.4: Comparación del resultado óptimo con la metaheurística *BSearchEx* y su tiempo de ejecución considerando diferentes valores de n y p.

Al tratarse de un método exacto, se obtiene la solución óptima para todos los problemas. Consigue los resultados óptimos en un menor tiempo de cómputo que el algoritmo *BSearch* en casi todas las instancias, estos se muestran en la Tabla 4.4.

Capítulo 5

Heurísticas de construcción para la resolución del problema del p-Centro

5.1. Heurísticas greedy

Los algoritmos voraces o *greedy* construyen una solución paso a paso. En cada paso del algoritmo se actualiza la solución, que se está construyendo, tomando la mejor decisión local posible. Todas las decisiones hechas por el algoritmo *greedy* siempre formarán parte de la solución final. Los problemas que pueden ser resueltos por algoritmos voraces normalmente pueden dividirse en subproblemas con la misma estructura general.

Las algoritmos *greedy* necesitan una información que usar como criterio para poder tomar decisiones. Esta información orientativa se llama heurística y normalmente se puede extraer de la estructura del problema. Escoger una buena heurística es fundamental para la corrección y efectividad del algoritmo *greedy*.

A pesar de que los algoritmos *greedy* no suelan solucionar óptimamente un problema, generalmente aportan buenas aproximaciones. Además, su mayor atractivo es su simplicidad y su eficiencia computacional. A continuación se explican dos algorimos *greedy* para el problema del p-centro con dos heurísticas distintas.

5.1.1. Greedy - 1

En el problema del p-centro, intuitivamente, al querer que el tiempo máximo de respuesta sea mínimo, nos interesa que las puntos de servicio estén ubicados lo más lejos posible unos de otros. De esta manera, todos los puntos de demanda tienen la posibilidad de tener un punto de servicio cercano y, los puntos de demanda que estén más alejados del resto de vértices no contribuyan demasiado a la distancia máxima.

El algoritmo *Greedy - 1* empieza escogiendo arbitrariamente un posible punto de servicio $i \in N$ e incluyéndolo en el conjunto solución S. A continuación se escoge el punto esté lo más alejado posible de S (que por ahora solo contiene i) y se inclu-

ye en la solución parcial. Generalizando, si se quieren abrir como mucho p puntos, mientras $|S| < p$ se encuentra el posible punto de servicio j que maximice $d(j, S)$ y se añade a S. Cuando se llegue a $|S| = p$, se para y S es el conjunto solución.

Podemos definir formalmente el algoritmo:

Algoritmo 5 Greedy - 1

1: $S \leftarrow \emptyset$
2: Se escoge un punto $i \in N$ arbitrariamente
3: $S \leftarrow \{i\}$
4: **repetir**
5: $\quad s = \arg\max_{s \in N} d(s, S)$
6: $\quad S \leftarrow S \cup \{s\}$
7: **hasta que** $|S| = p$
8: S **es el conjunto solución.**

5.1.2. Greedy - 2

En el algoritmo *Greedy - 1* se usa como heurística las distancias entre los puntos de servicio, pero no se tiene en cuenta el tiempo de respuesta máximo explícitamente. El algoritmo *Greedy - 2* lo tiene en cuenta.

Dada una solución $S \subseteq N$, sea $\delta_S(i) = \min_{j \in S} d_{i,j}$, se tiene que $C(S) = \max_{i \in M} \delta_S(i)$ es la distancia máxima de un punto de M a S. El algoritmo *Greedy - 2* empieza escogiendo arbitrariamente un posible punto de servicio $i \in N$ e incluyéndolo en el conjunto solución S. A continuación, ya obtenida una solución parcial $S \subseteq N$, se calcula para todos los puntos $j \in N \setminus S$ el criterio greedy $w(j) = C(S \cup \{j\})$. $w(j)$ indica la distancia máxima que habría de un punto de M a S, si añadieramos al punto j en la solución S. Es decir, la distancia máxima de $i \in M$ al conjunto $S \cup \{j\}$. Obviamente, en cada paso nos interesa encontrar el punto j que minimice el criterio $w(j)$.
Este algoritmo, al igual que todos los algoritmos voraces, es determinista. Formalmente lo podemos definir así:

Algoritmo 6 Greedy - 2

1: $S \leftarrow \emptyset$
2: Se escoge un punto $i \in N$ arbitrariamente
3: $S \leftarrow \{i\}$
4: **repetir**
5: **para** $j \in N \setminus S$ **hacer**
6: $w_j = C(S \cup \{j\})$
7: **fin para**
8: $s = \arg\min_j w_j$
9: $S \leftarrow S \cup \{s\}$
10: **hasta que** $|S| = p$
11: S **es el conjunto solución.**

5.1.3. Resultados Greedy - 1 vs Greedy - 2

				Greedy - 1			Greedy - 2		
Datos	n	p	Óptimo	Solución	Tiempo CPU	gap	Solución	Tiempo CPU	gap
pmed1	100	5	127	166	<0.01	30.71 %	138	<0.01	10.24 %
pmed2	100	10	98	143	<0.01	45.91 %	115	<0.01	17.35 %
pmed3	100	10	93	133	<0.01	43.01 %	142	0.01	52.69 %
pmed4	100	20	74	104	<0.01	40.54 %	100	0.02	35.14 %
pmed5	100	33	48	62	<0.01	29.16 %	70	0.04	66.6 %
pmed6	200	5	84	122	<0.01	45.24 %	99	0.04	17.86 %
pmed7	200	10	64	93	<0.01	45.31 %	78	0.07	21.88 %
pmed8	200	20	55	82	<0.01	49.10 %	79	0.08	43.64 %
pmed9	200	40	37	50	<0.01	35.14 %	64	0.13	72.97 %
pmed10	200	67	20	31	<0.01	55.00 %	36	0.31	80.00 %
pmed11	300	5	59	94	<0.01	59.32 %	68	0.05	15.25 %
pmed12	300	10	51	84	<0.01	64.71 %	60	0.10	17.65 %
pmed13	300	30	36	56	<0.01	55.55 %	60	0.38	66.66 %
pmed14	300	60	26	41	<0.01	57.69 %	44	0.59	69.23 %
pmed15	300	100	18	25	<0.01	38.89 %	36	0.97	100 %
pmed16	400	5	47	63	<0.01	34.04 %	52	0.1	10.64 %
pmed17	400	10	39	56	<0.01	43.59 %	48	0.18	23.10 %
pmed18	400	40	28	42	<0.01	50.00 %	43	0.81	53.57 %
pmed19	400	80	18	28	<0.01	55.56 %	35	1.89	94.44 %
pmed20	400	133	13	19	<0.01	46.15 %	26	2.05	100 %

Tabla 5.1: Comparación del resultado óptimo con los resultados obtenidos con las heurísticas Greedy - 1 y Greedy - 2 y su tiempo de ejecución considerando diferentes valores de n y p.

Si nos fijamos en la calidad de las soluciones, parece que Greedy - 2 funciona mejor cuando es menor el número de instalaciones, independientemente del número

de puntos totales. Según va aumentando el número de instalaciones a localizar, el método Greedy - 1 empieza a aportar mejores resultados. En cuanto al tiempo de cómputo, el método Greedy - 1 es significativamente más rápido. No es de gran importancia en nuestros problemas, ya que estos son de tamaño pequeño y mediano, pero tendrá una gran importancia en la siguiente sección, cuando se empleen los métodos multiarranque.

