Trabajo de Fin de Grado

Grado en Ingeniería Informática

Biblioteca Java para algoritmos generadores de la región eficiente de un MOILP

Java class library for generator algorithms of the efficient set of a MOILP

Natalie Dajakaj

La Laguna, 28 de Junio de 2017

D. **Jesús Manuel Jorge Santiso**, con N.I.F. 42.097.398-S profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

C E R T I F I C A

Que la presente memoria titulada:

“Biblioteca Java para algoritmos generadores de la región eficiente de un MOILP”

ha sido realizada bajo su dirección por D.ª **Natalie Dajakaj**,
con N.I.F. X-7.380.743-C.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firma la presente en La Laguna a 28 de Junio de 2017.

Agradecimientos

Quiero agradecer a mis amigos, con los que he pasado muy buenos ratos en los descansos que todos hacíamos del TFG. También quiero agradecer a mi pareja que me ha apoyado en los momentos duros y me ha hecho compañía en las largas noches de programación.

Licencia

Resumen

El objetivo de este trabajo es realizar una biblioteca Java de clases en la que se implementen los algoritmos exactos más relevantes actualmente para la generación de la región eficiente de un problema de optimización multiobjetivo lineal entero (MOILP). Se pone especial énfasis en el diseño de la biblioteca de clases, y se trata de generalizar suficientemente las clases que implementan los distintos algoritmos, de forma que se puedan extender fácilmente y/o combinar distintos procedimientos para obtener nuevos enfoques de resolución. Además se ha creado una API con los distintos solvers escalares para que el usuario pueda usarlos todos de la misma forma, sin tener que aprender a usar cada uno de ellos.

Palabras clave: multiobjetivo, optimización, MOILP, región eficiente, solver.

Abstract

The objective of this work is make a Java class library that implements the most relevants exact algorithms in generation of the efficient set of a integer linear multiobjective optimization problem (MOILP). Special emphasis is placed on the design of the class library and we try to generalize the classes that implements the different algorithms sufficiently, so they can be extended easily and/or combine the different procedures to get new approaches to resolution. We also create an interface with the solvers we use, so the user can use all of them in the same way, without having to learn how to use each one.

Keywords: multiobjective, optimization, MOILP, efficient set, solver.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1 Introducción

1.1 Antecedentes

La optimización multiobjetivo consiste en optimizar varias funciones objetivos sobre una determinada región factible. Este campo se está estudiando con gran interés y aún queda mucho por explorar. Ha ganado tanta importancia debido a que representa mucho mejor la realidad, pues lo más común es encontrar un problema con múltiples criterios a tener en cuenta, en lugar de tomar una decisión con un solo criterio.

Un ejemplo muy sencillo en el que podemos ver esto es en la búsqueda de una casa, nos interesa que sea grande, que esté cerca del centro de la ciudad y que sea lo más barata posible. En este ejemplo vemos que tenemos varios criterios a tener en cuenta, los cuales son:

- Maximizar el tamaño de la casa.
- Minimizar la distancia al centro de la ciudad.
- Minimizar el coste.

Además vemos que se pueden combinar funciones objetivo que maximizan y minimizan en un mismo problema.

En este TFG vamos a estudiar los problemas multiobjetivos lineales enteros (Multiobjective Integer Linear Programming - MOILP), que son aquellos en que todas las restricciones y funciones objetivos son lineales y las variables enteras.

1.2 Estado del arte

La optimización multiobjetivo presenta varios problemas con respecto a la optimización mono-objetivo.

En primer lugar, las funciones objetivo tienen cierto grado de conflicto, lo que hace que al mejorar una de las funciones sea muy común que otras de las funciones empeoren su valor objetivo. Siguiendo con el ejemplo de la casa vemos este conflicto, cuanto más grande sea la casa mayor será su precio, por lo que mejoramos un criterio

y empeoramos otro. Por esto, no existe una única solución que optimice todas las funciones, sino que existe un conjunto de soluciones no dominadas.

A raíz de esto surge otro problema, y es que este conjunto de soluciones puede tener un gran tamaño. Cuantas más soluciones tenga el problema multiobjetivo más difícil será obtenerlas todas, consumiendo cada vez más tiempo e incluso puede darse el caso en el que no se pueda resolver el problema debido a que existan infinitas soluciones.

Otro problema es que estamos trabajando en el caso entero, el cual es mucho más difícil de resolver que el caso continuo, debido a que el conjunto de soluciones factibles pierde la propiedad de convexidad.

Actualmente existen varios algoritmos que generan este conjunto de soluciones, siendo algunos pequeñas mejoras de algoritmos que ya existían. En estos algoritmos se resuelven varios problemas mono-objetivos, siempre intentando reducir el número total de problemas que se resolverán y la dificultad de ellos.

Estos algoritmos presentan una serie de problemas o dificultades a la hora de estudiarlos:

- No se han estudiado suficientemente: los algoritmos no se han comparado entre ellos por lo que no sabemos cuáles son mejores o peores. Esto hace que la elección de los algoritmos con los que alguien va a trabajar se haga a ciegas, sin saber cuál es la mejor opción en cada caso.

- Diferente notación: cada autor utiliza su propia notación, en algunos casos es similar, pero en otros es completamente diferente. Por esto tenemos que dedicar un tiempo extra al estudio del algoritmo, para poder entender la notación que se está usando.

- Lenguaje muy técnico: la forma en la que se explican los algoritmos es muy compleja, llegando a ser críptica en algunos casos. Esto hace que las personas que no son expertas en el tema necesiten mucho tiempo para entender qué es lo que se está explicando.

- Falta de ejemplos: son pocos los algoritmos que vienen acompañados de un ejemplo en el que se resuelva un pequeño problema paso a paso. En los que se incluyen este tipo de ejemplos, es mucho más sencillo entender cómo funciona.

Pero lo más común es que no aparezca un ejemplo, o esté a medias, y en estos casos es mucho más difícil entender la idea en la que se basa.

Estos problemas son los que nos han motivado a realizar este proyecto, unificando y comparando los algoritmos más relevantes para generar la región eficiente de un MOILP.

1.3 Objetivos

El primer objetivo es implementar algunos de los algoritmos más recientes y relevantes de la literatura, concretamente implementaremos cuatro algoritmos: el de Sylva y Crema [1] (2004), el de Boland, Charkhgard y Savelsbergh [2] (2017), y los dos algoritmos que se presentan en el artículo de Lokman y Köksalan [3] (2013).

Al implementar nosotros los cuatro objetivos, vamos a homogeneizar la notación usando la misma en todos los algoritmos. De esta forma solucionamos uno de los problemas que comentamos anteriormente, el tiempo extra que hay que dedicar a entender la nueva notación que usa el autor.

Cuando hayamos implementado todos los algoritmos, podremos probarlos con un conjunto de problemas de prueba, y realizar un estudio computacional en el que se compare el rendimiento de ellos. Aquí también solucionamos otro de los problemas que tenemos, pues podemos saber en qué casos trabajan mejor cada uno de los algoritmos. Esto es de gran utilidad cuando queremos decidir el algoritmo que mejor se adapte a nuestras necesidades.

Un objetivo que no está directamente relacionado con los algoritmos, pero que nos es de gran interés, es encapsular los solvers en una API homogeneizando el uso de todos ellos. Con este API común queremos ofrecer al usuario la posibilidad de trabajar de forma transparente con distintos solvers escalares, sin que necesite conocer los métodos propios y demás detalles de cada uno de ellos.

Como vemos, un punto muy importante en nuestro trabajo de final de grado es la homogeneización, estando presente en la implementación de los algoritmos y en la encapsulación de los solvers.

Capítulo 2 Conceptos básicos

2.1 Problema multiobjetivo

El problema de optimización multiobjetivo consiste en optimizar simultáneamente un conjunto de p funciones objetivo, z , donde $p >= 2$ sobre una determinada región factible $X \subseteq R^n$. Matemáticamente, el problema se puede formular como:

$$\begin{aligned} max \quad & z(x) \\ s.t. \quad & x \in X \end{aligned}$$

Donde $z(x) = (z_1(x), ..., z_p(x))$

Además nosotros estamos trabajando en el caso entero, por lo que en la región factible siempre exigiremos que todas las variables del problema sean enteras. Así, $X \subseteq Z^n$.

En estos problemas queremos conseguir el mejor valor para cada función objetivo, pero es muy raro encontrar una solución única que dé el valor óptimo de todas las funciones (solución ideal).

2.2 Solución eficiente

En lugar de tener una única solución, lo común en los problemas MOILP es tener un conjunto de soluciones eficientes.

Estas soluciones también son conocidas como solución no dominada o solución Pareto. Una solución factible $x' \in X$ se llama eficiente si no existe una solución $x \in X$ que cumpla:

$$z(x) \geq z(x') \quad donde \quad z(x) := (z_1(x), ..., z_p(x))$$

2.3 Espacio de decisión

Es el espacio en el que está contenida la región factible, es decir, el conjunto de posibles decisiones que tenemos disponibles. Los algoritmos que buscan en este espacio, deben explorar las soluciones factibles y descartar aquellas que sean dominadas. A veces se usan indistintamente los conceptos de región factible y espacio de decisión.

2.4 Espacio de criterios

Es el espacio en el que se encuentran los valores de las funciones objetivo. Es mucho más sencillo y rápido encontrar los puntos no dominados en este espacio debido a que usualmente tiene menor dimensionalidad que el espacio de decisión. Por ello los algoritmos que buscan en este espacio deberían ser más eficientes que los que buscan en el espacio de decisión.

2.5 Región factible

Es la región del espacio de decisión en la que las variables pueden tomar valores. Se corresponde con las restricciones del problema, pues son las que delimitan las zonas en las que estarán las soluciones.

2.6 Región eficiente

Región formada por todos los puntos o soluciones eficientes del problema, por lo que para construir este región necesitamos generar una a una las soluciones eficientes.

Capítulo 3 Solvers Escalares

3.1 Introducción

En este trabajo de final de grado se utilizarán varios solvers escalares para resolver los problemas mono-objetivos lineales que surjan al generar la región eficiente de los problemas multiobjetivo.

