Universidad del País Vasco Euskal Herriko Unibertsitatea
ZIENTZIA ETA TEKNOLOGIA FAKULTATEA
FACULTAD DE CIENCIA Y TECNOLOGÍA

Gradu amaierako lana /Trabajo fin de grado
Ingenieritza Elektroniko gradua/ Grado en Ingeniería Electrónica

Laneko Txanden Esleipen Automatikoa, Lehentasunak eta Murrizketak kontsideratuz / Asignación Automática de Turnos de Trabajo teniendo en cuenta Restricciones y Preferencias

Egilea/ Autor:
Aitor Salazar Vieira
Zuzendaria/Directora:
Amparo Varona Fernández

Leioa, 2023ko Ekainaren 22a / Leioa, 22 de junio de 2023

Índice general

Resumen

En las ciencias de la computación, el Problema de Organización de Enfermeras (*Nurse Scheduling Problem*, NSP) es un problema muy recurrente y del que se siguen investigando nuevas soluciones dadas sus aplicaciones en todos los sectores empresariales. En su forma más común se interpreta como "el problema de gestionar los turnos de una plantilla de varios trabajadores", pero también se puede utilizar para modelizar la gestión de recursos o vehículos de las industrias.

En este trabajo, se propone resolver dos casos reales, la plantilla de una cafetería y la de una clínica (o una planta de hospital). Los turnos de estas plantillas tienen que ser gestionados para que se adecúen a las necesidades de los trabajadores y se gestione el tiempo de la manera más eficaz posible. La solución que se propone consiste en, partiendo de la definición clásica del NSP, plantear cada uno de los dos problemas como un problema de programación lineal en enteros, y usar varios algoritmos de optimización diferentes para encontrar la mejor solución. Los métodos de resolución se pueden dividir principalmente en dos grupos: una implementación "más básica" de un algoritmo voraz para resolver el problema y otra implementación "más completa" capaz de usar cualquier algoritmo del que se disponga (en este trabajo concretamente se han usado el Simplex y el *Branch and cut*).

Capítulo 1

Introducción

La gestión de una plantilla de empleados es un problema real al que se enfrentan cientos de miles de personas cada día en todos los sectores del tejido empresarial. Vivimos en un mundo cada vez más competitivo donde prima la eficiencia en el trabajo y la buena gestión del tiempo puede ser una de las claves para lograrla. En este contexto aparece el Problema de Organización de Enfermeras (*Nurse Scheduling Problem*, NSP), que fue formulado por primera vez hace más de 40 años [1] precisamente con el propósito de encontrar la mejor forma de organizar los turnos de una plantilla de trabajadores.

En empresas pequeñas de poco más de un par de empleados es fácil gestionar los horarios y los turnos de los trabajadores, pero cuando se trata de grandes compañías multinacionales de varias decenas o cientos de trabajadores, se vuelve imposible llevar a cabo este proceso a mano. Cada trabajador tiene sus propias necesidades y prioridades que se tienen que gestionar e imprevistos a los que se tiene que poder responder con eficacia para garantizar el buen funcionamiento de la plantilla. En caso de no hacerlo, puede haber una gran pérdida de recursos o una bajada en la producción de los empleados.

Este es un problema muy relevante dentro del ámbitos de las ciencias de la computación y tiene aplicaciones que van más allá de organizar plantillas de trabajadores. Es por esto que tiene un gran valor académico y año tras año aparecen nuevas versiones que aportan novedosas soluciones que pueden ser adaptadas a otros problemas. Con la llegada de los ordenadores y las técnicas de optimización modernas, los avances en este campo han aumentado significativamente siendo algunos de los ejemplos más recientes el trabajo basado en un modelo preventivo-reactivo, de Otero-Caicedo et al. [2], el trabajo que plantea una solución basándose en el problema de flujo de múltiples productos, de Ali El Adoly et al. [3] o el trabajo de Hoshino et al. donde se plantea una solución basada en el algoritmo "branch and cut"[4]. Gracias a estos avances, ha sido posible resolver problemas donde se tienen en cuenta muchas más variables que benefician a los trabajadores, haciendo las asignaciones de turnos más flexibles y teniendo en cuenta parámetros como su experiencia en el puesto (o en la tarea que se desempeñan), su disponibilidad o su preferencia por un turno en concreto.

El objetivo de este trabajo es adaptar la definición clásica del NSP para resolver dos casos reales; una cafetería de media docena de empleados, y una clínica médica de unos veinte empleados y que consta de médicos, enfermeros, auxiliares, etc. Para resolver estos dos casos reales, la intención es plantearlos como problemas de programación lineal en

enteros, que es una rama del estudio de los problemas de optimización que se diferencia de las demás en que la función objetivo a optimizar es una expresión lineal. Después, este problema se resuelve utilizando los dos métodos de resolución que se han diseñado: el primero está programado desde cero y se basa en un algoritmo voraz, que es uno de los algoritmos más sencillos de implementar, aunque no por ello una mala opción. La particularidad de estos algoritmos es que toman la opción que parece mejor en cada momento sin molestarse en las consecuencias que pueda tener esa decisión. El segundo, es una implementación que hace uso de varias librerías de código abierto para utilizar algoritmos más complejos y completos. Concretamente, la librería principal que se ha usado es *PuLP*, un librería para modelización de problemas lineales.

Antes de dar la solución a esos dos problemas reales, se hace un breve repaso de los temas en los que se basa la solución propuesta por este trabajo. En el capítulo 2, se explican los fundamentos teóricos de este trabajo. Primero se explica qué es la programación lineal y los problemas de optimización, que son la teoría mediante la cual se resuelven la mayoría de problemas de optimización, especialmente los que son lineales. En segundo lugar, se habla sobre la programación dinámica, que está estrechamente ligada a los algoritmos voraces, que son los que se han usado para la implementación básica de este trabajo. Por último, se hace una breve introducción a la teoría de la complejidad computacional y se da una explicación sobre la complejidad del problema que trata de resolver este trabajo. Desde el punto de vista de la complejidad computacional, el problema NSP en particular se considera que es *NP-duro*, lo que lo convierte en uno de los más difíciles de resolver y se desconoce que pueda ser resuelto en tiempo polinómico.

A continuación, en el capítulo 3 se plantea el modelo matemático del problema. El NSP es un problema de optimización, así que los dos ejemplos que se van resolver se plantean también como tal. Para ello, se sigue el procedimiento habitual para definir un problema de optimización: se definen el problema y las variables del problema; después se define la función objetivo, que en este caso habrá que maximizar; y por último se definen las restricciones al problema.

El capítulo 4 es el eje central de este trabajo. En él se encuentra el desarrollo de la solución y los resultados a los dos problemas propuestos. A grandes rasgos, el problema se resuelve mediante un programa escrito en lenguaje Python [5]. Se ha optado por este lenguaje porque es el más versátil en ciencia gracias a las librerías de las que dispone y se ha convertido en el estándar en el estudio de la optimización y del aprendizaje automático. Además, a través de Numpy [6], todo el programa se ha realizado utilizando *arrays* en lugar de listas por los beneficios que aportan.

Finalmente, en el capítulo 5 se presentan las conclusiones a las que se han llegado durante el desarrollo de este trabajo. Todavía quedan preguntas sin resolver además de mucho margen de mejora, al fin y al cabo, no es casualidad que año tras años sigan apareciendo nuevas publicaciones que traten de resolver el problema.

Cabe mencionar que, anexado a este documento, puede encontrarse el código desarrollado al completo en forma de Apéndice. Junto al código se ha añadido una breve explicación de la instalación previa que se tiene que completar para poder usar el programa, además de una escueta guía de uso.

Capítulo 2

Fundamentos teóricos

Hay una gran variedad de algoritmos y técnicas que se han utilizado para tratar de resolver el Problema de Organización de Enfermeras que van desde algoritmos genéticos, computación paralela y optimización estocástica hasta la programación dinámica y los algoritmos voraces. Estos últimos sientan las bases de algoritmos más complejos, como por ejemplo, los algoritmos genéticos. De hecho, en cierto modo se pueden considerar a los algoritmos voraces como un caso particular de la programación dinámica. Por este motivo, es esencial dar una breve introducción sobre el funcionamiento y programación de esta clase de algoritmos que se utilizan para resolver problemas de optimización y de decisión, además de qué son y cómo se resuelven los problemas de optimización.

En este capítulo se hará una breve introducción a algunos de los conceptos que aparecen en el desarrollo de la solución del problema de NSP. Como este trabajo esta centrado en la optimización de un problema matemático, su desarrollo pasa por algunas áreas de las matemáticas que tratan sobre algoritmia y teoría de complejidad.

2.1. Problemas de optimización

Los problemas de optimización son los problemas que se centran en encontrar la mejor solución a partir de todas las formas factibles y están constituidos de muchas partes: la función objetivo, las restricciones, las variables, las soluciones y candidatas a solución... Dependiendo de cómo se definan y como estén interconectadas, los problemas serán de una forma u otra. Pero si hay una pieza que funciona de forma similar para todos los problemas de optimización es la función objetivo, pues todo problema de optimización tiene como objetivo maximizar o minimizar un valor, que será el resultado de esa función. Así, la combinación específica de variables que da ese valor como resultado de la función objetivo, será la solución al problema.

Las variables de optimización son lo que se quiere calcular resolviendo el problema de optimización y para ello se busca la combinación de ellas que mayor (o menor) resultado de en la función objetivo. Además, las variables estarán sometidas a restricciones que son ecuaciones e inecuaciones que imponen condiciones sobre los valores que las variables pueden tomar. La pieza que suele diferenciar a los problemas de optimización suele ser la función objetivo.

La función objetivo puede ser cualquier expresión matemática. Por ejemplo, el trans-

porte de un átomo en una trampa de potencial armónico seguirá una trayectoria que se puede describir mediante una serie de Fourier [7]. Para evitar la decoherecia de estado durante el transporte, se puede definir un coeficiente de "ruido durante el transporte"que, cuanto menor sea, mejor se mantendrá la coherencia y más rápido será el transporte. Este coeficiente de ruido se calcula integrando la expresión de la trayectoria de Fourier sobre un intervalo. El problema es que se desconocen los valores de los coeficientes de la serie de Fourier que se va a integrar. Solución; plantear un problema de optimización donde la función objetivo es la integral que se quiere minimizar, para así obtener los valores de los coeficientes, es decir, la solución.

Ese es un ejemplo de una función objetivo que no es lineal. Sin embargo, si se puede formular la función objetivo como una función lineal, que depende de restricciones que se pueden escribir como ecuaciones e inecuaciones, entonces el problema pertenece a un caso especial de problemas de optimización: los **problemas de programación lineal**.

Para entender mejor cuál es la estructura de un problema de optimización y, en concreto, uno de programación lineal, se propone el siguiente ejemplo. Una empresa que fabrica comida para gatos quiere fabricar su producto al menor coste posible pero cumpliendo con sus valores nutricionales mínimos. Cada producto consta de seis ingredientes, así que para abaratar los costes habrá que variar las cantidades hasta hacerlo lo más barato posible sin que incumpla ninguna de las condiciones nutricionales.

Cada ingrediente aporta una parte del total de proteínas, grasas, fibra y sal. En la Tabla 2.1 se muestra el coste por gramo de cada ingrediente y la contribución en gramos (por gramo de ingrediente) al total de cada valor nutricional:

Ingrediente	**Coste (€/g)**	**Proteína**	**Grasa**	**Fibra**	**Sal**
Pollo	0.013	0.100	0.080	0.001	0.002
Ternera	0.008	0.200	0.100	0.005	0.005
Cordero	0.010	0.150	0.110	0.003	0.007
Arroz	0.002	0.000	0.010	0.100	0.002
Trigo	0.005	0.040	0.010	0.150	0.008
Gelatina	0.001	0.000	0.000	0.000	0.000

Tabla 2.1: *Costes y aportes nutricionales de cada ingrediente.*

El primer paso es definir las variables del problema que serán los porcentajes de cada ingrediente que irán en cada lata de producto. Si cada lata de producto es de 100g, estos porcentajes son directamente la cantidad de ingrediente que habrá por lata.

$$
\begin{aligned}
x_1 &= porcentaje\ de\ pollo \\
x_2 &= porcentaje\ de\ ternera \\
x_3 &= porcentaje\ de\ cordero \\
x_4 &= porcentaje\ de\ trigo \\
x_5 &= porcentaje\ de\ arroz \\
x_6 &= porcentaje\ de\ gelatina
\end{aligned}
$$

Lo siguiente es plantear la función objetivo. Esta función es linealmente dependiente

de las variables y el objetivo es minimizar su resultado,

$$F = \text{mín}\{0{,}013\,x_1 + 0{,}008\,x_2 + 0{,}010\,x_3 + 0{,}002\,x_4 + 0{,}005\,x_5 + 0{,}001\,x_6\}. \tag{2.1}$$

A continuación se formulan las restricciones. La primera será que la suma del total de porcentajes debe ser 100,

$$x_1 + x_2 + x_3 + x_4 + x_5 + x_6 = 100; \tag{2.2}$$

y las siguientes serán restricciones que marcan los valores nutricionales a seguir. Por ejemplo,

$$\begin{aligned}
0{,}100x_1 + 0{,}200x_2 + 0{,}150x_3 + 0{,}000x_4 + 0{,}040x_5 + 0{,}0x_6 &\geq 8{,}0\\
0{,}080x_1 + 0{,}100x_2 + 0{,}110x_3 + 0{,}010x_4 + 0{,}010x_5 + 0{,}0x_6 &\geq 6{,}0\\
0{,}001x_1 + 0{,}005x_2 + 0{,}003x_3 + 0{,}100x_4 + 0{,}150x_5 + 0{,}0x_6 &\leq 2{,}0\\
0{,}002x_1 + 0{,}005x_2 + 0{,}007x_3 + 0{,}002x_4 + 0{,}008x_5 + 0{,}0x_6 &\leq 0{,}4.
\end{aligned}$$

Este es el planteamiento habitual de un problema de programación lineal. El siguiente paso sería resolver este problema lineal haciendo uso de algún algoritmo de optimización. Por lo tanto, después de ver este ejemplo se puede describir intuitivamente la estructura general de un problema de optimización:

1. **Definir las variables** del problema que se van a optimizar.
2. **Definir la función objetivo** como una combinación lineal de unos coeficientes y las variables de optimización. El objetivo será maximizar (o minimizar) el resultado de esta función.
3. **Formular las restricciones** y condiciones que deberá cumplir la solución al problema. Estas restricciones se escriben como ecuaciones e inecuaciones lineales. Es importante mencionar que no se permiten restricciones como inecuaciones estrictas.
4. **Resolver el problema** empleando un algoritmo de optimización y encontrar el valor óptimo de la función objetivo (ya sea el máximo o el mínimo). Además, normalmente lo que se busca no es el resultado de la función, sino la solución que devuelve ese valor; es decir, los valores de las variables dan ese resultado. Por ello, debe existir un mecanismo para guardar la solución de una iteración del programa al siguiente.