5.2. Calidad de aproximación de greedy

Como su nombre indica, los algoritmos de aproximación, proporcionan una aproximación a la solución de problemas de optimización. Este tipo de algoritmos no garantizan devolver la solución óptima, pero la solución aproximada tiene garantías de estar a menos de una determinada distancia de la solución óptima. Además, este tipo de algoritmos garantizan ejecutarse en tiempo polinómico.

***Greedy* es un algoritmo 2-aproximación**
Demostremos que *Greedy* es un algoritmo 2-aproximación:
Se tiene $S^* = \{s_1, \ldots, s_k\}$ como conjunto de la solución óptima y r^* es su radio o distancia máxima. Esta solución óptima divide a los puntos de demanda M en k clusters donde cada punto $i \in M$ pertenece al cluster M_l^* siendo l el punto de servicio más cercano a i.

Es trivial que cada par de puntos i y j que pertenezcan al mismo cluster están, como mucho, a $2r^*$ de distancia. Según la desigualdad triangular, la distancia $d(i, j)$ entre ellos es, a lo sumo, la suma de $d(i, s_l)$, la distancia entre i y el centro s_l y $d(j, s_l)$, la distancia desde el centro s_l hasta j. Se pueden redondear las dos distancias a r^*, entonces:

$$d(i, j) \leq d(i, s_l) + d(j, s_l) \leq r^* + r^* = 2r^*$$

Sea V el conjunto de todos los puntos. Ahora se considera el conjunto $S \subseteq V$ de puntos de servicio escogidos por el algoritmo *greedy* y se considera la primera iteración en la que se escoge un punto $i \in V_l^*$ que se añade a S habiendo escogido el algoritmo previamente un punto $i\prime \in V_l^*$ en una iteración anterior. Previamente a haber elegido $i\prime$, cada centro añadido a S había sido escogido de distintos clusters óptimos de S^*. Para todos los puntos j cubiertos por los centros añadidos previamente a $i\prime$, se tiene $d(j, S) \leq r^*$ por lo recientemente mencionado. Como el algoritmo *greedy* siempre escoge los puntos más alejados del conjunto de solución parcial S, entonces cualquier otro punto $j \in N \setminus S$ tiene su distancia acotada por:

$$d(j, S) \leq d(i', i)$$

En caso contrario, j tendría que haber sido añadido a S antes que i'. Sin embargo, i y i' pertenecen al mismo cluster V_l^* por lo que $d(i', i) \leq 2r^*$. Generalizando, todos los puntos cubiertos después de que i' haya sido añadido, la distancia entre ellos y su punto de servicio más cercano será como mucho $2r^*$. Entonces tenemos las distancias acotadas para todos los puntos $i \in V$:

$$d(i, S) \leq 2r^*$$

Si r representa el radio de S devuelto por el algoritmo *greedy*, entonces obtenemos:

$$r^* \leq r \leq 2r^*$$

Por lo que podemos concluir que el algoritmo devuelve una 2-aproximación. *Greedy* es un algoritmo 2-aproximación.

5.2.1. Mejora a la búsqueda dicotómica acotando con *greedy*

Volviendo a la búsqueda dicotómica, es necesario recordar que esta técnica se basa en la búsqueda entre dos cotas de la distancia máxima óptima para el problema del *Set Covering*. Sea r^* la solución que nos devuelve el algoritmo *greedy*, y r la solución óptima del problema del p-centro, se sabe que:

$$r^* \leq r \leq 2r^*$$

Por lo que se puede empezar la búsqueda dicotómica con cota inferior r^* y cota superior $2 \cdot r^*$ para disminuir el tamaño del espacio de búsqueda y el tiempo de cómputo.

5.2.2. Resultados Búsqueda Dicotómica vs Dicotómica Mejorada

			Búsqueda dicotómica		Búsqueda dicotómica cotas *greedy*	
Datos	n	p	Tiempo CPU	Iteraciones	Tiempo CPU	Iteraciones
pmed1	100	5	0.59	16	0.20	7
pmed2	100	10	0.22	16	0.16	7
pmed3	100	10	0.47	16	0.12	7
pmed4	100	20	0.14	16	0.09	6
pmed5	100	33	0.15	16	0.09	7
pmed6	200	5	1.37	16	1.22	7
pmed7	200	10	0.51	17	0.35	7
pmed8	200	20	0.55	17	0.19	7
pmed9	200	40	0.32	17	0.14	6
pmed10	200	67	0.25	16	0.09	5
pmed11	300	5	1.30	16	0.81	7
pmed12	300	10	2.55	17	1.65	6
pmed13	300	30	0.81	17	0.38	6
pmed14	300	60	0.58	16	0.17	5
pmed15	300	100	0.51	16	0.11	4
pmed16	400	5	2.87	16	2.18	6
pmed17	400	10	2.76	16	2.43	6
pmed18	400	40	2.49	17	0.50	6
pmed19	400	80	1.12	16	0.27	5
pmed20	400	133	1.25	17	0.12	4
pmed21	500	5	8.21	17	5.26	6
pmed22	500	10	11.72	17	9.77	6
pmed23	500	50	2.45	16	1.38	6
pmed24	500	100	1.64	16	0.28	5
pmed25	500	167	2.83	17	0.15	4
pmed26	600	5	8.55	16	5.95	5
pmed27	600	10	11.74	17	8.32	6
pmed28	600	60	10.19	17	2.14	5
pmed29	600	120	2.24	17	0.57	5
pmed30	600	200	4.76	17	0.16	4
pmed31	700	5	26.68	17	19.66	5
pmed32	700	10	24.27	18	22.89	6
pmed33	700	70	9.73	17	4.47	5
pmed34	700	140	4.88	17	0.26	4
pmed35	800	5	14.36	17	13.50	5
pmed36	800	10	35.88	17	34.46	6
pmed37	800	80	10.84	17	1.67	5
pmed38	900	5	34.94	18	30.12	5
pmed39	900	10	33.56	18	33.65	5
pmed40	900	90	19.02	17	8.12	5

Tabla 5.2: Comparación del tiempo de ejecución e iteraciones obtenidas con la búsqueda dicotómica y la búsqueda dicotómica con cotas considerando diferentes valores de n y p.

Se puede notar una reducción en el coste de cómputo al haber utilizado mejor cotas iniciales mediante el método *greedy*. Se emplean menos iteraciones para llegar a la solución.

Teniendo en cuenta que el espacio de búsqueda se reduce polinómicamente, hay una mejora pero no es muy notable. Será menos notable cuanto mayor sea el tamaño del problema. Reducimos el espacio de búsqueda de $|\max d(i,j)|$ a $|r^*|$.

5.3. Métodos multiarranque

5.3.1. Greedy multiarranque

El principal inconveniente de los algoritmos *greedy* es que obtienen soluciones que, a pesar de ser localmente óptimas, pueden estar muy distantes de la solución óptima global. Iniciar los algoritmos locales desde distintos puntos iniciales es una técnica cuyo uso está muy expandido que evita que se devuelvan malas soluciones óptimas. Siguen el esquema *Generar Punto Inicial* y *Algoritmo Greedy*. En ocasiones, en vez de generarse un único punto inicial, se generan l , $l > 1$ puntos y se inicia el algoritmo *greedy* sobre la solución parcial.

