Universidad de Valladolid

FACULTAD DE CIENCIAS

TRABAJO FIN DE GRADO

Grado en Estadística

# EL PROBLEMA DE LA MOCHILA MULTIDIMENSIONAL (MKP). MÉTODOS DE SOLUCIÓN Y APLICACIÓN A PROBLEMAS DE SELECCIÓN DE PROYECTOS.

Autor: Ricardo García Lara
Tutores: Ricardo Josa Fombellida y Jesús Sáez Aguado
2023

# Agradecimientos

Me gustaría hacer un agradecimiento a todas aquellas personas que me han acompañado durante esta etapa universitaria.

Muchas gracias a mis familiares y amigos por estar a mi lado durante estos años, tanto en los momentos buenos como en los malos.

También agradecer a todos los profesores del Grado por los conocimientos que me han aportado y en especial a Ricardo Josa y Jesús Sáez, que me han ayudado en el desarrollo de este Trabajo de Fin de Grado.

# Resumen

En este trabajo fin de grado se realiza un estudio teórico sobre los diferentes problemas MKP (multidimensional knapsack problem), junto con un desarrollo práctico en el que se implementan diferentes algoritmos para la resolución de dichos problemas. Para la aplicación práctica de dicho trabajo se utilizarán los datos de la librería ORLIB http://people.brunel.ac.uk la cual cuenta con una serie de problemas MKP los cuales están resueltos en varios artículos que tratan este tema. Además, se ha construido un algoritmo para la lectura de dichos datos de forma que podamos identificar y seleccionar de manera correcta los datos del problema. Finalmente se ha realizado una comparación entre los diferentes algoritmos utilizados para determinar cuál de ellos tiene un mejor funcionamiento dependiendo del tamaño del problema y las características del mismo.

**Palabras clave:** Problema de la Mochila Multidimensional, algoritmos greedy, aplicación a la selección de proyectos, programación lineal entera.

**ABSTRACT:** In this final degree thesis a theoretical study of the different MKP (multidimensional knapsack problem) problems is carried out, together with a practical development in which different algorithms are implemented for the resolution of these problems. For the practical application of this work, data from the ORLIB library http://people.brunel.ac.uk will be used, which has a series of MKP problems that have been solved in several articles on this subject. In addition, an algorithm for reading the data has been constructed so that we can identify and select the data of the problem correctly. Finally, a comparison has been made between the different algorithms used to determine which of them performs better depending on the size of the problem and the characteristics of the problem.

**Keywords:** Multidimensional Knapsack Problem, greedy algorithms, application to project selection, integer linear programming.

# Objetivos

El objetivo del trabajo será, en primer lugar, dar una explicación teórica del problema de la mochila y sus diferentes variantes, incluyendo como han ido evolucionando a lo largo de la historia las diferentes técnicas de resolución del mismo. Para finalizar se tratará de resolver una serie de dichos problemas con la aplicación de cuatro algoritmos diferentes y dar una comparativa de los mismos para ver su eficiencia.

# Índice General

# 1. El problema de la Mochila Multidimensional (MKP)

Durante este capítulo hablaremos sobre las diferentes variantes que tiene el problema MKP, se expondrán todas ellas junto con una explicación individual acompañada de la formulación de un modelo.

Además, se darán diferentes aplicaciones que puede tener el problema MKP en situaciones reales.

Todo esto viene perfectamente explicado en el artículo, *Laabadi et al. (2018).* Donde nos enseña perfectamente las variantes de estos problemas.

Las diferentes variantes que puede tener el problema MKP vienen dadas por las restricciones del problema, los tipos de variables, los tipos de datos, las dimensiones de la bolsa, los objetivos del problema, etc.

Según los tipos de datos, existen dos versiones de MKP: MKP determinista y MKP no determinista. En la primera versión, se supone que todos los datos se conocen de antemano. Sin embargo, no es así en la segunda versión.

Los diferentes tipos de problema MKP que estudiaremos serán: MKP estándar, MKP múltiple (MMKP), MKP con elección múltiple (MCMKP), MKP multiobjetivo (MOMKP), MKP estocástico y MKP difuso, siendo estos dos últimos no deterministas por su tipo de datos.

## 1.1 Variantes deterministas de MKP

### 1.1.1 MKP estándar

Dados $N$ artículos con beneficios $p_i > 0$ y una mochila con $d$ dimensiones. Cada artículo $i$ consume una cantidad $w_{ji} \geq 0$ de cada dimensión $j$. Sabiendo que cada dimensión tiene una capacidad $C_j > 0$, el objetivo es maximizar la suma de beneficios de los artículos de la mochila de forma que la suma de pesos en cada dimensión $j$ no supere $C_j$. Formalmente, el MKP estándar podría expresarse con un modelo de programación entera:

$$\begin{cases} Max \sum_{i=1}^{N} p_i x_i & (1) \\ s.t \sum_{i=1}^{N} w_{ji} x_i \leq C_j ; \forall j = 1, \cdots, d; & (2) \\ x_i \in \{0,1\}; \forall i = 1, \cdots, N; & (3) \end{cases}$$

La ecuación (2) es la restricción de capacidad de los recursos.

La ecuación (3) indica que $x_i$ es una variable de decisión binaria, igual a 1 si se selecciona el elemento *i-ésimo* y 0 en caso contrario.

Si $d = 1$, el MKP se reduce al problema de Knapsack:

$$\begin{cases} Max\sum_{i=1}^{N} p_i x_i & (4) \\ s.t\sum_{i=1}^{N} w_i x_i \le C; & (5) \\ x_i \in \{0,1\}; \forall i = 1,\cdots,N; & (6) \end{cases}$$

### 1.1.2 MKP múltiple

El MKP múltiple (MMKP) difiere del MKP en el número de mochilas. En lugar de una única mochila, consideramos múltiples mochilas en las que cada una, digamos *k*, tiene *d* dimensiones con capacidad limitada $C_{jk}$. Además, cada artículo i consume una cantidad $w^k$ de cada dimensión *j* de cada mochila *k*. La decisión no es sólo si se selecciona un artículo, sino también en qué mochila se empaqueta. Del mismo modo, introducimos una variable binaria de $x_{ik}$ =1 para representar que el artículo i ha sido seleccionado y empaquetado en la mochila k. Matemáticamente, el MMKP viene dado por:

$$\begin{cases} Max\sum_{k=1}^{M}\sum_{i=1}^{N} p_i x_{ik} & (7) \\ s.t\sum_{i=1}^{N} w_{ij}^k x_{ik} \le C_j^k; \forall j = 1,\cdots,d; \forall k = 1,\cdots,M; & (8) \\ \sum_{k=1}^{M} x_{ik} \le 1; \forall i = 1,\cdots,N; & (9) \\ x_{ik} \in \{0,1\}; \forall i = 1,\cdots,N; k = 1,\cdots,M; & (10) \end{cases}$$

La ecuación (8) significa que el tamaño total de los artículos no puede superar la capacidad de la dimensión de cada mochila.

La ecuación (9) garantiza que cada artículo aparezca como máximo una vez en todas las mochilas.

### 1.1.3 MKP con elección múltiple

El MKP de elección múltiple (MCMKP) es una variante más compleja del MKP. En

efecto, se nos da un conjunto de elementos *N* dividido en *n* grupos disjuntos $N = N_1 \cup N_2$

$\cup N_n$, para todo k $k \neq k \in \{1,2,\ldots,n\}$ tenemos $N_k \cap N_{k'} = \emptyset$.

Cada elemento *i*, $i \in N_k$, tiene un beneficio $p_{ik} > 0$, y requiere recursos dados por el vector de pesos $w_{ik}$ ( $w_{ik1}$, $w_{ik2}$,..., $w_{ikd}$) donde *d* es la dimensión de la mochila. Las cantidades de recursos disponibles vienen dadas por *Cj*. El objetivo del MCMKP es elegir exactamente un artículo de cada clase para maximizar el beneficio total de la mochila, sin violar la restricción de capacidad de cada dimensión. Del mismo modo, introducimos una variable binaria $x_{ik}$ que es igual a 1, si se selecciona el artículo *j* de la clase *k-ésima*, e igual a 0 en caso contrario.

Formalmente, el MCMKP puede plantearse de la siguiente manera:

$$\begin{cases} Max \sum_{k=1}^{n} \sum_{i \in N_k} p_{ik} x_{ik} & (11) \\ s.t \sum_{k=1}^{n} \sum_{i \in N_k} w_{ik}^{j} x_{ik} \le C_j; \forall j = 1, \cdots, d; & (12) \\ \sum_{i \in N_k} x_{ik} = 1; \forall k = 1, \cdots, n; & (13) \\ x_{ik} \in \{0,1\}; \forall k = 1, \cdots, n; i \in N_k; & (14) \end{cases}$$

La ecuación (12) significa que la suma de los elementos seleccionados no puede superar la capacidad de cada dimensión de la mochila.

La ecuación (13) garantiza que sólo se seleccione un artículo de cada clase.

### 1.1.4 MKP Multiobjetivo

El MKP Multiobjetivo (MOMKP) es un MKP con objetivos (o criterios) contradictorios. De hecho, se nos da una mochila con *d* dimensiones, un conjunto de *N* y un conjunto de *M* objetivos, donde $p^k$ es el beneficio del artículo *i* en relación con el objetivo *k* cuando se selecciona para la *j-ésima* dimensión de la mochila, $w_{ij}$ es el peso del artículo *i* cuando se selecciona para la *j-ésima* dimensión de la mochila, y $C_j$ es la capacidad de la *j-ésima* dimensión de la mochila. El objetivo es seleccionar un subconjunto de elementos de forma que el beneficio total de los elementos seleccionados para cada objetivo sea máximo, respetando la restricción de capacidad de cada dimensión de la mochila. Así, el MOMKP puede formularse de la siguiente manera:

$$\begin{cases} Max\ z_k = \sum_{i=1}^{N} \sum_{j=1}^{d} p_{ij}^{k} x_{ij}, \forall k = 1, \cdots, M; & (15) \\ s.t \sum_{i=1}^{N} w_{ij} x_{ij} \le C_j; \forall j = 1, \cdots, d; & (16) \\ x_{ij} \in \{0,1\}; \forall i = 1, \cdots, N;\ j = 1, \cdots, d; & (17) \end{cases}$$

La ecuación (16) es la restricción de capacidad de las dimensiones de la mochila.

La ecuación (17) significa que *xij* es una variable de decisión binaria, igual a 1 si el artículo *i* se selecciona para la dimensión *j-ésima* de la mochila, y 0 en caso contrario.

