Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Algoritmia básica

TP6: programación lineal

Autora:
Nerea Gallego Sánchez (801950)

10 de junio de 2022

Índice

1. Problema

El problema elegido, es el problema número 7 (problema de la mudanza) de la colección de problemas proporcionada. "*Los señores Pérez han comprado un piso nuevo y ha llegado el momento de hacer mudanza. Tienen que empaquetar n objetos de peso y volumen diferente y distribuirlos en un conjunto de cajas. Cada caja tiene una capacidad P_j (en peso), volumen V_j, y un coste c_j. Necesitan tu ayuda para encontrar la colocación óptima de los objetos en las cajas que minimice el coste total de cajas utilizadas.*" sacado del documento `Proyectos21-22` [1]

2. Análisis del problema

Parámetros del problema:

x_i : peso del objeto i
y_i : volumen del objeto i
c_j : coste de la caja j
P_j : peso máximo que admite la caja j
V_j : peso máximo que admite la caja j

Variables del problema:

w_{ij} : variable binaria con valor 1 si el objeto i se encuentra en la caja j. Tendrá el valor 0 en caso contrario.
z_j : variable binaria con valor 1 si la caja j contiene algún elemento. Tendrá el valor 0 en caso contrario.

Función objetivo:

$$\text{máx}\, c = -\sum_{j=1}^{k} c_j * z_j \tag{1}$$

Restricciones:

$$\sum_{i=1}^{n} x_i * w_{ij} \leq P_j * z_j \quad \forall j \tag{2}$$

El peso de los objetos que contiene la caja debe ser menor o igual que el peso máximo de la caja.

$$\sum_{i=1}^{n} y_i * w_{ij} \leq V_j * z_j \quad \forall j \tag{3}$$

El volumen de los objetos que contiene la caja debe ser menor o igual que el volumen máximo de la caja.

$$\sum_{j=1}^{n} w_{ij} = 1 \quad \forall i \tag{4}$$

Cada objeto solo puede estar en una caja.

$$\forall i \quad \forall j \quad w_{ij} \leq z_j \tag{5}$$

Si hay algún objeto en la caja j, z_j tiene valor 1, sino, tiene valor 0.

Se ha considerado que es un problema de programación lineal mixta, ya que los parámetros de peso y volumen, pueden tomar valores reales. De la misma manera, contiene las variables binarias de pertenencia o no a una caja y si la caja está vacía o no.

[1] Junio 2022 `https://moodle.unizar.es/add/pluginfile.php/6493131/mod_resource/content/1/Proyectos21-22.pdf`

3. Diseño e implementación

Para la implementación de la generación de instancias se ha elegido la herramienta de programación lineal para el lenguaje Python, PYTHON-MIP `https://python-mip.readthedocs.io/en/latest/install.html`. Además, se ha hecho uso de una clase python implementada por la empresa IBM destinada a la importación de datos. Esta clase, se ha obtenido de una herramienta desarrollada por IBM para la resolución de problemas de programación lineal [2] con distintas APIs. En el API de Python, se incluye esta clase para la resolución de problemas con el paquete cplex. Esta clase ha sido de gran utilidad para leer los datos de entrada desde un fichero con formato ".dat" y asignarlo a las variables correspondientes.

Los datos de entrada deben tener el formato de un fichero ".dat" de manera que la primera fila del fichero contiene un array con los pesos de los objetos que se quieren incluir en las cajas.Estos valores se almacenan de manera secuencial, correspondiendo el elemento i-esimo con el peso del i-eismo objeto que se quiere almacenar. En la segunda linea se incluyen los volúmenes de los objetos que se quieren incluir en las cajas. También se almacena de manera secuencial.
La siguiente fila contiene el coste de las cajas almacenado de manera secuencial. A continuación debe encontrarse el peso máximo que soportan las cajas y, por último el volumen máximo que almacenan las cajas. No debería haber nada más en el fichero de datos.
Toda esta información se almacena de manera secuencial y el forma de array. Los distintos valores se encuentran separados por comas. Por ejemplo:

```
[0.1, 0.2, 5, 3.8]
[0.3, 0.4, 7, 2.3]
[0.5, 1, 3]
[0.1, 4, 7]
[1, 5, 7]
```

Este fichero corresponde con el primer ejemplo de datos a probar. Contiene 4 objetos de pesos 0.1, 0.2, 5 y 3.8 respectivamente y, volúmenes 0.3, 0.4, 7 y 2.3.
En este caso hay tres cajas, en las que se pueden almacenar objetos, de costes 0.5, 1 y 3, en las que se puede almacenar un peso máximo de 0.1, 4 y 7, con las restricciones de volumen de 1, 5 y 7.