En primer lugar, probaremos la herramienta propia de cada solver, y así comprobaremos que podemos resolver la clase de problemas que necesitamos. Después de esto incluiremos las librerías en Java y realizaremos unos programas sencillos, para aprender cómo funcionan los métodos. Por último, encapsularemos los algoritmos en nuestra API *MySolver*.

3.2 Elección de solvers

En un principio se barajaron cuatro solvers: CPLEX [5], XPRESS [6], Gurobi [4] y Coin-OR [7]. Lo que tenemos que hacer es instalarlos y probarlos con problemas sencillos para decidir si finalmente trabajaremos con todos ellos o descartaremos alguno.

Primero vamos a descargar la herramienta y trabajar directamente con ella, es decir, aún no crearemos ningún programa en Java que resuelva los problemas. Cuando descarguemos las herramientas tendremos que pedir las correspondientes licencias académicas. Tras probar la instalación de los solvers ya podemos descartar uno de los ellos, pues XPRESS solamente deja descargar la versión para Windows y nosotros estamos trabajando en Mac. Además, tampoco se pudo descargar el solver Coin-OR, concretamente el proyecto clp de Coin-OR, porque no tiene instalador para Mac, pero no lo descartamos en este paso porque encontramos una librería Java de este solver. Con respecto a Gurobi y CPLEX, su instalación no dio ningún problema y su uso es sencillo.

Una vez instalados los solvers, pasamos a realizar un programa Java sencillo para cada solver, en el que leemos ficheros con problemas LP sencillos y los resolvemos. Haciendo esto identificamos los métodos que necesitaremos en nuestro proyecto y

aprenderemos a usar las librerías. Usando la librería de Coin-OR vemos que no está bien documentada y que le faltan métodos, por lo que también lo descartamos.

Finalmente solo trabajaremos con Gurobi y CPLEX, al ser los únicos solvers que no nos han dado ningún problema.

3.3 Funcionamiento de los solvers

A continuación, explicaremos de una forma resumida el funcionamiento de los solvers, comentando los métodos y tipos de datos que necesitaremos usar en nuestro proyecto. También comentaremos cómo incluir las librerías en Java para poder usar todos estos métodos.

No es necesario aprender a usar los solvers, pues con el API que implementaremos todos se podrán usar de la misma forma. Aunque no necesitemos esta sección es interesante tener un conocimiento básico de las herramientas que usa el proyecto.

CPLEX

En primer lugar, incluimos la librería en nuestro proyecto, desde eclipse solo tenemos que modificar el Build Path añadiendo el fichero .jar que estará en una ruta similar a la siguiente:

```
IBM/ILOG/CPLEX_Studio127/cplex/lib/cplex.jar
```

El tipo de dato más básico es `IloCplex`, es el encargado de llamar a la gran mayoría de los métodos para resolver los diferentes modelos de programación matemática y almacena los problemas. Otro tipo de dato es `IloLPMatrix`, los objetos de este tipo permiten trabajar de una forma sencilla con un conjunto de restricciones de rango (objetos de tipo `IloRange`) y variables (`IloNumVar`), lo que hace que el acceso a estos dos tipos de objetos sea rápido.

Tras hablar de los tipos de datos que usa la librería, vamos a comentar algunos de los métodos que usamos para resolver el problema y obtener las soluciones:

- `solve()`: resuelve el problema que está almacenado en el objeto `IloCplex`.

- `getObjValue()`: devuelve el valor objetivo que se obtiene tras resolver el problema.

- `getValues(IloNumVar[] arg)`: usamos este método para saber los valores que toman las distintas variables en la solución que se ha calculado.

- `getNumVars()`: en el método anterior hay que pasar como argumento un vector con las variables del problema. Estas las podemos obtener llamando a este método con el objeto `IloLPMatrix`.

Gurobi

La inclusión de la librería de este solver es un poco más compleja que la del solver anterior. Primero añadimos la librería al proyecto Java, de igual forma que con CPLEX, que estará en una ruta similar a la siguiente:

```
/Library/gurobi702/mac64/lib/gurobi.jar
```

Después de hacer esto tenemos que añadir un argumento a la ejecución del programa. En Eclipse, dentro de *Run configurations*, vamos a la pestaña *Arguments* y en el cuadro de *VM Arguments* escribimos el siguiente argumento:

```
-Djava.library.path=/Library/gurobi702/mac64/lib
```

Este argumento se tendrá que añadir siempre que se trabaje con Gurobi, si no hacemos esto no funcionará el programa, al no encontrar la librería.

Para especificar con qué solver vamos a trabajar, cambiamos el valor de una constante de tipo String del programa. Este es el único paso para trabajar con un solver u otro, todo lo demás se hará de forma automática.

La forma de almacenar el modelo en este solver es distinta que en CPLEX, pues utiliza dos tipos de datos en lugar de uno solo. El primer tipo de dato que usamos es `GRBEnv`, con esto creamos un entorno,necesario porque todos los modelos deben estar relacionados a un entorno para poder ser creados. El otro tipo de dato que usamos es `GRBModel` y es el que almacena el modelo matemático. Una vez tenemos almacenado el problema que queremos resolver, necesitamos también el tipo de dato `GRBVar` para guardar las variables, y así poder acceder a ellas posteriormente y recuperar los valores que toman.

Igual que hicimos con CPLEX vamos a explicar los métodos con los que resolvemos los problemas y obtenemos los resultados:

- `optimize()`: es el método que resuelve el problema almacenado en el modelo que realiza la llamada.

- `get(GRB.DoubleAttr.ObjVal)`: con esto podemos obtener el valor objetivo de la solución.

- `get(GRB.Attr.X, GRBVar[ ] vars)`: devuelve los valores que las variables toman en la solución.

3.4 API MySolver

Desde un principio se decidió que este proyecto incluiría más de un solver, para que el usuario tuviera varias opciones y pudiese comparar el efecto en el rendimiento de la biblioteca al usar cada uno de ellos.

Un problema que puede surgir con esta idea es que el usuario necesitaría aprender a usar cada uno de los solvers, pues usan distinta nomenclatura, distintos tipos de datos, métodos, etc.

La solución a este problema es hacer un API con los métodos que se van a necesitar, para que todos los solvers se usen de la misma forma. Internamente, sin que el usuario se de cuenta, se llamarán a los métodos propios de cada solver para que resuelva el problema. Necesitaremos crear nuestra propia estructura de clases para representar los problemas y tener un punto común que después transformemos. Esta estructura la explicaremos más adelante.

Hemos usado un patrón de diseño Strategy en el desarrollo de esta API. En la siguiente imagen podemos ver la estructura de la API que encapsula los solvers:

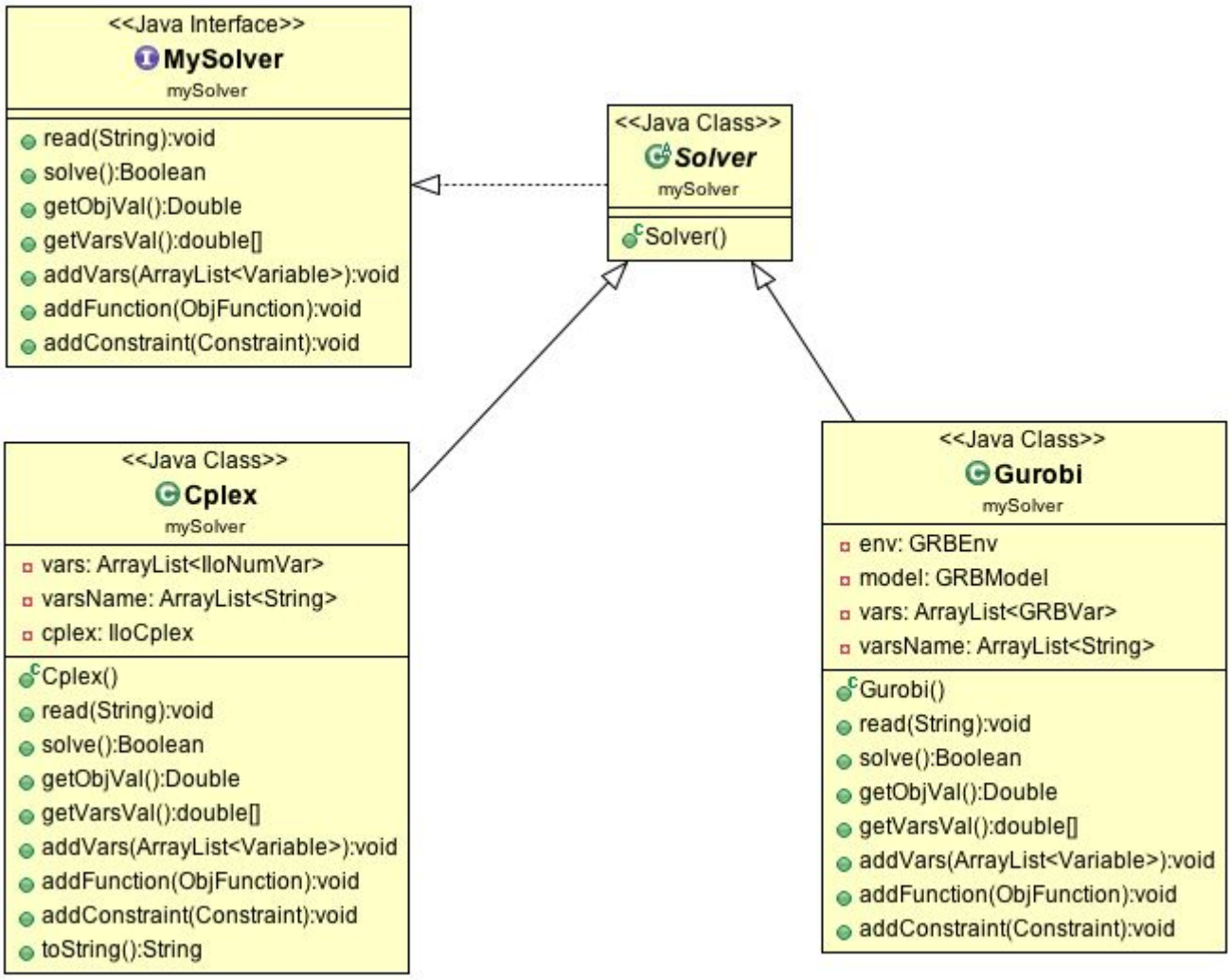

Figura 1: MySolver

El primer paso fue crear la interfaz `MySolver` en la que se definen los métodos que vamos a necesitar, pero sin incluir ninguna implementación. Esto lo hacemos para agrupar en un fichero las funciones que usaremos sin tener que ver si se trata de un solver u otro.