2.2. Programación lineal de optimización

Para describir las propiedades y los algoritmos de los problemas lineales existen dos formas canónicas: estándar y no estándar. Un programa lineal en forma **estándar** es aquel cuyas restricciones se describen mediante inecuaciones y se pretende maximizar el resultado de la función objetivo, mientras que uno expresado en forma **no estándar** es aquel en que las restricciones vienen dadas por ecuaciones. Lo más habitual es expresar los problemas de programación lineal en forma estándar (ver capítulo 29 de [8]).

Cualquier combinación lineal de las variables del problema que cumpla con las restricciones se denomina **solución viable** del problema lineal. Las restricciones, que habitualmente serán inecuaciones, forman una región en el espacio de soluciones que contiene a

las soluciones viables. Por ejemplo, si un problema tiene dos variables, forma un espacio de dos dimensiones y las restricciones serán rectas. Como son inecuaciones, un lado de la recta cumplirá la inecuación mientras que el otro no. La región que surge de la intersección de todas las rectas se conoce como **región viable.** Cada valor, calculado por la función objetivo con una solución que queda dentro de la región viable, se denomina **valor objetivo**, y este valor será el que tendrá que ser maximizado (o minimizado).

Los valores óptimos se encuentran en los contornos de las regiones viables, concretamente, en las intersecciones entre el contorno (o un vértice) de una región con una restricción, que se representa como una recta. Se aplica la misma lógica a medida que se aumentan las dimensiones de la espacio de soluciones (es decir, a medida que aumenta el número de variables), de forma que las restricciones pasan a ser semiplanos de varias dimensiones y las intersecciones de estos semiplanos forman la región viable convexa. De la misma forma que en dos dimensiones, las soluciones óptimas se encontrarán en los vértices de intersección. En este contexto, a la región viable que surge de estas intersecciones entre semiplanos se le llama **simplex.** Ahora la función objetivo es un hiperplano, que es convexo, por lo que existe una solución óptima en el vértice del simplex.

La definición del simplex conduce al algoritmo más común y usado para resolver problemas de programación lineal, el algoritmo simplex. El algoritmo simplex toma como entrada un problema lineal y devuelve la solución óptima. Para ello, empieza en uno de los vértices del simplex y con cada iteración se mueve de un vértice a otro colindante, siempre intentando aumentar el valor de la función objetivo. El algoritmo termina cuando llega a un máximo local, que es un vértice desde el cual todos los vértices colindantes tienen un valor objetivo menor. Como la región viable es convexa y la función objetivo es lineal, este máximo local es de hecho el máximo global.

2.2.1. Forma estándar

Cuando un problema de programación lineal está descrito en forma estándar, este estará definido por n números reales $c_1, c_2 \dots c_n$; m números reales $b_1, b_2 \dots b_m$ y $m \cdot n$ números reales a_{ij} donde $i = \{1, 2, \dots m\}$ y $j = \{1, 2, \dots n\}$. El objetivo del problema será encontrar los n números reales $x_1, x_2, \dots x_n$ tales que se consiga:

$$\text{maximizar} \qquad \sum_{j=1}^{n} c_j \, x_j \tag{2.3}$$

$$\begin{aligned} \text{sujeto a} \qquad & \sum_{j=1}^{n} a_{ij} \, x_j \;\leq\; b_i \qquad para\; i = 1, 2, \dots m \\ & x_j \;\geq\; 0 \qquad para\; j = 1, 2, \dots n \end{aligned} \tag{2.4}$$

La expresión 2.3 se llama **función objetivo** y las $n + m$ inecuaciones 2.4 son las restricciones al problema. Las segunda inecuación que solo depende de las **variables** se llama **restricción de no negatividad.** No todos los problemas lineales tendrán restricción de no negatividad pero en la forma estándar es un requerimiento. Otra manera de expresar un problema lineal, dentro de la forma estándar, es matricialmente.

$$\begin{aligned} \text{maximizar} \qquad & c^T x \\ \text{sujeto a} \qquad & A\, x \;\leq\; b \\ & x \;\geq\; 0. \end{aligned} \tag{2.5}$$

De esta manera se puede expresar un problema mediante la tupla (A, b, c).

Como ya se ha mencionado anteriormente, al conjunto de variables (o vector, si se da en forma matricial) $\overline{x}$ que satisface las restricciones del problema se le llama solución viable mientras que a uno que no satisface aunque sea solo una de las restricciones se le llama solución inviable. Cualquier solución al problema $\overline{x}$, ya sea viable o inviable tiene un valor objetivo $c^T x$. Una solución viable cuyo valor objetivo sea mayor que el resto es una **solución óptima** y se dice que tiene el **valor objetivo óptimo.** Si un problema lineal no tiene soluciones viables, se dice que es **inviable**. Si por el contrario sí que tiene soluciones viables pero no es posible encontrar un valor objetivo óptimo finito, entonces el problema **no está acotado.**

El algoritmo simplex definido anteriormente, es el algoritmo básico para resolver problemas lineales y debido a los pasos que sigue, funciona en todos los casos. Por ello, el teorema fundamental se define a través de este particular algoritmo[8].

> ***Teorema fundamental de la programación lineal.*** Cualquier problema lineal dado en forma estándar cumple una de las siguientes afirmaciones:
>
> - Tiene una solución óptima con un valor objetivo finito,
> - es inviable, o
> - no está acotada.
>
> Si el problema es inviable, el algoritmo simplex devolverá que es inviable y si no está acotado devolverá que no lo está. En cualquier otro caso devolverá una solución óptima con un valor objetivo finito.

2.2.2. Programación lineal en enteros

Si a un problema de programación lineal se le añade la condición de que las variables tengan que ser números enteros, entonces se dice que es un un problema de programación lineal en enteros o PLE (Integer Linear Problem, ILP) puro o mixto. Como ocurre en otras ramas de las matemáticas, los modelos que involucran números enteros son más complejos de resolver que los que tratan con números reales[9]. Los modelos de PLE más comunes suelen ser los que limitan los valores de sus variables a 0 y 1 (convirtiéndolos en problemas de decisión).

En programación lineal en enteros se distinguen dos formas de expresar los problemas (matricialmente), la forma canónica:

$$\begin{aligned} \text{maximizar} \quad & c^T x \\ \text{sujeto a} \quad & A\,x \leq b, \\ & x \geq 0, \\ \text{y} \quad & x \in \mathbb{Z}^n, \end{aligned} \tag{2.6}$$

y la forma estándar:

$$\begin{aligned} \text{maximizar} \quad & c^T x \\ \text{sujeto a} \quad & A\,x + s = b, \\ & s \geq 0, \\ & x \geq 0, \\ \text{y} \quad & x \in \mathbb{Z}^n \end{aligned} \tag{2.7}$$

2.3. Programación dinámica

Los problemas de optimización se pueden resolver mediante algoritmos de programación dinámica. La programación dinámica sigue una lógica que se podría resumir como "divide y vencerás". El fundamento básico de la programación dinámica se centra en resolver los subproblemas que surgen de tratar de resolver el problema principal, y después construir la solución al problema combinando las soluciones de los subproblemas. Concretamente, un algoritmo de programación dinámica se beneficia de los problemas en los que los subproblemas se repiten. En lugar de calcular las soluciones de todos los subproblemas una y otra vez (aunque se repitan) como harían otra clase de algoritmos, en programación dinámica se calcula el resultado de cada subproblema una vez y se guarda el resultado en una estructura que se asemeja a una tabla. De esta forma, si ese subproblema vuelve a aparecer mientras se resuelve otro subproblema, en lugar de volver a resolverlo, simplemente se consulta su resultado en la tabla (ver capítulo 15 de [8]).

Esta metodología ahorra el coste computacional que se añadiría si se tuviera que recalcular un valor ya conocido cada vez que se resolviera un subproblema, pero esto también significa que la programación dinámica emplea más memoria para ahorrar tiempo computacional. A pesar de ello, el ahorro conseguido suele ser sustancialmente mayor que la pérdida en espacio de memoria. Por ejemplo, una solución de tiempo exponencial puede ser transformada en una de tiempo polinómico gracias a este método.

Para ayudar a comprender los beneficios de la programación dinámica frente a un algoritmo por fuerza bruta común, se propone el siguiente ejemplo en el que se intenta cortar una barra en trozos de distintas longitudes maximizando el beneficio obtenido. Distintas longitudes proporcionan distintos beneficios, como se muestra en la Tabla 2.2.

longitud	1	2	3	4	5	6	7	8	9	10
precio	1	5	8	9	10	17	17	20	24	30

Tabla 2.2: *Tabla de precios para cada longitud de barra. Cuanto más larga la barra, mayor su precio de venta y mayor beneficio obtendrá la empresa.*

Una barra de longitud n se puede seccionar de distintas maneras, y con cada una se obtendrá un beneficio. Se puede escribir una descomposición de una barra de k secciones como una suma, $n = i_1 + i_2 + i_3 + \cdots + i_k$, de forma que la barra de longitud n está compuesta de k trozos de longitud i_k (por ejemplo, $8 = 3 + 3 + 2$). Mismamente, el beneficio asociado a una solución concreta de k cortes se expresa como $b_n = r_1 + r_2 + r_3 + \cdots + r_k$. El beneficio óptimo para una barra de longitud n puede calcularse eligiendo el máximo de entre todas las opciones de corte de esa barra, incluida la opción de no cortarla,

$$b_n = \text{máx}\,(p_n, b_1 + b_{n-1}, b_2 + b_{n-2}, \ldots, b_{n-1} + b_1).$$

El primer argumento corresponde al caso de no hacer ningún corte mientras que los siguientes se corresponden con los beneficios obtenidos de hacer dos secciones iniciales de longitudes i y $n-i$, para después seccionar cada uno de esos dos trozos de forma óptima sucesivamente hasta obtener los beneficios b_i y b_{n-i}.

Para resolver el problema original de la barra de longitud n, se resuelven primero subproblemas del mismo tipo, es decir, se considera cada uno de los dos trozos resultantes de un corte como su propio problema de maximizar el valor. La solución del problema principal combina las soluciones de los dos subproblemas relacionados para los cuales se maximiza el valor. Por este motivo, se dice que el problema de maximización del valor de barras tiene estructura óptima.

Este problema puede ser resuelto con un algoritmo recursivo que va resolviendo los subproblemas tal y como se ha descrito,

```
def cut_rod_recursive(p, n):
    """
    Esta función tiene una complejidad O(2^n).
    :param p: Array que contiene precios para cada longitud.
    :type p: numpy.ndarray
    :param n: Longitud de la barra.
    :type n: int
    :return: Beneficio óptimo.
    """
    if n == 0:
        return 0
    q = -np.inf
    for i in range(1, n + 1):
        q = max([q, p[i - 1] + cut_rod_recursive(p, n - i)])
    return q
```

Código 2.1: *Fragmento de código escrito en Python que resuelve el problema de corte de una barra en distintos trozos mediante un algoritmo recursivo de fuerza bruta.*

Esta implementación toma un array[1] de precios p y la longitud de la barra n y devuelve el beneficio óptimo que se puede obtener de esa barra. Para ello, primero comprueba si la longitud es igual a 0, porque en caso de serlo no hay beneficio que obtener. En el caso contrario, se inicializa el beneficio óptimo $q = -\infty$ de forma que el bucle `for` resuelve recursivamente el problema.

Esta forma de resolver el problema es tremendamente ineficiente porque se llama a sí mismo una y otra vez para resolver los mismos subproblemas. Si por ejemplo $n = 5$, al comienzo del bucle se llamará primero a `cut_rod_recursive(p, 4)`, después a `cut_rod_recursive(p, 3)`... y así sucesivamente hasta llegar a $n = 1$ y comenzará a iterar. Cuando acabe de iterar con $n = 1$, volverá a $n = 2$ y necesitará volver a completar un ciclo en $n = 1$ para poder obtener el resultado. Este proceso seguirá a medida que aumente n haciéndose más y más costoso.

Este proceso se puede recortar drásticamente usando programación dinámica. Generalmente, se pueden distinguir dos estrategias para implementar programación dinámica:

- **Orden natural con memoización.** En este enfoque, el código se escribe siguiendo la secuencia natural de eventos, de principio a fin, para que se resuelva recursivamente con la particularidad de ir guardando los resultados obtenidos de los subproblemas

[1] En todos los casos se presupone el uso de *Numpy* como: `import numpy as np`.

resueltos. Primero se comprueba si el subproblema ha sido resuelto previamente. Si es así, se consulta el resultado en una tabla (que habitualmente suele ser un array o una tabla *hash*), logrando ahorrar en tiempo computacional. Si no, se resuelve el subproblema y se guarda el valor en la tabla.