## 1.2 Variantes no deterministas de MKP

### 1.2.1 MKP Estocástico

En una versión estocástica del MKP, suponemos que los tamaños de los artículos son variables aleatorias independientes que cada tamaño sigue el mismo tipo de distribución de probabilidad,

no necesariamente con el mismo parámetro. Se impone una restricción probabilística conjunta a las restricciones de capacidad y la función objetivo es la misma que la del problema determinista. Denotamos los tamaños de los artículos por $\xi_{ji}$ (en lugar de $w_{ji}$), y formulamos el problema como una programación estocástica con restricciones probabilísticas, en la que las restricciones de capacidad del MKP "Ecuación (2)" se sustituyen por la siguiente restricción probabilística conjunta:

$$\begin{cases} Max \sum_{i=1}^{N} p_i x_i & (18) \\ Pr\left( \sum_{i=1}^{N} x_i \xi_{ji} \le C_j \right) \ge q; \forall j = 1, \cdots, d; & (19) \\ x_i \in \{0,1\}; \forall i = 1, \cdots, N; & (20) \end{cases}$$

donde q ∈ [0,1] es un nivel de probabilidad fijo (por ejemplo, 0,9, 0,95). Suponiendo que las variables aleatorias $\xi_{ji}$ son independientes, la restricción probabilística conjunta "Ecuación (19)" puede escribirse como sigue:

$$\prod_{j=1}^{d} Pr\left( \sum_{i=1}^{N} x_i \xi_{ji} \le C_j \right) \ge q. \qquad (21)$$

Nótese que, en algunas aplicaciones de la vida real, no podemos suponer que los vectores aleatorios sean independientes. En este caso, tenemos la siguiente desigualdad:

$$Pr\left( \sum_{i=1}^{N} x_i \xi_{ji} \le C_j \text{ for } j = 1, \cdots, d \right) \ge \prod_{j=1}^{d} Pr\left( \sum_{i=1}^{N} x_i \xi_{ji} \le C_j \right) \ge q. \qquad (22)$$

Entonces, sustituyendo la "Ecuación (19)" por la "Ecuación (21)", el problema queda como sigue:

$$\begin{cases} Max \sum_{i=1}^{N} p_i x_i & (23) \\ \prod_{j=1}^{d} Pr\left( \sum_{i=1}^{N} x_i \xi_{ji} \le C_j \right) \ge q; & (24) \\ x_i \in \{0,1\}; \forall i = 1, \cdots, N; & (25) \end{cases}$$

### 1.2.2 MKP Difuso

Hay muchos problemas de tipo knapsack que implican elementos cuyos pesos o beneficios no se conocen con precisión. Uno de los métodos para tratar la imprecisión es aplicar la teoría de los conjuntos difusos. Así, los números difusos se han aplicado a modelar los pesos y beneficios imprecisos en el MKP. El objetivo de esta variación del MKP es alcanzar un determinado nivel de beneficio aceptado sin exceder una determinada capacidad de cada dimensión de la mochila. Sea $\tilde{p}_i$ un intervalo difuso que modela el beneficio impreciso del artículo $i$ y sea $w_{ji}$ un intervalo difuso que modela el peso impreciso del artículo $i$ en la dimensión $j$. Así, el MKP difuso (FMKP) puede enunciarse como sigue:

$$\begin{cases} Max \sum_{i=1}^{N} \tilde{p}_i x_i & (26) \\ \sum_{i=1}^{N} x_i w_{ji} \le C_j; \forall i = 1, \cdots, N; \forall j = 1, \cdots, d; & (27) \\ x_i \in \{0,1\}; \forall i = 1, \cdots, N; & (28) \end{cases}$$

## 1.3 Aplicaciones

Son muchos los problemas que pueden surgir en escenarios reales y que podemos modelar como un MKP, en este apartado daremos alguno de los ejemplos los cuales tratan diferentes áreas de negocio: Logística, informática, finanzas, etc.

### 1.3.1 Subastas combinatorias multiunidades

Las subastas combinatorias son ventas públicas de bienes al mejor postor, de hecho, la subasta se realiza en dos pasos principales. En primer lugar, los participantes presentan sus ofertas. En segundo lugar, el subastador se enfrenta a un problema de asignar precios a las propiedades, con el fin de maximizar sus ingresos, que son relativos a la suma de todas las ofertas presentadas por los participantes y aceptadas por el subastador. Este problema se denomina Problema de Determinación del Ganador (WDP). Una variante destacada de las CAs (Subastas combinatorias) es la subasta combinatoria multiunidad (MUCA). Difiere de las CAs, o más exactamente de las CAs de una unidad, en el número de copias del mismo tipo de bien. En las CA de una sola unidad, solo existe una única unidad.

En el artículo *Pfeiffer et al (2017).* Se formula el problema WDP como una variante del MKP. El beneficio $p_i$ usado en el MDKP corresponde al precio de la *i- esima* puja. $u_j$ representa la cantidad disponible de la unidad *j*, mientras que el consumo de recursos $r_{ij}$ se considera el número de unidades *j* solicitadas en la *i-ésima* puja. La variable de decisión $x_i$ es igual a 1 si la *i-ésima* puja es aceptada por el subastador, y es igual a 0 en caso contrario. Por tanto, la formulación puede ser la siguiente:

$$\begin{cases} Max \sum_{i} p_i x_i & (29) \\ s.t \sum_{i} r_{ij} x_i \le u_j; \forall j; & (30) \\ x_i \in \{0,1\}; \forall i; & (31) \end{cases}$$

A pesar de la estrecha relación entre el MKP y el WDP, la literatura aborda ambos problemas de forma independiente. Holte, R.C. fue el primero en establecer esta relación.
Recientemente, *Kelly (2004)* introdujo una comparación directa entre el WDP de los distintos tipos de subastas y su correspondiente familia de knapsack, demostrando la eficacia de los algoritmos desarrollados para el MKP para resolver el WDP.

### 1.3.2 Asignación de recursos con demandas estocásticas

El conocido problema de la atribución de frecuencias que consiste en asignar el espectro

electromagnético a bandas de frecuencia se hace cada vez más complejo debido a la popularidad y complejidad de las aplicaciones en red más recientes. Éstas se apoyan en una gran variedad de servicios de sistemas finales en la nube y en diferentes tipos de redes. Los recursos de los sistemas informáticos distribuidos incluyen los de dispositivos finales como la CPU, la memoria y el disco, así como los del sistema en red, como conmutadores y enrutadores. Para dar servicio a múltiples usuarios que ejecutan simultáneamente, necesitamos satisfacer sus peticiones sin violar la capacidad de los recursos. En otras palabras, consideramos el sistema informático como una unidad que tiene recursos limitados y sólo admite un subconjunto de usuarios/tareas al mismo tiempo. Asignamos peso y beneficio a cada usuario. Así, el objetivo es responder a solicitudes máximas de usuarios o tareas sin sobrepasar las limitaciones de recursos. *Chen et al (2012)* tratan la asignación de recursos cuando las demandas pueden cambiar con el tiempo y sólo cuando sus estadísticas o distribuciones se conocen a priori. Han modelado este problema de asignación como un MKP estocástico, donde los recursos corresponden a las dimensiones de la mochila, mientras que los usuarios corresponden a los artículos que deben almacenarse en la mochila. Cada usuario $j$ solicita $A_{ij}$ de los recursos $i$, donde $A_{ij}$ es una matriz aleatoria basada en una cierta distribución o que tiene algunas estadísticas conocidas, suponemos que: $A_{ij}$ y $A_{ik}$ son independientes si $j \neq k$ , si tienen demandas correlacionadas, las consideramos como una sola tarea y fusionamos sus demandas. Los beneficios utilizados en el MKP corresponden a las demandas satisfechas $c_j$ de cada usuario $j$, la capacidad $b_i$ de cada recurso $i$ representa el subconjunto de usuarios que pueden ser admitidos al mismo tiempo por el recurso *i-ésimo*. Dejemos que $p$ denote la probabilidad de desbordamiento que indica la frecuencia máxima en la que los usuarios/tareas admitidos pueden violar las restricciones de capacidad. La variable de decisión binaria $x_j$ es igual a 1 si se satisfacen las demandas del usuario $j$, e igual a 0 en caso contrario. Así, el problema se formula de la siguiente manera:

$$\begin{cases} Max \sum_j c_j x_j & (32) \\ s.t\ Pr\left( \sum_j A_{ij} x_j > b_i \right) \leq p; \forall i; & (33) \\ x_i \in \{0,1\}; \forall j; & (34) \end{cases}$$

Este problema puede resolverse mediante métodos probabilísticos que utilizan la aproximación por escenarios o la aproximación media simple (SAA). Para algunas distribuciones como Bernoulli, el cálculo de la probabilidad $Pr(\sum_j A_{ij} x_j > b_i)$ es muy difícil.

### 1.3.3 Problema del presupuesto de capital

El problema del presupuesto de capital consiste en seleccionar los proyectos, como la inversión en I+D o la apertura de una nueva sucursal, que merece la pena llevar a cabo. Este problema es uno de los principales temas de investigación e interés en la gestión de proyectos. Este problema puramente financiero fue una de las primeras aplicaciones del MKP en la literatura. Existe toda una literatura que establece el vínculo entre el MKP y la presupuestación de capital para beneficiarse de sus sólidos enfoques. *Khalili-Damghani, K. y Taghavifard, M.* modelizaron el problema como un FMKP, considerando el beneficio aportado por cada proyecto como incierto

y persiguiendo dos objetivos clave: Minimizar el coste de la inversión y maximizar el rendimiento del proyecto. Así, el problema de selección puede traducirse de la siguiente manera: Los elementos están representados por los proyectos que deben seleccionarse, se definen por el número de recursos necesarios para la ejecución de los proyectos, así como por el beneficio que aportan a la organización. Ambas características se consideran, respectivamente, tamaño y beneficio de los elementos. Es decir, las dimensiones del saco de nudos corresponden al tipo de recursos disponibles (por ejemplo, materia prima), y la capacidad de cada dimensión viene determinada por la cantidad de recursos disponibles de cada tipo. Añadimos una restricción adicional relativa a la capacidad global de la mochila, que es el presupuesto dedicado a la cartera de proyectos. Antes de modelizar este problema, introducimos algunos índices y parámetros:

*j*: Number of projects, $j = 1, 2, \cdots, n$.
*i*: Type of human resources, $i = 1, 2, \cdots, m$.
*k*: Machine kind, $k = 1, 2, \cdots, s$.
*o*: Type of raw material, $o = 1, 2, \cdots, z$.
$\tilde{H}_i$ : Available human resource of type *i*.
$\tilde{h}_{ij}$ : Requirements of human resource *i* in project *j*.
$\tilde{M}_k$ : Available machine—hour of type *k*.
$\tilde{m}_{kj}$ : Requirements of machine—hour of type *k* in project *j*
$\tilde{R}_o$ : Maximum available raw material of type *o*.
$\tilde{r}_{oj}$ : Requirements of raw material o in project *j*.
$\tilde{B}_j$ : Maximum available budget for project *j*.
$\tilde{C}_i$ : Cost of human resourceper hour *i*.
$\tilde{C}_k$ : Cost of machine type per hour *k*.
$\tilde{C}_o$ : Unit cost of raw material *o*.
$\tilde{p}_j$ : Total net profit of projects *j*.

$$\begin{cases} Max \sum_{j=1}^{n} x_j \left[ p_j - \sum_{i=1}^{m} \tilde{h}_{ij} \tilde{C}_i - \sum_{k=1}^{s} \tilde{m}_{kj} \tilde{C}_k - \sum_{o=1}^{z} \tilde{r}_{oj} \tilde{C}_o \right] & (45) \\ s.t \sum_{i=1}^{m} \tilde{h}_{ij} x_j \le \tilde{H}_i; \forall i = 1, \cdots, m; & (46) \\ \sum_{k=1}^{s} \tilde{m}_{kj} x_j \le \tilde{M}_k; \forall k = 1, \cdots, s; & (47) \\ \sum_{o=1}^{z} \tilde{r}_{oj} x_j \le \tilde{R}_o; \forall o = 1, \cdots, z; & (48) \\ \left( \sum_{i=1}^{m} \tilde{h}_{ij} \tilde{C}_i + \sum_{k=1}^{s} \tilde{m}_{kj} \tilde{C}_k + \sum_{o=1}^{z} \tilde{r}_{oj} \tilde{C}_o \right) x_j \le \tilde{B}_j; \forall j = 1, \cdots, n; & (49) \\ \left( \sum_{i=1}^{m} \tilde{h}_{ij} \tilde{C}_i + \sum_{k=1}^{s} \tilde{m}_{kj} \tilde{C}_k + \sum_{o=1}^{z} \tilde{r}_{oj} \tilde{C}_o \right) x_j < \tilde{p}_j; \forall j = 1, \cdots, n; & (50) \\ \sum_{j=1}^{n} x_j \ge 1; & (51) \\ x_j \in \{0,1\}; \forall j = 1, 2, \cdots, n; & (52) \end{cases}$$

La ecuación (45) significa que el problema actual tiene más de un objetivo, pretende maximizar el beneficio de los proyectos seleccionados y minimizar su coste en términos de recursos humanos, máquinas y materiales.