Si el problema es muy grande, se pueden realizar k arranques, asegurándose de que no se repitan los puntos iniciales generados. Si el problema es de un tamaño no demasiado grande, se puede iterar el punto inicial por todos los posibles puntos iniciales.

Los métodos *greedy* multiarranque siguen esta lógica:

Algoritmo 7 Greedy multiarranque

1: **para** $i \in N$ **hacer**
2: La mejor solución obtenida hasta el momento es S^{min}
3: $S \leftarrow \{i\}$
4: $S^* =$ Conjunto solución para algoritmo ***greedy*** con conjunto inicial S
5: **si** $f(S^*) < f(S^{min})$ **entonces**
6: $S^{min} = S^*$
7: **fin si**
8: **fin para**
9: S^{min} **es el conjunto solución.**

En este caso, como se puede ver en el bucle, el método multiarranque toma todos los puntos iniciales posibles. Si el tamaño del problema es muy grande, puede no ser practicable.

Como el algoritmo *greedy* asegura terminar en un tiempo polinómico $O(pv)$ con $v = n+m$, introducirlo en el bucle lo convierte en $O(pvn) = O(pn(n+m))$, asumible si el tamaño del problema es pequeño o mediano.

En este caso, se van a implementar unos métodos *greedy* multiarranque en los que solo se va a recorrer todos los puntos iniciales posibles. Sin embargo, existen más

posibilidades como, en vez de inicializar un único punto en cada iteración, inicializar q puntos siempre que $q < p$, habiendo $n \cdot (n-1) \ldots (n-q)$ posibles soluciones iniciales.

5.3.2. Resultados Greedy Multiarranque

				Greedy - 1 Multistart			**Greedy - 2 Multistart**		
Datos	n	p	Óptimo	Solución	Tiempo CPU	gap	Solución	Tiempo CPU	gap
pmed1	100	5	127	155	0.01	30.71 %	129	0.58	10.24 %
pmed2	100	10	98	117	0.01	45.91 %	108	1.80	17.35 %
pmed3	100	10	93	124	0.01	43.01 %	101	2.10	52.69 %
pmed4	100	20	74	92	0.03	40.54 %	92	2.00	35.14 %
pmed5	100	33	48	62	0.05	29.16 %	70	4.36	66.66 %
pmed6	200	5	84	110	0.03	45.24 %	84	4.59	17.86 %
pmed7	200	10	64	85	0.06	45.31 %	70	15.92	21.88 %
pmed8	200	20	55	71	0.08	49.10 %	72	47.17	43.64 %
pmed9	200	40	37	49	0.20	35.14 %	58	31.68	72.97 %
pmed10	200	67	20	29	0.40	55.00 %	36	49.76	80.00 %
pmed11	300	5	59	68	0.05	59.32 %	60	12.31	15.25 %
pmed12	300	10	51	66	0.10	64.71 %	59	83.66	17.65 %
pmed13	300	30	36	49	0.38	55.55 %	49	84.79	66.66 %
pmed14	300	60	26	36	0.55	57.69 %	44	220.15	69.23 %
pmed15	300	100	18	23	1.00	38.89 %	36	240.33	100 %
pmed16	400	5	47	52	0.10	34.04 %	49	28.67	10.64 %
pmed17	400	10	39	48	0.58	43.59 %	43	65.32	23.10 %
pmed18	400	40	28	39	1.63	50.00 %	43	315.80	53.57 %
pmed19	400	80	18	27	1.96	55.56 %	35	>600	94.44 %
pmed20	400	133	13	17	2.62	46.15 %	26	>600	100 %

Tabla 5.3: Comparación del resultado óptimo con los resultados obtenidos con las heurísticas Greedy - 1 y Greedy - 2 Multiarranque y su tiempo de ejecución considerando diferentes valores de n y p.

Los métodos multistart ofrecen mejores resultados al explorar todos los puntos iniciales posibles de la solución, sacrificando así el coste de cómputo para obtener soluciones de mayor calidad. Las tendencias comentadas en la sección anterior siguen iguales.

En este caso, es notable el coste de cómputo del método Greedy-2, que es significativamente mayor que el del método Greedy-1. Esto se refleja en los tiempos de CPU mucho más elevados para Greedy-2 en comparación con Greedy-1.

5.3.3. Greedy aleatorizado

Dentro de los métodos multiarranque, uno de los más importantes es el método *Greedy Aleatorizado*. Su principal punto fuerte es eliminar el carácter determinista de los métodos *greedy*. Hasta ahora, si se tenía n posibles puntos de servicio, un algoritmo *greedy* solo podía conseguir n soluciones, problema inexistente al eliminar el determinismo.

El algoritmo se basa en construir una lista de candidatos (RCL) en cada iteración (en vez de elegir directamente el mejor elemento localmente) y se escoge al azar un elemento de la lista. Este elemento pasa a formar parte de la solución.

El tamaño de la RCL puede decidirse de dos formas: mediante cardinal (escogiendo los K mejores candidatos) o mediante valor. Ferone *et al.* [4] proponen formar la RCL calculando los valores:

$$z_{min} = \min_{j \in N \setminus P} w(j) \quad \text{y} \quad z_{max} = \max_{j \in N \setminus P} w(j)$$

Siendo $w(j)$ el mismo criterio empleado por el método *Greedy - 2* explicado anteriormente. Entonces, tomando el valor de *cut-off*:

$$\mu = z_{min} + \beta(z_{max} - z_{min})$$

donde el parámetro $\beta \in \{0, 1\}$, la RCL está formada por todos los $j \in N \setminus P$ con $w(j) \leq \mu$.

Los dos parámetros que se fijan en un algoritmo *greedy* aleatorizado son $nIter$ (número de iteraciones) y K o β en función de si se forma la RCL por cardinal o por valor respectivamente. Podemos definir el algoritmo mediante el siguiente pseudocódigo:

Algoritmo 8 Greedy aleatorizado

para $i = 1$ hasta $iter\_max$ **hacer**
    La mejor solución obtenida hasta el momento es S^{min}
    $i =$ punto aleatorio $\in N$
    $S = \{i\}$
    mientras $|S| < p$ **hacer**
        $RCL = \{\}$
        mientras $nK < K$ **hacer**
            $j = \arg\min_{j \in N \setminus S} d_{j,S}$
            $RCL \leftarrow RCL \cup \{j\}$
        fin mientras
        $r \leftarrow$ Candidato escogido aleatoriamente
        $S \leftarrow S \cup \{r\}$
    fin mientras
    si $f(S) < f(S^{min})$ **entonces**
        $S^{min} = S$
    fin si
fin para
S^{min} **es el conjunto solución.**

5.3.4. Resultados Greedy Aleatorizado

				Greedy Aleatorizado		
Datos	n	p	Óptimo	Solución	Tiempo CPU	gap
pmed1	100	5	127	152	0.01	19.69 %
pmed2	100	10	98	120	0.03	8.33 %
pmed3	100	10	93	107	0.03	18.81 %
pmed4	100	20	74	96	0.04	29.73 %
pmed5	100	33	48	62	0.07	61.90 %
pmed6	200	5	84	102	0.02	21.43 %
pmed7	200	10	64	84	0.04	31.25 %
pmed8	200	20	55	69	0.11	25.45 %
pmed9	200	40	37	51	0.19	37.84 %
pmed10	200	67	20	29	0.28	45.00 %
pmed11	300	5	59	72	0.03	22.03 %
pmed12	300	10	51	69	0.09	35.29 %
pmed13	300	30	36	49	0.23	36.11 %
pmed14	300	60	26	37	0.42	42.31 %
pmed15	300	100	18	23	0.67	27.78 %
pmed16	400	5	47	55	0.03	17.02 %
pmed17	400	10	39	50	0.07	28.21 %
pmed18	400	40	28	40	0.34	42.86 %
pmed19	400	80	18	27	0.70	50.00 %
pmed20	400	133	13	18	1.06	38.46 %

Tabla 5.4: Comparación del resultado óptimo con la heurística *greedy* aleatorizado y su tiempo de ejecución considerando diferentes valores de n y p

Obtiene mejores resultados que los métodos *greedy* multiarranque. Se consigue un *gap* medio mucho menor con una notable reducción del coste computacional como se puede observar en la Tabla 5.4.