Lo siguiente es hacer una clase abstracta vacía, sin ningún método abstracto dentro, que añada la interfaz que acabamos de crear. Este paso es para forzar a todas las clases que hereden de `Solver` a implementar los métodos de la interfaz, y así garantizar que todos los solvers que incluyamos tendrán todas las funciones necesarias.

Por último, creamos una clase que herede de `Solver` por cada solver con el que queramos trabajar, y dentro de estas clases definimos cómo se resolverán los problemas, es decir, en esta parte es en la que se usan los métodos propios de cada solver. Esta es la parte que transforma nuestra estructura de datos en la estructura que usa Gurobi o CPLEX.

La forma en la que utilizamos esto en nuestro proyecto es la siguiente:

- Tenemos un objeto `Solver` que instanciamos según el solver que se especifique en el programa, es decir, llamaremos al constructor de una de las clases hijas dependiendo del valor que se guarde.

- Este solver usará internamente los métodos propios, pero para el usuario sólo serán los de nuestra API.

- Al ser de tipo `Solver` solo tenemos que tener en cuenta el tipo de solver al crearlo, pero en el resto del programa se usan todos de la misma forma.

Capítulo 4 Algoritmos generadores

4.1 Introducción

Este capítulo tiene un apartado por cada conjunto de autores estudiado, en el que se explica la idea en la que se basa cada algoritmo para obtener las soluciones. Todos ellos construyen varios problemas lineales con los que van obteniendo las soluciones. Cuando expliquemos la forma de construir estos problemas usaremos la misma notación en todos, teniendo presente uno de los principales objetivos del trabajo, conseguir la homogeneización de la notación. La forma de definir los problemas lineales mono-objetivo es:

$$\begin{aligned} & max \quad Cx \\ & \quad s.t.\ x \in X = \{x \in Z^n \ / \ Ax = b,\ x \geq 0\} \end{aligned}$$

También usaremos un problema sencillo para mostrar cómo se aplica lo explicado, pues ver en funcionamiento los algoritmos con un ejemplo real ayuda a entender los conceptos. El problema que usaremos es el siguiente:

$$\begin{aligned} (P):\ & max \quad x_1 - 2x_2 \\ & max \quad -x_1 + 3x_2 \\ & \quad s.t.\ x_1 - 2x_2 \leq 0 \\ & \quad\quad x_1, x_2 \in \{0, 1, 2\} \end{aligned}$$

4.2 Algoritmo de Sylva y Crema [1]

Los autores presentan este algoritmo como un planteamiento teórico sencillo para enumerar todas las soluciones no dominadas de un MOILP. La idea principal de este algoritmo es resolver una serie de problemas lineales enteros cada vez más restringidos, consiguiendo en cada iteración obtener una nueva solución no dominada.

La forma de construir estos problemas cada vez más restringidos es la siguiente:

$$
\begin{aligned}
(P_l):\ max \quad & \sum_{k=1}^{p} \lambda_k z_k(x) \\
s.t.\ \ & x \in X \\
& z_k \geq (z_k^s(x) + 1)\, y_k^s - M_k (1 - y_k^s), \\
& \quad for\ s = 1, ..., l; \quad k = 1, ..., p, \\
& \sum_{k=1}^{p} y_k^s \geq 1, \quad y_k^s \in \{0, 1\} \\
& \quad for\ s = 1, ..., l; \quad k = 1, ..., p, \\
& x \geq 0, \quad x \in Z^n
\end{aligned}
$$

Explicaremos cada una de las partes que participan en la construcción de los problemas:

- Función objetivo: es la suma de las funciones objetivo del problema multiobjetivo multiplicadas por un vector de pesos. Todos los pesos deben ser estrictamente positivos.

- Restricciones del problema original: en todos los problemas que se resuelvan estarán presentes las restricciones del problema inicial, que se representan como $x \in X$, y conforman la región factible del problema.

- Nuevas restricciones: en cada iteración (l) se añaden $p + 1$ restricciones, siendo p el número de funciones objetivo del MOILP. $z(x)$ son las funciones objetivo, y $(z_k^s(x))$ es el valor que toma la función objetivo k en la solución s. y_k^s es el nombre que se da a las nuevas variables del problema que se añaden en cada paso, y $-M$ es un vector de cotas inferiores, es decir, los valores mínimos que pueden tomar las funciones objetivo. Hasta aquí se han añadido p restricciones, y la que falta garantiza que la suma de las nuevas variables que aparecen en esta iteración va a valer al menos uno, lo que fuerza que al menos

una de las restricciones anteriores se active. Esto significa que al menos uno de los valores objetivos que se obtuvo en el paso anterior va a ser mejorado, y por lo tanto, la nueva solución que obtengamos será no dominada.

Este algoritmo acaba cuando llegamos a un problema no factible, es decir, no se ha podido encontrar una solución que lo satisfaga. Los problemas que se resuelven al principio son sencillos, pero con cada iteración se vuelven más difíciles. Esto se traduce en que el crecimiento del conjunto de soluciones eficientes está directamente relacionado con el aumento de la dificultad del problema. Pero los puntos fuertes de este algoritmo son:

- Obtención rápida de un subconjunto de soluciones: al ser sencillos los primeros problemas que se resuelven, podemos limitar el número de soluciones que vamos a obtener, y en poco tiempo tener un subconjunto de ellas.

- Nueva solución en cada iteración: este algoritmo consigue proporcionarnos una nueva solución no dominada en cada paso, algo que no pueden hacer otros de los algoritmos que hemos estudiado. Esto influye en la ventaja anterior, pues si no pudiera garantizar esto, al limitar el número de pasos que realiza podemos obtener menos soluciones que iteraciones realizadas, ya que algunos modelos que se resuelvan pueden obtener soluciones repetidas o tratarse de problemas no factibles.

Vamos a mostrar una traza de cómo se aplica este algoritmo en el problema especificado en el apartado anterior. No resolveremos todo el problema, solo mostraremos las tres primeras iteraciones:

Vector de pesos:
$\lambda = [4, 3]$
Función objetivo:
$4 (x_1 - 2 x_2) + 3 (- x_1 + 3 x_2) = 4 x_1 - 3 x_1 - 8 x_2 + 9 x_2 = x_1 + x_2$

Iteración 0:

```
Maximize
    + 1.0 x1 + 1.0 x2
Subject To
    1.0 x1 -2.0 x2 <= 0
    x1 -> {0, 2}
    x2 -> {0, 2}
P0 SOLUTION: (2.0, 2.0), (-2, 4)
```

Iteración 1:

```
Maximize
    + 1.0 x1 + 1.0 x2
Subject To
    1.0 x1 -2.0 x2 <= 0
    1.0 x1 -2.0 x2 -3.0 y1-1 >= -4
    -1.0 x1 3.0 x2 -7.0 y1-2 >= -2
    1.0 y1-1 1.0 y1-2 >= 1
    x1 -> {0, 2}
    x2 -> {0, 2}
    y1-1 -> {0, 1}
    y1-2 -> {0, 1}
P1 SOLUTION: (1.0, 2.0, 0.0, 1.0),(-3, 5)
```

Iteración 2:

```
Maximize
    + 1.0 x1 + 1.0 x2
Subject To
    1.0 x1 -2.0 x2 <= 0
    1.0 x1 -2.0 x2 -3.0 y1-1 >= -4
    -1.0 x1 3.0 x2 -7.0 y1-2 >= -2
    1.0 y1-1 1.0 y1-2 >= 1
    1.0 x1 -2.0 x2 -2.0 y2-1 >= -4
    -1.0 x1 3.0 x2 -8.0 y2-2 >= -2
    1.0 y2-1 1.0 y2-2 >= 1
    x1 -> {0, 2}
    x2 -> {0, 2}
    y1-1 -> {0, 1}
    y1-2 -> {0, 1}
    y2-1 -> {0, 1}
    y2-2 -> {0, 1}
P2 SOLUTION: (2.0, 1.0, 1.0, -0.0, 1.0, 0.0), (0, 1)
```

En el apéndice uno [A1] se puede ver la traza completa de un problema de mayor tamaño (cuatro funciones objetivo, una restricción y diez variables). En ese documento se puede ver claramente cómo se empieza resolviendo problemas muy sencillos y se acaba con problemas muy complejos. En la traza se muestra el problema que se debe resolver en cada paso y la solución obtenida en cada paso.

4.3 Algoritmo de Boland, Charkhgard y Savelsbergh [2]

Este algoritmo destaca por ser uno de los primeros que busca las soluciones en el espacio de criterios, en lugar de buscar en el espacio de decisión. Como hemos explicado anteriormente, los algoritmos que buscan en el espacio de criterios encuentran las soluciones de una forma más sencilla y rápida, por lo que este algoritmo debería ser más eficiente que el anterior. En el capítulo del experimento computacional podremos comprobar si esto se cumple. Lo que se puede ver con la traza es que el tamaño del problema varía dentro de un margen, en lugar de crecer en cada paso se mantiene constante.

En este algoritmo trabajaremos con problemas que minimizan, pero en los demás algoritmos se maximizará.

El algoritmo tiene una lista de objetos que representan diferentes zonas del espacio de criterios que aún no han sido explorados, esta lista se representa con P. Los objetos que se incluyen en esta lista cumplen este formato, $O(\tilde{Y}, y, y^L)$, donde $\tilde{Y}$ representa el límite superior del subespacio, y es el punto guía de búsqueda, e y^L es el límite inferior (para más detalles consultar el artículo [2]). Cada vez que exploramos un objeto de esta lista existe la posibilidad de que se incluyan nuevos subespacios en la lista, pero el crecimiento de esta lista es controlado, ya que solo se añaden cero, uno o dos nuevos objetos.

Al comienzo del algoritmo se inicializa la lista de objetos, P, con el objeto $O(\emptyset, null, null)$, y la lista de soluciones, L_E, está vacía. En cada iteración se extrae un elemento de la lista P, y el algoritmo termina cuando esta lista está vacía, pues significa que no queda ningún subespacio por explorar.