La ecuación (46) indica que los recursos humanos necesarios para el trabajo del proyecto no deben superar los recursos humanos disponibles.

Las ecuaciones (47), (48) tienen la misma descripción que la Ecuación (46), pero se aplican para la hora-máquina y las materias primas, respectivamente.

Las ecuaciones (49), (50) garantizan que el coste total de un proyecto seleccionado *j* sea inferior a su presupuesto y a su beneficio, respectivamente.

La ecuación (51) garantiza que debe seleccionarse al menos un proyecto.

La ecuación (52) indica que $x_j$ es una variable binaria, que denota si el proyecto *j* se selecciona ($x_j$ =1) o no ($x_j$ = 0).

Vistas ya algunas de las posibles aplicaciones que pueden tener las diferentes versiones del problema MKP, nos centraremos particularmente en la aplicación de los modelos para la resolución de problemas de selección de proyectos.

# 2. Métodos de resolución: algoritmos greedy

Un abordaje inicial para la resolución de los problemas MKP sería utilizar un solver de programación entera como puede ser Xpress, en dicho solver se incluyen ciertos algoritmos que nos pueden facilitar la resolución del problema.

Xpress tiene implementados algoritmos del tipo branch and bound, el algoritmo Branch and Bound es un método de búsqueda completa utilizado en la optimización combinatoria para encontrar la solución óptima de un problema de optimización en un espacio de búsqueda grande.

La idea detrás del algoritmo es dividir el espacio de búsqueda en pequeñas subregiones o ramas, y luego explorar cada rama de manera exhaustiva para determinar si puede conducir a una solución óptima. Si no es posible encontrar una solución en una rama, se descarta esa rama y se explora la siguiente rama. Para acelerar el proceso de búsqueda, el algoritmo utiliza una técnica de cota que reduce el espacio de búsqueda descartando ramas que no pueden contener la solución óptima. La cota se actualiza continuamente a medida que se explora cada rama, lo que permite al algoritmo descartar rápidamente las ramas que no tienen posibilidad de contener la solución óptima.

El problema que tenemos es que este tipo de solvers son comerciales y de elevado coste por lo que si queremos resolver alguno de estos problemas deberíamos crear nuestros propios algoritmos heurísticos que puedan ser implementados posteriormente.

Durante este capítulo estudiaremos en profundidad los algoritmos de tipo greedy para la resolución del MKP, concretamente aquellos que están basados en una función de pseudototalidad.

Distinguimos entre dos tipos de algoritmos en función de la solución inicial: primales (ADD) o duales (DROP), la principal diferencia es que los primales comienzan con la solución nula y van añadiendo ítems, mientras que los duales comienzan con la solución que incluye todos los ítems.

La continuación a estos algoritmos, clasificados como heurísticas de construcción, serían las heurísticas de mejora, como, por ejemplo, la búsqueda local. Durante este trabajo no vamos a tratar con las heurísticas de mejora, debido a que estas podrían ocupar un solo trabajo para tratar ese tema, y nos quedaría un trabajo demasiado extenso.

Aun así, veremos una implementación en el código que utiliza la búsqueda local para mejorar los resultados que se obtienen con algunos de los algoritmos, aunque no tratemos este tema a fondo.

## 2.1 Historia de los Algoritmos Heurísticos

*Senju y Toyoda (1968)* sugirieron una heurística doble para los MKP empezando por asignar unos a todas y cada una de las variables y asignando de cero a uno a los valores de estas variables simultáneamente de acuerdo con ratios crecientes hasta que se cumplan los requisitos de viabilidad. Por el contrario, *G. Kochenberger et al (1974), Toyoda (1975), Loulou y Michaelides (1979)* desarrollaron una serie de métodos para los MKP partiendo del origen y asignando unos a los valores de los parámetros de acuerdo con ratios decrecientes hasta el punto en que la adición de variables adicionales violaría las restricciones.

*Hillier (1969)* presentó algoritmos multietapa y rutas internas para el MKP que se concentraban en el simplex formado por la solución óptima del LP (programación lineal) y sus puntos más alejados cercanos como punto de partida de la búsqueda lineal. La primera etapa identifica una ruta que conduce de la solución óptima del LP a otra solución adyacente perteneciente a la región viable entera. Después, el algoritmo recorre esta ruta para determinar una mejor solución entera posible en la segunda etapa. Y en la etapa final, la búsqueda local se lleva a cabo en un intento de mejorar la solución posible actual mediante la modificación de una variable o más al mismo tiempo.

Utilizando instancias MKPs de tamaño moderado, *Zanakis (1977)* demostró que el algoritmo de *Hillier* era más preciso que los algoritmos fundamentales primal/dual greedy. Un procedimiento de los más conocidos basados en LP para identificar soluciones aproximadas a los populares problemas lineales es el llamado método de "Pivote y Complemento" que fue establecido originalmente por *Balas y Martin (1980)*. Ellos recomendaron un algoritmo aproximado para el MKP para resolver el problema central, que es un problema de mochila determinado sobre un pequeño subconjunto de ítems existentes, de forma que habrá una alta posibilidad de descubrir un óptimo global en el núcleo, demostrando que la probabilidad para la heurística de descubrir la solución óptima crece con el tamaño de la instancia. El protocolo comienza resolviendo la relajación del LP mediante un método simplex de variable acotada estándar y procede aplicando una serie de pivotes con la intención de colocar los parámetros acotados en la base con el mínimo coste. A continuación, una etapa complementaria trata de mejorar la solución obtenida en los pivotes. Además, también se han obtenido resultados favorables para programas lineales puros utilizando combinaciones de búsqueda tabú y una heurística de pivote y complemento.

*Freville y Plateau (1994)* sugirieron un eficaz algoritmo de procesamiento previo al MKP que asigna límites agudos bajo y alto al valor óptimo reduciendo el conjunto continuo posible y eliminando variables y restricciones. *Magazine y Oguz (1984)* integraron el algoritmo binario de *Senju y Toyoda (1968)* utilizando un enfoque de relajación de *Lagrange* que permitía fijar los valores de los parámetros a sus valores asignados en el conjunto de soluciones óptimas. Posteriormente, su investigación fue ampliada por *Volgenant y Zoon (1990)*. *Freville y Plateau (1994)* se concentraron en las relajaciones lagrangeanas y sustitutivas. Sugirieron tres enfoques para las soluciones utilizando fijación acelerada (muchas variables fijadas simultáneamente), método de ruido y variables fuertemente determinadas, y restricciones sustitutas.

*Pirkul (1987)* construyó una técnica general más sencilla para resolver el MKP que contenía un método de descenso para determinar las restricciones sustitutas. A continuación, demostraron que este método codicioso era en general más rápido que la heurística de pivote y complemento y que producía soluciones comparables, en términos de calidad de la solución alcanzada, a instancias de 20 restricciones y 200 variables.

*Lee y Guignard (1988)* sugirieron una técnica multietapa para resolver el MKP afinado con pocas variables que regula el compromiso entre tiempo de computación y calidad de la solución, cuyos valores eran definidos por el usuario. Documentaron que las mejoras en el tiempo de computación y la calidad de solución mediante resultados numéricos para 48 problemas de prueba con 6-500 parámetros y 5-20 restricciones.

*Hanafi y Freville (1998)* desarrollaron un algoritmo multietapa simple para el MKP incorporando varios principios heurísticos como noising, threshold accepting, simulated annealing y greedy de

forma flexible. Comenzando con un grupo de posibles soluciones aleatorias, la primera fase realizaba muchas operaciones de búsqueda local y luego una fase extra, basada en pasos greedy reiterados intentaba mejorar la posible solución actual. *Balas et al (2001)* introdujeron una búsqueda local compleja en la vecindad del entero de la solución fraccionaria LP destinada a resolver problemas puros.

*Martello y Toth (1988)* establecieron un algoritmo eficiente para problemas de gran tamaño sobre la base de la utilización de un algoritmo codicioso para resolver grandes problemas knapsack. Es capaz de resolver el problema central y alcanzar una solución óptima mediante branch- and-bound, obteniendo así una cota baja apreciablemente buena.

*Plateau et al (2002)* probaron un procedimiento multietapa mediante el uso de metaheurísticas y técnicas de punto interior donde la primera etapa abarca una búsqueda híbrida que emplea una técnica de punto interior para producir puntos germinales fraccionarios, una búsqueda local para recuperar la viabilidad, y un generador de cortes para diversificar la población de las posibles soluciones de partida. Por otro lado, la segunda etapa realiza un número constante de ensayos de reenlace de rutas entre grupos de pares de soluciones elegidos de la población de partida. Consiguieron resolver el MKP y compararon sus resultados con los del algoritmo genético de *Chu y Beasley (1998)*. En conclusión, los resultados del estudio anterior mostraban perspectivas alentadoras para la aplicación de técnicas de puntos interiores como guías para el re-enlazamiento de rutas fomentado/mejorado, y enfoques de operaciones de búsqueda dispersa o local.

*Balev et al (2008)* sugirieron una heurística que utilizaba la programación dinámica de forma relevante para obtener una posible solución mediante mejoras progresivas de la solución de redondeo LP, y examinaron su rendimiento en todos los conjuntos estándar encontrados en la literatura. Su heurística resultó ser notablemente rápida y potente en comparación con los mejores métodos de búsqueda tabú.

*Frieze y Clarke (1984)* recomendaron un método de aproximación polinómica sobre la base del uso de un algoritmo simplex dual para la programación lineal y probaron las características asintóticas de un modelo aleatorio específico. *Rinnooy Kan et al (1993)* ofrecieron un grupo de algoritmos del tipo codicioso generalizado en el que los elementos se seleccionan en función de sus ratios de disminución de beneficios y de las sumas ponderadas de sus coeficientes de recursos.

*Fox y Scudder (1985)* sugirieron una heurística basada en un inicio que establece todos los valores de las variables en cero (uno) y selecciona progresivamente las variables que se establecerán en uno (cero). Presentaron los resultados computacionales para problemas de prueba MKP producidos aleatoriamente que comprendían hasta 100 restricciones y variables.

## 2.2 Algoritmos greedy duales (DROP)

Los algoritmos greedy duales, también conocidos como algoritmos DROP (Dual Reduction Optimization Procedure), son una clase de algoritmos greedy utilizados en problemas de optimización combinatoria. Estos algoritmos se enfocan en el uso de información dual del problema para realizar reducciones y tomar decisiones de manera iterativa.

La idea principal detrás de los algoritmos DROP es utilizar información dual para identificar y

eliminar componentes no prometedores del problema. Al eliminar estas componentes, se reduce el tamaño del problema, lo que puede conducir a una solución óptima más rápidamente o permitir el uso de algoritmos más eficientes.

A continuación, se presenta una descripción general del proceso seguido por los algoritmos greedy duales (DROP):

1. Inicialización: Se inicia con una solución inicial que puede ser factible o no factible.
2. Cálculo de información dual: Se calcula información dual asociada al problema, como los precios duales, los coeficientes duales o cualquier otra medida que refleje la importancia relativa de los componentes.
3. Reducción de componentes: Se identifican y eliminan componentes no prometedores basados en la información dual calculada. Esto puede implicar eliminar elementos, restricciones o cualquier otra parte del problema que se considere no rentable.
4. Actualización de la solución: Después de cada reducción de componentes, se actualiza la solución actual para reflejar los cambios realizados.
5. Convergencia: El proceso de reducción y actualización se repite iterativamente hasta que se alcance un criterio de convergencia predefinido. Esto puede ser un número máximo de iteraciones, una mejora mínima en la solución o cualquier otro criterio establecido.

Es importante destacar que la selección de los componentes no prometedores y la actualización de la solución en los algoritmos DROP pueden variar según el problema específico que se esté abordando. Estas decisiones se basan en la información dual calculada y en los objetivos del problema.