Capítulo 6

Heurísticas de búsqueda local en la resolución del problema del p-Centro

La mayoría de los algoritmos heurísticos tienen la búsqueda local como su base. Este metodo heurístico se emplea cuando se tienen problemas en los que se quiere minimizar o maximizar una función objetivo y hay varias soluciones candidatas.

La búsqueda local inicia con una solución inicial y consiste en pasar de una solución candidata a otra vecina aplicando cambios locales iterativamente hasta un límite de intercambios o de tiempo. Normalmente, cada solución candidata tiene más de una solución vecina, la elección de la siguiente solución se tomará solo en base a la información de las soluciones vecinas de la actual. La búsqueda local no asegura conseguir la solución óptima y es común que se quede atascado en un mínimo local. Este problema se puede solucionar empleando métodos multiarranque, introduciendo aleatorización o esquemas más complejos.

Para poder moverse por el espacio de soluciones, los algoritmos de búsqueda local emplean la relación llamada vecindad. Se dice que una solución s es vecina de orden k de otra s' y viceversa si se puede llegar de una solución a otra realizando exactamente k movimientos. En el contexto del p-centro, un movimiento consiste en cambiar una instalación de un posible punto de servicio a otro.

Dado un punto $i \in P$ y otro punto $j \notin P$:

$$P^{nueva} = P \setminus \{i\} \cup \{j\}$$

6.1. Neighborhood Search

La búsqueda local más básica es el método *Neighborhood Search* o Intercambio 1-1. Este método fue propuesto por Teitz y Bart en 1968 [14]. Explora todas las soluciones vecinas de la actual $i \in S$ por $j \notin S$ a partir de una solución inicial S_{ini}. Este método es muy sensible a la solución inicial y al orden en el que se comprueban los intercambios. Si se alcanza una solución cuyo vecindario no tiene una solución

mejor, se termina el algoritmo.

Su mayor atractivo es la sencillez del concepto y de implementación y su eficacia en problemas de pequeñas y medianas dimensiones. Por ello es muy popular usar un método *greedy* para obtener la solución inicial S_{ini} y emplear NS para obtener una estimación aceptable en un tiempo de computación asumible. A pesar de esto, su rendimiento no es bueno cuando el problema tiene una gran dimensión y, consecuentemente, los vecindarios que tiene que explorar son muy grandes.

Cuando se tiene un número p grande, esto provoca que muchas soluciones candidatas se encuentren en *mesetas*, es decir, es posible que una gran parte de los posibles movimientos no afecten al valor objetivo y no haya soluciones vecinas que lo mejoren, solo lo empeoran. Hay muchos mínimos locales vecinos.

Se puede definir mediante pseudocódigo de la siguiente forma:

Algoritmo 9 Neighborhood Search

$S = S_{ini}$
repetir
    $\Delta_{max} = 0$
    $z_S = C(S)$
    para todo $i \in S$, $j \notin S$ **hacer**
        $z_{int} = C(S \setminus \{i\} \cup \{j\})$
        $\Delta = z_{sol} - z_{int}$
        si $\Delta > \Delta_{max}$ **entonces**
            $\Delta_{max} = \Delta$
            $i^* = i$
            $j^* = j$
        fin si
    fin para todo
    si $\Delta_{max} > 0$ **entonces**
        $S = Intercambio(i^*, j^*)$
    fin si
hasta que $mejora\_max \leq 0$
S **es el conjunto solución.**

6.2. Neighborhood Search Multiarranque

Como se ha mencionado anteriormente, este algoritmo es sensible a la solución inicial S_{ini}, por lo que es recomendable iterar el algoritmo y quedarnos con la mejor solución obtenida. Este enfoque multiarranque es muy eficiente en problemas pequeños pero inabarcable en problemas de grandes dimensiones. En estos casos, la práctica más común es generar la solución inicial aleatoriamente en vez de usar un algoritmo *greedy*.

6.3. Resultados Neighborhood Search

				Greedy + NS			**Greedy + NS Multiarranque**		
Datos	n	p	Óptimo	Solución	Tiempo CPU	gap	Solución	Tiempo CPU	gap
pmed1	100	5	127	133	1.28	4.72 %	127	110.28	0 %
pmed2	100	10	98	109	2.54	11.22 %	100	203.11	2.04 %
pmed3	100	10	93	102	5.68	9.68 %	96	379.75	3.22 %
pmed4	100	20	74	94	7.67	27.03 %	79	344.56	6.76 %
pmed5	100	33	48	58	9.46	20.83 %	52	503.92	8.33 %
pmed6	200	5	84	91	7.67	8.33 %	84	1416.35	0 %
pmed7	200	10	64	79	14.78	23.44 %	68	2742.93	6.25 %
pmed8	200	20	55	65	27.55	18.18 %	61	5639.34	10.91 %
pmed9	200	40	37	49	31.01	35.14 %	44	6142.44	18.92 %
pmed10	200	67	20	29	45.41	55.00 %	26	9050.03	80.00 %
pmed11	300	5	59	60	32.14	1.69 %	59	9503.43	0 %
pmed12	300	10	51	74	68.50	45.10 %	60	18085.39	17.65 %
pmed13	300	30	36	51	165.26	41.67 %	44	45012.23	22.22 %
pmed14	300	60	26	36	214.79	38.46 %	-	-	- %
pmed15	300	100	18	24	117.52	33.33 %	-	-	- %
pmed16	400	5	47	48	131.45	2.13 %	-	-	- %
pmed17	400	10	39	44	163.51	12.82 %	-	-	- %
pmed18	400	40	28	39	682.29	39.29 %	-	-	- %
pmed19	400	80	18	28	401.81	55.56 %	-	-	- %
pmed20	400	133	13	19	648.66	46.15 %	-	-	- %

Tabla 6.1: Comparación del resultado óptimo con las heurísticas *Neighborhood Search* y *Neighborhood Search* Multiarranque y su tiempo de ejecución considerando diferentes valores de n y p

En la Tabla 6.1 se puede ver que NS obtiene mejoras para todos los resultados obtenidos mediante *greedy*, se queda muy cerca del óptimo en los problemas más sencillos, con un número de instalaciones p bajo. El coste computacional crece rápidamente según la dimensión de los problemas crece, especialmente cuando hay un p alto. El método multiarranque ofrece una mejora de las soluciones a cambio de un alto coste computacional. El coste computacional del método multiarranque con los datos pmed14-pmed20 es inasumible.