Cada vez que se extrae un objeto se busca una solución no dominada en ese subespacio, la forma de construir el modelo que se debe resolver para el objeto $O(\tilde{Y}, y, y^L)$ es:

$$(P): \ min \ \sum_{j=i}^{p} z_j^a$$

$$s.t. \ x \in X$$

$$z_j^a = z_j(x) \qquad \forall j \in \{1, ..., p\}$$

$$z_j^a \leq (y_j^i - \varepsilon - M_j)\, b_{ij} + M_j \qquad \forall i \in \{i, ..., l\}, \ \forall j \in \theta(y^i)$$

$$\sum_{j \in \theta(y^i)} b_{ij} = 1 \qquad \forall\, i \in \{i, ..., l\}$$

$$z_j^a \leq (y_j - \varepsilon - M_j)\, b'_j + M_j \qquad \forall\, j \in \theta(y)$$

$$\sum_{j \in \theta(y)} b'_j = 1$$

$$z_j^a \in Q \qquad \forall\, j \in \{1, ..., p\}$$

$$b_{ij} \in \{0, 1\} \qquad \forall\, i \in \{i, ..., l\},\ \forall\, j \in \theta(y^i)$$

$$b'_j \in \{0, 1\} \qquad \forall\, j \in \theta(y)$$

Vamos a explicar las partes que componen este algoritmo, y traducir alguno de los elementos más ofuscados:

- Función objetivo: suma de tantas variables z_j^a como funciones objetivo tenga el problema multiobjetivo. Estas variables no serán enteras aunque estemos en el caso entero, van a ser racionales para permitir coeficientes decimales en las funciones objetivo.

- Región factible: siempre se van a añadir las restricciones originales del MOILP.

- Restricciones de igualdad: las nuevas variables se van a igualar con las funciones objetivo correspondientes.

- Primeras restricciones de tipo menor igual: aparece una nueva variable en cada una de ellas, b_{ij}, y se usa $\theta(y^i)$. Donde $\theta(y^i) = \{j \in \{0, ..., p\} : y_j^i > -\infty\}$, y y_j^i son los puntos que forman el conjunto $\tilde{Y}$.

- Segundas restricciones de tipo menor igual: también aparece una nueva variable en cada restricción, b'_j. Se define $\theta(y)$ como $\theta(y) = \{j \in \{0, ..., p\} : y_j > -\infty\}$

- Sumatorios: van a garantizar que las nuevas soluciones sean no dominadas.

Una vez se obtiene una solución resolviendo el problema que construimos con la explicación anterior, se sigue el siguiente algoritmo:

Algorithm 1: A new algorithm for computing the nondominated frontier of a MOIP.

Initialize the list of obtained efficient solutions L_E to be empty
Initialize the priority queue P with $(\emptyset, null, z^B)$
while P *is not empty* **do**
  Pop an element off P and denote it by $(\tilde{\mathcal{Y}}, y, y^L)$
  $x^n \leftarrow Find - NDP - Obj(\tilde{\mathcal{Y}}, y)$
  if $x^n \neq null$ **then**
    Add x^n to L_E
    $\hat{y}^L \leftarrow (\max(\min(y_1, z_1(x^n)), y_1^L), \ldots, \max(\min(y_p, z_p(x^n)), y_p^L)))$
    if $y \neq null$ **then**
      for $j = 1$ *to* p **do**
        $y'_j \leftarrow -\infty$
        $y''_j \leftarrow -\infty$
        if $z_j(x^n) > y_j$ **and** $z_j(x^n) > \hat{y}_j^L$ **then**
          $y'_j \leftarrow z_j(x^n)$
        if $z_j(x^n) < y_j$ **and** $y_j > \hat{y}_j^L$ **then**
          $y''_j \leftarrow y_j$
      if $y' \neq (-\infty, \ldots, -\infty)$ **and** $y'' \neq (-\infty, \ldots, -\infty)$ **then**
        Add $(\tilde{\mathcal{Y}} \cup \{y', y''\}, null, \hat{y}^L)$ to the priority queue P
    for $j = 1$ *to* p **do**
      if $\hat{y}_j^L > y_j^L$ **then**
        $\hat{y}_j \leftarrow \hat{y}_j^L$
      else
        $\hat{y}_j \leftarrow -\infty$
    if $\hat{y} \neq (-\infty, \ldots, -\infty)$ **then**
      Add $(\tilde{\mathcal{Y}}, \hat{y}, y^L)$ to the priority queue P
return L_E

Figura 2: Algoritmo de Boland, Charkhgard y Savelsbergh[2]

A diferencia del algoritmo de Sylva y Crema, no en todas las iteraciones se va a obtener una solución eficiente, pero en problemas de gran tamaño no se van a tener que resolver modelos con un gran número de restricciones y variables. Un problema que vemos en este algoritmo, es que en el peor de los casos resolveremos

$2^{nsol+1} - 1$ modelos, siendo $nsol$ el número de soluciones no dominadas del problema. Al resolver tantos problemas es posible que en más de una ocasión obtengamos soluciones repetidas pero en cada rama del árbol binario de problemas generado no pueden repetirse. La profundidad de este árbol en el caso peor es $nsol + 1$. Además, puede ocurrir que muchos de los modelos resueltos sean no factibles.

Como hicimos con el algoritmo de Sylva y Crema vamos a mostrar algunos pasos de la traza del problema descrito en la introducción, para ver mejor cómo se construyen los modelos según los objetos que se estén explorando:

Iteración 0:

```
P = {( empty , null , null )}
pop() = ( empty , null , null )
Maximize
    - 1.0 za1 - 1.0 za2
Subject To
    1.0 x1 -2.0 x2 <= 0
    1.0 za1 1.0 x1 -2.0 x2 = 0
    1.0 za2 -1.0 x1 3.0 x2 = 0
    x1 -> {0, 2}
    x2 -> {0, 2}
    za1 -> {-∞, ∞}
    za2 -> {-∞, ∞}
SOLUTION: (-0.0, 2.0, 4.0, -6.0) - (4, -6)
P.add( empty , ( 4 -6 ) , null )
```

Iteración 1:

```
P = {( empty , ( 4 -6 ) , null )}
pop() = ( empty , ( 4 -6 ) , null )
Maximize
    - 1.0 za1 - 1.0 za2
Subject To
    1.0 x1 -2.0 x2 <= 0
    1.0 za1 1.0 x1 -2.0 x2 = 0
    1.0 za2 -1.0 x1 3.0 x2 = 0
    1.0 za1 1.0 b1 <= 4
    1.0 za2 9.0 b2 <= 2
    1.0 b1 1.0 b2 = 1
    x1 -> {0, 2}
```

```
    x2 -> {0, 2}
    za1 -> {-∞, ∞}
    za2 -> {-∞, ∞}
    b1 -> {0, 1}
    b2 -> {0, 1}
SOLUTION: (1.0, 2.0, 3.0, -5.0, 1.0, 0.0) - (3, -5)
P.add({( -∞ -5 )( 4 -∞ )}, null , ( 3 -6 ))
P.add(empty , ( 3 -6 ) , null)
```

Iteración 2:

```
P = {({( -∞-5 )( 4 -∞)}, null , ( 3 -6 )),
      (empty , ( 3 -6 ) , null)}
pop() = ({( -∞-5 )( 4 -∞)}, null , ( 3 -6 ))
Maximize
    - 1.0 za1 - 1.0 za2
Subject To
    1.0 x1 -2.0 x2 <= 0
    1.0 za1 1.0 x1 -2.0 x2 = 0
    1.0 za2 -1.0 x1 3.0 x2 = 0
    1.0 za2 8.0 b12 <= 2
    1.0 b12 = 1
    1.0 za1 1.0 b21 <= 4
    1.0 b21 = 1
    x1 -> {0, 2}
    x2 -> {0, 2}
    za1 -> {-∞, ∞}
    za2 -> {-∞, ∞}
    b12 -> {0, 1}
    b21 -> {0, 1}
SOLUTION: No sol.
```

En el apéndice dos [A2] se puede ver la traza de este algoritmo con un problema de mayor tamaño. En ese problema se podrá ver cómo se repiten en más de una ocasión las soluciones.

4.4 Lokman y Köksalan[3]

El artículo de Lokman y Köksalan presenta dos algoritmos que pueden parecer similares, pero son completamente distintos. El primero de ellos es una pequeña mejora del algoritmo de Sylva y Crema, en el que se consigue añadir menos restricciones y variables a los modelos que construyen. El otro se caracteriza por resolver varios problemas muy sencillo en cada iteración, en lugar de resolver problemas muy complejos.

Algoritmo 1

Como ya hemos dicho, la idea en la que se basa este algoritmo es la misma idea que usa el algoritmo de Sylva y Crema. Son prácticamente iguales, solo cambia la forma en la que se construye la función objetivo del problema mono-objetivo, y que en cada paso son menos las variables y restricciones nuevas.

También encuentra una nueva solución no dominada en cada paso, y el algoritmo para cuando se llega a un problema no factible. La forma de construir los modelos es la siguiente:

$$
\begin{aligned}
(P):\ & max\, z_p(x) + \varepsilon \sum_{j \neq p} z_j(x) \\
& s.t.\ x \in X \\
& z_j(x) \geq (z_j^l + 1)\, y_{tj} - M(1 - y_{lj}) \quad \forall\, j \neq p,\ \forall\, l \\
& \sum_{j \neq p} y_{lj} = 1 \qquad \forall\, l \\
& y_{lj} \in \{0, 1\} \quad \forall\, l \quad j \neq m
\end{aligned}
$$

Como hemos hecho hasta ahora, vamos a explicar las distintas partes de este modelo:

- Función objetivo: se suma la última función objetivo con el resto de funciones multiplicadas por un peso $\varepsilon > 0$ lo suficientemente pequeño. En este caso no tenemos un vector de pesos, sino una variable de peso igual para todas las funciones.

- Región factible: en todas las iteraciones aparecerán las restricciones del problema multiobjetivo.

- Restricciones mayor igual: por cada función objetivo, excluyendo la última, se añade una restricción en la que se hace uso de una nueva variable, y_{tj}. Los valores que se usan son las soluciones obtenidas en los pasos anteriores. $-M$ es un vector de cotas inferiores de las funciones objetivo del MOILP.