En cuanto a la generación automática de instancias del problema, se ha construido de la siguiente manera:

Inicialmente se han obtenido los parámetros del problema con ayuda de la clase input data construida por IBM, incluida en el fichero *inputdata.py* (junto con los ficheros entregados). El programa espera ser invocado con el fichero de entrada de dónde se quieren coger los datos. Guarda en las variables correspondientes, los parámetros introducidos por el usuario.
Una vez obtenidos los datos comprueba que los datos introducidos sean correctos, es decir, es necesario que compruebe que la cantidad de pesos introducidos para los objetos es igual a la cantidad de volúmenes introducidos para los objetos, y su correspondencia con coste, peso y volumen para las cajas.

Una vez se ha comprobado que los parámetros introducidos son correctos, se crea el modelo y se introducen las variables. En este problema, hay dos variables, una matriz binaria, con valores 0 o 1 que contiene información de si el objeto i está dentro de la caja j. A esta variable se le ha llamado w. La otra variable del problema, contiene información de si una caja contiene algún objeto dentro, llamada z. Esta variable también es del tipo binario, y de la misma manera, adopta los valores 0 o 1.

```
# se crea el modelo del problema
m = Model("Mudanza")
w = [[ m.add_var(var_type=BINARY) for j in range(nbCajas)] for i in range(nbObjetos)] #
    el objeto i esta en la caja j
z = [m.add_var(var_type=BINARY) for j in range(nbCajas)] # hay algun objeto en la caja
```

A continuación, se establece la función objetivo del programa. Como ya se ha observado en la formalización del problema. El objetivo principal es minimizar el coste de las cajas a utilizar de manera

[2] Junio 2022. Vesión 22.1.0 `https://www.ibm.com/es-es/products/ilog-cplex-optimization-studio`

que se introduzcan todos los objetos. Su traducción a la formalización se realiza como el máximo del negado de la suma de los costes de las cajas utilizadas (1). Esta función se traduce a la herramienta de programación lineal utilizada de la siguiente manera:

```
# funcion objetivo: minimizar el coste
m.objective = maximize(-xsum(c[j]*z[j] for j in range(nbCajas)))
```

Por último, hay que añadir las restricciones al problema.
Para ello, ha sido necesario traducir la formalización de las restricciones al lenguaje utilizado. Para ello, ha sido de gran utilidad la función "*xsum*"[3], incluida en la documentación de la herramienta de programación lineal utilizada. Esta función calcula la suma de expresiones lineales. Como todas las restricciones del problema son sumatorios y comparaciones, con estas aproximaciones ha sido suficiente para construir el modelo.

La primera restricción es que cada objeto solo puede estar en una sola caja, formalizado en la ecuación 4. Para ello, se ha iterado sobre cada objeto, y se ha añadido que, la suma de todas las posibles cajas en las que puede estar contenido, tiene que ser exactamente 1. De esta manera se puede asegurar que cada objeto está en exactamente una caja.

```
# restriccion: cada objeto solo puede estar en una caja
for i in range(nbObjetos):
    m += xsum(w[i][j] for j in range(nbCajas)) == 1
```

A continuación, se añaden las restricciones de peso y volumen. Para cada caja j, entre las posibles cajas, la suma total del peso de los objetos tiene que ser menor que el peso que soporta la caja (si esa caja es utilizada). En caso de que la caja no sea utilizada, no debería haber ningún objeto en ella. Para saber cuáles son los objetos contenidos en la caja, se utiliza la variable w. Se multiplica el peso del objeto por 1 si está en la caja o por 0 si no está. De esta manera, se logra sumar únicamente los objetos que están en esa caja.
Con la restricción de volumen ocurre exactamente lo mismo.

```
# restriccion: la cantidad de peso que tiene una caja es menor que la cantidad de peso
    que soporta
for j in range(nbCajas):
    m += xsum(x[i]*w[i][j] for i in range(nbObjetos)) <= P[j] * z[j]

# restriccion: la cantidad de volumen que puede contener una caja es menor que la
    cantidad de volumen maxima de una caja
for j in range(nbCajas):
    m += xsum(y[i]*w[i][j] for i in range(nbObjetos)) <= V[j] * z[j]
```

Por último, queda añadir que la variable z tendrá valor 1 si esa caja contiene algún objeto. Es decir, si la columna j de la matriz w contiene algún 1 en sus filas. Esta restricción se traduce de la siguiente manera:

```
# restriccion: si hay algun objeto en la caja, la caja esta ocupada
for j in range(nbCajas):
    for i in range(nbObjetos):
        m += w[i][j] <= z[j]
```

Una vez se tienen todas las restricciones en el modelo, queda resolverlo mediante la opción "optimize()" de dicho modelo.
Una vez se resuelve, se comprueba si se ha obtenido una solución óptima y se muestra el estado final de la solución.