Capítulo 7

Metaheurísticas para el problema de p-centro

En el contexto del problema del p-centro, los algoritmos metaheurísticos [6] son técnicas de optimización que se utilizan para encontrar soluciones aproximadas o subóptimas a problemas de optimización combinatoria, como el problema del p-centro. Estos algoritmos son especialmente útiles cuando el espacio de búsqueda es muy grande y complejo, lo que hace que los métodos exactos sean computacionalmente costosos o incluso impracticables.

Los algoritmos metaheurísticos son técnicas generales y flexibles que se inspiran en principios naturales o en fenómenos observados en sistemas complejos para guiar la búsqueda hacia regiones prometedoras del espacio de soluciones. Algunos ejemplos de algoritmos metaheurísticos comúnmente utilizados incluyen:

- Algoritmos genéticos: Estos algoritmos están inspirados en la evolución biológica y operan con una población de soluciones candidatas que se someten a operadores como selección, cruce y mutación para generar nuevas soluciones.
- Recocido simulado (Simulated Annealing): Basado en el proceso metalúrgico de recocido, este algoritmo simula el enfriamiento gradual de un material para encontrar soluciones óptimas explorando el espacio de búsqueda de manera probabilística. Esta metaheurística tiene muchos derivados.
- Búsqueda tabú: Este método mantiene una lista de movimientos prohibidos (llamados tabúes) para evitar ciclos y estancamientos, permitiendo una exploración más efectiva del espacio de soluciones.
- Algoritmos de búsqueda local iterativa: Estos algoritmos comienzan desde una solución inicial y la mejoran iterativamente mediante movimientos locales que mejoran gradualmente la calidad de la solución.

7.1. GRASP

GRASP (Greedy Randomized Adaptive Search Procedure) es un algoritmo metaheurístico usado para encontrar soluciones cercanas al óptimo. GRASP esencialmente es la combinación de un algoritmo *Greedy* Aleatorizado al que se le aplica una mejora mediante cualquier tipo búsqueda local. GRASP normalmente está conformado por los siguientes pasos:

1. Inicialización: Se parte de una solución vacía $S = \emptyset$
2. Greedy aleatorizado: En cada iteración el método greedy aleatorizado construye una solución parcial.
3. Búsqueda local: Después de conseguir la solución parcial, se aplica una búsqueda local para mejorar la calidad de la solución.
4. Actualización de la solución: Si la solución obtenida después de la búsqueda local es mejor que la mejor solución hasta ahora, se convierte en la nueva mejor solución.
5. Criterio de parada: El algoritmo se repite hasta que se alcance condición de fin, ya sea un tiempo límite o un número de iteraciones máximas.

Podemos definir un *GRASP* básico de la siguiente forma:

Algoritmo 10 GRASP

```
S ← ∅
iter ← 0
mientras iter < iter_max hacer
    S ← GreedyAleatorizado(N, M)
    S' ← BusquedaLocal(S)
    si f(S') < f(S) entonces
        S ← S'
    fin si
    iter ← iter + 1
fin mientras
```

En mi caso, el número de iteraciones empleadas han sido 10 y he optado por implementar la búsqueda local mediante una mejora estocástica. Esta mejora estocástica consiste en buscar soluciones inmediatamente vecinas de forma aleatoria, se puede definir así:

Algoritmo 11 Búsqueda local: Mejora estocástica

$nFracasos = 0$
$final \leftarrow False$
$S \leftarrow \emptyset$
mientras $final = False$ **hacer**
  $S' \leftarrow IntercambioAleatorio(S)$
  si $f(S') < f(S)$ **entonces**
    $S \leftarrow S'$
    $nFracasos \leftarrow 0$
  si no
    $nFracasos \leftarrow nFracasos + 1$
  fin si
  si $nFracasos = fallosMax$ **entonces**
    $final \leftarrow True$
  fin si
fin mientras
return S

El número de fallos seguidos permitido por cada búsqueda local es 10000, para poder explorar el vecindario de la solución actual.

7.1.1. Resultados GRASP

En la Tabla 7.1 se muestra la comparación del resultado óptimo con el resultado obtenido mediante GRASP y su tiempo de ejecución.

Datos	n	p	Óptimo	GRASP	Tiempo CPU	gap
pmed1	100	5	127	127	58.92	0 %
pmed2	100	10	98	98	51.46	0 %
pmed3	100	10	93	96	50.26	3.23 %
pmed4	100	20	74	83	49.41	12.16 %
pmed5	100	33	48	57	58.25	18.75 %
pmed6	200	5	84	84	70.47	0 %
pmed7	200	10	64	69	71.60	7.81 %
pmed8	200	20	55	64	72.31	16.36 %
pmed9	200	40	37	48	76.27	29.73 %
pmed10	200	67	20	29	73.87	45.00 %
pmed11	300	5	59	59	76.78	0 %
pmed12	300	10	51	55	81.53	7.84 %
pmed13	300	30	36	45	126.64	25 %
pmed14	300	60	26	37	130.41	42.31 %
pmed15	300	100	18	23	133.65	27.78 %
pmed16	400	5	47	47	185.64	0 %
pmed17	400	10	39	43	200.09	10.26 %
pmed18	400	40	28	37	204.99	32.14 %
pmed19	400	80	18	27	201.56	50.00 %
pmed20	400	133	13	18	257.59	38.46 %

Tabla 7.1: Comparación del resultado óptimo con la metaheurística GRASP y su tiempo de ejecución considerando diferentes valores de n y p

Se podrían obtener ligeras mejoras en los resultados si el número de fallos por la mejora estocástica fuera mayor, especialmente en los problemas de mayor dimensión, pero el tiempo de cálculo sería excesivamente elevado. En los problemas de mayor dimension, la mejora estocástica funciona peor.

7.2. Simulated Annealing

Simulated Annealing (SA) es un método de optimización local para resolver problemas de optimización combinacionales. El interés en este método empezó en 1983 con la publicación de Kirkpatrick *et al* en *Science* [12]. Desde entonces, Simulated Annealing se ha empleado en un gran número de problemas de optimización. Especialmente en aquellos en los que se tiene un espacio de soluciones S y se quiere encontrar la solución $s \in S$ que optimice una función $f(s)$.

SA es un algoritmo atractivo ya que es fácil de implementar, generalmente consigue muy buenos resultados y se puede aplicar a gran diversidad de problemas. Sus mayores defectos son su gran coste computacional y la falta de memoria, puede que evalúe una misma solución una gran cantidad de veces.

Igual que el resto de algoritmos de búsqueda local, SA empieza con una solución inicial ya dada e intenta mejorar la solución moviéndose a soluciones vecinas con un mejor valor para la función objetivo. Una característica que le distingue de muchos algoritmos de búsqueda local es que, con el objetivo de no quedarse atascado en

nínimos locales, el algoritmo permite en ciertas ocasiones moverse a un vecino con un peor valor para la función objetivo bajo una cierta probabilidad que depende del parámetro *temperatura*.