- Restricción sumatorio: garantiza que al menos una de las nuevas restricciones se active para obtener una nueva solución no dominada.

Las soluciones del problema se obtendrán en orden decreciente según el valor objetivo de la última función, pues es la que tiene más peso en la nueva función objetivo de los modelos que vamos a resolver, y al estar maximizando buscará primero los valores más altos.

Este algoritmo tiene los mismos puntos fuertes y débiles que el algoritmo de Sylva y Crema. Comparándolo con el algoritmo de Sylva y Crema, puede que nos parezca que la mejora es insignificante, pero en problemas con un gran número de soluciones eficientes ahorramos mucho tiempo al disminuir considerablemente el número de restricciones y variables.

A continuación podemos ver parte de la traza del problema que hemos usado en los demás apartados:

Peso:

$\varepsilon = 0.0001$

Iteración 0:

```
Maximize
    - 0.9999 x1 + 2.9998 x2
Subject To
    1.0 x1 -2.0 x2 <= 0
    x1 -> {0, 2}
    x2 -> {0, 2}
P0 SOLUTION: (0.0, 2.0), (-4, 6)
```

Iteración 1:

```
Maximize
    - 0.9999 x1 + 2.9998 x2
Subject To
    1.0 x1 -2.0 x2 <= 0
    1.0 x1 -2.0 x2 -1.0 y1-1 >= -4
    1.0 y1-1 >= 1
    x1 -> {0, 2}
    x2 -> {0, 2}
    y1-1 -> {0, 1}
P1 SOLUTION: (1.0, 2.0, 1.0), (-3, 5)
```

Iteración 2:

```
Maximize
    - 0.9999 x1 + 2.9998 x2
Subject To
    1.0 x1 -2.0 x2 <= 0
    1.0 x1 -2.0 x2 -1.0 y1-1 >= -4
    1.0 y1-1 >= 1
    1.0 x1 -2.0 x2 -2.0 y2-1 >= -4
    1.0 y2-1 >= 1
    x1 -> {0, 2}
    x2 -> {0, 2}
    y1-1 -> {0, 1}
    y2-1 -> {0, 1}
P2 SOLUTION: (2.0, 2.0, 1.0, 1.0), (-2, 4)
```

En el apéndice tres [A3] se puede ver la traza completa de este algoritmo con un problema más complejo.

Algoritmo 2

Para intentar solucionar el problema que surge al tener que resolver modelos muy complejos cuando el conjunto de soluciones es grande, los autores presentan un segundo algoritmo en el que siempre se resuelven problemas muy sencillos. Para poder tener problemas sencillos, lo que se debe hacer es resolver varios en cada iteración.

Los problemas que se tienen que resolver siguen la siguiente estructura:

$$(P^b): \ max\, z_p(x) + \varepsilon \sum_{j \neq p} z_j(x)$$
$$s.t. \ x \in X$$
$$z_j(x) \geq b_j \quad j = 1, ..., p-1$$

Vamos a explicar cada una de las partes que constituirán nuestros modelos y explicar cómo se resuelven varios en cada iteración:

- Función objetivo: se construye de la misma forma que la función objetivo del primer algoritmo que presentan.

- Región factible: en todos los modelos que se tengan que resolver, se mantendrán las restricciones del MOILP.

- Nuevas restricciones: se obliga que las funciones objetivo del problema multiobjetivo sean mayores o iguales que unas cotas que se calculan en cada iteración.

En cada iteración hay que generar un conjunto de índices, K, para establecer las cotas de cada modelo a resolver. K se define como:

$$K = \{k \in Z^{p-2} \ / \ k_i \in \{0, ..., l\} \quad \forall\, i \in \{1, ..., p-2\}\}$$

Una vez tenemos K, pasamos a calcular todos los vectores de cotas que tendrá esa iteración, que serán $(l+1)^{p-2}$. Con cada uno de esos vectores habrá que resolver un problema. La forma de construir las cotas es la siguiente:

$$b_j^{k,l} = \begin{cases} -M & si\ k_j = 0 \\ z_{k_j j} + 1 & en\ otro\ caso \end{cases}$$

$$b_{p-1}^{k,l} = \begin{cases} -M & si\ S_n^k = \emptyset \\ max_{z^t \in S_l^k} \{z_{t(p-1)}\} + 1 & en\ otro\ caso \end{cases}$$

El conjunto S_l^k lo definimos como:

$$S_l^k = \{z^t : z_{tj} \geq b_j^{k,l} \quad j = 1, ..., p-2, z^t \in S_l\}$$

Donde S_l son las l primeras soluciones no dominadas que se han obtenido.

Como resolvemos más de un problema en cada iteración, vamos a obtener más de una solución en cada paso. Pero al final solo se añadirá una de ellas, la que tenga mayor valor en la última función objetivo.

En este algoritmo es muy útil resolver un ejemplo como se ha hecho con todos los demás, pues el tema de las cotas es difícil de entender si no se ve exactamente cómo se está haciendo. Sin embargo, en esta ocasión usaremos otro problema de mayor dimensión para realizar la traza al objeto de ilustrar cómo se construye k. El problema con el que se va a trabajar es el siguiente:

$$Maximize\ 77x_1 + 26x_2 + 57x_3 + 22x_4 + 10x_5 + 32x_6 + 98x_7 + 77x_8 + 43x_9 + 19x_{10}$$
$$Maximize\ 56x_1 + 22x_2 + 28x_3 + 37x_4 + 89x_5 + 17x_6 + 65x_7 + 85x_8 + 10x_9 + 41x_{10}$$
$$Maximize\ 89x_1 + 33x_2 + 47x_3 + 63x_4 + 31x_5 + 39x_6 + 27x_7 + 66x_8 + 49x_9 + 13x_{10}$$

Subject To

$$91x_1 + 95x_2 + 87x_3 + 42x_4 + 12x_5 + 55x_6 + 24x_7 + 17x_8 + 96x_9 + 77x_{10} \leq 298$$
$$x_i \in \{0, 1\} \qquad i = 1, ..., 10$$

La traza correspondiente a las tres primeras iteraciones es:

Peso:

$\varepsilon = 0.001$

Problema a resolver en cada paso:

```
Maximize
    + 25.228 x8 + 89.102 x9 + 59.073 x10 + 71.222 x1 +
41.081x2
     + 23.132 x3 + 44.122 x4 + 61.130 x5 + 97.088 x6 +
27.190x7
Subject To
    91.0 x1 + 95.0 x2 + 87.0 x3 + 42.0 x4 + 12.0 x5 + 55.0 x6
       + 24.0 x7 + 17.0 x8 + 96.0 x9 + 77.0 x10 <= 298
    77.0 x1 + 26.0 x2 + 57.0 x3 + 22.0 x4 + 10.0 x5 + 32.0 x6
       + 98.0 x7 + 77.0 x8 + 43.0 x9 + 19.0 x10 >= b1
    56.0 x1 + 22.0 x2 + 28.0 x3 + 37.0 x4 + 89.0 x5 + 17.0 x6
       + 65.0 x7 + 85.0 x8 + 10.0 x9 + 41.0 x10 >= b2
    89.0 x1 + 33.0 x2 + 47.0 x3 + 63.0 x4 + 31.0 x5 + 39.0 x6
       + 27.0 x7 + 66.0 x8 + 49.0 x9 + 13.0 x10 >= b3
```

Iteración 0 (l = 0):

k = [0, 0] COTAS: [0, 0, 0] **SOL: (337, 322, 301, 370)**

Iteración 1 (l = 1):

k = [0, 0] COTAS: [0, 0, 302] SOL: (316, 349, 315, 325)
k = [0, 1] COTAS: [0, 323, 0] **SOL: (237, 325, 301, 357)**
k = [1, 0] COTAS: [338, 0, 0] SOL: (351, 340, 299, 304)
k = [1, 1] COTAS: [338, 323, 0] SOL: (351, 340, 299, 304)

Iteración 2 (l = 2):

k = [0, 0] COTAS: [0, 0, 302] SOL: (316, 349, 315, 325)
k = [0, 1] COTAS: [0, 323, 302] SOL: (316, 349, 315, 325)
k = [0, 2] COTAS: [0, 326, 0] SOL: (313, 353, 265, 340)
k = [1, 0] COTAS: [338, 0, 0] SOL: (351, 340, 299, 304)
k = [1, 1] COTAS: [338, 323, 0] SOL: (351, 340, 299, 304)
k = [1, 2] COTAS: [338, 326, 0] SOL: (351, 340, 299, 304)
k = [2, 0] COTAS: [238, 0, 302] SOL: (316, 349, 315, 325)
k = [2, 1] COTAS: [238, 323, 0] **SOL: (313, 353, 265, 340)**
k = [2, 2] COTAS: [238, 326, 0] SOL: (313, 353, 265, 340)

En esta traza vemos como el número de modelos que hay que resolver en cada paso aumenta muy rápidamente. La traza completa de este problema se puede ver en

el apéndice cuatro [A4]. En esta traza la última iteración es la 10 y el vector de cotas contiene 121 combinaciones.

Una forma de reducir el número de problemas que se tienen que resolver, es guardar en un HashMap las cotas que ya se han resuelto con sus soluciones. Esto es de utilidad porque más de una vez se tendrán las mismas cotas, y podemos aprovechar que ya lo hemos resuelto.

Capítulo 5 Implementación

5.1 Clases Relativas a problemas MOILP

En el capítulo de los solvers, se comentó que necesitaremos una estructura común que luego traduzcamos en las estructuras que usa cada solver. Siguiendo esto en este apartado comentaremos cuáles son esas clases y por qué decidimos crearlas.

La primera clase es *MOILP*, está no será usada por el solver, pero nos servirá para guardar todo lo relacionado con el MOILP. Tiene un conjunto de funciones objetivo, una lista de restricciones para representar la región factible, y un conjunto de variables. También tiene algunos métodos y elementos que siempre encontramos en los problemas multiobjetivo, como puede ser el cálculo del vector M, que siempre se utiliza en los algoritmos que hemos estudiado para construir los problemas. Además esta clase hace uso de otra que hemos creado, la clase *Reader* que explicaremos más adelante.