A continuación, se muestra la resolución del problema del corte de la barra empleando memoización.

```
def cut_rod_memoized(p, n, r):
    """
    Esta función tiene una complejidad O(n^2). La diferencia con la
   anterior es que guarda los valores de los subproble-
    mas ya calculados en r.
    :param p: Array que contiene precios para cada longitud.
    :type p: numpy.ndarray
    :param n: Longitud de la barra.
    :type n: int
    :param r: Array que guarda los valores ya calculados.
    :type r: numpy.ndarray
    :return: Beneficio óptimo.
    """
    if r[n] >= 0:
        return r[n]
    if n == 0:
        q = 0
    else:
        q = -np.inf
        for i in range(1, n + 1):
            q = max([q, p[i - 1] + cut_rod_memoized(p, n - i, r)])
    r[n] = q
    return q
```

Código 2.2: *Fragmento de código que es similar al Código 2.1 con la diferencia de que se comprueba si el subproblema ya ha sido resuelto y guardado en el array r.*

- **Orden invertido.** En este otro enfoque se considera que existe una manera de ordenar los subproblemas "por tamaño". Aprovechándose de esta propiedad, se ordenan los subproblemas de menor a mayor y se resuelven en orden de forma que resolver algún subproblema en particular solo dependa de haber resuelto los que son más pequeños que él. Así, cuando se vaya a resolver un subproblema, todos aquellos en los que este dependa ya se habrán resuelto y se habrán guardado sus resultados. Con este método, cada subproblema se resuelve solo una vez.

```
def cut_rod_bottom_up(p, n):
    """
    Esta función tiene una compleidad O(n^2).
    :param p: Array que contiene precios para cada longitud.
    :type p: numpy.ndarray
    :param n: Longitud de la barra.
    :type n: int
    :return: Beneficio óptimo.
    """
    r = np.full((numb + 1), -np.inf)
    r[0] = 0
    for j in range(1, n + 1):
        q = -np.inf
        for i in range(1, j + 1):
```

```
15                 q = max([q, p[i - 1] + r[j - i]])
16             r[j] = q
17     return r[n]
```

Código 2.3: *Fragmento de código que muestra la implementación de orden invertido que no usa recursividad.*

La implementación de este método para resolver el problema de la barra es incluso más sencilla que la memoización del método recursivo. Dado que los problemas de tamaño i son más pequeños que los de tamaño j, para cada j, si el subproblema no ha sido resuelto ya lo resuelve y lo guarda, si no simplemente lo consulta.

Este ejemplo demuestra que una implementación basada en programación dinámica puede mejorar notoriamente la eficiencia y rapidez de un algoritmo. Para poder implementar esta metodología deben cumplirse una serie de condiciones, como por ejemplo, que el problema exhiba estructura óptima, como lo hace el ejemplo resuelto anteriormente. Sin embargo, este método no es siempre la mejor opción, dado que hay veces en las que no es necesario resolver los subproblemas de un problema, pues la opción más "fácil" en ese momento puede que sea la mejor.

2.3.1. Características de la programación dinámica

A pesar de ser aplicable a una gran variedad de casos, la programación dinámica requiere que se cumplan dos condiciones en el problema de optimización en que quiera ser empleada: que el problema exhiba estructura óptima y que contenga subproblemas que comparten subproblemas.

- **Estructura óptima.** Por definición, un problema exhibe estructura óptima si una solución que lo resuelva contiene soluciones óptimas a subproblemas. Esto tiene sentido dado que en programación dinámica la solución se construye a partir de las soluciones a sus subproblemas. El proceso de encontrar las soluciones a los subproblemas puede verse mejorado si se mantiene el espacio de subproblemas reducido, es importante elegir un espacio de subproblemas que se adecúe a las necesidades del problema en cuestión.

  En función del tipo de problema que se afronte, los problemas pueden variar en: cuántos subproblemas utiliza una solución óptima, y en cuántas decisiones pueden hacerse a la hora de determinar cuántos subproblemas deben usarse. Esto se ve claro en el ejemplo de la barra; solo utiliza un único subproblema de tamaño $n - i$, sin embargo, se deben tomar n decisiones de i para poder determinar cuál es la que obtiene la solución óptima.

- **Subproblemas que comparten subproblemas.** Esta propiedad es la clave que diferencia la programación dinámica del resto. En esencia, significa que es aplicable solo cuando los subproblemas se repiten, son siempre los mismos, en lugar de aparecer nuevos en cada iteración. Esto es lo que sucede con el algoritmo recursivo del ejemplo de la barra, cuando va pasando de un valor de n al siguiente, se encuentra con subproblemas que ya han sido resueltos y en lugar de volver a resolverlos simplemente consulta su solución en un array.

Como se muestra en la Figura 2.1, el algoritmo recursivo del ejemplo de la barra resolverá los mismos subproblemas una y otra vez dado que no guarda los valores de

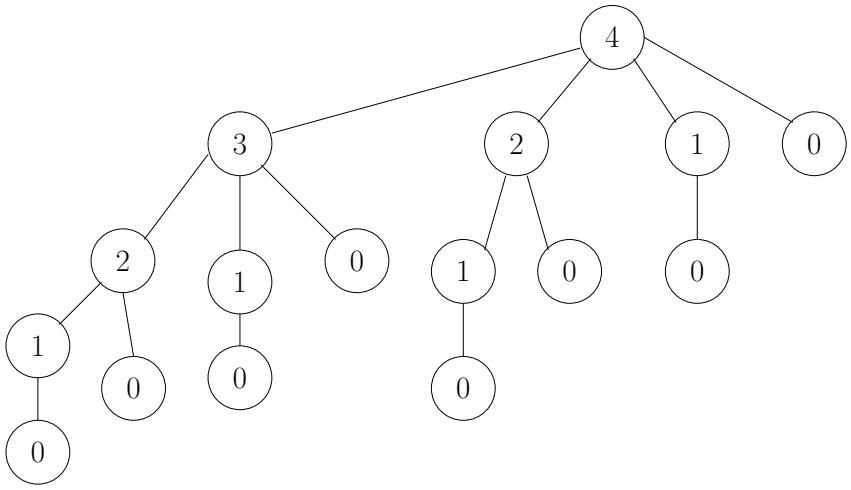


Figura 2.1: *Grafo que muestra las llamadas que hace el algoritmo recursivo por fuerza bruta para resolver el beneficio óptimo de cada longitud. Este grafo muestra el caso de* $n = 4$.

los subproblemas resueltos previamente, dando como resultado un tiempo de ejecución exponencial. Por el contrario, si se guardan los valores calculados, el grafo se simplifica, como se indica en la Figura 2.2.

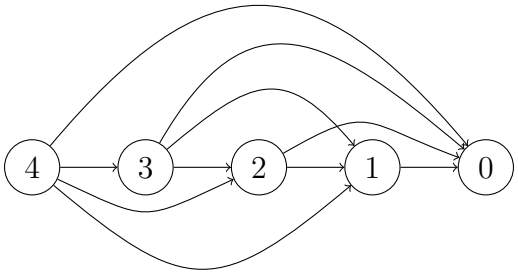


Figura 2.2: *Grafo simplificado que muestra que guardar los resultados de los subproblemas reduce las llamadas que hace el algoritmo recursivo por fuerza bruta.*

Evidentemente, muchas veces se requiere guardar no solo los valores de los subproblemas resueltos, sino también las decisiones tomadas en sus soluciones para obtener ese resultado. Normalmente, además del resultado al problema se necesita saber la solución que lo da, y para ello es necesario guardar las soluciones de los subproblemas que componen la solución óptima. En el problema de la barra, el resultado al problema es el beneficio óptimo conseguido (por ejemplo, para $n = 5$, $b_{op} = 13$) y la solución al problema es la combinación de segmentos de la barra que resultan en ese beneficio óptimo (para $n = 5$, la combinación sería $5 = 2 + 3$).

2.3.2. Algoritmos voraces

La subsección anterior se ha centrado en la programación dinámica, que es la más eficaz a la hora de afrontar todo tipo de problemas de optimización. Sin embargo, hay muchos casos en que no son la opción más eficiente ni tampoco la mejor. Dada su estructura de

resolución de subproblemas para construir la solución final, pasan por todas las opciones disponibles antes de elegir la más adecuada. Este procedimiento puede ser demasiado exhaustivo para resolver algunos problemas en los que simplemente elegir la opción que parece mejor en ese momento es suficiente, y así resolver el subproblema que se obtiene de esa decisión, sin resolver todos los posibles subproblemas relacionados a este. Los algoritmos que siguen esta estrategia se llaman **algoritmos voraces**, porque toman una decisión *voraz*, sin "pensar" en las consecuencias que pueden aparecer a causa de esa decisión (como se explica en el capítulo 16 de [8]).

Puede parecer que esta simplicidad implícita de los algoritmos voraces es siempre una ventaja frente a los dinámicos y que se podrían sustituir por voraces en todos los casos. Sin embargo, esto no es así porque los algoritmos voraces no devuelven siempre la solución óptima. Los voraces obtienen la solución después de tomar una serie de decisiones y su criterio para tomarlas es elegir la opción que parece más adecuada en cada momento. A causa de esa naturaleza heurística que tienen, no siempre tomarán las decisiones que dan el mejor resultado a largo plazo. Por ello, hay algunos problemas en los que esta estrategia funcionará mejor que en otros.

Los pasos a seguir para diseñar un algoritmo voraz suelen ser los siguientes:

1. Plantear el problema de optimización como uno en que a cada paso se toma una decisión y se resuelve tan solo el subproblema que queda.
2. Demostrar que se puede encontrar una solución óptima al problema tomando decisiones voraces, de forma que es seguro aplicar esta estrategia.
3. Demostrar que el problema tiene estructura óptima de forma que, si se resuelven los subproblemas que van quedando tras cada decisión, se pueden combinar las soluciones a los subproblemas para conseguir la solución al problema original.

Tal y como sucedía con la programación dinámica, hay dos puntos clave que se deben cumplir para que puedan aplicarse los voraces. Como se ha mencionado anteriormente, una solución basada en programación dinámica es posible en cualquier problema en que los algoritmos voraces funcionen, mientras que lo contrario no se cumple. Para que los voraces funcionen se deben cumplir la condición de elección voraz y que el problema tenga estructura óptima.

- **Elección voraz**. En esta propiedad es donde los algoritmos voraces difieren de la programación dinámica. Cuando se emplea la programación dinámica, las decisiones que se toman entre paso y paso van dictadas por las soluciones de los subproblemas que se van resolviendo. Además, normalmente se empieza con los subproblemas más pequeños, y a medida que se van resolviendo se va construyendo la solución hasta llegar a los más grandes, es decir, usando orden invertido (a menos que se use memoización, en ese caso se usaría orden natural, como se explica en la sección 2.3).

  Un algoritmo voraz hace todo lo contrario, simplemente toma la decisión que parece mejor en ese momento. Esa decisión puede estar influenciada por las decisiones tomadas en el pasado pero nunca estará condicionada por lo que pueda pasar en el futuro. Es decir, mientras que en programación dinámica se resuelven subproblemas antes de tomar la primera decisión, lo primero que hace hará un algoritmo voraz es tomar una decisión y después resolver los subproblemas que salen de esa elección. Asimismo, la estructura con la que resuelven también es la opuesta a la más habitual

de los dinámicos, pues suelen seguir el orden natural. De esta forma, empiezan con el subproblema más grande y a medida que se van tomando decisiones se va reduciendo el tamaño del problema, mejorando así la probabilidad de que la próxima decisión sea la mejor.

- **Estructura óptima.** Como ya se había explicado en la sección anterior, se dice que un problema tiene estructura óptima si la solución que lo resuelve está compuesta por soluciones a sus subproblemas. Esta pieza que es clave para aplicar programación dinámica, también es necesaria para los voraces.

Los algoritmos voraces presentan algunas ventajas sobre la programación dinámica que puede ser útil en ciertas ocasiones. Por un lado, los algoritmos voraces son habitualmente más rápidos porque resuelven menos subproblemas. Además, son iterativos por naturaleza así que tendrán una mejor complejidad computacional y no requieren más espacio de memoria que el que necesita el problema original. Los dinámicos son recursivos por naturaleza (aunque es cierto que se pueden adaptar para ser iterativos), lo que significa que necesitan más espacio de memoria para almacenar los resultados que se vayan obteniendo. Sin embargo, no hay que olvidar que los voraces no siempre encontrarán la mejor solución, mientras que con los dinámicos se garantiza que se encontrará la mejor. Para entender mejor la diferencia entre los dinámicos y los voraces, se propone un ejemplo, concretamente, el clásico problema de la mochila, pero en su variante fraccional (también denominada "contínua").

El problema de la mochila tiene el siguiente planteamiento: se dispone una mochila con capacidad W y una mesa llena de objetos, cada uno con peso ω_i y valor por peso v_i. El objetivo es llenar la mochila consiguiendo transportar el mayor valor total posible. Es decir, si hay n objetos,

$$\begin{aligned} &\text{maximizar} && \sum_{i}^{n} \omega_i \, v_i \\ &\text{sujeto a} && \sum_{i}^{n} \omega_i \leq W. \end{aligned} \tag{2.8}$$

A este en concreto se le conoce como "el problema de la mochila 0-1", por que los objetos son indivisibles y se resuelve mediante programación dinámica. Sin embargo, hay otra variante que considera los objetos divisibles, de forma que se pueden coger "partes" de cada uno. A este segundo problema se le conoce como "el problema de la mochila fraccional" y se puede resolver usando algoritmos voraces. Para ello, se puede modificar el problema 2.8 para que sea fraccional:

$$\begin{aligned} &\text{maximizar} && \sum_{i}^{n} x_i \, v_i \\ &\text{sujeto a} && \sum_{i}^{n} x_i \leq W. \\ & && 0 \leq x_i \leq \omega_i \qquad \text{para } i = 1, 2, \dots n \end{aligned} \tag{2.9}$$

La resolución de este problema tendría dos fases. Primero, se ordenan los objetos por su valor por peso v_i, de mayor a menor. Después, se van tomando los objetos de la lista siguiendo el orden de mayor v_i a menor. Cuando se llegue al punto en que el siguiente objeto sobrepasaría la capacidad máxima, se toma de él la cantidad que equivale a la diferencia entre la capacidad W y la suma de x_i calculada hasta ese momento. De esta forma, se resuelve el problema.