Los algoritmos greedy duales (DROP) son útiles cuando se dispone de información dual relevante y se puede utilizar para guiar el proceso de búsqueda. Sin embargo, es importante tener en cuenta que estos algoritmos también pueden sufrir de suboptimización si la información dual no es precisa o si la estrategia de reducción no es adecuada.

En resumen, los algoritmos greedy duales (DROP) utilizan información dual para reducir componentes no prometedores en un problema de optimización combinatoria. A través de iteraciones de reducción y actualización, estos algoritmos buscan obtener soluciones óptimas o de alta calidad de manera eficiente.

### 2.2.1 Algoritmo de Senju y Toyoda

El algoritmo de Senju y Toyoda, propuesto en 1968, es un método dual de gradiente utilizado para resolver problemas de programación lineal. Este algoritmo se utiliza para encontrar la solución óptima de un problema de programación lineal mediante la actualización iterativa de los multiplicadores lagrangianos duales.

A continuación, explicaré los pasos básicos del algoritmo de Senju y Toyoda:

1. Inicialización: Se inicia con una solución primal y dual factible.
2. Cálculo de los multiplicadores lagrangianos: Se calculan los multiplicadores lagrangianos duales asociados a las restricciones del problema.
3. Cálculo de la dirección de búsqueda: Se calcula la dirección de búsqueda utilizando el

gradiente dual. Esta dirección indica hacia dónde se debe mover la solución dual para mejorar el valor de la función objetivo.

4. Actualización de los multiplicadores lagrangianos: Se actualizan los multiplicadores lagrangianos en función de la dirección de búsqueda y un tamaño de paso determinado.
5. Comprobación de la condición de parada: Se verifica si se ha alcanzado la condición de parada, que puede ser una mejora mínima en la función objetivo o una convergencia satisfactoria.
6. Actualización de la solución primal: Si la condición de parada no se cumple, se actualiza la solución primal utilizando los multiplicadores lagrangianos actualizados y se vuelve al paso 2.

El algoritmo de Senju y Toyoda utiliza el gradiente dual para determinar la dirección de búsqueda, que es la dirección en la que se debe mover la solución dual para mejorar la función objetivo. La actualización de los multiplicadores lagrangianos se realiza para encontrar el tamaño de paso óptimo que garantice la convergencia hacia la solución óptima.

Este algoritmo tiene la ventaja de ser aplicable a problemas de programación lineal con restricciones generales, no solo a problemas de programación lineal pura. Sin embargo, también tiene algunas limitaciones, como la posible convergencia lenta o la sensibilidad a la elección del tamaño de paso.

Es importante mencionar que han surgido variaciones y mejoras del algoritmo original propuesto por Senju y Toyoda a lo largo de los años. Estas variantes pueden incluir técnicas adicionales para mejorar la eficiencia y la convergencia del algoritmo.

En resumen, el algoritmo de Senju y Toyoda es un método dual de gradiente utilizado para resolver problemas de programación lineal. A través de la actualización iterativa de los multiplicadores lagrangianos duales, busca encontrar la solución óptima del problema. Sin embargo, como con cualquier algoritmo, es importante tener en cuenta las características específicas del problema y ajustar los parámetros adecuadamente.

## 2.3 Algoritmos greedy primales (ADD)

Los algoritmos greedy primales, también conocidos como algoritmos ADD (Addition of Duals to a Direct heuristic), son otro enfoque para resolver problemas de optimización combinatoria. A diferencia de los algoritmos duales que se centran en el uso de información dual, los algoritmos primales ADD se basan en la adición de componentes directamente a la solución primal de manera iterativa.

Esta es una descripción general de los algoritmos greedy primales ADD:

1. Inicialización: Se comienza con una solución inicial vacía.
2. Cálculo de valores relativos: Se calcula un valor relativo para cada componente no seleccionada en función de su contribución a la función objetivo.

3. Selección de componentes: Se selecciona el componente que maximiza el valor relativo entre los componentes no seleccionados.
4. Agregar componente a la solución: Se agrega el componente seleccionado a la solución primal.
5. Comprobación de la condición de parada: Se verifica si se ha alcanzado la condición de parada, que puede ser una mejora mínima en la función objetivo o una convergencia satisfactoria.
6. Actualización de valores relativos: Después de agregar un componente a la solución, se actualizan los valores relativos para reflejar los cambios realizados.
7. Repetir los pasos 3 a 6 hasta que se cumpla la condición de parada.

La clave de los algoritmos primales ADD radica en la selección de componentes basada en los valores relativos. Estos valores reflejan la contribución esperada de cada componente a la función objetivo y ayudan a guiar la selección de componentes en cada iteración.

Es importante tener en cuenta que los algoritmos primales ADD son heurísticas y no garantizan la obtención de la solución óptima. Sin embargo, pueden proporcionar soluciones aproximadas de buena calidad en muchos casos.

En resumen, los algoritmos primales ADD son enfoques greedy para resolver problemas de optimización combinatoria. A través de la adición iterativa de componentes a la solución primal, basándose en valores relativos, buscan encontrar una solución de alta calidad. Estos algoritmos son relativamente simples de implementar y pueden ser eficientes en la práctica.

### 2.3.1 Algoritmo de Toyoda

El algoritmo de Toyoda, propuesto por Toyoda en 1975, es un método primal de gradiente utilizado para resolver problemas de programación lineal. Este algoritmo se basa en la optimización de la función objetivo primal a través de actualizaciones iterativas.

A continuación, se presenta una descripción general del algoritmo de Toyoda:

1. Inicialización: Se selecciona una solución inicial factible para el problema de programación lineal.
2. Cálculo de los valores duales: Se calculan los valores duales asociados a las restricciones del problema. Estos valores representan el gradiente de la función objetivo dual.
3. Cálculo de la dirección de búsqueda: Se determina la dirección de búsqueda primal utilizando los valores duales calculados. La dirección de búsqueda se obtiene mediante una combinación lineal de las restricciones del problema.
4. Tamaño del paso: Se determina el tamaño del paso o la longitud del paso en la dirección de búsqueda. Esto se puede hacer mediante una búsqueda de línea o utilizando un tamaño de paso fijo predefinido.
5. Actualización de la solución: Se actualiza la solución primal moviéndose en la dirección de búsqueda con el tamaño del paso determinado. Esto implica modificar los valores de las variables de decisión de acuerdo con el tamaño del paso y la dirección de búsqueda.
6. Convergencia: Los pasos 2 a 5 se repiten iterativamente hasta que se alcance un criterio de

convergencia predefinido. Esto puede ser un número máximo de iteraciones, una mejora mínima en la solución o cualquier otro criterio establecido.

El algoritmo de Toyoda es un método primal de gradiente que busca mejorar iterativamente la solución primal moviéndose en la dirección de búsqueda óptima. Al actualizar la solución en cada iteración, el algoritmo busca acercarse a la solución óptima del problema de programación lineal.

Es importante destacar que el algoritmo de Toyoda no garantiza la convergencia a la solución óptima en todos los casos. Sin embargo, en la práctica, puede proporcionar soluciones factibles de alta calidad y es ampliamente utilizado en la optimización lineal.

# 3. Extensiones de los algoritmos greedy

En este punto trataremos de describir otros de los algoritmos más recientes como son los algoritmos de Loulou-Michaelides (1979), y el algoritmo de Pirkul (1987). Los cuales fueron nombrados en el capítulo anterior, pero serán descritos en mayor profundidad para de esta forma comprender mejor su funcionamiento y el porqué de la importancia de dichos algoritmos.

## 3.1 Algoritmos de Loulou-Michaelides

Los algoritmos de Lolou-Michaelides son un conjunto de técnicas de optimización desarrolladas por C. P. Lolou y E. Michaelides en 1979. Estos algoritmos se utilizan para resolver problemas de optimización no lineales con restricciones.

La principal ventaja de los algoritmos de Lolou-Michaelides es que son muy eficientes y pueden converger rápidamente a la solución óptima. Estos algoritmos también son muy robustos y pueden manejar una amplia gama de problemas de optimización no lineales.

### 3.1.1 Algoritmo de Proyección Activa

El algoritmo de proyección activa (PA) es uno de los algoritmos más populares desarrollados por Lolou- Michaelides para la resolución de problemas de optimización no lineales con restricciones. El algoritmo de PA utiliza una combinación de proyecciones y un enfoque de gradiente para encontrar la solución óptima. En el algoritmo de PA, se comienza con una solución inicial factible y se realiza una búsqueda en la dirección del gradiente negativo. Si la solución resultante cumple con las restricciones, se acepta como nueva solución. Si la solución no cumple con las restricciones, se proyecta en el conjunto de soluciones factibles más cercano y se repite el proceso.

La proyección se lleva a cabo en cada iteración del algoritmo para garantizar que la solución se mantenga dentro del conjunto factible de soluciones. La proyección se realiza mediante la minimización de la distancia entre la solución actual y la solución proyectada en el conjunto de soluciones factibles. La solución proyectada se calcula como la solución más cercana al conjunto de soluciones factibles.
El algoritmo de PA es muy eficiente en la resolución de problemas de optimización no lineales con restricciones, especialmente cuando el número de restricciones es grande. El algoritmo es capaz de converger rápidamente a la solución óptima debido a su enfoque de gradiente y su capacidad para manejar eficientemente las restricciones.

Además, el algoritmo de PA tiene la capacidad de manejar problemas no convexos y no diferenciables. Esto lo hace útil en una amplia gama de aplicaciones en ingeniería, ciencias y finanzas.

### 3.1.2 Algoritmo de Dirección de Búsqueda

El algoritmo de dirección de búsqueda (BD) es otro algoritmo desarrollado por Lolou-

Michaelides para resolver problemas de optimización no lineales con restricciones. A diferencia del algoritmo de proyección activa, el algoritmo de BD utiliza una estrategia de búsqueda indirecta para encontrar la solución óptima. En el algoritmo de BD, se comienza con una solución inicial factible y se genera una dirección de búsqueda utilizando una combinación de las direcciones de gradiente y de las direcciones normales a las restricciones activas. Las restricciones activas son aquellas que se cumplen con igualdad en la solución actual.

La dirección de búsqueda se utiliza para generar una nueva solución factible, que se acepta como nueva solución si cumple con las restricciones y es mejor que la solución anterior. Si la nueva solución no cumple con las restricciones, se genera una nueva dirección de búsqueda y se repite el proceso.

El algoritmo de BD es eficiente en la resolución de problemas de optimización no lineales con un número moderado de restricciones. La estrategia de búsqueda indirecta permite al algoritmo explorar el espacio de soluciones de manera más efectiva que el algoritmo de proyección activa en algunos casos.

Este algoritmo también puede resolver problemas no convexos y no diferenciables, lo que también lo hace bastante útil en diferentes ramas de las finanzas e ingeniería.

### 3.1.3 Algoritmo de Gradiente Reducido

El algoritmo de gradiente reducido (GR) es una técnica de optimización desarrollada por Michael J. D. Powell en 1964, más adelante implementado en los algoritmos de Lolou-Michaelides, que ha sido ampliamente utilizado en la resolución de problemas de optimización sin restricciones. El algoritmo de GR es un método de optimización de primer orden que utiliza únicamente información del gradiente para encontrar la solución óptima.

El algoritmo de GR comienza con una solución inicial y calcula el gradiente de la función objetivo en ese punto. Luego, se busca una dirección de descenso utilizando una estrategia de búsqueda de línea, como la regla del punto mínimo o la regla de Armijo. La dirección de descenso se utiliza para generar una nueva solución, que se acepta si mejora la función objetivo.

A diferencia de otros algoritmos de optimización de primer orden, como el método del gradiente descendente, el algoritmo de GR utiliza información adicional del gradiente para mejorar la eficiencia y convergencia del algoritmo. En lugar de calcular el gradiente completo de la función objetivo, el algoritmo de GR utiliza una aproximación del gradiente en la dirección de la última búsqueda para reducir el costo computacional.