El algoritmo generalmente empieza con una solución aleatoria o proporcionada por otro algoritmo más básico, por ejemplo, un algoritmo *greedy*. Se explora el vecindario de la solución, calculándose la función objetivo. Si se encuentra un mejor valor para la función objetivo, se reemplaza la solución actual por la solución vecina, en caso contrario, la solución vecina se acepta con una determinada probabilidad. Esta probabilidad viene dada por la diferencia entre los valores de la función objetivo de las dos soluciones candidatas Δ y la temperatura actual T: $prob = \exp(-\Delta/T)$. Es trivial deducir que será más probable aceptar soluciones cuya diferencia Δ sea menor, aquellas soluciones con un valor muy alejado al actual son penalizadas. Por otra parte, al principio del algoritmo, cuando T tiene un valor alto, la mayoría de movimientos serán aceptados mientras que, según vaya avanzando el algoritmo y la temperatura descendiendo, casi ningún movimiento será aceptado.

La temperatura T empieza con un valor alto al principio para que el algoritmo no se quede atrapado en mínimos locales. Para poder implementar SA, se necesitan fijar varios parámetros:

1. Temperatura inicial T_0
2. Funcion de reducción de la temperatura
3. Número de iteraciones L a hacer para cada valor de la temperatura T sin encontrar mejora
4. Criterio de parada del algoritmo

Para la reducción de la temperatura se fija un parámetro $\alpha = 0{,}9$ y cuando se llega al máximo de iteraciones L para $T = t$, se actualiza la temperatura:

$$T = \alpha \cdot T$$

En cuanto al criterio de parada, se escoge un número máximo de iteraciones. Cuanto mayor sea el tamaño del problema y el número p de instalaciones que abrir, se aumenta la temperatura inicial T_0 o se aumenta el número de iteraciones. Cuanto mayor sea el valor de estos parámetros, se explorarán mas soluciones y la solución obtenida será mejor pero mayor será el coste computacional.

El algoritmo se puede definir de la siguiente forma:

Algoritmo 12 Simulated Annealing

1: $S \leftarrow S_{ini}$
2: $S \leftarrow S_{opt}$
3: $T \leftarrow T_0$
4: $t \leftarrow 0$ Inicializamos el contador de cambio de temperatura
5: **repetir**
6: $t \leftarrow t + 1$
7: $S' \leftarrow Shake(S)$
8: **si** $f(S') < f(S)$ **entonces**
9: $S \leftarrow S'$
10: **si** $f(S') < f(S_{opt})$ **entonces**
11: $S_{opt} \leftarrow S'$
12: $t \leftarrow 0$
13: **fin si**
14: **si no**
15: **si** $t \leq L$ **entonces**
16: $S_{opt} \leftarrow S'$ con prob $= \exp(-\Delta/T)$, $\Delta = f(S') - f(S_{opt})$
17: **si no**
18: $T \leftarrow \alpha \cdot T$
19: $t \leftarrow 0$
20: **fin si**
21: **fin si**
22: **hasta que** $iter = max\_iter$
23: S^{min} **es el conjunto solución.**

La solución inicial S_{ini} se obtiene mediante el método *Greedy - 1.*

7.2.1. Resultados Simulated Annealing

En la Tabla 7.2 se muestra la comparación del resultado óptimo con el resultado obtenido mediante *Simulated Annealing* y su tiempo de ejecución.

				Simulated Annealing		
Datos	n	p	Óptimo	Solución	Tiempo CPU	gap
pmed1	100	5	127	127	30.00	0 %
pmed2	100	10	98	102	40.23	4.08 %
pmed3	100	10	93	98	32.52	5.38 %
pmed4	100	20	74	84	34.85	13.50 %
pmed5	100	33	48	54	41.67	12.50 %
pmed6	200	5	84	84	116.04	0 %
pmed7	200	10	64	66	130.48	3.13 %
pmed8	200	20	55	58	138.99	5.45 %
pmed9	200	40	37	40	140.52	8.11 %
pmed10	200	67	20	22	220.43	10.00 %
pmed11	300	5	59	59	142.97	0 %
pmed12	300	10	51	57	148.61	11.76 %
pmed13	300	30	36	41	410.50	13.89 %
pmed14	300	60	26	33	651.85	26.92 %
pmed15	300	100	18	23	954.49	27.78 %
pmed16	400	5	47	47	156.90	0 %
pmed17	400	10	39	41	594.24	5.13 %
pmed18	400	40	28	34	1004.16	21.43 %
pmed19	400	80	18	23	1893.38	27.78 %
pmed20	400	133	13	17	2229.77	30.77 %

Tabla 7.2: Comparación del resultado óptimo con la metaheurística *Simulated Annealing* y su tiempo de ejecución considerando diferentes valores de n y p

Obtiene el valor óptimo para los archivos más sencillos, en los que solo hay que ubicar 5 instalaciones. Obtiene mejores resultados que las heurísticas de construcción pero requiere un tiempo de computación mucho mayor. Se podrían obtener mejores resultados para los archivos con un gran número p, pero esto supondría un coste de computación enorme.

7.3. Threshold Accepting

Threshold Accepting (TA) es una metaheurística derivada de la recién vista Simulated Annealing, fue propuesta por Dueck y Scheuer [2]. Al igual que SA, es un algoritmo de optimización estocástico que busca optimizar una función objetivo buscando en el vecindario de la solución actual. Este algoritmo sigue una estructura aún más simple que SA ya que se elimina la probabilidad de aceptar una solución vecina con peor valor de la función objetivo que el actual.

Este elemento probabilístico es sustituido por un elemento determinista, el *threshold* (límite). TA aceptará una solución peor que la actual si su diferencia con la solución actual es menor o igual que el *threshold*. Los elementos claves que se debe fijar es la función de reducción del *threshold* a lo largo de la ejecución del método, el método de obtención de la solución inicial y la obtención de las soluciones vecinas de la

solución actual. La calidad de las soluciones obtenidas dependen mucho de estos elementos.

Los puntos fuertes del TA son muy similares al SA. Su simplicidad conceptual y su buen rendimiento en problemas de optimización combinacional destacan. Se puede definir el método de la siguiente forma:

Algoritmo 13 Threshold accepting

```
1: S ← S_ini
2: S_opt ← S_ini
3: T = T_0
4: L número máximo de iteraciones
5: mientras T > T_min hacer
6:     iter = 0
7:     mientras iter < L hacer
8:         iter = iter + 1
9:         S' = Shake(S)
10:        si f(S') < f(S) + T entonces
11:            S = S'
12:            si f(S') < f(S_opt) entonces
13:                S_opt = S'
14:            fin si
15:        fin si
16:    fin mientras
17:    T = α · T
18: fin mientras
19: S^min es el conjunto solución.
```

Al igual que en SA, se obtiene una buena solución inicial mediante el método *Greedy - 1*. Al final del bucle exterior, se actualiza el *threshold* multiplicándolo por un factor α. La definición de vecindad se mantiene igual que en los métodos anteriores.

Al eliminar el cálculo de probabilidades, se consigue una reducción del coste computacional y, en general, obtiene mejores resultados que SA.

7.3.1. Resultados Threshold Accepting

En la Tabla 7.3 se muestra la comparación del resultado óptimo con el resultado obtenido mediante *Threshold Accepting* y su tiempo de ejecución.