Si no se ordenan previamente los objetos, lo único que haría el algoritmo es buscar cuál tiene la mayor v_i en cada iteración, es decir, optaría por la elección voraz. El Código 2.4 muestra la resolución del problema si los objetos no están ordenados:

```
def knapsack_fractional(objects, max_w):
    """
    This function returns the maximum possible total value from the
   objects dictionary by searching with a greedy
    algorithm.
    :param objects: dictionary that contains each weight associated to
   its value/weight rate.
    :type objects: dict
    :param max_w: maximum capacity of the knapsack
    :type max_w: int
    :return: tuple with the total weight and total value
    """
    objects_w = list(objects.keys())
    objects_v = list(objects.values())
    sum_w, sum_v = 0, 0
    while True:
        max_v = max(objects_v)
        i_v = objects_v.index(max_v)
        w_i = objects_w[i_v]
        if sum_w + w_i < max_w:
            sum_v += max_v * w_i
            sum_w += w_i
            objects_v.pop(i_v)
            objects_w.pop(i_v)
        else:
            dif = max_w - sum_w
            sum_w += dif
            sum_v += max_v * dif
            break
    return sum_v, sum_w
```

Código 2.4: *Fragmento de código que resuelve el problema de la mochila fraccional tomando siempre la elección voraz.*

2.4. Complejidad computacional

Como ya se ha mencionado, el problema NSP es un problema "NP-duro". Esta clasificación hace referencia a su complejidad computacional, según la cuál pueden distinguirse varias clases de complejidad que forman conjuntos de problemas de decisión. El problema que nos concierne pertenece a la clase de problemas de tiempo polinómico no determinista (*Nondeterministic Polynomial time*, NP), que se define por ser aquellos que pueden ser resueltos en tiempo polinómico por una máquina de Turing no determinista. Para poder entender el problema NSP en su totalidad, es conveniente conocer qué significa que un problema sea NP-duro.

2.4.1. Clases P, NP y NP-completo

Los problemas de decisión que pueden resolverse en tiempo polinómico se dividen en tres conjuntos: P, NP y NP-C (estos últimos son los problemas NP-completos). Los problemas de las clase P (o problemas P) son aquellos que pueden ser resueltos en tiempo polinómico mediante una máquina de Turing determinista, más concretamente, en un tiempo $O(n^k)$ donde k es una constante y n es el tamaño del problema a resolver[8].

La clase NP es el conjunto de problemas que se puede verificar que pueden ser resueltos en tiempo polinómico mediante una máquina de Turing no determinista. La clase NP se diferencia de la clase P en que la solución que se propone para el problema se genera de forma no determinista y después un algoritmo determinista verifica si la solución al problema es válida. Dicho de otra forma, *si el problema puede ser resuelto en tiempo polinómico por una máquina de Turing determinista, se trata de un problema de la clase P, mientras que si es por una máquina no determinista, pertenece a la clase NP.* Cualquier problema en P está también en NP, dado que si está en P es porque se puede resolver en tiempo polinómico sin necesidad de verificarlo también en tiempo polinómico. Sin embargo, que un problema sea verificable en tiempo polinómico no significa que se pueda resolver en tiempo polinómico, es decir, que un problema esté en NP no implica que esté en P.

Finalmente, un problema es de la clase NP-C (o es NP-completo) si está en NP y es al menos tan difícil (o duro) de resolver como cualquier problema en NP. Esto significa que *si cualquier problema NP-completo puede ser resuelto en tiempo polinómico, entonces todos los problemas en NP pueden ser resueltos en tiempo polinómico.* Todavía a día de hoy no se ha demostrado que los problemas NP-completos sean resolubles en tiempo polinómico, aunque la mayoría de la comunidad científica concuerda en que no lo son.

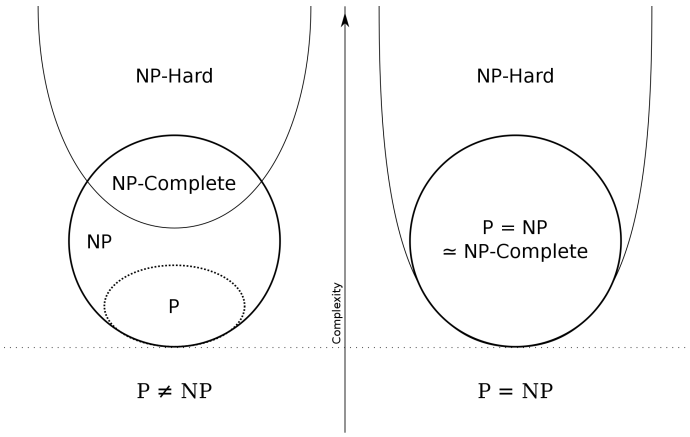


Figura 2.3: *Diagrama de Euler de los conjuntos P, NP, NP-duro y NP-completo. Asumiendo que $P \neq NP$, se ha demostrado la existencia de problemas contenidos en NP que no están contenidos en P o NP-completo. Si se llegase a encontrar algún problema NP-completo resoluble en tiempo polinómico, entonces, por corolario, se cumpliría $P = NP$.*

La teoría de complejidad NP no es aplicable directamente a los problemas de optimización, por este motivo es importante no confundir los problemas de decisión con los problemas de optimización. Un problema de optimización busca encontrar una solución óptima a la que va asociada un valor óptimo mientras que un problema de decisión obtiene

como solución un valor booleano, sí o no. Sin embargo, siempre es posible obtener una relación directa entre un problema de optimización y su respectivo problema de decisión si se impone un límite al problema de optimización. Un ejemplo de esto sería un problema de optimización para encontrar el camino más corto entre los vértices u y v de un grafo $G = (V, E)$. Bastaría con añadir un límite k de aristas en las que conseguir la solución, que dejaría el problema de decisión: ¿Es posible encontrar el camino más corto entre los vértices u y v que tenga como máximo k aristas?

De esta manera, es posible caracterizar la complejidad de un problema de optimización conociendo la de su equivalente de decisión. Dicho de otra forma, si se puede demostrar que un problema de decisión es duro, entonces también que su problema de optimización relacionado lo es.

2.4.2. NP-duro

Como ya se puede intuir por la Figura 2.3, un problema de decisión M es NP-duro cuando cualquier problema N de NP es reducible en tiempo polinómico a M. Esto significa que si efectivamente se confirmase que $P \neq NP$, entonces no existiría ningún algoritmo capaz de resolver un problema NP-duro en tiempo polinómico.

Además, a pesar de que los problemas NP-duros cumplen con la condición de ser al menos tan difíciles como el problema más difícil de NP, al igual que lo hacen los problemas NP-completos, no cumplen con la condición de pertenecer a NP. Esto es así porque, pueden no ser decidibles en tiempo finito. Todos los problemas en NP deben ser decidibles en tiempo finito.

Un ejemplo de problema que no es decidible en tiempo finito es el problema de la parada, en el que se presupone un programa con un input y se hace la siguiente pregunta: ¿Parará alguna vez el programa? Este es un problema de decisión y es NP-duro. Como cualquier problema de NP se puede reducir en tiempo polinómico a un problema NP-duro y la clase NP-completo está contenida en NP, el "problema de satisfabilidad booleana" (que es NP-completo) puede ser reducido al problema de la parada si se transforma a una máquina de Turing que comprueba todos los combinaciones verdaderas hasta encontrar una que satisfaga la ecuación del problema. Cuando encuentre esa asignación de valores la máquina se detendrá, si no seguirá ejecutándose hasta el infinito. Esto hace que el problema de la parada, que es NP-duro, sea indecidible.

Capítulo 3

Planteamiento matemático del problema de planificación de turnos

Para solucionar los dos problemas propuestos en este trabajo (el de la cafetería y el de la clínica), se parte del problema NSP clásico y se adaptan las restricciones y la función objetivo para que se adecúen a cada ejemplo. En este trabajo se ha optado por plantear el problema como un problema de programación lineal en enteros (PLE), lo que significa que tanto la función objetivo como las restricciones serán expresiones lineales en sus variables y estas variables serán números enteros. La idea es que la función objetivo del PLE maximice puntos que indican la preferencia de cada empleado (o lo bien que encaja cada empleado) por cada turno, teniendo en cuenta todo tipo de restricciones que tenga cada empleado para cada turno.

3.1. Definición y función objetivo

La manera más sencilla de plantear este problema es mediante un problema de decisión donde las variables a buscar $X_{e,d,s}$ representan si un empleado e trabaja el día d en cierto turno s o no [4]. Estas variables se usan para formular la función objetivo del problema, que será una expresión lineal de las variables $X_{e,d,s}$ y de una expresión de los coeficientes $P_{e,d,s}$ y $S_{e,s}$. El primero indica la "preferencia" que tiene un empleado sobre un turno de trabajo y el segundo indica la experiencia o la "antigüedad" del empleado en el trabajo que va a desempeñar. Definiendo el problema en forma estándar,

$$\begin{aligned} \text{maximizar} \quad & f(S,P)^T x \\ \text{sujeto a} \quad & T\,x \leq b \\ & x \geq 0. \end{aligned} \tag{3.1}$$

donde $f(S,P)$ y b son vectores de dimensión n y T es una matriz de dimensión $n \times n$.

n es un parámetro que depende de los cuatro conjuntos que constituyen este problema: el conjunto de empleados E, el de días D, el de roles R y el de bloques de turnos B, para así tener $n = |D| \cdot |B| \cdot |R| \cdot |E|$. Las variables de decisión del problema tendrán un subíndice por cada uno de estos conjuntos que serán los grados de libertad de cada variable.

- **Empleados** E. Este conjunto contiene una cantidad de empleados e y será de la forma $E = \{1, 2, 3 \dots, e\}$. Cada número es una etiqueta que representa a un empleado, es decir, 1 equivale a empleado_1, 2 a empleado_2, etc.

- **Días** D. Es el conjunto de días que conforman un ciclo. Habitualmente, los ciclos serán de 5 o 7 días, pero podrían ser mucho más extensos, como de un mes. Este conjunto tendrá la forma $D = \{1, 2, 3 \dots\}$ y cada número representa un día de cada ciclo (por ejemplo, si el ciclo es de una semana, entonces serían 1 = Lunes, 2 = Martes, etc.).

- **Roles** R. Este conjunto representa los diferentes oficios que cada trabajador desempeña dentro de un negocio. Por ejemplo, en una cafetería habrá camareros, cocineros y ayudantes de cocina, encargados de barra, etc. que pueden ir rotando de un puesto a otro.

  Además del caso anterior, es cada vez más habitual que los puestos de trabajo estén cubiertos por trabajadores altamente especializados. Por eso, este conjunto podría extenderse en significado a "obligaciones" en lugar de oficios.

  El conjunto de roles, como los anteriores, se define mediante números que representan los diferentes puestos de trabajo o tareas de un grupo de trabajadores.

- **Bloques de turnos** B. Este conjunto está compuesto de bloques de trabajo que están definidos por la unión de los intervalos de horas que van desde i hasta j, $\{\bigcup_{1 \leq i \leq f}[i, j]\}$. Los intervalos de horas disponibles son los que pertenecen al conjunto de horas $H = \{1, 2, 3 \dots\}$. En el conjunto H cada número representa un intervalo de horas de un día laboral. Por ejemplo, volviendo al ejemplo de la cafetería, la hora para entrar a trabajar puede ser las 6 : 00 y la hora de salida más tardía las 23 : 00. Entonces, 1 = 6 : 00 a 7 : 00, 2 = 7 : 00 a 8 : 00,... 17 = 22 : 00 a 23 : 00.

  Por lo tanto, el bloque $[1, 17]$ corresponde a un turno que cubre el total de las 17 horas disponibles en un día mientras que el bloque $[8, 8]$ corresponde al turno de 1 hora de la 13 : 00 a las 14 : 00. Si hay $|H| = n$ horas en el conjunto H, entonces hay $\binom{n+1}{2}$ donde n es la cantidad de elementos en el conjunto que son tomados en grupos de r, como $r = 2$, $\binom{n+r-1}{r} = \binom{n+1}{2}$.

A través de estos cuatro conjuntos se puede definir la función objetivo a maximizar, que dependerá de dos coeficientes: antigüedad y preferencia.

- **Antigüedad** ($S_{r,e}$). Para cada $e \in E$, $S_{r,e}$ expresa la antigüedad o *seniority* de cada empleado desempeñando el rol r. Cuanto mayor sea la experiencia de un empleado en un rol, mayor será su valor de antigüedad en ese rol. Los coeficientes de antigüedad son números enteros positivos, porque son iguales a la cantidad de años que el trabajador tenga de experiencia en el rol.

- **Preferencia** ($P_{d,b,r,e}$). Este coeficiente expresa la preferencia (o disposición) que tiene un empleado para trabajar en el rol r, el día d, en el bloque b. Dependiendo del valor que tenga, el valor de preferencia se interpreta de las siguientes formas:

  - $P_{d,b,r,e} \geq 0$, e está disponible para ese $[d, b, r]$ y cuanto mayor sea, mayor es su disposición a ese turno, es decir, más conveniente le es ese turno.

  - $P_{d,b,r,e} = 0$, e está disponible para ese $[d, b, r]$ pero su disposición es mínima, es decir, no desea trabajar en ese turno.