El algoritmo de GR es conocido por su capacidad para converger rápidamente a soluciones óptimas en problemas convexos. Además, el algoritmo de GR también es capaz de manejar problemas no convexos y no diferenciables, aunque puede ser menos eficiente en estos casos.

### 3.1.4 Algoritmo de Gradiente Proyectado

El algoritmo de gradiente proyectado (PG) es un método popular para resolver problemas de optimización convexa con restricciones lineales. Este algoritmo fue desarrollado por primera vez por el matemático soviético Boris Polyak en la década de 1960, más tarde, también lo implementarían Lolou-Michaelides en sus algoritmos.

El algoritmo de PG utiliza una combinación de la dirección de gradiente y la proyección en el conjunto de soluciones factibles para encontrar la solución óptima. En cada iteración, se calcula la dirección de gradiente de la función objetivo y se proyecta en el conjunto de soluciones factibles. La solución resultante se utiliza como punto de partida para la siguiente iteración.

La proyección en el conjunto de soluciones factibles es un paso crítico en el algoritmo de PG, ya que garantiza que la solución permanezca dentro del conjunto de soluciones factibles. La proyección se lleva a cabo mediante la minimización de la distancia entre la solución actual y la solución proyectada en el conjunto de soluciones factibles.

El algoritmo de PG es muy eficiente en la resolución de problemas de optimización convexa con restricciones lineales. El algoritmo es capaz de converger rápidamente a la solución óptima debido a su enfoque de gradiente y su capacidad para manejar eficientemente las restricciones lineales.

Además, el algoritmo de PG es fácil de implementar y es ampliamente utilizado en una variedad de aplicaciones, incluyendo la optimización de carteras de inversión, la planificación de la producción, la optimización de redes y la planificación de proyectos.

### 3.1.5 Algoritmo de Gradiente Mejorado

El algoritmo de gradiente mejorado (IG) es otro algoritmo utilizado para resolver problemas de optimización no lineales con restricciones. Fue desarrollado por Fiacco y McCormick en la década de 1960 y es una extensión del método del gradiente tradicional, que más tarde implementarían Loulou-Michaelides en sus algoritmos.

El algoritmo IG combina el uso del gradiente y de la función objetivo para mejorar la eficiencia de la búsqueda de la solución óptima. En cada iteración, se utiliza el gradiente para determinar la dirección de búsqueda y se utiliza la función objetivo para determinar el tamaño de paso adecuado.

El algoritmo IG también utiliza una técnica de proyección para garantizar que la solución se mantenga dentro del conjunto factible de soluciones. La proyección se realiza en cada iteración para garantizar que la solución se mantenga dentro del conjunto de soluciones factibles.

El algoritmo IG es muy eficiente para resolver problemas de optimización no lineales con

restricciones, especialmente cuando el número de restricciones es grande. El enfoque combinado del gradiente y la función objetivo permite al algoritmo encontrar soluciones óptimas de manera más efectiva que el método del gradiente tradicional.

Al igual que el resto de los algoritmos vistos con anterioridad durante este punto, el algoritmo IG tiene la capacidad de manejar problemas no convexos y no diferenciables y también lo hace útil en diferentes ramas de aplicaciones en ingeniería, ciencias y finanzas.

## 3.2 Algoritmo de Pirkul

El algoritmo de Pirkul es un algoritmo de programación entera que se utiliza para resolver el problema de localización de facilidades (PFL) con restricciones de capacidad. Este problema consiste en seleccionar las ubicaciones óptimas para un conjunto de facilidades, de tal manera que se satisfagan las demandas de un conjunto de clientes, y se cumplan ciertas restricciones de capacidad en las facilidades.

El algoritmo de Pirkul se basa en la técnica de branch and cut, y utiliza una relajación lineal del problema original para construir una serie de problemas de programación lineal enteros (PLE) más pequeños y manejables. En cada iteración del algoritmo, se resuelve el PLE mediante el método de punto interior y se utiliza una técnica de corte para descartar soluciones no factibles. Si la solución encontrada es entera, se devuelve como solución óptima. Si no es entera, se selecciona una variable fraccional para ser ramificada, y se crean dos subproblemas PLE, uno con la restricción de que la variable seleccionada es menor o igual a su parte entera, y otro con la restricción de que la variable es mayor o igual a su parte entera más uno.

El algoritmo de Pirkul ha demostrado ser eficaz para resolver instancias de tamaño moderado del PFL con restricciones de capacidad. Sin embargo, debido a la complejidad computacional del problema, puede requerir mucho tiempo para resolver instancias grandes. Por lo tanto, se han propuesto varias variantes y mejoras del algoritmo, como la técnica de relajación dual, la reducción de subproblemas y la selección de variable dinámica, para mejorar su eficiencia.

### 3.2.1 Técnica de relajación dual

La técnica de relajación dual es una técnica de optimización que se utiliza para resolver problemas de programación entera y mixta-entera. Esta técnica se basa en la relajación lineal del problema original, donde se eliminan las restricciones enteras y se permite que las variables tomen valores fraccionarios. La relajación lineal se puede resolver con algoritmos de programación lineal convencionales, lo que permite obtener una solución óptima o cercana a la óptima del problema original.

En la técnica de relajación dual, además de resolver la relajación lineal, se construye un problema dual, que se obtiene intercambiando las variables y restricciones de la relajación lineal. El problema dual es un problema de maximización que se resuelve con algoritmos de programación lineal convencionales. La solución óptima del problema dual

proporciona una cota inferior para la solución óptima del problema original.

La técnica de relajación dual se utiliza en la resolución de problemas de programación entera y mixta-entera porque puede proporcionar una buena aproximación de la solución óptima, lo que permite reducir el espacio de búsqueda y acelerar la convergencia de los algoritmos de ramificación y corte. Además, la técnica de relajación dual también se utiliza para la generación de cortes, que son restricciones adicionales que se agregan al problema original para mejorar la eficiencia de los algoritmos de branch and cut.

### 3.2.2 Técnica de selección de variable dinámica

La técnica de selección de variable dinámica es una técnica utilizada en branch and cut para seleccionar la variable a ramificar en cada iteración del algoritmo de programación entera. En lugar de seleccionar una variable de manera predefinida, como se hace en la selección de variable estática, la técnica de selección de variable dinámica utiliza información obtenida durante la resolución del problema para elegir la variable a ramificar.

Existen varias estrategias de selección de variable dinámica, pero una de las más populares es la estrategia de selección de variable basada en la función objetivo. En esta estrategia, se calcula la contribución de cada variable a la función objetivo del problema y se utiliza esta información para determinar qué variable ramificar. La idea es seleccionar la variable que tenga la mayor contribución a la función objetivo en términos de mejora de la solución, es decir, la variable cuyo cambio de valor más probablemente conduzca a una mejora en la solución óptima.

Otra estrategia de selección de variable dinámica es la estrategia de selección de variable basada en la viabilidad. En esta estrategia, se selecciona la variable cuyo valor fraccional es más grande y, por lo tanto, es más probable que conduzca a una solución no factible. La idea es ramificar en esta variable para forzar una solución entera y reducir la relajación lineal del problema.

En general, la selección de variable dinámica es una técnica poderosa para mejorar la eficiencia y la efectividad de los algoritmos de programación entera. Permite adaptar la selección de variable a las características específicas del problema y a la información obtenida durante la resolución del problema, lo que puede conducir a una mejor exploración del espacio de soluciones y a una convergencia más rápida hacia la solución óptima.

En resumen, el algoritmo de Pirkul es un algoritmo de programación entera que utiliza la técnica de ramificación y corte para resolver el problema de localización de facilidades con restricciones de capacidad. Es un algoritmo eficaz para instancias de tamaño moderado, pero puede requerir mucho tiempo para resolver instancias grandes.

# 4. Aplicaciones de los métodos vistos a los problemas MKP

Durante este punto veremos la aplicación práctica de los algoritmos antes vistos para la resolución de diferentes problemas MKP.

Para ello vamos a utilizar diferentes problemas planteados en http://people.brunel.ac.uk , estos problemas cuentan con números de mochilas y objetos diferentes para que de esta forma podamos comparar los diferentes algoritmos teniendo en cuenta estas características.

Los códigos utilizados para resolver dichos problemas, incluyendo el algoritmo para la lectura de datos y el algoritmo para conseguir el resultado óptimo del problema se encuentran en el anexo.

Para la solución óptima del problema se ha construido un programa en Mosel para la obtención de la misma.

El resto de los códigos han sido planteados en lenguaje Java, siguiendo el paradigma de la Programación Orientada a Objetos, a continuación, daré una pequeña explicación del funcionamiento de cada uno de los códigos:

**Algoritmo DROP:** Se parte de una solución completa no factible, realizamos el cálculo de los valores relativos, a partir de aquí, en cada iteración, y hasta que se cumpla el criterio de parada, que no haya datos o solución factible, iremos seleccionando el componente peor para cada mochila.

**Algoritmo ADD:** El algoritmo comienza con una solución inicial vacía, a partir de ahí realiza un cálculo de los valores relativos de cada objeto en cada mochila, ordenándolos para posteriormente escoger los mejores objetos, de esta forma iteraremos hasta cumplir la condición de parada, que en este caso será que los valores relativos estén vacíos, y de esta forma se irán añadiendo componentes de manera óptima a la solución inicial vacía.

**Algoritmo de Loulou-Michaelides (Proyección Activa):** Comenzamos inicializando las asignaciones iniciales de manera aleatoria hasta que no quede espacio, pasamos a la mejora que incluye el algoritmo de búsqueda local y proyección activa en la que eliminamos un elemento, de forma ordenada, de izquierda a derecha y proyectamos con una combinación de los elementos no seleccionados, actuamos de esta forma hasta que se cumpla la condición de parada, que en este caso viene dada por una constante que establecemos igual a mil iteraciones.

**Algoritmo de Pirkul:** Al igual que en el anterior algoritmo, las asignaciones iniciales se dan mediante un proceso aleatorizado hasta que no quede espacio. Para mejorar la solución iteraremos hasta llegar a la condición de parada, mil iteraciones, si se encuentra una mejora, volvemos a comenzar desde 0, a diferencia del algoritmo de Loulou-Michaelides este algoritmo solo contiene un bucle a diferencia de los dos del código anterior.

A parte de los códigos de los algoritmos se han incluido clases auxiliares que acompañan a dichos algoritmos en la resolución:

**MultidimensionalKnapsack.java:** Es la clase principal en la cual se ejecutan los diferentes algoritmos para la obtención de los resultados, en primer lugar, se indica el fichero en el que se encuentran los datos del problema que queremos resolver, los cuales serán leídos gracias al método *readKnapsackDataFromFile*, en este método se implementa un algoritmo en el que leemos del fichero el número de mochilas y de objetos del problema, en función de estos datos obtenemos las capacidades de cada mochila, los pesos de cada objeto y sus restricciones, finalmente leemos el valor óptimo del problema.

Después de la lectura del fichero y la obtención de los datos, se ejecuta cada uno de los algoritmos incluyendo el tiempo de ejecución y el resultado obtenido por el mismo.

**KnapsackData.java:** Esta clase contiene el modelo en el que los atributos son los datos del problema Knapsack, incluyendo el número de mochilas, el número de objetos, los pesos, las capacidades, los items y el valor óptimo. Se implementan los diferentes métodos get para la obtención de los valores de dichos atributos.

**KnapsackItem.java:** Esta clase contiene el modelo en el que los atributos son los datos del KnapsackItem, incluyendo los pesos y las restricciones. Se implemntan los métodos get y set para la obtención y modificación de los valores de dichos atributos.

**Pair.java:** Esta clase será una clase auxiliar que nos ayudará en la selección de las diferentes mochilas para cada uno de los objetos.