Una clase muy similar a esta es la clase *LP*, que tiene los elementos propios de un problema mono-objetivo lineal. Como hemos estado vienda hasta ahora, lo que vamos a tener que resolver son problemas mono-objetivo, por lo que esta clase será la encargada de comunicarse con el solver. Tiene un método resolver que llama a *solve()* de la API *MySolver*, y en este método inyecta los datos al solver para que este tenga el modelo almacenado.

Otra clase que el solver no utilizará, pero nos ayuda a la hora de almacenar los resultados, es la clase *Solution*. En esta clase se almacenan las soluciones, y los valores que toman las variables en cada uno de ellos. Con la API *MySolver*, podemos obtener ambos datos por separado, y uniéndolos trabajamos de una forma más

sencilla. Además, tenemos la función *toString()*, que no es de gran utilidad cuando queremos hacer una traza del algoritmo.

Las tres clases que vamos a comentar son las que guardan los distintos datos de los problemas. Todas ellas tienen el método *toString()* para poder mostrar la traza de cada algoritmo de una forma rápida, sencilla y limpia. El solver, en sus métodos, especifica que estas clases que vamos a explicar son los tipos de datos con los que va a trabajar para construir su modelo. Estas clases son:

- *ObjFunction*: guarda un conjunto de nombres de variables y coeficientes. Otro atributo que vemos es *max*, un booleano que usamos para saber si la función maximiza o no. Esto nos hace falta porque en ocasiones tendremos que resolver problemas con funciones que maximicen y otras que minimicen. Por esto también tenemos el método *changeSymbol()*, que multiplicará toda la función por menos uno para pasar de maximizar a minimizar o viceversa.

- *Constraint*: guarda un conjunto de nombre de variables, coeficientes, el tipo de desigualdad del que se trata ($\leq \ = \ \geq$) y el valor al que se iguala esa restricción.

- *Variable*: cada variable tiene un nombre, una cota inferior, una cota superior y una variable booleana que indica si es entera o no.

Como usaremos todos estos tipos de datos en las demás clases, hemos incluido los getters y setters para los distintos atributos de ellas.

La última clase común a todos los algoritmos es la clase *Reader*, que ya hemos nombrado con anterioridad. Esta clase será la encargada de leer los ficheros que contienen los problemas multiobjetivo, y almacenarlos en las clases que hemos implementado para representar ese mismo problema con nuestra estructura. La clase *MOILP* llama a las funciones del *Reader* que devuelven los distintos tipos de datos y los almacena en sus atributos.

En estas clases se implementan todas las características comunes de los problemas multiobjetivo, y los elementos necesarios para resolver los problemas. El agrupar todas las características comunes de estos problemas nos permite evitar tener código repetido en las clases de cada algoritmo. También hace que las clases sean más sencillas al descomponerse los elementos en otros más sencillos con sus métodos específicos.

5.2 Diagrama UML

En el siguiente diagrama UML se muestran las clases que se han explicado en el apartado anterior:

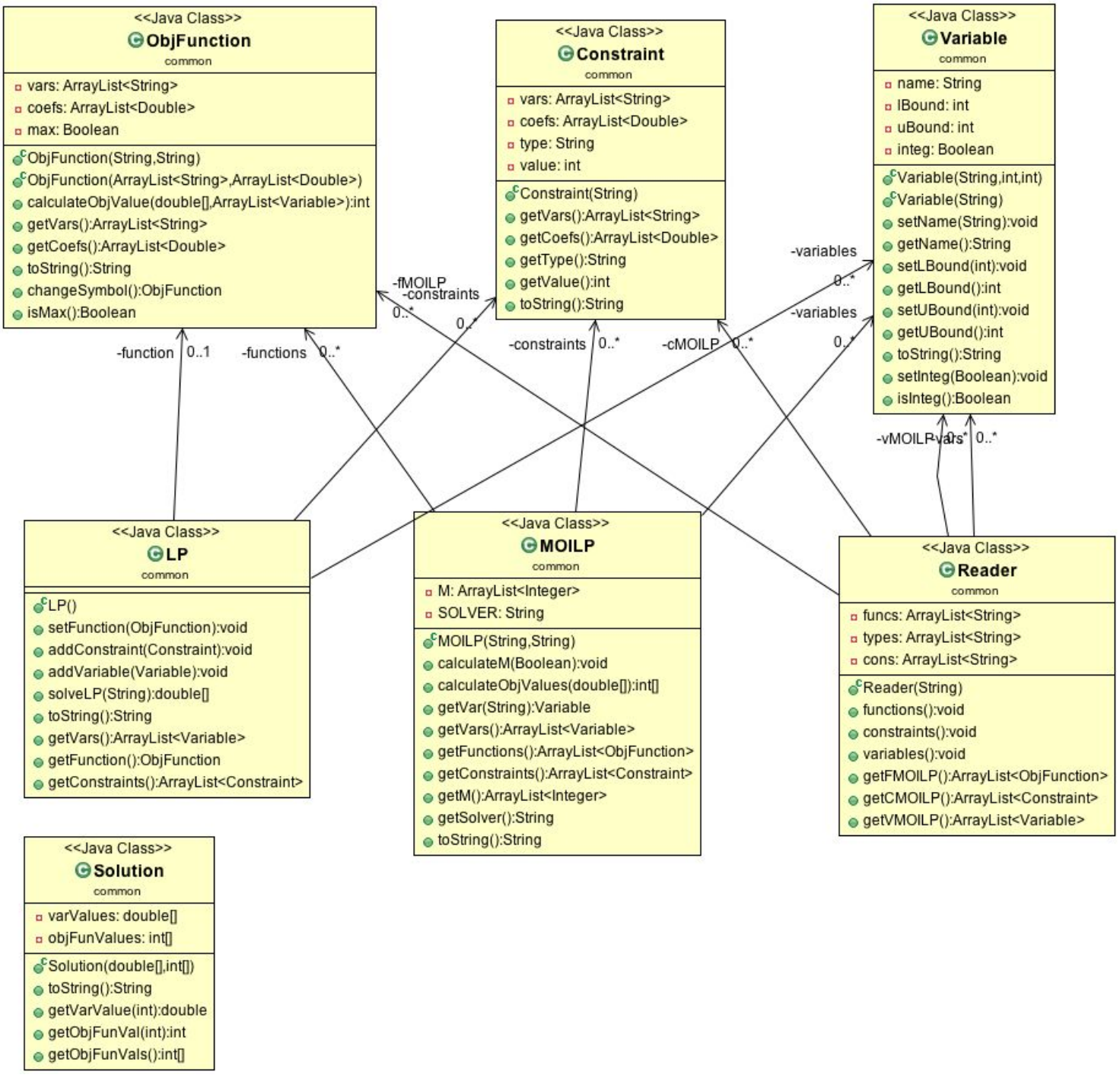

Figura 3: Clases comunes

La distribución de los ficheros y los paquetes es la siguiente:

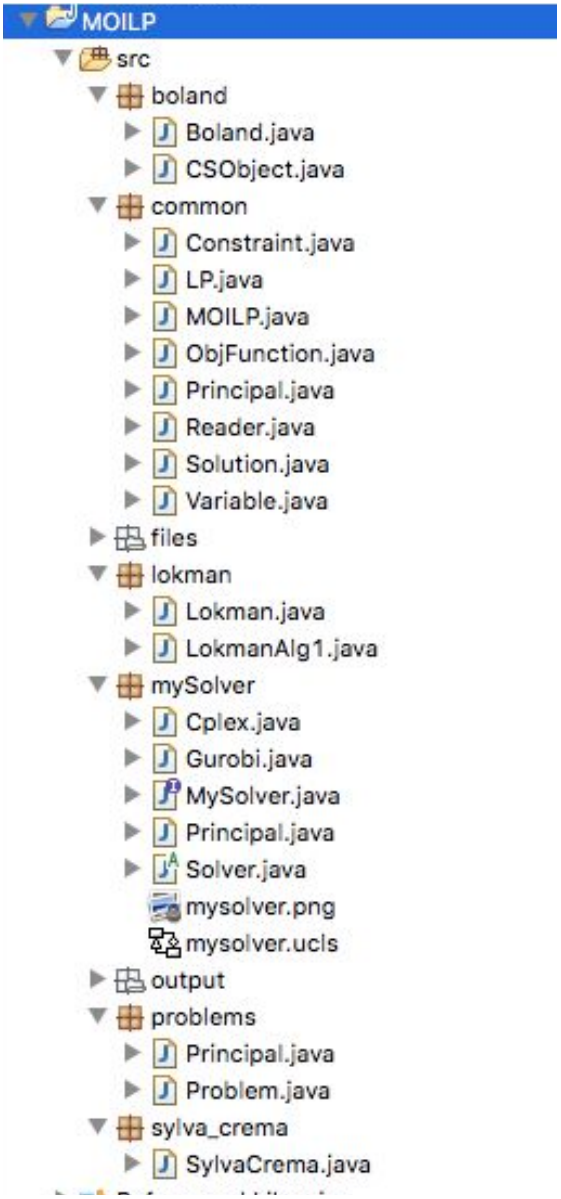

Figura 4: Paquetes

Hay tres paquetes que no hemos nombrado aún:

- files: en esta carpeta guardamos los diagramas de clases UML.

- problems: en este paquete está el generador de problemas aleatorio que hemos implementado. Es muy útil para el experimento computacional, en el cual hay que generar un gran número de problemas multiobjetivo aleatorios.

- output: carpeta a la que se dirigen todos los problemas que hemos generado.

5.3 Algoritmo de Sylva y Crema

El algoritmo de Sylva y Crema está implementado en una clase aparte, que se resume en la siguiente imagen:

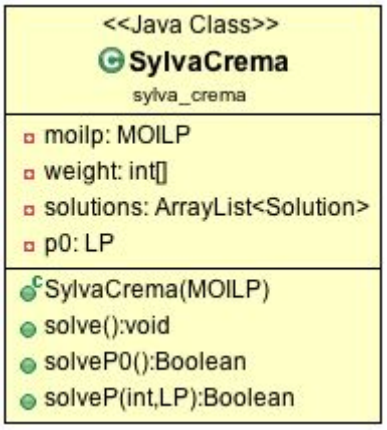

Figura 5: Clase *SylvaCrema*

Empezaremos comentando los atributos que forman la clase:

- `MOILP`: como es obvio almacenamos en la clase el problema multiobjetivo que queremos resolver.