- $P_{d,b,r,e} \leq 0$, e no está disponible para ese turno. Este último puede interpretarse como una restricción dura o como una restricción blanda, dependiendo de como se defina la disponibilidad de los empleados. Si se considera que, aunque un empleado no este disponible se le puede aun así asignar el turno en caso de verdadera necesidad, en ese caso se estaría hablando de una restricción blanda. Sin embargo, si asignar un turno con preferencia negativa a un empleado está absolutamente prohibido, entonces sería una restricción dura.

La función objetivo se basará en una función que dependerá de $S_{r,e}$ y $P_{d,b,r,e}$ que representa la conveniencia de asignar un turno específico a un empleado e y también de una variable binaria $X_{d,b,r,e}$ que describe si el empleado trabaja o no el día d, en el rol r, en el bloque b. Sea $X_{d,b,r,e}$ la siguiente variable dinámica:

$$X_{d,b,r,e} = \begin{cases} 1 & \text{si el empleado e, trabaja el día d, en el rol r en el bloque b,} \\ 0 & \text{en caso contrario.} \end{cases} \tag{3.2}$$

Entonces, se puede definir la función objetivo como sigue,

$$F(X_{d,b,r,e}) = \sum_{d \in D} \sum_{b \in R} \sum_{r \in R} \sum_{e \in E} f(S_{r,e}, P_{d,b,r,e})\, X_{d,b,r,e}. \tag{3.3}$$

Para este ejemplo, se ha usado f como $f(S, P) = S_{r,e} \cdot P_{d,b,r,e}$.

La elección de f como una multiplicación de los dos coeficientes está motivada por tener la mayor sencillez posible. Es por esto que esta función podría variarse para ver como cambian los resultados y para ver si mejora o empeora la conformidad de los trabajadores con la asignación de turnos calculada.

3.2. Restricciones del problema

A cada trabajador se le puede asignar como máximo un bloque en un rol por día. Es decir, un empleado e de la cafetería no puede trabajar de 9 : 00 a 12 : 00 como camarero y después de 17 : 00 a 20 : 00 como cocinero, porque se rompen las dos condiciones: dos bloques en un día y dos roles en un día. Así que un turno de un empleado debe ser un único bloque continuo de trabajo. De esta manera, se pueden definir las dos primeras restricciones:

$$X_{d,b,r,e} \in \{0, 1\} \qquad \forall\, e \in E, d \in D, r \in R, b \in B \tag{3.4}$$

$$\sum_{r \in R} \sum_{b \in B} X_{d,b,r,e} \leq 1 \qquad \forall\, e \in E, d \in D \tag{3.5}$$

La primera de las dos indica que las variables de optimización serán binarias, mientras que la segunda indica que para todo empleado y cualquier día, la suma de turnos asignados sobre todos los roles y todos los bloques debe ser como máximo igual a 1, es decir, solo puede tener asignado como máximo un bloque en un rol por día.

Se pueden poner restricciones sobre la cantidad de horas que un empleado puede trabajar. Para ello, se debe conocer cuantas horas tiene el bloque que se le puede asignar. Para medir esto, se define la **longitud de bloque** como $len(b) = h_2 - h_1 + 1$ si $b = [h_1, h_2]$

(se suma 1 porque no puede haber bloques de longitud 0, porque eso significaría que no se trabaja en el turno de ese bloque).

Se define $MAXHR_{d,e}$ como el máximo de horas que un trabajador e desea trabajar el día d y $MINHR_{d,e}$ como el mínimo. Por lo tanto, para todo $e \in E$, $d \in D$ y $r \in R$

$$X_{d,b,r,e} = 0 \quad \forall\, b \in B \text{ donde } len(b) > MAXHR_{d,e} \tag{3.6}$$
$$X_{d,b,r,e} = 0 \quad \forall\, b \in B \text{ donde } 0 < len(b) < MINHR_{d,e} \tag{3.7}$$

Las ecuaciones 3.6 y 3.7 indican que la cantidad de horas que se asignen a un empleado e en un día estarán acotadas entre los valores de $MAXHR_{d,e}$ y $MINHR_{d,e}$. Además, tampoco se permite asignar una cantidad de horas negativa.

Se pueden generalizar las dos restricciones anteriores para el total de días del conjunto D, $|D|$. Así que, si se toma a $HOURS_e$ como el máximo de horas trabajadas para un trabajador e por ciclo $|D|$ (si el ciclo fuera de una semana, $|D| = 7$) y $DAYS_e$ como máximo de días trabajados por $|D|$:

$$\sum_{d\in D}\sum_{b\in B}\sum_{r\in R} len(b)\, X_{d,b,r,e} \leq HOURS_e \quad \forall\, e \in E \tag{3.8}$$
$$\sum_{d\in D}\sum_{b\in B}\sum_{r\in R} X_{d,b,r,e} \leq DAYS_e \quad \forall\, e \in E \tag{3.9}$$

Retomando una vez más el ejemplo de la cafetería, un camarero que trabaja 40 horas a la semana y 3 días como máximo tendría $HOURS_{camarero} = 40$ en la ecuación 3.8 y $DAYS_{camarero} = 3$ en 3.9.

Otra condición que se puede plantear es que un empleado que cierra un turno no pueda abrir al día siguiente, es decir, alguien con el último turno de un día no puede tener el primero del día siguiente). Por lo tanto,

$$\sum_{r\in R}\sum_{b\ni|H|} X_{d,b,r,e} + \sum_{r\in R}\sum_{b\ni 1} X_{d,b,r,e} \leq 1 \quad \forall\, e \in E, d \in D \tag{3.10}$$

Estas restricciones hacen referencia a los deseos de los empleados. Además de esos, se pueden definir restricciones que reflejen las preferencias de un negocio sobre sus trabajadores. $MAXNUM_{d,r,h}$ será el máximo de empleados que deben estar cubriendo el rol r, el día d a la hora h, mientras que $MINNUM_{d,r,h}$ será el mínimo. Por consiguiente, se tiene que

$$\sum_{b\ni h}\sum_{e\in E} X_{d,b,r,e} \leq MAXNUM_{d,h,r} \quad \forall\, d \in D, r \in R, h \in H, \tag{3.11}$$
$$\sum_{b\ni h}\sum_{e\in E} X_{d,b,r,e} \geq MINNUM_{d,h,r} \quad \forall\, d \in D, r \in R, h \in H. \tag{3.12}$$

De esta forma, si $MINNUM = 3$ para las 14 : 00, los martes y para el rol de camareros, significa que a esa hora deberá haber como mínimo 3 camareros trabajando. La misma lógica se aplica a $MAXNUM$.

El PLE deberá maximizar la función objetivo 3.3 para que cumpla con las 9 restricciones. En principio se tomarán todas estas incuaciones como restricciones duras aunque como se verá más adelante, alguna de ellas tiene cierto margen para pasar de ser dura a blanda, cambiando así el resultado de la función objetivo y finalmente la solución final.

Capítulo 4

Desarrollo y solución

Para resolver y buscar la solución óptima a los dos casos de estudio de este trabajo usando el modelo matemático de la sección anterior, se ha optado por emplear todo el potencial del lenguaje de programación por excelencia para resolución de problemas de optimización y desarrollo de aprendizaje automático, *Python* [5]. La planificación y la forma del programa que calcula la asignación de turnos, ha ido cambiando a lo largo de varios meses hasta concluir finalmente en una estructura que consta de tres clases; una clase principal que contiene el código de resolución principal (`ShiftSolver`) y dos clases auxiliares (`Employee` y `Preference`) que manipulan y preparan los datos de entrada para "facilitarle el trabajo"a la clase principal.

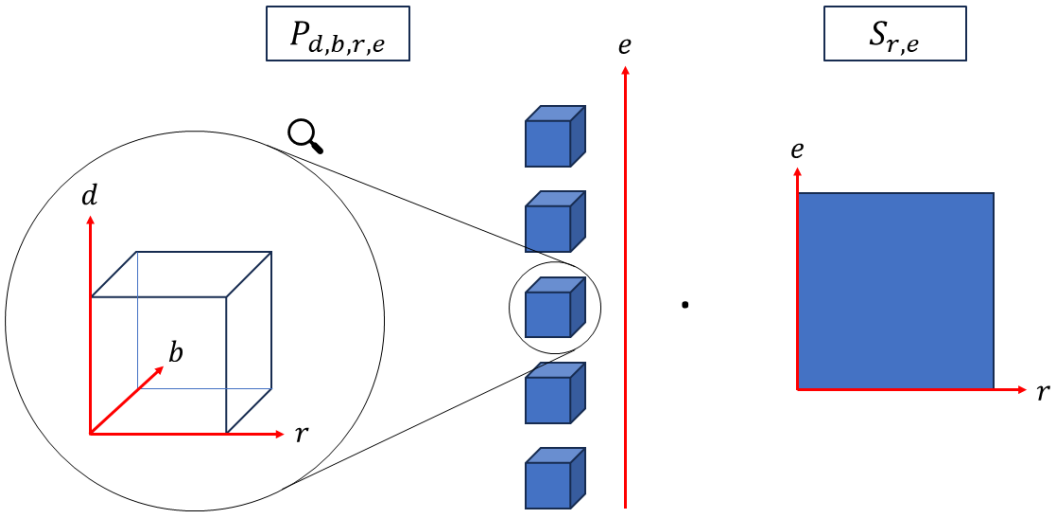


Figura 4.1: *La función* `solve()` *calcula* $f(S, P)$ *mediante la multiplicación escalar entre* $P_{d,b,r,e}$ *y* $S_{r,e}$. *El algoritmo voraz escogerá los valores máximos del tensor resultante, como se explica en el paso 3.*

La estructura general del programa es la siguiente:

1. `ShiftSolver` recibe los datos de la entrada y utiliza a `Employee` para parsear y extraer los datos de cada empleado. De este proceso saldrá una lista de objetos `employee` listos para ser usados.

2. A continuación `ShiftSolver` crea las dos matrices que se van a usar para el desarrollo para la Función 3.3, una matriz bidimensional $S_{r,e}$ formada por los valores de antigüedad de cada empleado y un tensor de cuatro dimensiones $P_{d,b,r,e}$ que contiene los valores de preferencia de cada empleado sobre los diferentes bloques. Para generar $P_{d,b,r,e}$, `ShiftSolver` llama a la clase `Preference` que utiliza los datos extraídos por `Employee`.

3. Una vez estén listos $S_{r,e}$ y $P_{d,b,r,e}$, el programa ya tiene toda la información necesaria para resolver el PLE. Este puede ser resuelto con la función `solve()` o con la función `pulp_solve()`. `solve()` es una implementación voraz que toma el valor máximo de $f(S_{r,e}, P_{d,b,r,e})$ (que se calcula tal y como se muestra en la Figura 4.1) en cada iteración y va construyendo la solución rellenando los turnos y comprobando que se cumplan las condiciones que se han detallado en el capítulo anterior. `pulp_solve()` es una implementación mediante la programación lineal que se basa en la librería *PuLP* [10]. Este permite usar programas externos "como cerebro" para los cálculos y así mejorar el rendimiento de la función de resolución básica o añadir o modificar restricciones con más facilidad.

A continuación, se detallan las características y funcionalidades de las tres clases que conforman el programa.

4.1. Clase auxiliar `Employee`

La clase `Employee` se ha diseñado como la encargada del tratamiento de los datos de entrada, para así dejarlos preparados para la clase `ShiftSolver`. Como ya se ha mencionado en la introducción de esta sección, finalmente se ha optado por separar en dos clases el tratamiento de estos datos (por cuestiones de orden y estructura, principalmente).

En la versión actual, se pueden introducir los datos de dos formas: escribiéndolos en un documento de texto sin formato, con extensión de archivo *txt*; o mediante un documento con datos separados por coma, con extensión *csv* (por comodidad, el programa reconocerá como separador de archivos csv tanto la coma "," como el punto y coma ";", ya que algunos de los programas de edición de hojas de cálculo más usados emplean el punto y coma en lugar de la coma). En ambos casos los datos van separados por columnas y filas, de forma que cada fila del documento representa a un empleado y contiene toda la información relevante a ese empleado separada en columnas. Cada una de estas columnas contiene información como el máximo de horas que puede trabajar en un día ($MAXHR$), la cantidad de días que puede trabajar en un ciclo ($DAYS$) o su disponibilidad en cada día del ciclo. En la Tabla 4.1 se muestra en detalle el formato del documento de datos de entrada.

Normalmente, "antigüedad" será más de una columna porque lo más habitual es que haya más de un rol (o tarea) que asignar. Por ejemplo, en el caso de la cafetería habría tres columnas y en cada una se especificaría la antigüedad que tiene el trabajador en ese puesto. Si el trabajador en cuestión nunca a trabajado en ese rol se pondrá 0.

En la columna de "disponibilidad" se indica la franja horaria en la que el trabajador

Empleado	$MAXHR$	$HOURS$	$DAYS$	**Antigüedad**	**Disponibilidad**
Empleado 1	7	35	5	6	9:00-18:00 . 8:00-16:00 ...
Empleado 2	8	32	4	2	. . 14:00-21:00 ...

Tabla 4.1: *Estructura que tienen los datos de entrada. Es muy importante respetar este formato ya que en caso de que no se cumpla, el programa se detendrá y devolverá una excepción indicando que el documento no es válido.*

está disponible para que se le puedan asignar turnos. Las franjas horarias se escriben separadas con un espacio unas de otras y se interpretan en el mismo orden en que están ordenados los días del ciclo. Es decir, si el ciclo es de siete días, una semana, la primera franja será la correspondiente al lunes, la segunda al martes, y así sucesivamente. Para indicar que un trabajador no está disponible un día, se escribe un punto en lugar de una franja, este también deberá tener una separación de un espacio a sus lados.