Pasamos a ejecutar estos códigos con una selección de diferentes problemas MKP obteniendo la siguiente tabla de resultados, en la cual, el tiempo (T) está representado en Milisegundos y los diferentes problemas expresan entre paréntesis el número de mochilas y de objetos respectivamente:

| **Problema** | **DROP** | **T** | **ADD** | **T** | **LOULOU** | **T** | **PIRKUL** | **T** | **ÓPTIMO** |
|---|---|---|---|---|---|---|---|---|---|
| **Weing1 (2,28)** | 138168 | 21 | 139278 | 0 | 115255 | 38 | 140532 | 10 | 141278 |
| **Weing2 (2,28)** | 118885 | 17 | 124155 | 0 | 101609 | 38 | 112325 | 7 | 130883 |
| **Weing3 (2,28)** | 76560 | 22 | 93278 | 0 | 42799 | 42 | 94908 | 9 | 95677 |
| **Weing4 (2,28)** | 101017 | 18 | 115831 | 0 | 107487 | 39 | 116622 | 9 | 119337 |
| **Weing5 (2,28)** | 76560 | 19 | 93278 | 0 | 40734 | 40 | 94248 | 7 | 98796 |
| **Weing6 (2,28)** | 118885 | 19 | 123645 | 0 | 99655 | 55 | 121051 | 19 | 130623 |
| **Weing7(2,105)** | 1084386 | 19 | 1090337 | 0 | 1071287 | 187 | 1089797 | 18 | 1095445 |
| **Weing8(2,105)** | 577197 | 19 | 620872 | 0 | 428577 | 691 | 588845 | 156 | 624319 |
| **Weish1(5,30)** | 4341 | 19 | 4534 | 0 | 4411 | 83 | 4549 | 10 | 4554 |
| **Weish2(5,30)** | 4341 | 20 | 4504 | 0 | 4158 | 81 | 4515 | 12 | 4536 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Weish3(5,30)** | 3439 | 20 | 4051 | 0 | 3985 | 85 | 4115 | 15 | 4115 |
| **Weish4(5,30)** | 4505 | 18 | 4505 | 0 | 4145 | 82 | 4527 | 11 | 4561 |
| **Weish5(5,30)** | 4341 | 18 | 4514 | 0 | 3879 | 80 | 4451 | 11 | 4514 |
| **Weish6(5,40)** | 5097 | 18 | 5452 | 0 | 5384 | 144 | 5537 | 12 | 5557 |
| **Weish7(5,40)** | 5097 | 18 | 5452 | 0 | 5356 | 163 | 5542 | 13 | 5567 |
| **Weish8(5,40)** | 5097 | 19 | 5452 | 0 | 5142 | 141 | 5507 | 11 | 5605 |
| **Weish9(5,40)** | 5019 | 17 | 5212 | 0 | 4324 | 138 | 5210 | 18 | 5246 |
| **Weish10(5,50)** | 5294 | 18 | 5880 | 0 | 5548 | 214 | 6327 | 27 | 6339 |
| **Weish11(5,50)** | 4539 | 21 | 5203 | 0 | 5283 | 216 | 5643 | 26 | 5643 |
| **Weish12(5,50)** | 5294 | 19 | 5440 | 0 | 6007 | 210 | 6339 | 20 | 6339 |
| **Weish13(5,50)** | 4971 | 18 | 5745 | 0 | 5643 | 212 | 6088 | 33 | 6159 |
| **Weish14(5,60)** | 5683 | 21 | 6787 | 0 | 6395 | 327 | 6885 | 36 | 6954 |
| **Sento1(30,60)** | 7155 | 18 | 7606 | 0 | 6971 | 1338 | 7761 | 218 | 7772 |
| **Sento2(30,60)** | 8466 | 21 | 8709 | 1 | 8482 | 1391 | 8640 | 154 | 8722 |

Con los resultados de los diferentes problemas recogidos en la tabla anterior pasaremos ahora al siguiente punto en el que trataremos de exponer algunas conclusiones a cerca del funcionamiento de los cuatro algoritmos, haciendo una comparación entre los resultados obtenidos durante las ejecuciones y los tiempos de ejecución que estos conllevan.

Además, expondremos cual es el algoritmo con un mejor, y peor, funcionamiento dependiendo del tamaño de cada problema.

# 5. Conclusiones

Durante este punto, trataremos de dar una comparativa de los algoritmos anteriormente utilizados para resolver nuestros problemas, en primer lugar, daremos una descripción de lo visto en la tabla de resultados para cada algoritmo, seguido de una comparativa entre los cuatro algoritmos y para finalizar trataremos de determinar cuál de los cuatro algoritmos es el que mejor funcionamiento tiene, asi como, el que peor funcionamiento tiene.

**Algoritmo DROP**

Para el algoritmo DROP podemos observar en la tabla de resultados que se obtienen unos valores moderadamente próximos al óptimo, los cuales son mejorables, en algunos casos obtenemos buenos resultados en comparación con el óptimo, pero en líneas generales los resultados distan del óptimo que buscamos.

Respecto a los tiempos de ejecución, son relativamente bajos, para nada son tiempos largos de ejecución, respecto a lo que tiempos de ejecución se refiere se podría decir que este algoritmo funciona correctamente.

Fijándonos en los tamaños de los diferentes problemas, observamos que el algoritmo funciona de una manera bastante homogénea tanto para problemas pequeños como para problemas grandes.

**Algoritmo ADD**

Para el algoritmo ADD podemos observar en la tabla de resultados que se obtienen unos valores bastante próximos al óptimo, los cuales son muy poco mejorables, todos los valores obtenidos se encuentran bastante cercanos a el óptimo, podríamos decir que obtenemos muy buenos resultados con este algoritmo.

Respecto a los tiempos de ejecución, son inmejorables, son mínimos, respecto a lo que tiempos de ejecución se refiere se podría decir que este algoritmo es perfecto.

Fijándonos en los tamaños de los diferentes problemas, observamos que el algoritmo funciona de una manera bastante homogénea tanto para problemas pequeños como para problemas grandes.

**Algoritmo de Loulou-Michaelides (Proyección Activa)**

Para el algoritmo de Loulou-Michaelides podemos observar en la tabla de resultados que se obtienen unos valores que en líneas generales no se aproximan lo suficiente al valor óptimo del problema, son bastante mejorables, en algunos casos obtenemos buenos resultados, pero por lo general, estos resultados no son del todo próximos al óptimo.

Respecto a los tiempos de ejecución, son bastante altos, respecto a lo que tiempos de ejecución se

refiere se podría decir que este algoritmo podría ser bastante mejorable.

Fijándonos en los tamaños de los diferentes problemas, observamos que el algoritmo obtiene tiempos de ejecución menores para problemas más pequeños y mayores en el caso contrario, para los resultados obtenidos, no se observa diferencia entre problemas de menor o mayor tamaño.

**Algoritmo de Pirkul**

Para el algoritmo de Pirkul podemos observar en la tabla de resultados que se obtienen unos valores prácticamente similares al óptimo, llegando en algún punto, incluso, a un valor igual a el valor óptimo del problema, los resultados que obtenemos para este algoritmo son realmente buenos.

Respecto a los tiempos de ejecución, son moderadamente bajos, no son tiempos largos de ejecución, respecto a lo que tiempos de ejecución se refiere se podría decir que este algoritmo funciona correctamente.

Fijándonos en los tamaños de los diferentes problemas, observamos que el algoritmo funciona mejor, en tiempos de ejecución, para problemas de tamaño pequeño, a medida que se incrementa el tamaño del problema, aumenta también el tiempo de ejecución, para los resultados obtenidos, no se observa diferencia entre problemas de menor o mayor tamaño.

**Comparativa de los diferentes algoritmos**

Como podemos observar en la tabla de resultados, el algoritmo de Loulou-Michaelides sería el peor de los cuatro, ya que sus tiempos de ejecución son más elevados que los del resto de los algoritmos, además podemos ver como los resultados que se obtienen, en líneas generales, no son para nada los mejores.

Después, en esta clasificación, estaría el algoritmo DROP, tiene tiempos de ejecución menores, para problemas más grandes, que el algoritmo de Pirkul, pero los resultados que se obtienen no son tan próximos al óptimo como podrían ser los obtenidos con el algoritmo ADD o el algoritmo de Pirkul.

Para determinar cuál de los dos últimos algoritmos funciona mejor, la decisión no es tan clara como antes, ya que el algoritmo de Pirkul obtiene unos valores ligeramente más cercanos al óptimo del problema, pero con un tiempo de ejecución más alto.

Teniendo en cuenta la proximidad al valor óptimo y que los tiempos de ejecución están medidos en milisegundos, vamos a considerar como el mejor algoritmo, en este caso, al algoritmo de Pirkul.

# 6. Anexo

## 6.1 Programa que calcula el valor óptimo del problema:

mkp_optimo.mc

```
! Problema MKP, soluci□n exacta

model "Problemas MKP OR-Library"
uses "mmxprs"
uses "mmsystem"


declarations
    m, n:integer
    zopt, z:real
    !archivo_datos = "c:/datos/MKP/mknap1/mknap1_7.dat"
    !archivo_datos = "c:/datos/MKP/mknap2-1/weish01-1.dat"
    archivo_datos = "weing1.txt"
    !archivo_datos = "sento2.txt"
end-declarations

fopen(archivo_datos,F_INPUT)
readln(m,n,m)

declarations
    objetos = 1..n
    recursos = 1..m
    p: array(objetos) of real
    r: array(recursos,objetos) of integer
    b: array(recursos) of integer

    x: array(objetos) of mpvar
    tiempo_bb = 30

end-declarations

forall(j in objetos)read(p(j))
forall(i in recursos)read(b(i))
forall(i in recursos,j in objetos)read(r(i,j))

read(zopt)
fclose(F_INPUT)

forall(j in objetos) x(j) is_binary

objetivo := sum(j in objetos) p(j)*x(j)

forall(i in recursos) res1(i):= sum(j in objetos) r(i,j)*x(j)<=b(i)

writeln("Problema MKP: ", archivo_datos,"\nn = ",n,", m = ",m)

exportprob(EP_MAX,"generado",objetivo)

t1:=gettime
setparam("XPRS_MAXTIME",tiempo_bb)
maximize(objetivo)
z:=objetivo.sol
writeln("\nSolucion entera: z = ",z.sol,", solucion libreria: ",zopt)
writeln("\ntiempo: ",gettime-t1)

!maximize(objetivo)
writeln("Valor final: ",getobjval)

!writeln("\nj\tx\n")
!forall(j in objetos)writeln(j,"\t",x(j).sol)
end-model
```