[3]Junio 2022 `https://python-mip.readthedocs.io/en/latest/classes.html`

Se han generado unos planes de prueba de manera que se compruebe el análisis de sensibilidad del tiempo de solución a las instancias del problema.
Los planes establecidos son: pequeñas instancias del problema con datos que llevan a una solución óptima o no, para verificar la corrección del problema. A continuación se realizaron pruebas con muchas instancias de cajas o otras con muchas instancias de objetos comparando los tiempos de ejecución que toman para resolver los mismos. Por último, se ha realizado una prueba de carga para observar el tiempo que le toma resolver algunas instancias del problema.

4. Experimentación

En este apartado se describen las pruebas que se han realizado para verificar la corrección del algoritmo y comprobar los parámetros del problema que influyen en el tiempo de solución.

Para realizar las pruebas de corrección, se utilizado primero el fichero de "*datos.dat*" que contiene 4 objetos y 3 cajas de manera que se debe colocar el primer objeto en la primera caja, el segundo y el cuatro objeto en la segunda caja y el tercer objeto en la última caja. Se obtiene como resultado un coste de 4.5, ya que se utilizan las 3 cajas y se corrobora así, la corrección de asignación de cajas cuando esta es posible.

La siguiente prueba de corrección realizada se encuentra en el fichero "*datos2.dat*". En esta prueba, se deben asignar todos los objetos a la caja 2, ya que es la que produce un menor coste a pesar de que otros objetos podrían asignarse a otras cajas (aunque produciría mayor coste). Como se resultado se obtiene que asigna todos los objetos a la caja 2 ya que caben todos y tiene un menor coste.

La última prueba de verificación se realiza con una instancia del problema que no tiene solución. Es decir, no hay manera alguna de distribuir todos los objetos en cajas. Esta instancia se encuentra en el fichero "*datos3.dat*". Esta instancia no devuelve distribución de los objetos en las cajas y por lo tanto, es correcta la solución.

También se realizaron pruebas con los casos triviales: cuando no hay objetos para asignar a las cajas y cuando no hay cajas para asignar los objetos. En el primer caso, devuelve una solución con coste 0 y la caja vacía. En el segundo caso, no devuelve solución válida ya que las variables a optimizar son nulas.
Una vez concluidas estas pruebas, se puede confirmar que el algoritmo implementado para generar instancias del problema es correcto.

Para poder localizar cuáles son los parámetros que influyen en el tiempo de solución del problema, se han realizado pruebas de manera que se incrementen iterativamente la cantidad de parámetros del problema y se han comparado los tiempos de solución obtenidos.

Primero, se han realizado pruebas incrementando la cantidad de objetos.
Las pruebas se han realizado manteniendo 3 cajas de manera que todos los objetos deben repartirse en dos cajas para lograr la solución óptima.
Como se puede observar en la figura 1 el tiempo de de ejecución en CPU aumenta lentamente a pesar de que se aumentan la cantidad de objetos a asignar en las cajas. Es admisible el tiempo que aumenta, ya que es normal que le cueste más resolver un problema que tenga más objetos que asignar, pero dicho incremento es aceptable dada la carga de trabajo del sistema. Llaman la atención los picos de ejecución que tiene el programa. Estos picos son debidos a descomposiciones caprichosas que realiza la herramienta utilizada con algunas instancias del problema. A pesar de que no son nada beneficiosas para el objetivo principal del proyecto, se ha considerado que son dignos de mención.

La segunda prueba realizada, utiliza un aumento de las cajas en las que hay que asignar objetos. Todas las cajas son idénticas a excepción de las tres primeras (la primera caja tiene peso y volumen pequeños y además poco coste, la segunda caja es algo más grande y tiene poco coste pero no caben todos los objetos en ella, la tercera caja es más grande, caben todos los objetos que se quieren asignar y tiene el mismo coste que las demás). La instancia del problema utilizado contiene 4 objetos de poco peso y volumen de manera que se asigna siempre el primer objeto a la primera caja y el resto de objetos en cualquiera de las cajas con índice mayor que 2. Con esta asignación se logra el mínimo coste del problema. Como se