Si los datos de entrada se entregan en el formato correcto, la clase principal `ShiftSolver` recibirá el documento como entrada y lo separará en filas. Cada una de estas filas será la entrada de `Employee` y este se encargará de interpretar los datos que contiene cada fila. De esta forma, `Employee` creará un objeto de su misma clase con cada fila que se le de, es decir, un objeto `Employee` por cada empleado. Cada uno de estos objetos guardará los datos del empleado al que representa en sus atributos y estarán procesados para que las clases `Preference` y `ShiftSolver` puedan usarlos. En total, cada objeto constará de cinco atributos, como se puede ver en el diagrama de la Figura 4.2.

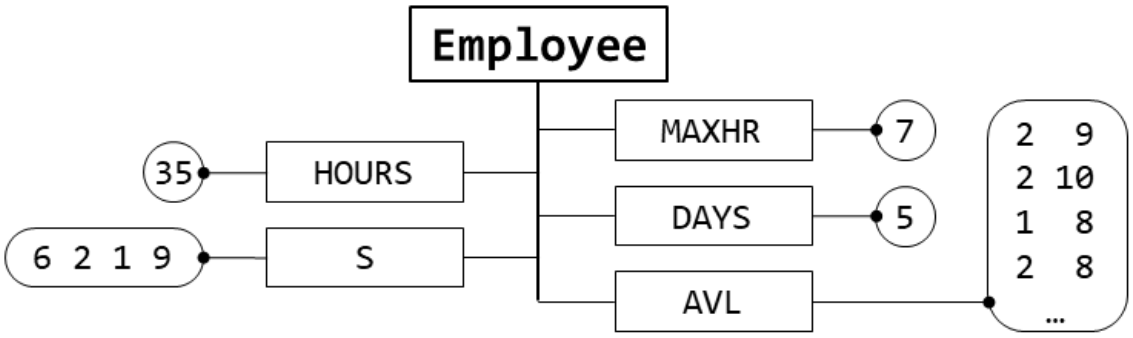


Figura 4.2: *Cada empleado tendrá su propio objeto de la clase* `Employee` *que guardará sus datos tal y como se muestra en la imagen.*

- **.MAXHR**. Este atributo contiene el valor de la segunda columna, el máximo de horas que un trabajador podrá trabajar en un día. En esta versión se ha decidido que la cantidad de horas del turno sea $MAXHR$ o menos, pero nunca más, para así facilitar su implementación.
- **.HOURS**. En este atributo se guarda la cantidad máxima de horas que podrá trabajar en un ciclo.
- **.DAYS**. Este contiene el máximo de días que puede trabajar a la semana. En caso de que no se cumpla $MAXHR = DAYS \cdot HOURS$, saltará una excepción indicándolo.
- **.S**. Este atributo es un array de la librería *Numpy* que contiene la antigüedad del empleado en cada rol, en orden. Si entre los valores de antigüedad se encuentra el

0 porque el empleado no tiene experiencia en esa tarea, se sustituirá por el valor infinito de la librería Numpy. Esto se hace para evitar futuras indeterminaciones de $0 \cdot \infty$ (ver último párrafo de la Sección 4.2).

- **.AVL**. Finalmente, el atributo AVL es un array de dos dimensiones de tamaño $|D| \times 2$ donde $|D|$ es la cantidad de días de un ciclo. Cada fila de la matriz que guarda tiene la hora de inicio y la de finalización de la franja horaria en la que el trabajador está disponible, en cada día. Si el trabajador no se encuentra disponible en uno o varios días del ciclo, las filas que corresponden a esos días se dejan con valor nulo.

Una vez que todas las filas del documento de datos hayan sido analizadas y parseadas por `Employee`, el siguiente paso es utilizar los datos que han sido recogidos para crear las estructuras de datos que se van a utilizar en la resolución del problema. Uno de ellos es el tensor cuatridimensional de valores de preferencia, del que se va a encargar la segunda clase auxiliar `Preference`.

4.2. Clase auxiliar `Preference`

`Preference` cumple con la tarea de crear el tensor de cuatro dimensiones que contiene los valores de preferencia de cada empleado en todas las combinaciones de turnos posibles. Los cuatro ejes de este tensor son los días, los bloques, los roles y los empleados por lo que forman un espacio de cuatro dimensiones con las coordenadas d, b, r y e. El tensor contiene un total de $|D| \cdot |B| \cdot |R| \cdot |E|$ valores o combinaciones posibles y cada uno de estos valores se calcula usando los datos de cada empleado extraídos por `Employee`. Por lo tanto, cada posición en el espacio de cuatro dimensiones del tensor contiene una combinación única.

Este tensor se usa junto con la matriz $S_{r,e}$ formada por los valores de antigüedad de los empleados en cada rol para calcular la Función Objetivo 3.3, dado que la función $f(S, P)$ se ha definido como la multiplicación escalar de estos dos. En la sección 4.3 se explicará más en detalle.

A nivel de estructura, esta clase toma como entrada los datos de los empleados y los bloques de turnos (un array de todos los que hay disponibles, que se calcula sabiendo cuales son todas las longitudes de bloque que están permitidas, por ejemplo 5h, 6h, 8h...) y los utiliza para generar el tensor, que será un array de cuatro dimensiones. Para crear el tensor, esta clase comprueba cuatro condiciones, dos de las cuales son las restricciones de las Ecuaciones 3.6 y 3.7. Primero se crea un tensor vacío de las dimensiones correctas y después se va rellenando de valores empezando desde el eje e hasta el eje b haciendo las siguientes comprobaciones:

1. Se empieza considerando el eje de empleados (i) y de cada empleado el eje días (j). El eje de días tiene la misma dimensión y el mismo orden que las filas de la matriz de disponibilidad que guarda el objeto `employee` (el atributo .AVL), así que la primera comprobación es ver si la franja horaria $employees[i]$.AVL$[j]$ es nula (igual a $[0, 0]$). Si es que sí, significa que el empleado no se encuentra disponible ese día así que se le asigna un valor de penalización en los valores de ese empleado en ese día en todos los roles y en todos los bloques. En cambio, si no es nula, se continúa con la siguiente condición.

   Como ya se adelantó en la sección 3.1, el comportamiento del programa ante el

valor de penalización que indica que un trabajador no está disponible, marcará las decisiones que se tomen para la solución final y alterará considerablemente el valor objetivo. Esto se verá con más claridad en el apartado de `ShiftSolver`.

2. Una vez que se ha verificado que la franja horaria de la iteración actual no es nula, lo siguiente será iterar sobre el eje de bloques para buscar bloques compatibles. En cada bloque por el que pasa el programa se comprueba si la longitud del bloque es mayor o menor que $MAXHR$. Si la longitud es mayor, entonces el bloque se descarta insertando un valor de penalización en el tensor en el lugar que se corresponde con este bloque (para este empleado i, en el día j en todos los roles), mientras que si es menor o igual a $MAXHR$ entonces se pasa al siguiente paso.

   Como ya se ha mencionado previamente en esta sección, para simplificar la implementación se ha optado por usar solo $MAXHR$ sin considerar $MINHR$, de forma que el turno asignado no puede tener más horas que $MAXHR$ pero no tiene cota inferior.

3. Después de ver si el bloque tiene la longitud adecuada, la última comprobación que hay que hacer antes de poder asignarle un valor de preferencia (o conveniencia) es ver que el bloque esté dentro de los límites de la franja horaria del día j. Si el bloque empieza antes o termina después de los valores que haya proporcionado el empleado en su franja horaria, entonces el bloque se penaliza y se descarta.

4. Si el bloque ha pasado por todas las condiciones anteriores sin problemas, entonces sí, se le asigna un valor de preferencia. Este valor se calcula de la siguiente manera,

$$P_{d,b,r,e} = 10 \left(\frac{len(b)}{MAXHR_e} \right). \tag{4.1}$$

   De esta forma, el máximo valor posible es 10, que se alcanza en el caso en que el bloque tiene una longitud de exactamente $MAXHR$.

Después de considerar todo el eje de empleados, el tensor de preferencia estará completo y tendrá todos sus valores asignados, ya sean de penalización (probabilidad nula) o de probabilidad no nula. El valor de penalización que se inserta en las posiciones que no son viables es $-\infty$ (de la librería Numpy), para asegurarse de que los bloques con ese valor nunca sean elegidos y sean verdaderamente inaccesibles. Este es el motivo por el cual se sustituyen los valores de la antigüedad que son nulos por ∞ cuando se están guardando en el atributo .S de `Employee`. Si no se hiciera, al hacer la multiplicación $P_{d,b,r,e} \times S_{r,e}$ habría indeterminaciones en todas las posiciones donde hay un bloque en un rol en el que empleado no tiene experiencia y además no está disponible. El valor lógico para este caso en particular sería $-\infty$ pero en vez de eso habría NaN. Poniendo ∞ en lugar de 0 se consigue que la multiplicación de los dos valores devuelva $-\infty$, solucionando el problema.

Por desgracia esto causa otro problema, el caso en el que hay un bloque disponible con probabilidad no nula n para un rol en el que el empleado no tiene experiencia. Esto normalmente no causaría ningún problema dado que devolvería un valor nulo de $f(S, P)$, $n \cdot 0$. Sin embargo, como se ha hecho el cambio de 0 a ∞ en todos los valores nulos de S, ahora el valor resultante de la multiplicación es $n \cdot \infty = \infty$. La mejor solución a esto es cambiar uno a uno todos los ∞ que vayan apareciendo, tal y como se explica en la sección siguiente.

4.3. Clase principal `ShiftSolver`

`ShiftSolver` es la pieza central de este trabajo, contiene el código que resuelve el problema y devuelve la asignación de turnos. En el mismo archivo que contiene a `ShiftSolver`, hay ocho funciones que cumplen todo tipo de tareas que van desde verificar algunas de las restricciones (ver Sección 3.2) hasta guardar la solución en un archivo *csv* de forma que sea legible. Este apartado tiene como objetivo explicar el funcionamiento de los métodos de resolución en profundidad.

La clase `ShiftSolver` se inicializa con los siguientes parámetros:

- `file[str]` El archivo *txt* o *csv* de entrada que contiene los datos de los empleados.
- `tc[char]` Letra que indica la extensión del archivo `file`. Acepta 't' para *txt*, 'c' para *csv* y 'cex' para *csv* creado por Excel (este programa en particular utiliza ';' como separador de los archivos *csv* en lugar de la habitual ',').
- `emp_amount[int]`=`DEFAULT_E` Cantidad de empleados.
- `n_days[int]`=`DEFAULT_DAYS` Cantidad de días del ciclo.
- `n_rolls[int]`=`DEFAULT_R` Cantidad de roles.
- `shift_lengths[numpy.ndarray]`=`None` Array que contiene las longitudes de bloque permitidas, por ejemplo $[6, 7, 8]$.
- `shift_start_end[numpy.ndarray]`=`None` Array que contiene todas los intervalos de horas que forman el conjunto H (ver Sección 3.1).
- `minnum[numpy.ndarray]`=`MINNUM` Array donde cada valor es la cantidad mínima de trabajadores que debe haber en cada turno de cada rol.
- `maxnum[numpy.ndarray]`=`MAXNUM` Array donde cada valor es la cantidad mínima de trabajadores que debe haber en cada turno de cada rol.

Como se puede ver, la mayoría de parámetros se inicializan con un valor predeterminado (el del ejemplo de la cafetería).

Después de guardar todas las especificaciones del problema y generar la lista de objetos `employee` además de los tensores $S_{r,e}$ y $P_{d,b,r,e}$, el siguiente paso es resolver el problema con una de las dos funciones disponibles.

4.3.1. Resolución mediante `solve()`

La función `solve()` es la que se usa por defecto para resolver problemas de programación lineal en este trabajo. Esta función es una implementación voraz que escoge la combinación con mayor valor de $f(S, P)$ en cada iteración. Estos son los pasos que sigue la función:

1) En primer lugar calcula el tensor T que representa a $f(S, P)$. Este es el resultante de la multiplicación de $P_{d,b,r,e} \cdot S_{r,e}$. La idea detrás de este tensor es similar al cálculo del valor por peso en el problema de la mochila, que se usa para elegir que objetos meter en la mochila para poder completar el peso con el máximo valor posible. Los valores de T indican cómo de adecuado es el turno que se encuentra en cada posición del tensor.

Los turnos que se vayan asignando se guardan en un diccionario de Python llamado `assigned_shifts`, de manera que cada *item* del diccionario tiene como clave el identificador del empleado y como valor una lista de los turnos que se le hayan asignado. El identificador tendrá este formato: `enp`i, donde i será un número entero de $[0, enp\_amount - 1]$.

2) Después de calcular el tensor T, se abre un bucle en el que en cada iteración se busca el valor máximo y se obtienen las coordenadas de su posición en las cuatro dimensiones (que serán los índices d, b, r y e del array T). Este bucle sólo se detendrá si se han asignado todos los turnos que se pueden asignar a los empleados (hay que recordar que no puede asignárseles más de uno por día, restricción 3.5) y/o si el valor máximo que se encuentra en la matriz no es mayor que $-\infty$.

Una vez se tienen los índices del valor máximo, antes de comprobar si el bloque de esa posición cumple con las diversas restricciones, se verifica si es ∞. Hay que recordar que, como ya se mencionó en la sección anterior, la solución al problema de las indeterminaciones $0 \cdot \infty$ provocaba tener posiciones con valor ∞ cuando se trataba de un turno para un rol con $S > 0$ pero en un bloque para el cual el empleado no se encontraba disponible. Por esto mismo se hace está comprobación, para evitar falsos positivos. Lo único que se hace en este caso es cambiar el signo del infinito y continuar con el siguiente valor (con este *modus operandi*, el programa primero "barre" todos los falsos positivos antes de empezar con los valores reales).

3) Si el valor máximo encontrado no es ∞, entonces se procede con las restricciones. Si el turno de esta posición en T cumple con todas, entonces será asignado al empleado que le corresponde (como una de las cuatro dimensiones es el eje de empleados, el empleado será el que indique el índice e).