- `weight`: vector de tipo entero que guarda los pesos por los que se multiplicarán cada una de las funciones objetivo.

- `solutions`: ArrayList en el que se van añadiendo las nuevas soluciones que se obtienen.

- `p0`: en la explicación del algoritmo vemos que en cada iteración se añaden nuevas restricciones y variables, es decir, podemos reutilizar el modelo que se resolvió en la iteración anterior e incluir las nuevas variables y restricciones. Este problema se construye por primera vez en el método `solveP0()`, y se usará en todas las iteraciones del algoritmos, para no tener que volver a generar modelos que ya tenemos.

Después de comentar los elementos de la clase, vamos a hablar de las funciones que utiliza el algoritmo para resolver el problema. Son muy pocos métodos que

implementan de una forma clara y sencilla el algoritmo. A continuación, indicaremos la función de cada uno de ellos:

- Constructor: se comprueba que todas las funciones objetivo son de maximizar, porque el algoritmo de Sylva y Crema solo trabaja con problemas que maximizan. Si hay alguna función es de minimizar, se cambia el signo con el método propio de la clase *objFunction*.

- `solve()`: va a garantizar que una vez se llegue a un problema no factible, se pare la ejecución del algoritmo.

- `solveP0()`: construye el primer modelo que hay que resolver, sumando las funciones objetivo multiplicadas por los pesos. Una construido este primer modelo, se resuelve.

- `solveP(int iter, LP lp)`: para el resto de iteraciones se construyen los modelos a partir de los que se han resuelto en el paso anterior, añadiendo las variables y restricciones correspondientes. Cuando tenemos el modelo lo resolvemos y devolvemos la solución.

Lo más destacable de esta clase es la reutilización de modelos anteriores, para no tener que construir todas las restricciones de nuevo en cada paso.

5.4 Boland, Charkhgard y Savelsbergh

Para implementar este algoritmo necesitamos dos clases, una de ellas la que incluye los métodos de este algoritmo, y la otra es para crear los objetos que representan los subespacios del espacio de criterios. Esta segunda clase facilitará mucho tener la lista de subespacios en el algoritmo.

Simplificaremos los nombres de las clases llamando *Boland* a la clase que implementa el algoritmo (nombre del primer autor que aparece en el artículo), y *CSObject* a la clase para representar los subespacios del espacio de criterios (Criteria Space Object).

El siguiente diagrama UML muestra la relación de estas dos clases de una forma clara y sencilla:

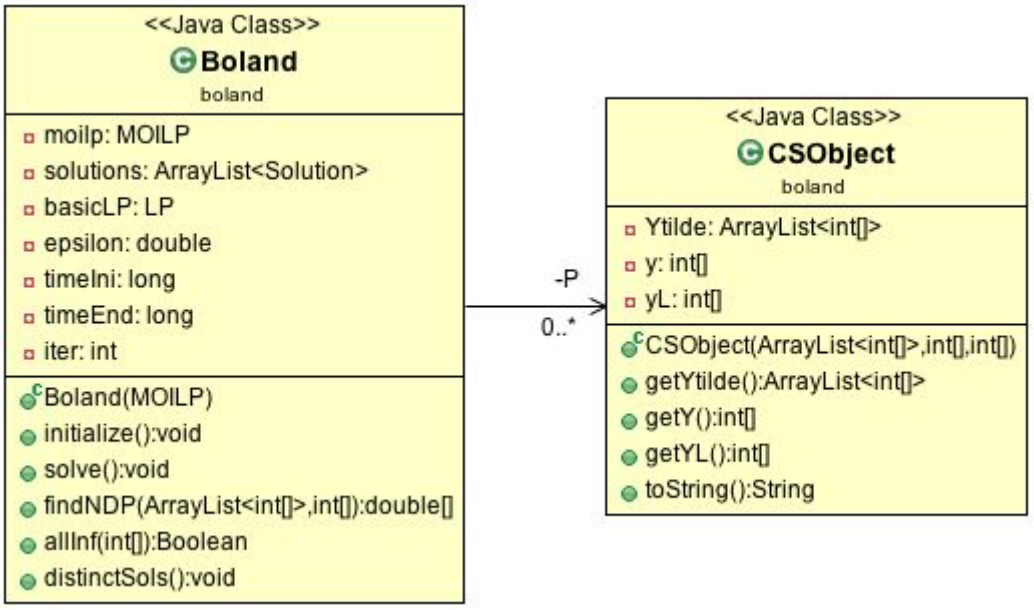

Figura 6: Clase *Boland* y clase *CSObject*

Comentaremos brevemente ambas clases, hablando tanto de sus atributos, como de las funciones que utiliza. Empezaremos con la clase *CSObject*:

- Atributos

    - `YTilde`: en el algoritmo es un conjunto de puntos, por lo que aquí lo representamos con una lista de vectores.

    - `y` e `yL`: ambos atributos son puntos que se guardan como vectores de tipo entero.

- Métodos

    - Constructor: recibe los distintos elementos del objeto y los agrupa.

    - `toString()`: este método, aunque sencillo, es muy importante al facilitar la traza del algoritmo.

Esta clase, al igual que la que representa el algoritmo de Sylva y Crema, no tiene un gran número de elementos, pero separa los pasos lo suficiente como para facilitar la comprensión de la ejecución del algoritmo. Los atributos y métodos que forman la clase *Boland* son:

- Atributos

    - `MOILP`: problema multiobjetivo que se quiere resolver.

    - `solutions`: lista que almacena las soluciones que se obtienen a lo largo de la ejecución del algoritmo. Al poder obtener soluciones repetidas se debe controlar si una solución ya está contenida o no en la lista antes de añadirla.

    - `basicLP`: en este algoritmo no se va aumentando el tamaño del problema en cada paso, sino que todos los modelos tienen una parte común. Esta parte se almacena en este atributo para agilizar la construcción de los modelos.

    - `epsilon`: variable que guarda el peso que se va a utilizar en el algoritmo.

    - `P`: lista de subespacio que aún no han sido explorados en busca de una solución no dominada.

- Métodos

    - Constructor: hace lo mismo que el constructor de la clase *SylvaCrema*, pero en lugar de hacer que el problema de maximizar, hace que sea de minimizar. En la traza de este algoritmo que se mostró en el capítulo anterior, vemos que se maximiza en lugar de minimizar, simplemente lo que se ha hecho es cambiar el signo de la función objetivo, pues nuestra API solo resuelve problemas que maximicen.

    - `initialize()`: crea el problema base que se usará en todos los pasos.

    - `solve()`: se sigue ejecutando el algoritmo hasta que no queden subespacios que explorar. En cada paso busca una solución no dominada y aplica la lógica del algoritmo para añadir los nuevos subespacios a la lista.

    - `findNDP()`: busca una solución no dominada en el subespacio que recibe como argumento.

- `allInf()`: comprueba si todos los elementos de un vector son -∞. Esto se usa a la hora de añadir nuevos subespacios a la lista.

La parte más importante de la implementación de este algoritmo, es la clase *CSObject* que facilita la representación de los objetos.

5.5 Lokman y Köksalan

Algoritmo 1

La clase que contiene la implementación de este algoritmo se llama *LokmanAlg1*, y es la siguiente:

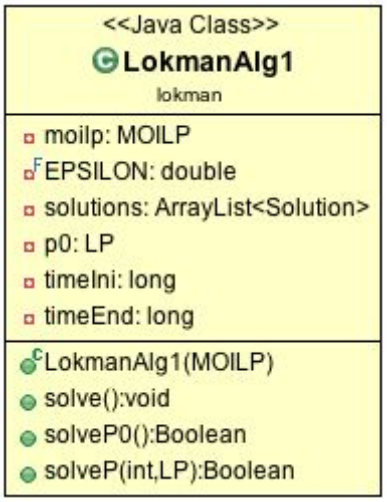

Figura 7: Clase *LokmanAlg1*

Este algoritmo es tan parecido al de Sylva y Crema que no necesita que se explique. Tiene los mismos métodos, sólo que en cada paso crece un poco menos el problema que se tiene que resolver. La única diferencia en los atributos es que el peso es un sólo número, no es un vector.

Algoritmo 2

La clase *Lokman*, que contiene el segundo algoritmo es la que más difiere del resto. Teniendo métodos que no se han visto en ninguna otra clase y necesitando atributos especiales.

El diagrama UML muestra un resumen de ella:

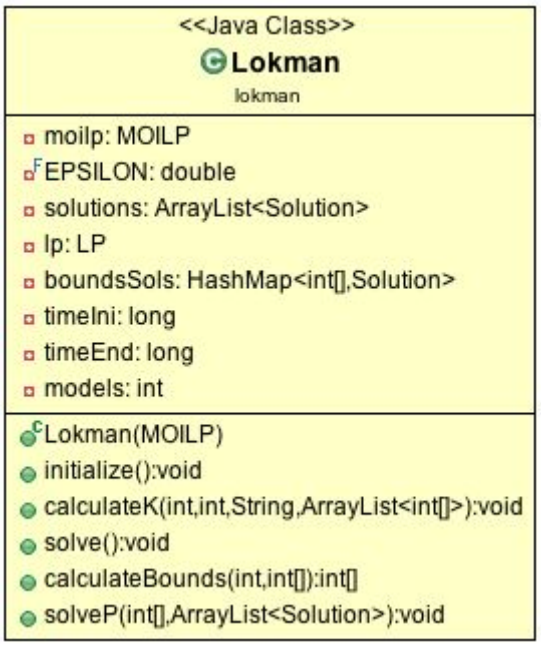

Figura 8: Clase *Lokman*

Vamos a explicar los atributos y métodos de esta clase para entender qué es lo que la diferencia tanto del resto de clases. Como siempre empezaremos hablando de los atributos:

- `MOILP, EPSILON, solutions`: estos tres atributos son iguales que los atributos de la clase del primer algoritmo. No necesitamos volver a explicarlos.