## 6.2 MultidimensionalKnapsack.java

```
package project;


import java.io.*;
import java.util.*;


import project.algorithms.PAKnapsack;
import project.algorithms.PirkulKnapsack;
import project.algorithms.GreedyKnapsack;
import project.model.KnapsackData;
```

```
import project.model.KnapsackItem;

public class MultidimensionalKnapsack {
private final static String INPUT = "mknap2/sento2.txt";
private final static boolean DEBUG = false;

public static void main(String[] args) throws IOException {


KnapsackData knapsackData = readKnapsackDataFromFile(INPUT);


long startTime = System.currentTimeMillis();
int result = GreedyKnapsack.dualRun(knapsackData, DEBUG);
long endTime = System.currentTimeMillis();
long executionTime = endTime - startTime;
System.out.println(String.format("\n\tAlgoritmo greedy Dual: %d. Miliseg: %d",
result, executionTime));
startTime = System.currentTimeMillis();
result = GreedyKnapsack.primalRun(knapsackData, DEBUG);
endTime = System.currentTimeMillis();
executionTime = endTime - startTime;
System.out.println(String.format("\n\tAlgoritmo greedy Primal: %d. Miliseg: %d",
result, executionTime));
startTime = System.currentTimeMillis();
result = PAKnapsack.run(knapsackData, DEBUG);
endTime = System.currentTimeMillis();
executionTime = endTime - startTime;
System.out.println(String.format("\n\tAlgoritmo Proyección Activa : %d. Miliseg:
%d", result, executionTime));


startTime = System.currentTimeMillis();
result = PirkulKnapsack.run(knapsackData, DEBUG);
endTime = System.currentTimeMillis();
executionTime = endTime - startTime;
System.out.println(String.format("\n\tAlgoritmo Pirkul : %d. Miliseg: %d", result,
executionTime));


}


private static KnapsackData readKnapsackDataFromFile(String file) throws
IOException {
BufferedReader br = new BufferedReader(new FileReader(INPUT));


int numKnapsacks = 0, numItems = 0, count = 0, optimumValue = 0;
int[] weights = null, capacities = null;
int[][] constraints = null;


String line;
while ((line = br.readLine()) != null) {
String[] items = line.split(" ");
for (String itemTxt : items)
if (itemTxt.trim().length() > 0){
```

```
int item;
try {
item = Integer.parseInt(itemTxt.trim());
}catch (Exception e) {
continue;
}


if (count == 0) {
numKnapsacks = item;
capacities = new int[numKnapsacks];
}else if (count == 1) {
numItems = item;
weights = new int[numItems];
constraints = new int[numKnapsacks][numItems];
}else if (count < numItems + 2) {
weights[count-2] = item;
}else if (count < numItems + 2 + numKnapsacks) {
capacities[count-numItems-2] = item;
}else if (count < numItems + 2 + numKnapsacks + numItems*numKnapsacks) {
int numKnapsack = (count-numItems-2-numKnapsacks)/numItems;
int numItem = (count-numItems-2-numKnapsacks)%numItems;
constraints[numKnapsack][numItem] = item;
}else
optimumValue = item;
count++;
}

}
/*weights                    =                    Arrays.stream(br.readLine().split("
")).mapToInt(Integer::parseInt).toArray();


int[]          capacities          =          Arrays.stream(br.readLine().split("
")).mapToInt(Integer::parseInt).toArray();


int[][] constraints = new int[numKnapsacks][numItems];
for (int i = 0; i < numKnapsacks; i++) {
constraints[i]                =                Arrays.stream(br.readLine().split("
")).mapToInt(Integer::parseInt).toArray();
}*/


List<KnapsackItem> itemsKnapsack = new ArrayList<>();
for (int i = 0; i < numItems; i++) {
int[] itemConstraints = new int[numKnapsacks];
for (int j = 0; j < numKnapsacks; j++)
itemConstraints[j] = constraints[j][i];
itemsKnapsack.add(new KnapsackItem(weights[i], itemConstraints));
}

//int optimumValue = Integer.parseInt(br.readLine());
br.close();


return    new    KnapsackData(numKnapsacks,    numItems,    weights,    capacities,
itemsKnapsack, optimumValue);
```

```
}


}
```

### 6.3 KnapsackData.java

```
package project.model;


import java.util.List;


public class KnapsackData {
private int numKnapsacks;
private int numItems;
private int[] weights;
private int[] capacities;
private List<KnapsackItem> items;
private int optimumValue;


public KnapsackData(int numKnapsacks, int numItems, int[] weights, int[]
capacities, List<KnapsackItem> items, int optimumValue) {
this.numKnapsacks = numKnapsacks;
this.numItems = numItems;
this.weights = weights;
this.capacities = capacities;
this.items = items;
this.optimumValue = optimumValue;
}


public int getNumKnapsacks() {
return numKnapsacks;
}


public int getNumItems() {
return numItems;
}


public int[] getWeights() {
return weights;
}


public int[] getCapacities() {
return capacities;
}


public List<KnapsackItem> getItems() {
```

```
return items;
}


public int getOptimumValue() {
return optimumValue;
}

}
```

### 6.4 KnapsackItem.java

```
package project.model;


public class KnapsackItem {
private int weight;
private int[] constraints;


public KnapsackItem(int weight, int[] constraints) {
super();
this.weight = weight;
this.constraints = constraints;

}


public int getWeight() {
return weight;
}


public void setWeight(int weight) {
this.weight = weight;
}


public int[] getConstraints() {
return constraints;
}


public void setConstraints(int[] constraints) {
this.constraints = constraints;
}


}
```

### 6.5 Pair.java

```
package project.model;
```

```
public class Pair {
private int index;
private double contributedValue;

public Pair(int index, double contributedValue) {
super();
this.index = index;
this.contributedValue = contributedValue;
}


public int getIndex() {
return index;
}


public double getContributedValue() {
return contributedValue;
}


@Override
public String toString() {
return "\n\tPair [index=" + index + ", contributedValue=" + contributedValue +
"]";
}


}
```

### 6.6 GreedyKnapsack.java

```
package project.algorithms;


import java.util.ArrayList;
import java.util.Comparator;


import project.model.KnapsackData;
import project.model.KnapsackItem;
import project.model.Pair;


public class GreedyKnapsack {


public static int primalRun(KnapsackData knapsackData, boolean debug) {
if (debug)
System.out.println("\n***************       Algoritmo       greedy       Primal
**********************" );
```

```
int numKnapsacks = knapsackData.getNumKnapsacks(); // N�mero de mochilas
//Inicializaci�n: Se comienza con una soluci�n inicial vac�a.
int solution = 0;
int[] restCapacities = knapsackData.getCapacities().clone();


//    C�lculo de valores relativos
ArrayList<Pair> contributedValue = calculateContributedValues(knapsackData, debug,
true);


//iteracion hasta que se cumpla condicion de parada lanzada con break
while (true) {
/*
Selecci�n de componente mas optimo para cualquier mochila:
al tener ya los valores ordenados solo tenemos que coger el primero si es que hay
valores aun
*/

//condicion de parada: no hay datos
if (contributedValue.isEmpty())
break;
//Mientra haya datos continuamos
//obtenemos el objeto
KnapsackItem                                selectedItem                                =
knapsackData.getItems().get(contributedValue.get(0).getIndex());

if (debug)
System.out.println("\n\nItem seleccionado " + selectedItem.getWeight() + ", peso
mochila 0: " + selectedItem.getConstraints()[0] + ", peso mochila 1: " +
selectedItem.getConstraints()[1]);

//Actualizaci�n de valores relativos: 
//hay capacidad: lo eliminamos de las listas de las otras mochilas para que no sea
elegido
boolean ability = true;
for (int j = 0; j < numKnapsacks; j++)
if(restCapacities[j] < selectedItem.getConstraints()[j])
ability = false;
if (ability) {
//Agregar componente a la soluci�n: Se agrega el componente seleccionado a la
soluci�n primal.
solution += selectedItem.getWeight();
for (int j = 0; j < numKnapsacks; j++)
restCapacities[j] -= selectedItem.getConstraints()[j];
}else
if (debug)
System.out.println("DESCARTADO" );

//lo eliminamos de la lista de valores ya sea porque pasa a la solucion o porque
no hay hueco
contributedValue.remove(0);

if (debug) {
System.out.println("Solucion: " + solution);
System.out.print("\tCapacidades: ");
```

```
for (int cap : restCapacities)
System.out.print(cap + " ");
System.out.print("\n\tRestantes: " );
for (Pair par : contributedValue)
System.out.print(knapsackData.getItems().get(par.getIndex()).getWeight() + " ");
}
}
if (debug)
System.out.println("\n***************       Algoritmo       greedy       Primal
**********************" );


return solution;
}


public static int dualRun(KnapsackData knapsackData, boolean debug) {
if (debug)
System.out.println("\n***************       Algoritmo       greedy       Dual
**********************" );


int numKnapsacks = knapsackData.getNumKnapsacks(); // N�mero de mochilas
//Inicializaci�n: Se comienza con una soluci�n maxima no factible.
int                                solution                                =
knapsackData.getItems().stream().mapToInt(KnapsackItem::getWeight).sum();
int[] actualCapacities = new int[numKnapsacks];
for (int j = 0; j < numKnapsacks; j++) {
int finalJ = j;
actualCapacities[j]    =    knapsackData.getItems().stream().mapToInt(item    ->
item.getConstraints()[finalJ]).sum();
}


//    C�lculo de valores relativos
ArrayList<Pair> contributedValue = calculateContributedValues(knapsackData, debug,
false);


//iteracion hasta que se cumpla condicion de parada lanzada con break
while (true) {
/*
Reducci�n de componentes: Selecci�n de componente menos optimo para cualquier
mochila:
al tener ya los valores ordenados solo tenemos que coger el primero si es que hay
valores aun
*/

//condicion de parada: no hay datos o solucion factible
boolean solutionOK = true;
for (int j = 0; j < numKnapsacks; j++)
if(actualCapacities[j] > knapsackData.getCapacities()[j])
solutionOK = false;

if (contributedValue.isEmpty() || solutionOK)
break;

//Reducci�n de componentes:
```

```
//    obtenemos el objeto
KnapsackItem                                selectedItem                                =
knapsackData.getItems().get(contributedValue.get(0).getIndex());

if (debug)
System.out.println("\n\nItem seleccionado " + selectedItem.getWeight() + ", peso
mochila 0: " + selectedItem.getConstraints()[0] + ", peso mochila 1: " +
selectedItem.getConstraints()[1]);

//lo eliminamos de la solucion
solution -= selectedItem.getWeight();
for (int j = 0; j < numKnapsacks; j++)
actualCapacities[j] -= selectedItem.getConstraints()[j];


//lo eliminamos de la lista de valores
contributedValue.remove(0);

if (debug) {
System.out.println("Solucion: " + solution);
System.out.print("\tCapacidades: ");
for (int cap : actualCapacities)
System.out.print(cap + " ");
System.out.print("\n\tRestantes: " );
for (Pair par : contributedValue)
System.out.print(knapsackData.getItems().get(par.getIndex()).getWeight() + " ");
}
}
if (debug)
System.out.println("\n**********************        Algoritmo        greedy        Dual
**********************" );
return solution;
}


/*    C�lculo de valores relativos: Se calcula un valor relativo para cada
componente no seleccionada en funci�n de su
contribuci�n a la funci�n objetivo.

En este caso se calculan para cada mochila y cada elemento donde impera el
coeficiente de
beneficio/coste.
*/
private     static     ArrayList<Pair>     calculateContributedValues(KnapsackData
knapsackData, boolean debug, boolean orderBetter) {
int numItems = knapsackData.getNumItems(); // N�mero de objetos
int numKnapsacks = knapsackData.getNumKnapsacks(); // N�mero de mochilas
ArrayList<Pair> contributedValue = new ArrayList<Pair>();
for (int i = 0; i < numItems; i++) {
double coef = 1.0;
KnapsackItem item = knapsackData.getItems().get(i);
for (int j = 0; j < numKnapsacks; j++) {
coef += item.getConstraints()[j];
}
contributedValue.add(new Pair(i, item.getWeight() / coef));
}

//las ordenamos para que los mejores valores sean los elegidos
```

```
contributedValue.sort(new Comparator<Pair>() {
@Override
public int compare(Pair o1, Pair o2) {
return                          orderBetter?Double.compare(o2.getContributedValue(),
o1.getContributedValue()):Double.compare(o1.getContributedValue(),
o2.getContributedValue());
}
 });

if (debug) {
System.out.println("valores calculados" );
for (int i = 0; i < numItems; i++)
System.out.println("\t"                                                          +
knapsackData.getItems().get(contributedValue.get(i).getIndex()).getWeight()  +  "-
>" + contributedValue.get(i).getContributedValue());
}


return contributedValue;
}
}
```