En la función `solve()`, las restricciones se comprueban con funciones específicas creadas para ello. Como se explicará más adelante, en la función `pulp_solve()` que hace uso de la librería PuLP, se utiliza un método distinto para añadir y verificar las restricciones.

En primer lugar, se comprueba si el empleado e tiene asignado algún turno el día d. Para ello, se accede a los turnos asignados a e del diccionario `assigned_shifts` y se busca un turno que coincida con el día d, como se ve en el Código 4.1

```
1  def e_free_for_day_d(e0, d0, shift_dict):
2      """
3      This function checks if a given employee is free for a given day, as
       each employee can only work in one shift per
4      day.
5      :param e0: index of the employee to check for.
6      :type e0: numpy.ndarray
7      :param d0: index of the day to check for.
8      :type d0: numpy.ndarray
9      :param shift_dict: Dictionary containing all the assigned shifts for
       each employee.
10     :type shift_dict: dict
11     :return: True if e0 is has no assigned shifts day d0, False
      otherwise.
12     :rtype: bool
13     """
14     e_shifts = shift_dict[f"enp{e0}"]
```

```
15      for d, b, r in e_shifts:
16          if d == d0:
17              return False
18      else:
19          return True
```

Código 4.1: *Fragmento de código que contiene la función que comprueba la Restricción 3.5*

A continuación, como se muestra en el Código 4.2, se comprueba si al empleado e ya se le han asignado todos los turnos. Es decir, si su valor $DAYS = 4$, al empleado solo se le podrán asignar cuatro turnos, uno por día. La función que hace esta comprobación simplemente cuenta cuántos turnos se le han asignado hasta ese momento.

```
1  def e_days_filled(e0, employee, shift_dict):
2      """
3      This function checks if a given employee e0 has all of its days
       already occupied.
4      :param e0: index of the employee to check for.
5      :type e0: numpy.ndarray
6      :param employee: Employee type object of index e0 in the list of
       employees.
7      :type employee: Employee
8      :param shift_dict: Dictionary containing all the assigned shifts for
       each employee.
9      :type shift_dict: dict
10     :return: True if e0 has a shift assigned to each day.
11     :rtype: bool
12     """
13     e_shifts = shift_dict[f"enp{e0}"]
14     if len(e_shifts) >= employee.DAYS:
15         return True
16     return False
```

Código 4.2: *Fragmento de código que contiene la función que verifica la restricción 3.9.*

La siguiente restricción a comprobar será la 3.8 (ver Código 4.3). Esta se hará de manera similar a la anterior, se cuenta el total de horas que han sido asignadas al empleado e y si no se le pueden asignar más el turno, se descarta.

```
1  def e_hours_filled(e0, employee, shift_dict):
2      """
3      This function checks if a given employee e0 has all of its hours
       already occupied.
4      :param e0: index of the employee to check for.
5      :type e0: numpy.ndarray
6      :param employee: Employee type object of index e0 in the list of
       employees.
7      :type employee: Employee
8      :param shift_dict: Dictionary containing all the assigned shifts for
       each employee.
9      :type shift_dict: dict
10     :return: True if all of e0 shifts sum equal to its HOURS property.
11     :rtype: bool
12     """
13     e_shifts = shift_dict[f"enp{e0}"]
14     h_total = 0
15     for d, b, r in e_shifts:
16         h_total += Preference.len_b(b)
```

```
17      if h_total >= employee.HOURS:
18          return True
19      return False
```

Código 4.3: *Esta función utiliza la función* `len_b` *de la clase* `Preference` *para calcular la longitud de los bloques ya asignados y accede al atributo HOURS para hacer la comprobación de la Restricción 3.8.*

Después, se continúa con las restricciones del negocio, en primer lugar se mira si se ha cubierto el mínimo de trabajadores requeridos para ese rol ese día. Si no se ha cubierto el mínimo automáticamente se le asigna el turno al empleado e. Si el mínimo ya se ha cubierto, entonces se comprueba si ya se ha llegado al máximo. Si se ha alcanzado el máximo entonces el turno se descarta, si no, se asigna (ver Código 4.4).

```
1  def occupation_less_than_min_for_day_d(d0, r0, shift_dict, minnum):
2      """
3      This function checks if the occupation for a certain rol r0 in a
       given day d0 has reached it's minimum or not.
4      :param d0: index of day to check for.
5      :type d0: numpy.ndarray
6      :param r0: index of rol to check for.
7      :type r0: numpy.ndarray
8      :param shift_dict: Dictionary containing all the assigned shifts for
       each employee.
9      :type shift_dict: dict
10     :param minnum: Array containing the minimum number of employees
       needed in each rol.
11     :return: True if minimum is filled, False otherwise.
12     :rtype: bool
13     """
14     min_for_this_rol = minnum[r0]
15     counter = 0
16     all_shifts = []
17     for val in shift_dict.values():
18         all_shifts.extend(val)
19     for d, b, r in all_shifts:
20         if d == d0:
21             if r == r0:
22                 counter += 1
23                 if counter >= min_for_this_rol:
24                     return False
25     return True
```

Código 4.4: *Fragmento de código que se encarga de comprobar la Restricción 3.12, la del mínimo de empleados. La del máximo es igual solo que con* `maxnum` *en lugar de* `minnum`.

En todos los casos anteriores, si se añade o se descarta el turno, se cambia su valor por $-\infty$ para evitar que vuelva a ser elegido en la siguiente iteración. Además, si se asigna el turno también se añade el valor de T de esa posición a la función objetivo.

Con esta metodología, el programa ha demostrado dar buenos resultados en los ejemplos a los que ha sido sometido (ver Sección 4.4). El hecho de usar arrays de Numpy en lugar de listas de Python convencionales, aligera la carga de procesamiento y acelera el proceso de resolución. Sin embargo, hay una marcada diferencia entre la solución obtenida por `solve()` y la obtenida por `pulp_solve()` como ya se adelantó en la Sección 3.1 y está

relacionada con como se interpreta la "no disponibilidad" de los empleados. Más adelante, cuando se muestren los resultados obtenidos por ambos, se explicará esta diferencia.

4.3.2. Resolución mediante `pulp_solve()`

PuLP es un programa de modelado de problemas lineales escrito en Python [10]. Es capaz de generar y resolver problemas lineales llamando a diversas librerías y programas de optimización tanto de código abierto como GLPK [11] o comerciales como CPLEX [12] o Gurobi [13]. Con PuLP, se simplifica el proceso de definir las restricciones y resolver el problema. Además, permite usar programas externos con implementaciones basadas en otro tipo de algoritmos además de los voraces.

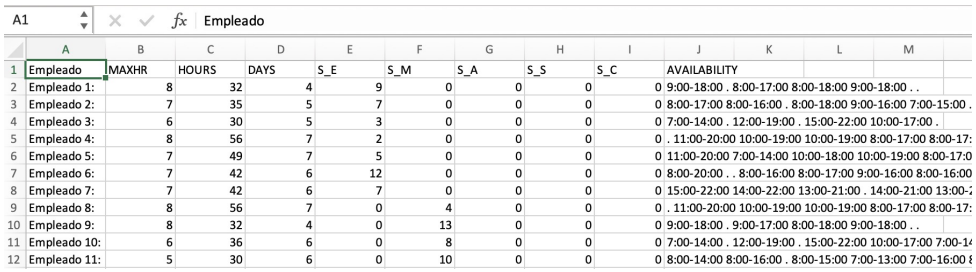
El procedimiento a seguir es el siguiente,

1. Definir todas las variables del problema, mediante `pulp.LpVariable`. Habrá un total de $|D| \cdot |B| \cdot |R| \cdot |E|$ variables.
2. Definir el problema y la función objetivo usando `pulp.LpProblem` y `pulp.LpAffine-Expression`. El primero de los dos crea un objeto `problem` que será el problema lineal. A este se le añadirán las restricciones, la función objetivo y el estado del problema como atributos. La función objetivo se define como una expresión lineal de las variables creadas previamente y se añade al objeto `problem`. En la función `solve()`, bastaba con calcular la multiplicación vectorial entre $P_{d,b,r,e}$ y $S_{r,e}$ y ajustar los valores infinitos para usarlo dentro de la función objetivo. Esta segunda implementación sin embargo, no admite valores infinitos así que se cambia por un valor muy negativo en comparación con los valores "normales", $-100\,000$. Esto significa que el valor de penalización de las posiciones que no cumplen con las restricciones pasa de ser "inalcanzable" a "altamente improbable", pero no imposible.
3. Definir las restricciones al problema, que serán interpretadas como un diccionario de objetos `pulp.LpConstraint`. Estas también se añaden como expresión lineal al objeto `problem`.
4. A continuación el problema puede ser resuelto con el *solver* (programa de resolución externo) elegido. Por defecto, el problema será resuelto con el solver propio de PuLP aunque en este trabajo se ha probado con dos solvers más además de con el que viene por defecto, GLPK y Gurobi. La solución puede ser reconstruida a partir de las variables definidas anteriormente.

4.4. Comparación entre los métodos de resolución

Las dos funciones de resolución que incluye el programa se han escrito de forma que devuelvan el valor objetivo y el diccionario `assigned_shifts`. La solución final se guarda en un archivo *csv* mediante la función `load_to_csv`, para que sea accesible desde cualquier aplicación de hojas de cálculo.

Para ver si el programa funciona correctamente, se proponen dos ejemplos. El primero de ellos es el ejemplo de la cafetería al que se ha hecho alusión repetidas veces a lo largo de este trabajo. El segundo es un ejemplo que representa una clínica o una planta de hospital que cuenta con más trabajadores y más roles que el ejemplo de la cafetería. En la Figura 4.3 se muestra un ejemplo de cómo se han de introducir los datos.



	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Empleado	MAXHR	HOURS	DAYS	S_E	S_M	S_A	S_S	S_C	AVAILABILITY			
2	Empleado 1:	8	32	4	9	0	0	0	0	9:00-18:00 . 8:00-17:00 8:00-18:00 9:00-18:00 . .			
3	Empleado 2:	7	35	5	7	0	0	0	0	8:00-17:00 8:00-16:00 . 8:00-18:00 9:00-16:00 7:00-15:00 .			
4	Empleado 3:	6	30	5	3	0	0	0	0	7:00-14:00 . 12:00-19:00 . 15:00-22:00 10:00-17:00 .			
5	Empleado 4:	8	56	7	2	0	0	0	0	. 11:00-20:00 10:00-19:00 10:00-19:00 8:00-17:00 8:00-17:			
6	Empleado 5:	7	49	7	5	0	0	0	0	11:00-20:00 7:00-14:00 10:00-18:00 10:00-19:00 8:00-17:0			
7	Empleado 6:	7	42	6	12	0	0	0	0	8:00-20:00 . . 8:00-16:00 8:00-17:00 9:00-16:00 8:00-16:00			
8	Empleado 7:	7	42	6	7	0	0	0	0	15:00-22:00 14:00-22:00 13:00-21:00 . 14:00-21:00 13:00-			
9	Empleado 8:	8	56	7	0	4	0	0	0	. 11:00-20:00 10:00-19:00 10:00-19:00 8:00-17:00 8:00-17:			
10	Empleado 9:	8	32	4	0	13	0	0	0	9:00-18:00 . 9:00-17:00 8:00-18:00 9:00-18:00 . .			
11	Empleado 10:	6	36	6	0	8	0	0	0	7:00-14:00 . 12:00-19:00 . 15:00-22:00 10:00-17:00 7:00-14			
12	Empleado 11:	5	30	6	0	10	0	0	0	8:00-14:00 8:00-16:00 . 8:00-15:00 7:00-13:00 7:00-16:00 8			

Figura 4.3: *Recorte tomado de la aplicación Microsoft Excel que muestra los datos de entrada introducidos como csv.*

Cafetería

Este problema lineal tiene las siguientes características:

- 6 empleados.
- Longitudes de bloque permitidas: 6, 7 y 8 horas.
- Primer turno a las $7:00$ y último acaba a las $22:00$.
- 3 roles: barra, camarero y cocina. Mínimo número de trabajadores en cada rol (en orden): $[1, 2, 1]$, máximo número de trabajadores en cada rol: $[2, 3, 2]$.
- Duración del ciclo, 5 días.

Clínica

Al tener más variables, el problema de la clínica tiene mayor complejidad:

- 20 empleados.
- Longitudes de bloque permitidas: 5, 6, 7 y 8 horas.
- Primer turno a las $7:00$ y el último acaba a las $22:00$.
- 5 roles: enfermero, médico, auxiliar, secretario, celador. Mínimo número de trabajadores en cada rol: $[3, 3, 1, 1, 1]$ (9 empleados como mínimo). Máximo número de trabajadores en cada rol: $[5, 6, 2, 2, 2]$ (17 empleados como máximo).
- Duración del ciclo: 7 días.

4.4.1. Solución al problema de la cafetería

En la Tabla 4.2 se muestran las soluciones y los valores objetivo obtenidos con los dos métodos de resolución. El primero, `solve()` es la implementación que usa un algoritmo voraz mientras que los siguientes tres están calculadas usando `pulp_solve()`. La diferencia entre los últimos tres está en qué algoritmo se ha usado.