				Threshold Accepting		
Datos	n	p	Óptimo	Solución	Tiempo CPU	gap
pmed1	100	5	127	127	5.43	0 %
pmed2	100	10	98	98	6.22	0 %
pmed3	100	10	93	94	7.19	0 %
pmed4	100	20	74	77	8.26	4.05 %
pmed5	100	33	48	48	9.82	0 %
pmed6	200	5	84	84	25.29	0 %
pmed7	200	10	64	64	25.57	0 %
pmed8	200	20	55	57	31.14	3.64 %
pmed9	200	40	37	40	34.20	8.11 %
pmed10	200	67	20	21	438.27	5.00 %
pmed11	300	5	59	59	193.76	0 %
pmed12	300	10	51	51	198.16	0 %
pmed13	300	30	36	38	213.05	5.56 %
pmed14	300	60	26	29	710.22	11.54 %
pmed15	300	100	18	20	800.99	11.11 %
pmed16	400	5	47	47	142.57	0 %
pmed17	400	10	39	39	432.44	0 %
pmed18	400	40	28	30	806.99	7.14 %
pmed19	400	80	18	22	1443.13	22.22 %
pmed20	400	133	13	17	1983.27	30.77 %

Tabla 7.3: Comparación del resultado óptimo con la metaheurística *Threshold Accepting* y su tiempo de ejecución considerando diferentes valores de n y p

Se obtienen los resultados óptimos para todos los datos con 5 o 10 puntos de servicio a localizar. Es notable el aumento de tiempo de computación al aumentar el tamaño del problema, especialmente cuando el número p de instalaciones también es alto. Esta metaheurística podría haber conseguido mejores resultados que los obtenidos para los archivos con mayor tamaño (pmed14 - pmed20) si se le hubiera permitido más iteraciones, pero el coste computacional sería enorme.

7.4. Tabu Search

Tabu Search o Búsqueda Tabú es una metaheurística propuesta por Glover [5] [7] y formalizada en 1989. Partiendo de una solución inicial, realiza una búsqueda local en los vecinos de la solución actual manteniendo una lista llamada *tabú* de elementos que no se pueden modificar durante un número fijado de iteraciones. En el caso del p-centro, se usa la misma relación de vecindad explicada anteriormente.

Uno de los mayores problemas de las búsquedas locales es quedarse atascado en un mínimo local o en mesetas, para solucionarlo, la búsqueda tabú permite al principio de cada paso aceptar una solución que pueda empeorar la solución. Además, tiene memoria a corto plazo de las soluciones recientemente visitadas, estas son introducidas en una lista tabú y no se permite modificarlas durante un determinado tiempo. En verdad, tener una memoria de todas las soluciones visitadas sería muy costoso,

en el problema del p-centro simplemente se guardan los últimos puntos que se han intentado (o conseguido) añadir a la solución. Si en una iteración no se consigue encontrar un mejor valor de la función objetivo, se realiza un intercambio aleatorio para que el algoritmo no se quede estancado.

La lógica detrás de los elementos tabú, es comprobar cuál es el efecto de introducir un elemento en la solución. Si se ha añadido recientemente un elemento y las soluciones vecinas desde su inclusión obtienen mejores valores para la función objetivo, se considera un buen movimiento. Se consideraría un mal movimiento en caso contrario.

Se puede definir el algoritmo de la siguiente forma:

Algoritmo 14 Búsqueda Tabú

$S \leftarrow S_{ini}$
$S_{min} \leftarrow S_{ini}$
$LT \leftarrow \{\}$
L número máximo de iteraciones
S^{min} **es el conjunto solución.**
$iter \leftarrow 0$
mientras $iter < L$ **hacer**
    $iter \leftarrow iter + 1$
    $f_{min} = \infty$
    para todo $i \notin LT, i \in S,\ j \notin LT,\ j \notin S$ **hacer**
        $S' = S \cup \{j\} \setminus \{i\}$
        si $f(S') < f_{min}$ **entonces**
            $S_{min} = S'$
            $i^* \leftarrow i$
            $j^* \leftarrow j$
        fin si
    fin para
    si $f(S_{min}) < f(S)$ **entonces**
        $S \leftarrow S_{min}$
    si no
        $Shake(S)$
    fin si
    $LT \leftarrow LT \cup \{i^*, j^*\}$
    Eliminar de LT los elementos que ya no son tabú
fin mientras
S es el conjunto solución

Es fundamental elegir un buen valor para el **tabu tenure** o **duración tabú**. Este parámetro determina cuánto tiempo permanece un movimiento en la lista tabú después de ser aplicado. Puede ser un valor fijo o ajustarse dinámicamente según ciertos criterios. Una duración de tabú más corta fomenta la exploración de soluciones diversas, mientras que una duración de tabú más larga puede ayudar al algoritmo a escapar de óptimos locales. Conociendo las características del problema del p-centro, se sabe que hay más óptimos locales y mesetas cuanto mayor sea el valor de p por lo que hay que aumentar el valor de $tabuTenure$ en consecuencia.

7.4.1. Resultados Búsqueda Tabú

En la Tabla 7.3 se muestra la comparación del resultado óptimo con el resultado obtenido mediante *Tabu Search* y su tiempo de ejecución.

				Threshold Accepting		
Datos	n	p	Óptimo	Solución	Tiempo CPU	gap
pmed1	100	5	127	127	5.43	0 %
pmed2	100	10	98	98	6.22	0 %
pmed3	100	10	93	94	7.19	0 %
pmed4	100	20	74	74	8.26	0 %
pmed5	100	33	48	48	9.82	0 %
pmed6	200	5	84	84	25.29	0 %
pmed7	200	10	64	64	25.57	0 %
pmed8	200	20	55	55	31.14	0 %
pmed9	200	40	37	37	34.20	0 %
pmed10	200	67	20	21	50.36	5.00 %
pmed11	300	5	59	59	45.93	0 %
pmed12	300	10	51	51	54.56	0 %
pmed13	300	30	36	36	60.27	0 %
pmed14	300	60	26	26	75.45	0 %
pmed15	300	100	18	20	90.03	11.11 %
pmed16	400	5	47	47	82.75	0 %
pmed17	400	10	39	39	104.93	0 %
pmed18	400	40	28	30	169.11	0 %
pmed19	400	80	18	19	240.39	5.56 %
pmed20	400	133	13	15	304.46	15.38 %

Tabla 7.4:

Tabla 7.5: Comparación del resultado óptimo con la metaheurística *Tabu Search* y su tiempo de ejecución considerando diferentes valores de n y p

Búsqueda tabú es la metaheurística que mejores resultados obtiene de todas las empleadas en este trabajo. El uso de la lista tabú ayuda a obtener mejores soluciones al evitar que el algoritmo se quede atascado en una meseta. Además, también reduce el tiempo de cómputo drásticamente respecto a SA o TA ya que se prueban muchas menos soluciones vecinas al evitar repetir soluciones ya visitadas recientemente.