### 6.7 Helper.java

```
package project.algorithms;


import project.model.KnapsackData;


public class Helper {

public static int[] calculateTotalCapacities(int[] assignment, KnapsackData data)
{
int[] totalAbility = new int[data.getNumKnapsacks()];
// Inicializar asignaciones iniciales: en este caso repartiendo por orden
for (int i = 0; i < data.getNumKnapsacks(); i++) {
totalAbility[i] = 0;
 for (int j = 0; j < data.getNumItems(); j++) {
      totalAbility[i]               +=               assignment[j]               *
data.getItems().get(j).getConstraints()[i];
 }
}
return totalAbility;
}




public static int calculateTotalWeight(int[] assignment, KnapsackData data) {


int totalWeight = 0;
for (int j = 0; j < data.getNumItems(); j++) {
totalWeight += assignment[j] * data.getWeights()[j];
```

```
}
return totalWeight;
}

public static boolean evalueAbility(int[] assignment, KnapsackData data, int
indexElement) {
boolean ability = true;
int[] totalWeight = calculateTotalCapacities(assignment, data);
for (int k = 0; k < data.getNumKnapsacks(); k++)
if(totalWeight[k]   +   data.getItems().get(indexElement).getConstraints()[k]   >
data.getCapacities()[k])
ability = false;
return ability;
}


public static void printAssignment(int[] assignment, KnapsackData data) {
System.out.print("\tWeight : " + calculateTotalWeight(assignment, data) + " -> ");
for (int j = 0; j < data.getNumItems(); j++)
if (assignment[j] == 1)
System.out.print(data.getItems().get(j).getWeight() + "(" + j + ") ");
int[] totalWeight = calculateTotalCapacities(assignment, data);
System.out.print("\n\tCapacidades : ");
for (int k = 0; k < data.getNumKnapsacks(); k++)
System.out.print(totalWeight[k] + "(" + k + ") ");
System.out.println();
}
}
```

### 6.8 PAKnapsack.java

```
package project.algorithms;


import java.util.ArrayList;
import java.util.List;
import project.model.KnapsackData;


public class PAKnapsack {

private final static int N_ITERATION = 1000;


public static int run(KnapsackData data, boolean debug) {


int numItems = data.getNumItems();
int[] assignment = new int[numItems];
// Inicializar asignaciones iniciales: en este caso repartiendo al azar hasta que
no haya espacio
List<Integer> negativeAssignment = new ArrayList<Integer>();
//recopilamos los objetos no asignados: todos
for (int j = 0; j < numItems; j++)
negativeAssignment.add(j);
```

```
//elegimos elementos mientras no queden objetos
while (negativeAssignment.size() > 0) {
int    selectedItem    =    negativeAssignment.remove((int)    (Math.random()    *
negativeAssignment.size()));
if (Helper.evalueAbility(assignment, data, selectedItem))
assignment[selectedItem] = 1;
}

if (debug) {
 System.out.println("ASIGNACION INICIAL:");
 Helper.printAssignment(assignment, data);
}
// Condicion de parada: recorremos todos los elementos N_ITERATION veces
for (int iter = 0; iter < N_ITERATION; iter++)
for (int selectedIndex = 0; selectedIndex < numItems; selectedIndex++) {
//b?squeda local y la proyecci?n activa: eliminamos un elemento de manera ordenada
de izq a derecha y proyectamos con una combinaci?n de los elementos no
seleccionados
if (assignment[selectedIndex] == 1) {
int[] tempAssignment = assignment.clone();
tempAssignment[selectedIndex] = 0;

      //Proyecci?n Activa
      negativeAssignment = new ArrayList<Integer>();
      //rellenamos hasta el tope con datos al azar: primero recopilamos los objetos
no asignados (sin el que acabamos de eliminar):
      for (int j = 0; j < numItems; j++)
      if (assignment[j] == 0)
            negativeAssignment.add(j);
 //elegimos elementos mientras no queden objetos
      while (negativeAssignment.size() > 0) {
      int   selectedItem   =   negativeAssignment.remove((int)   (Math.random()   *
negativeAssignment.size()));
      if (Helper.evalueAbility(tempAssignment, data, selectedItem))
            tempAssignment[selectedItem] = 1;
      }

 if (debug) {
       System.out.println("SOLUCION TEMPORAL: ");
       Helper.printAssignment(tempAssignment, data);
 }

      // Evaluar intercambios y proyecciones
      if        (Helper.calculateTotalWeight(tempAssignment,        data)        >
Helper.calculateTotalWeight(assignment, data)) {
            assignment = tempAssignment;
            //si hay mejora empezamos desde el primer elemento nuevamente
            selectedIndex = 0;
            if (debug)
                  System.out.println("MEJORADA");
      }
}
}
if (debug) {
 System.out.println("SOLUCION FINAL:");
 Helper.printAssignment(assignment, data);
}
return Helper.calculateTotalWeight(assignment, data);
}
```

```
}
```

### 6.9 PirkulKnapsack.java

```
package project.algorithms;


import java.util.ArrayList;
import java.util.List;
import project.model.KnapsackData;


public class PirkulKnapsack {

private final static int N_ITERATION = 1000;
private final static int N_DELETES = 2;


public static int run(KnapsackData data, boolean debug) {


int numItems = data.getNumItems();
int[] assignment = new int[numItems];
// Inicializar asignaciones iniciales: en este caso repartiendo al azar hasta que
no haya espacio
List<Integer> negativeAssignment = new ArrayList<Integer>();
//recopilamos los objetos no asignados: todos
for (int j = 0; j < numItems; j++)
negativeAssignment.add(j);
//elegimos elementos mientras no queden objetos
while (negativeAssignment.size() > 0) {
int    selectedItem    =    negativeAssignment.remove((int)    (Math.random()    *
negativeAssignment.size()));
if (Helper.evalueAbility(assignment, data, selectedItem))
assignment[selectedItem] = 1;
}

if (debug) {
 System.out.println("ASIGNACION INICIAL:");
 Helper.printAssignment(assignment, data);
}
// Iterar para mejorar las asignaciones: si en el numero indicado numIterations no
encontramos mejora paramos. Si encontramos mejora reiniciamos
for (int iter = 0; iter < N_ITERATION; iter++) {
// Evaluacion de vecindario: eliminamos uno al azar y rellenamos al azar hasta el
tope
int[] tempAssignment = assignment.clone();
List<Integer> positiveAssignment = new ArrayList<Integer>();
for (int j = 0; j < numItems; j++)
      if (assignment[j] == 1)
            positiveAssignment.add(j);
for (int j = 0; j < N_DELETES; j++) {
```

```
int selectedIndex = positiveAssignment.remove((int) (Math.random() *
positiveAssignment.size()));
 tempAssignment[selectedIndex] = 0;
}


negativeAssignment = new ArrayList<Integer>();
//rellenamos hasta el tope con datos al azar: primero recopilamos los objetos no
asignados (sin el que acabamos de eliminar):
for (int j = 0; j < numItems; j++)
      if (assignment[j] == 0)
            negativeAssignment.add(j);
 //elegimos elementos mientras no queden objetos
while (negativeAssignment.size() > 0) {
int selectedItem = negativeAssignment.remove((int) (Math.random() *
negativeAssignment.size()));
//movimientos permitidos
if (Helper.evalueAbility(tempAssignment, data, selectedItem))
tempAssignment[selectedItem] = 1;
}

if (debug) {
 System.out.println("SOLUCION TEMPORAL: ");
 Helper.printAssignment(tempAssignment, data);
}
// Evaluar intercambios y ajustar asignaciones para mejorar funci□n objetivo
//si la solucion mejora nos quedamos con ella, sino se desecha
//criterio de aceptacion
if (Helper.calculateTotalWeight(tempAssignment, data) >
Helper.calculateTotalWeight(assignment, data)) {
assignment = tempAssignment;
iter = 0;
if (debug)
System.out.println("MEJORADA");
}

}
if (debug) {
 System.out.println("SOLUCION FINAL:");
 Helper.printAssignment(assignment, data);
}
return Helper.calculateTotalWeight(assignment, data);
}

}
```

# 7. Bibliografía


1. Holte, R.C. (2001) Combinatorial Auctions, Knapsack Problems, and Hill-Climbing Search. Proceedings of the 14th Conference of the Canadian Society for Computational Studies of Intelligence (AI 2001), Ottawa, 7-9 June 2001, 57-66.
2. Kelly, T. (2004) Generalized Knapsack Solvers for Multi-Unit Combinatorial Auctions: Analysis and Application to Computational Resource Allocation. Proceedings of the International Workshop on Agent Mediated-Electronic Commerce VI: Theories for and Engineering of Distributed Mechanisms and Systems (AMEC 2004), New York, NY, 19 July 2004, 73-86.
3. Chen, F., La Porta, T. and Srivastava, M.B. (2012) Resource Allocation with Stochastic Demands. Proceedings of the IEEE 8th International Conference on Distributed Computing in Sensor Systems (DCOSS 2012), Hangzhou, 16-18 May 2012, 257-264.
4. Y. Toyoda, A simplified algorithm for obtaining approximate solutions to zero–one programming problems, Management Sciences 21 (1975) 1417–1427.
5. S. Senju, Y. Toyada, An approach to linear programming problems with 0–1 variables, Management Sciences 15 (1968) 196–207.
6. G. Kochenberger, G. McCarl and F. Wymann. A heuristic for general integer programming. Decision Sciences,. 5 (1974), 36-44.
7. R. Loulou and E. Michaelides. New greedy-like heuristics for the multidimensional 0–1 knapsack problem. Operations Research. 27 (1979), 1101-1114.
8. F. S. Hillier. Efficient heuristic procedures for integer linear programming with an interior. Operations Research. 17 (1969), 600–637.
9. S. H. Zanakis. Heuristic 0–1 linear programming: An experimental comparison of three methods. Management Science. 24 (1977), 91–104.
10. E. Balas and C. H. Martin. Pivot and complement – A heuristic for 0–1 programming. Management Science. 26 (1980), 86-96.
11. A. Freville and G. Plateau. An efficient preprocessing procedure for the multidimensional 0-1 knapsack problem. Discrete Applied Mathematics. 49 (1994), 189–212.
12. A. Freville and G. Plateau. Heuristics and reduction methods for multiple constraints 0–1 linear programming problems. European Journal of Operational Research. 24 (1986), 206–215.
13. J. S. Lee and M. Guignard. An approximate algorithm for multidimensional zero–one knapsack problems. A parametric approach. Management Science. 34 (1988), 402– 410.
14. S. Hanafi and A. Freville. An efficient tabu search approach for the 0–1 multidimensional knapsack problem. European Journal of Operational Research. 106 (1998), 659–675.
15. E. Balas, S. Ceria, M. Dawande, F. Margot and G. Pataki. A new heuristic for pure 0–1 programs Operations Research, OCTANE. 49 (2001), 207–225.
16. S. Martello and P. Toth. A new algorithm for the 0-1 knapsack problem. Management Science. 34 (1988), 633-644.
17. A. Plateau, D. Tachat and P. Tolla. A hybrid search combining interior point methods and metaheuristics for 0–1 programming. International Transactions in Operational Research. 9 (2002), 731–746.
18. P. C. Chu and J. E. Beasley. A genetic algorithm for the multidimensional knapsack problem. Journal of Heuristics. 4 (1998), 63-86.
19. S. Balev, A. Freville, N. Yanev and R. Andonov. A dynamic programming based reduction procedure for the multidimensional 0–1 knapsack problem. European Journal of Operations Research,. (1) 186 (2008), 63-76.

20. A. M. Frieze and M. R. B. Clarke. Approximation Algorithms for the m- Dimensional 0-1 Knapsack Problem: Worst-Case and Probabilistic Analysis. European Journal of Operational Research. 15 (1984), 100-109.
21. A. H. G. Rinnooy Kan, L. Stougie and C. Vercellis. A Class of Generalized Greedy Algorithms for the Multi-knapsack Problem. Discrete Applied Mathematics. 42 (1993), 279–290.
22. G. E. Fox and G. D. Scudder. A Heuristic with Tie Breaking for Certain 0-1 Integer Programming Models. Naval Research Logistics Quarterly. 32 (1985), 613–623.
23. Laabadi, S., Naimi, M., El Amri, H. and Achchab, B. (2018) The 0/1 Multidimensional Knapsack Problem and Its Variants: A Survey of Practical Models and Heuristic Approaches. American Journal of Operations Research, 8, 395-439.