- `lp`: problema que se va a tener que resolver con cada vector de cotas. Siempre será el mismo problema, solo cambiando el valor de la desigualdad, por lo que lo almacenamos y así solo lo construimos una vez.

- `boundsSols`: HashMap que relaciona las cotas con una solución, de esta forma no tendremos que volver a resolver un modelo cuando se repite un vector de cotas.

Las funciones que incluye esta clase son:

- Constructor e `initialize()`: funcionan de la misma forma que en el primer algoritmo de estos autores.

- `calculateK()`: K es el conjunto de todas las combinaciones posibles de k. Para obtener este conjunto necesitamos una función que haga uso de la

recursividad. La función es la siguiente:

```
public void calculateK(int n, int size, String string,
ArrayList<int[]> K) {
          String[] k = new String[n + 1];
          for (int i = 0; i <= n; i++) {
               k[i] = string + i + " ";
               if (size > 1) {
                    calculateK(n, size - 1, k[i], K);
               } else {
                    String[] tokens = k[i].split(" ");
                    int[] aux = new int[k[i].length() / 2];
                    for (int j = 0; j < tokens.length; j++) {
                         aux[j] = new Integer (tokens[j]);
                    }
                    K.add(aux);
               }
          }
     }
```

- `solve()`: en cada iteración se resuelve un modelo por cada elemento de K, hasta que no se obtiene ni una sola solución en un paso.

- `calculateBounds()`: construye los vectores de cotas siguiendo la lógica que se presentó en el capítulo anterior.

- `solveP()`: resuelve un problema según las cotas que recibe y lo guarda en una lista de soluciones. Esta es una lista de soluciones temporales que sirve para elegir la que tenga mayor valor en la última función objetivo. La solución que se elige de esta lista temporal es la que se añade en la lista de soluciones del problema.

Lo más importante de la implementación de este algoritmo es la forma en la que se calcula las cotas, y el HashMap que permite ahorrar tiempo al reducir el número de problemas que tenemos que resolver.

Capítulo 6 Experimento Computacional

En este capítulo resolveremos una batería de problemas de prueba con todos los algoritmos que hemos implementado, mostrando los resultados en una serie de tablas. Con esto podremos comparar el rendimiento de cada uno de ellos. Por cada tipo de problema se han generado 5 ejemplos, y lo que se muestran son las medias.

El nombre de los problemas indica el tamaño del problema, el primer número es la cantidad de funciones objetivo, las variables del problema están en segundo lugar, y por último está el número de restricciones.

En el quinto apéndice [A5] se puede ver con detalle de donde se han obtenido estos números, se trata de una serie de hojas de cálculo que muestran los tiempos y modelos resueltos para cada tipo problema.

Para simplificar y respetar el límite de páginas de la memoria, el experimento solo resolveremos problemas con tres funciones objetivo.

	Sylva y Crema				
	Soluciones (Nsol)	**Modelos resueltos (MR)**	**Tiempo resolución (T) ms**	**T / Nsol**	**MR / Nsol**
3x4x1	6.2	7.2	58	7.96	1.32
3x4x3	7	8	105	12.1	1.18
3x4x5	4	5	29.8	7.44	1.31
3x5x1	20.2	21.2	826.8	21.34	1.11
3x5x3	7.2	8.2	135.2	18.02	1.2
3x5x5	2.6	3.6	26.4	7.2	1.53
3x6x1	53.2	54.2	28916.8	255.49	1.03
3x6x3	27	28	3034.8	57.94	1.08
3x6x5	10.6	11.6	204.2	19.89	1.12
	Boland, Charkhgard y Savelsbergh				
3x4x1	6.2	12.8	49.6	7.17	2.42
3x4x3	7	13.8	76.2	11.76	2.04
3x4x5	4	8.4	43.4	10.71	2.2
3x5x1	20.2	37.8	213.6	9.08	2.082
3x5x3	7.2	14.2	99.6	12.54	2.05
3x5x5	2.6	5.8	29.4	8.63	2.47
3x6x1	53.2	108.4	836.6	16.82	2.05

3x6x3	27	61	480	16.29	2.16
3x6x5	10.6	22.2	179	17.95	2.11
	Lokman y Köksalan 1				
3x4x1	6.2	7.2	62.6	8.55	1.32
3x4x3	7	8	60.6	7.61	1.18
3x4x5	4	5	28.2	8.54	1.31
3x5x1	20.2	21.2	212	6.76	1.11
3x5x3	7.2	8.2	74.2	9.85	1.2
3x5x5	2.6	3.6	17.2	5.93	1.53
3x6x1	53.2	54.2	5454.2	63.98	1.03
3x6x3	27	28	1719.2	33.55	1.08
3x6x5	10.6	11.6	152.6	14.72	1.12
	Lokman y Köksalan 2				
3x4x1	6.2	33.8	127.4	17.66	4.82
3x4x3	7	38.2	105.8	15.52	4.7
3x4x5	4	14	43.6	11.3	3.4
3x5x1	20.2	167.4	522	18.99	6.54
3x5x3	7.2	37.4	131.4	18.84	4.74
3x5x5	2.6	9.2	35.2	11.5	3.23
3x6x1	53.2	857	3538.6	46.85	13.11
3x6x3	27	269.6	772.2	20.42	7.25
3x6x5	10.6	61.6	351.4	32.91	5.57

Tabla 1: Experimento computacional

Capítulo 7 Conclusiones y líneas futuras

Después de realizar el experimento computacional, podemos sacar unas conclusiones de los resultados:

- El algoritmo de Boland, Charkhgard y Savelsbergh, suele dar los mejores tiempos. Esto demuestra lo que dijimos al comienzo de la memoria, los algoritmos que buscan en el espacio de criterios son más eficientes.

- En los problemas con muy pocas soluciones se comportan mejor los algoritmos de Sylva y Crema, y el primero de Lokman y Köksalan. Esto es porque los modelos que se construyen no son muy complejos y se resuelven menos para obtener las soluciones.

- El segundo algoritmo de Lokman y Köksalan resuelve muchísimos más modelos que los demás algoritmos. Si se resuelven problemas con más de tres funciones objetivo se ve que este número crece aún más, pero no hemos resuelto ese tipo de problemas debido al tiempo que consumen.

En este trabajo hemos sufrido los problemas que explicamos desde el comienzo de la memoria. La falta de homogeneidad en la notación y lo compleja que es ha hecho que necesitemos mucho más tiempo para entender estos algoritmos. Además, las ideas en las que se basan estos algoritmos suelen ser bastante complejas, y hace difícil entender cómo funciona el algoritmo.

Por la dificultad del tema este proyecto ha sido todo un reto, que nos ha enseñado un tema de gran importancia en la actualidad.

Tras la presentación de TFG, se pretende seguir trabajando en los siguientes puntos:

- Refinar la implementación de los algoritmos y la estructura de clases utilizada.

- Conseguir optimizar una función objetivo sobre la región eficiente de un MOILP, basándose en los algoritmos implementados en este proyecto, pero sin tener que generar todas las soluciones eficientes.

- Difundir los resultados en congresos y/o revistas nacionales e internacionales.

Capítulo 8 Summary and Conclusions

In this project we implemented four algorithms to generate the efficient set of a multiobjective integer linear program. We also made a computational experiment to compare these algorithms to see which one is better.

After the computational experiment we have concluded the following:

- Boland, Charkhgard and Savelsbergh algorithms on most occasions is the best one. With this we prove what we said before, that the algorithms that search in the criteria space are more efficient.

- The multiobjective integer linear programs with very few solutions are best when you use Sylva and Crema algorithm, because the models aren't complex at the beginning and we have less to solve.
- Comparing the second algorithm of Lokman and Köksalan with the other ones, this algorithm solves a lot of models. If we use problems with more than three objective functions, we can see the difference is bigger.

This project was challenging because all the problems described at the beginning of this document occurred. The challenge involved a big learning curve that was very stimulating and I found it very interesting to work on.

Capítulo 9 Presupuesto

En la siguiente tabla se muestra el presupuesto de este trabajo:

Ítem	Cantidad	valor unidad	total
Hora de programación	360	30€	10800
Licencia CPLEX	1	*	
Licencia Gurobi	1	14000$	14000$

* CPLEX no muestra una lista de precios, hay que ponerse en contacto para saber el precio de la licencia que se necesite.

Apéndices

[A1] Traza del algoritmo de Sylva y Crema:
https://docs.google.com/document/d/1FJyg8m426Rsu9zptmMm_3_AaimiYPmio150nP-3-Khk/edit?usp=sharing

[A2] Traza del algoritmo de Boland, Charkhgard y Savelsbergh:
https://docs.google.com/document/d/1_C5_nECxAhVIScsKmj8RdzM9zhZR1mO2046EpPDzehY/edit?usp=sharing

[A3] Traza del primer algoritmo de Lokman y Köksalan:
https://docs.google.com/document/d/15nd0gN1UGLV1zIOIsu1v2zedg6ZukE8F4IFvdQz_mA4/edit?usp=sharing

[A4] Traza del segundo algoritmo de Lokman y Köksalan:

https://docs.google.com/document/d/1EX4rO7IlanoPUJ7q-87ZiP7bR2aJwpIQueWUIYvc4eE/edit?usp=sharing

[A5] Experimento computacional: https://docs.google.com/spreadsheets/d/1NE8sAp03RcWurdB2mta4M603TNLYwnoFyY9RHsflJzk/edit?usp=sharing

Bibliografía

[1] Sylva, J., Crema, A., 2004. A method for finding the set of non-dominated vectors for multiple objective integer linear programs. European journal of operational research 158, 46-55.

[2] Boland, N., Charkhgard, H., Savelsbergh, M., 2017. A New Method for Optimizing a Linear Function over the Efficient Set of a Multiobjective Integer Program. European journal of operational research 260, 904-919.

[3] Lokman, B., Köksalan, M., 2013. Finding all nondominated points of multi-objective integer programs. J Glob Optim 57, 347-365.

[4] Gurobi: http://www.gurobi.com/downloads/download-center

[5] CPLEX: https://www.ibm.com/developerworks/community/blogs/jfp/entry/CPLEX_Is_Free_For_Students?lang=en

[6] XPRESS: http://www.fico.com/en/products/fico-xpress-optimization-suite#overview

[7] Coin-OR: https://projects.coin-or.org/Clp