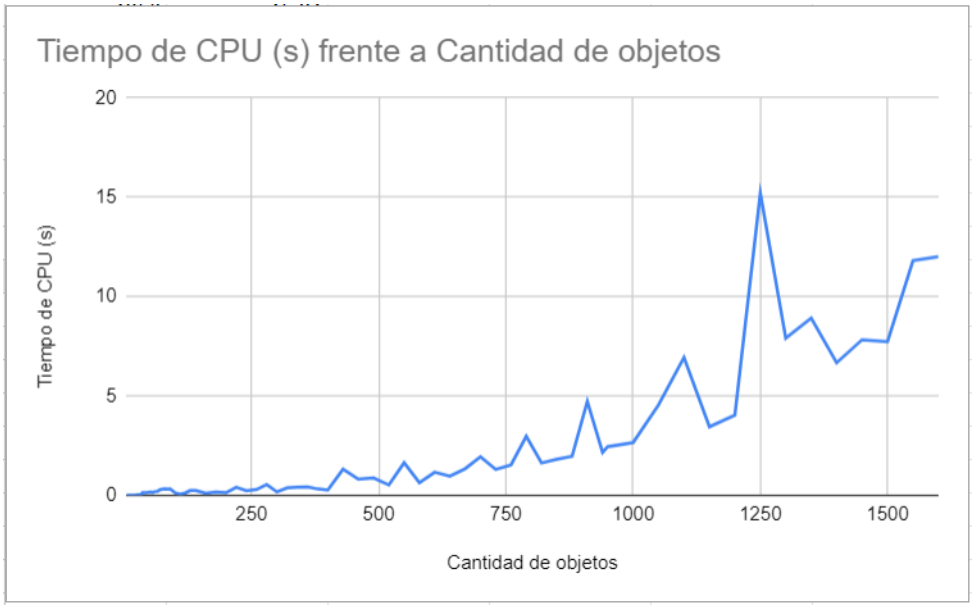

Figura 1: Relación entre la cantidad de objetos a asignar y el tiempo de CPU

puede observar en la figura 2 el tiempo de ejecución en CPU aumenta muy rápidamente al aumentar la cantidad de cajas en las que se deben almacenar los objetos. Al igual que ocurría en el caso anterior, debido a ejecuciones caprichosas, muestra tiempos irregulares, pero se puede observar como el incremento del tiempo es mucho mayor que en el experimento realizado aumentando la cantidad de objetos.

Como conclusión de las 2 últimas pruebas realizadas, se puede concluir que la cantidad de cajas entre las que hay que repartir los objetos, en el caso de que haya varias opciones para repartirlos y sea indistinto la caja que elegir, la cantidad de cajas es el parámetro que más influye en el tiempo de solución del problema. Las pruebas nombradas anteriormente se encuentran en los ficheros "*datos5.dat*" y "*datos6.dat*".

Por último se ha realizado una prueba de carga para observar el tiempo que le costaría resolver un problema asemejado a una instancia de problema real. En este caso, se han decidido repartir 200 objetos entre 50 cajas. Esto le cuesta resolverlo 662.46 segundos (11 minutos 2 segundos 46 milisegundos). A partir de esta carga de trabajo, se considera que el problema es demasiado grande y habría que buscar otras herramientas de optimización para resolverlo.

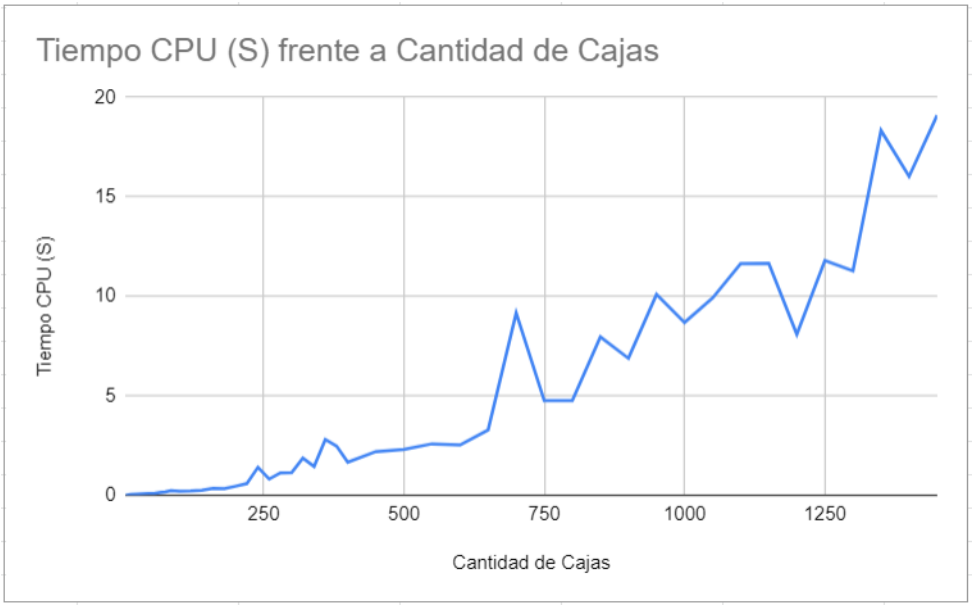

Figura 2: Relación entre la cantidad de cajas en las que se pueden asignar objetos y el tiempo de CPU