Capítulo 8

Conclusiones

En este trabajo se ha estudiado el problema del p-Centro y diferentes métodos que permiten obtener soluciones aproximadas a la solución óptima. La implementación de todos estos métodos se puede consultar accediendo al `GitLab` cuyo enlace figura en el apéndice A. Resolviendo el problema con la formulación clásica mediante el *solver* de Xpress, se obtienen las formulaciones exactas, pero de una forma muy ineficiente, tanto que la computadora se queda sin memoria cuando se intenta resolver un problema de tamaño medio. Debido a esto, se han empleado dos enfoques distintos para resolver el problema: búsquedas dicotómicas (binarias) ayudándose del problema de *Set Covering* asociado o heurísticas de construcción y metaheurísticas. La búsqueda dicotómica clásica obtiene buenos resultados en un tiempo medio de solo 7.48s para los 40 archivos con los que se ha experimentado. Con las cotas que nos brinda el algoritmo *greedy*, se consigue una media de 5.35s al realizar la búsqueda, se obtiene por tanto una mejora del 28.47 %. Obteniendo la solución exacta de la formulación alternativa Elloumi-Labbé-Pochet, se llega al mismo problema que con la formulación clásica. Sin embargo, esta formulación nos brinda una nueva forma de realizar la búsqueda dicotómica; el algoritmo *BSearch*. Este algoritmo consigue obtener un tiempo de cómputo medio inferior al de la búsqueda dicotómica básica: 4.98s. Sin embargo, *BSearch* no aporta la solución exacta, sino una estimación muy cercana (la cota superior), por ello se usa *BSearchEx*, que se ejecuta en un tiempo medio de 6.43s. Es importante remarcar que el tiempo de ejecución disminuye en *BSearchEx* respecto a *BSearch* en todos los problemas menos en la mayoría de aquellos en los que *BSearch* no converge. Ilhan-Pinar es el último método que se basa en una búsqueda dicotómica que se ha usado en el trabajo. Este algoritmo obtiene los mejores resultados que se han obtenido en este trabajo a nivel de coste computacional. El tiempo medio de ejecución sobre los 40 archivos de experimentación es de apenas 1.67s, aproximadamente un tercio del tiempo de *BSearch* y la búsqueda dicotómica mejorada. Estos algoritmos consiguen, apoyándose en el problema del *Set Covering*, reducir las dimensiones. Se tratan de algoritmos *Divide y vencerás*.

Si bien estos métodos mencionados son los que mejor funcionan, se necesita de un *solver* de Programación Entera para obtener las soluciones exactas de los subproblemas de *Set Covering*, este *software* tiene un coste de adquisición muy alto y no es fácilmente accesible. Existen heurísticas y metaheurísticas efectivas e independientes de un *software* externo, lo que permite resolver problemas del p-Centro sin necesidad de hacer una inversión monetaria.

La heurística *greedy* o voraz nos aporta una estimación muy rápida aunque no de

muy buena calidad. Sin embargo, nos interesa su característica de ser un algoritmo 2-aproximación. No alcanza la solución óptima en ninguna de las 20 instancias con ninguna de sus dos variantes. Como *greedy* es muy sensible al punto inicial que se escoja, se emplea el método *greedy* multiarranque. Nos proporciona mejores resultados pero emplea más tiempo. Especialmente el método *greedy - 2* multiarranque no parece muy útil por su alto coste computacional, ya que en cada iteración debe recorrer todo el espacio de posibles puntos de servicio. El algoritmo *greedy* aleatorizado obtiene resultados mucho mejores que los anteriores con un coste computacional bajo, solo 0.22s de media. Es un algoritmo muy eficaz si se quiere obtener una buena estimación rápidamente.

Prácticamente todas las metaheurísticas en el problema del p-Centro se basan en la búsqueda local. De las cuatro empleadas, las dos más destacadas son Threshold Accepting y Tabu Search. Threshold Accepting llega al óptimo en la mitad de los problemas y el gap es de 10,91 % en la mitad en la que no llega a la solución óptima. Tabu Search por otra parte alcanza la solución óptima en 16 de los 20 problemas y el gap medio es de 9,26 %. Fijándonos en estos parámetros, Tabu Search funciona mejor, pero la mayor diferencia entre Tabu Search y Threshold Accepting es su eficiencia computacional, TA obtiene un tiempo medio de ejecución de 376s mientras el de TS es de apenas 71s. Simulated Annealing obtiene resultados ligeramente peores que Threshold Accepting en un tiempo de computación. De misma forma GRASP obtiene resultados un poco peores que Simulated Annealing pero en este caso, GRASP requiere mucho menos coste computacional.

En general, a pesar de que se tenga acceso a un *solver* de programación entera, no es recomendable emplear métodos exactos basados en la formulación clásica para resolver el problema del p-Centro, pues son muy lentos y se vuelven inviables en problemas de tamaño medio o grande. La mejor solución encontrada en este trabajo es emplear los algoritmos de búsqueda dicotómica apoyados en el problema de *Set Covering*. Cualquiera de ellos funciona mejor que el resto de métodos empleados en este trabajo, especialmente Ilhan-Pinar destaca sobre las otras dos búsquedas binarias. Sin embargo, para emplear estos métodos se requiere un *solver*. En caso de no tenerlo, se pueden emplear metaheurísticas, dentro de estas destaca Tabu Search por obtener los mejores resultados en el mínimo tiempo en comparación con el resto de metaheurísticas que se han probado. Si se quiere optar por obtener una buena estimación en el mínimo tiempo posible, *greedy* aleatorizado es la mejor opción.

Apéndice A

Software programado

Todos los archivos *mosel* que se han desarrollado a lo largo del TFG con el fin de experimentación han sido alojados en el *GitLab* de la Universidad de Valladolid. Se pueden consultar accediendo a siguiente página: Código desarrollado. Todos los nombres de los archivos son autodescriptivos, cada archivo tiene como nombre el método que implementa. El archivo *lectura_datos.mos* es el único que no implementa un método para la resolución del problema del p-Centro, sino que contiene el algoritmo de Floyd-Warshall para la construcción de la matriz de distancias a partir de los datos de los ficheros de datos de la *OR-Library*.

Bibliografía

[1] M. S. Daskin. *Network and Discrete Location: Models, Algorithms, and Applications*. John Wiley & Sons, Hoboken, NJ, 2nd edition, 2013.

[2] G. Dueck and T. Scheuer. Threshold accepting: A general purpose optimization algorithm appearing superior to simulated annealing. *Journal of Computational Phyisics*, 90:161–175, 1990.

[3] S. Elloumi, M. Labbé, and Y. Pochet. A new formulation and resolution method for the p-center problem. *INFORMS Journal on Computing*, pages 84–94, 2004.

[4] D. Ferone, P. Festa, A. Napoletano, and M. Resende. A new local search for the p-center problem based on the critical vertex concept. *Learning and Intelligent Optimization*, pages 79–92, 2017.

[5] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research*, 13:pages 533–549, 1986.

[6] F. Glover and G. A. Kochenberger. *Handbook of Metaheuristics*. Springer, Boston, MA, 2003.

[7] F. Glover and M. Laguna. *Tabu Search*. Springer, Boston, MA, 1997.

[8] S. Hakimi. Optimum locations of switching centers and the absolute centers and medians of a graph. *Oper Res*, 12:450—-459, 1964.

[9] S. L. Hakimi. Optimum distribution of switching centers in a communication network and some related graph theoretic problems. *Oper Res*, 13:462—475, 1965.

[10] T. Ilhan and M. C. Pinar. An Efficient Exact Algorithm for the Vertex p-Center Problem. *Preprint[Online]*, 2001.

[11] O. Kariv and S. L. Hakimi. An algorithmic approach to network location problems. i: The p-centers. *SIAM Journal on Applied Mathematics*, 37:pages 513–538, 1979.

[12] G. C. Kirckpatrick S. and V. M. Optimization by simulated annealing. *Science*, 220:pages 671–680, 1983.

[13] E. Minieka. The m-center problem. *SIAM Rev.*, pages 138–139, 1970.

[14] M. B. Teitz and P. Bart. Heuristic methods for estimating the generalized vertex median of a weighted graph. *Operations Research*, 16:pages 955–961, 1968.