En esta resolución, se ha optado por usar tres variantes de `pulp_solve()`, como se puede ver en los resultados. El primero de los tres usa el *solver* por defecto de PuLP, que implementa el algoritmo "Branch and cut". Este algoritmo resuelve el problema lineal sin la condición de que las variables sean enteras usando el algoritmo Simplex y después se usa un algoritmo de corte de plano para encontrar más restricciones que se añaden al problema con la esperanza encontrar soluciones enteras a las variables y hacer el problema "menos

Programa	Algoritmo	Valor objetivo	Tiempo de ejecución 5 veces, 2 rep	Tiempo de ejecución 10 veces, 2 rep
ShiftSolver.solve()	Voraz	1170,0	$778{,}83 \pm 121{,}73\,\mu s$	$548{,}82 \pm 33{,}22\,\mu s$
PULP_CBC_CMD	*Branch and cut*	-198835.71	$96{,}51 \pm 2{,}40\,ms$	$94{,}87 \pm 0{,}91\,ms$
GLPK_CMD	Simplex	-198835.71	$89{,}46 \pm 0{,}17\,ms$	$89{,}15 \pm 0{,}12\,ms$
GUROBI	Simplex	-198835.71	$68{,}96 \pm 0,01\,ms$	$91{,}37 \pm 31{,}82\,ms$

Tabla 4.2: *Resultados del problema lineal de la cafetería obtenidos con los dos métodos de resolución. El primero, es el resultado de la función* `solve()` *que es la que implementa el algoritmo voraz y como se puede ver es significativamente más rápida pero es más básica que los otros. Los siguientes tres se obtienen con* `pulp_solve()` *e implementan algoritmos más complejos.*

fraccional". Los dos siguientes usan los *solvers* GLPK y Gurobi, y los dos implementan el algoritmo Simplex del cual ya se habló en la Sección 2.2.

Salta a la vista lo diferente que es el valor objetivo obtenido por el algoritmo voraz comparado con los demás algoritmos, además de que es el único que ha obtenido un valor positivo, es decir, un valor objetivo sin penalizaciones. Aunque pueda parecer que alguno de los resultados obtenidos es erróneo, un análisis más exhaustivo muestra que no es así. Han funcionado exactamente como se esperaba que lo hicieran, la diferencia radica en que `solve()` y `pulp_solve()` no se comportan igual ante la "no disponibilidad" de un empleado.

Por un lado, para la función `solve()` que un empleado no esté disponible se interpreta de la misma forma que una restricción, un bloque de un empleado no disponible es absolutamente inalcanzable. Como ya se ha mencionado anteriormente, el bucle principal que controla la ejecución de `solve()` solo se detendrá si se cumple una de estas condiciones (o las dos):

- Los empleados ya no admiten más turnos, se han asignado todos los que se podían.
- El siguiente valor máximo no es mayor que $-\infty$, es decir, es $-\infty$.

Si el que un empleado no este disponible se trata igual que una restricción, el programa terminará sin asignar todos los turnos que podía haber asignado porque solo quedarán posiciones con valor $-\infty$.

Por otro lado, la función `pulp_solve()` no trata a las posiciones en las que el empleado no está disponible como si fueran restricciones. Las restricciones deben cumplirse, y si no se cumplen, se descarta la posición. Sin embargo, ¿Qué ocurre cuando hay un turno que debe ser cubierto pero no hay ninguna posición con valor positivo que pueda cumplir todas las restricciones para que le sea asignado? Lo que ocurre es que a `pulp_solve()` no le queda más remedio que asignarle el turno a un empleado que se ha declarado no disponible para ese bloque.

Por lo tanto, lo que está ocurriendo es que `solve()` decide violar la Restricción 3.12 que indica que debe haber un mínimo de trabajadores en todo momento para respetar la "no disponibilidad" de los empleados. Dicho de otra forma, no puede asignar un turno con valor $-\infty$ por construcción, mientras que `pulp_solve()` asigna el puesto a un empleado aunque no esté disponible porque el valor que tiene en el tensor T no lo hace inalcanzable,

solo lo convierte en muy difícil de ser elegido. Así que, en primera instancia la solución de `solve()` sería inviable mientras que la de `pulp_solve()` sería viable.

Este problema puede solucionarse sin mayor dificultad, añadiendo una restricción más al modelo de PuLP para que lo considere también una restricción y se comporte más como lo hace `solve()`, o si no cambiando el valor que se asigna a una posición de un empleado que no está disponible por un valor muy negativo pero mayor que $-\infty$ para que `solve()` se comporte más como `pulp_solve()`. Sin embargo, se ha decidido dejarlo así porque ambos comportamientos son válidos si se sabe interpretar la solución que devuelve cada uno de ellos. Esta disparidad podría tener utilidad a la hora de estructurar una plantilla; por ejemplo, podría indicar que existe una necesidad de contratar a más trabajadores para cubrir todos los turnos establecidos.

	A	B	C	D	E	F
1	employee	day0	day1	day2	day3	day4
2	enp0	9:00-17:00 Barra			8:00-16:00 Barra	9:00-17:00 Barra
3	enp1	8:00-15:00 Camarero	8:00-15:00 Camarero		8:00-15:00 Camarero	9:00-16:00 Camarero
4	enp2	7:00-13:00 Barra		12:00-18:00 Barra		15:00-21:00 Barra
5	enp3		11:00-19:00 Cocina	10:00-18:00 Cocina	10:00-18:00 Cocina	8:00-16:00 Cocina
6	enp4	11:00-18:00 Cocina	7:00-14:00 Camarero	10:00-17:00 Camarero	10:00-17:00 Camarero	8:00-15:00 Camarero
7	enp5	8:00-15:00 Camarero			8:00-15:00 Camarero	8:00-15:00 Camarero

a

	A	B	C	D	E	F
1	employee	day0	day1	day2	day3	day4
2	enp0	10:00-18:00 Barra			9:00-17:00 Barra	10:00-18:00 Barra
3	enp1	9:00-16:00 Camarero	8:00-15:00 Camarero		10:00-17:00 Camarero	9:00-16:00 Camarero
4	enp2	8:00-14:00 Camarero	8:00-15:00 Barra	13:00-19:00 Barra		15:00-21:00 Barra
5	enp3		11:00-19:00 Cocina	10:00-18:00 Cocina	10:00-18:00 Cocina	9:00-17:00 Cocina
6	enp4	11:00-18:00 Cocina	8:00-14:00 Camarero	10:00-17:00 Camarero	11:00-18:00 Camarero	10:00-17:00 Camarero
7	enp5	12:00-19:00 Camarero		10:00-16:00 Camarero	9:00-16:00 Camarero	8:00-15:00 Camarero

b

Figura 4.4: *En (a) se muestra la solución de* `solve()` *mientras que en (b) se muestra la asignación obtenida con* `pulp_solve()` *usando el módulo PULP_CBC_CMD.*

Dejando a parte ese detalle, los cuatro obtienen una asignación de turnos muy similar y tienen tiempos de ejecución similares. La rapidez del algoritmo voraz en comparación a los demás puede achacarse a que los demás ejecutan más subprocesos de optimización y los muestran en pantalla mientras se están ejecutando.

Las dos soluciones que se muestran en la Figura 4.4 son prácticamente iguales, la única diferencia es que `pulp_solve()` ha asignado dos turnos más. Esto no es casualidad, ya que concuerda con los valores objetivo obtenidos por cada uno de ellos. En los datos de los empleados de la cafetería entregados al programa, enp_2 se mostraba no disponible el martes, al igual que enp_5 tampoco estaba disponible el miércoles. Tal y como se ha explicado, `solve()` ha violado la Restricción 3.12 evitando asignar más turnos en esos días y por eso ha obtenido un valor objetivo mucho mejor. Por el contrario, `pulp_solve` ha asignado dos turnos a empleados que no estaban disponibles, lo que ha supuesto sumar dos penalizaciones a su valor objetivo.

$$1165 + 2 \cdot (-100000) = -198835 \simeq -198835{,}71 = F\_obj_{pulp\_cbc},$$

lo que significa que si no hubiera asignado esos dos turnos con penalización, los valores objetivo obtenidos con `pulp_solve()` habrían sido muy similares, probablemente iguales;

$$\text{sin penalizaciones} \qquad F\_obj_{pulp} \approx 1164{,}29 \leq 1170 = F\_obj_{solve}$$

4.4.2. Solución al problema de la clínica

Para el problema de la clínica se considera una plantilla de veinte trabajadores y cinco roles, lo que aumenta significativamente el tamaño del problema a resolver. Aunque, también es cierto que para este caso se ha considerado una situación más real para una clínica en la que un trabajador solo tiene experiencia en un único rol. Además, se considera un ciclo de siete días en lugar de cinco por lo que hay que considerar más intervalos de disponibilidad.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	Empleado	MAXHR	HOURS	DAYS	S_E	S_M	S_A	S_S	S_C	AVAILABILITY					
2	Empleado 1:	8	32	4	9	0	0	0	0	9:00-18:00 . 8:00-17:00 8:00-18:00 9:00-18:00 . .					
3	Empleado 2:	7	35	5	7	0	0	0	0	8:00-17:00 8:00-16:00 . 8:00-18:00 9:00-16:00 7:00-15:00 .					
4	Empleado 3:	6	30	5	3	0	0	0	0	7:00-14:00 . 12:00-19:00 . 15:00-22:00 10:00-17:00 .					
5	Empleado 4:	8	56	7	2	0	0	0	0	. 11:00-20:00 10:00-19:00 10:00-19:00 8:00-17:00 8:00-17:00 10:00-18:00					
6	Empleado 5:	7	49	7	5	0	0	0	0	11:00-20:00 7:00-14:00 10:00-18:00 10:00-19:00 8:00-17:00 . 8:00-17:00					
7	Empleado 6:	7	42	6	12	0	0	0	0	8:00-20:00 . . 8:00-16:00 8:00-17:00 9:00-16:00 8:00-16:00					
8	Empleado 7:	7	42	6	7	0	0	0	0	15:00-22:00 14:00-22:00 13:00-21:00 . 14:00-21:00 13:00-20:00 15:00-22:00					
9	Empleado 8:	8	56	7	0	4	0	0	0	. 11:00-20:00 10:00-19:00 10:00-19:00 8:00-17:00 8:00-17:00 10:00-18:00					
10	Empleado 9:	8	32	4	0	13	0	0	0	9:00-18:00 . 9:00-17:00 8:00-18:00 9:00-18:00 . .					
11	Empleado 10:	6	36	6	0	8	0	0	0	7:00-14:00 . 12:00-19:00 . 15:00-22:00 10:00-17:00 7:00-14:00					
12	Empleado 11:	5	30	6	0	10	0	0	0	8:00-14:00 8:00-16:00 . 8:00-15:00 7:00-13:00 7:00-16:00 8:00-16:00					
13	Empleado 12:	7	35	5	0	6	0	0	0	8:00-17:00 8:00-16:00 . 8:00-18:00 9:00-16:00 7:00-15:00 .					
14	Empleado 13:	6	36	6	0	7	0	0	0	8:00-20:00 . . 8:00-16:00 8:00-17:00 9:00-16:00 8:00-16:00					
15	Empleado 14:	8	48	6	0	0	5	0	0	9:00-18:00 11:00-20:00 . 10:00-19:00 8:00-17:00 8:00-17:00 10:00-18:00					
16	Empleado 15:	7	42	6	0	0	3	0	0	8:00-20:00 . 7:00-16:00 8:00-16:00 8:00-17:00 9:00-16:00 8:00-16:00					
17	Empleado 16:	7	42	6	0	0	9	0	0	8:00-17:00 8:00-16:00 10:00-17:00 8:00-18:00 9:00-16:00 7:00-15:00 .					
18	Empleado 17:	7	49	7	0	0	0	2	0	11:00-20:00 7:00-14:00 . 10:00-19:00 8:00-17:00 7:00-15:00 8:00-17:00					
19	Empleado 18:	8	40	5	0	0	0	5	0	9:00-18:00 8:00-18:00 9:00-17:00 8:00-18:00 9:00-18:00 . .					
20	Empleado 19:	8	40	5	0	0	0	0	6	9:00-18:00 . 9:00-17:00 8:00-18:00 9:00-18:00 9:00-19:00 .					
21	Empleado 20:	6	42	7	0	0	0	0	3	7:00-14:00 9:00-16:00 12:00-19:00 . 8:00-15:00 10:00-17:00 7:00-14:00					

Figura 4.5: *Datos de entrada para el problema de la clínica escritos en Excel en formato csv.*

Este segundo ejemplo es más completo que el anterior y se ha considerado una plantilla con más trabajadores que la máxima cantidad que va a haber trabajando en cualquier momento. De esta manera, se evita (o al menos se reduce) la posibilidad de que no haya ningún trabajador disponible para cubrir un puesto que debe ser cubierto. Por lo tanto, este es un buen ejemplo para analizar el rendimiento de la función `solve`. En la Figura 4.5 se muestran los datos que se le han dado como input al programa para este ejemplo y en la Tabla 4.3 aparecen las soluciones y los valores objetivo que resuelven el problema de la clínica.

Programa	**Algoritmo**	**Valor objetivo**	**Tiempo de ejecución**	
			5 veces, 2 rep	**10 veces, 2 rep**
ShiftSolver.solve()	Voraz	6350	$5{,}51 \pm 1{,}12\,ms$	$4{,}71 \pm 1{,}03\,ms$
PULP_CBC_CMD	*Branch and cut*	6320	$943{,}09 \pm 5{,}14\,ms$	$946{,}18 \pm 4{,}34\,ms$
GLPK_CMD	Simplex	6320	$1{,}00 \pm 0{,}00\,ms$	$1{,}00 \pm 0{,}00\,ms$
GUROBI	Simplex	6320	$794{,}94 \pm 34{,}89\,ms$	$772{,}25 \pm 2{,}04\,ms$

Tabla 4.3: *Resultados del problema lineal de la clínica.*

Los valores objetivo calculados indican, sorprendentemente, que la función más sencilla `solve()` que implementa el algoritmo voraz, logra una puntuación mayor que la función `pulp_solve()` usando los distintos programas comerciales. Además de eso, tarda apenas un 0,5 % del tiempo que tardan los demás en calcular la solución.

Como se puede ver en la Figura 4.6, la asignación que calcula cada función es notablemente diferente. La función `solve()` tiene una tendencia a escoger turnos que empiezan

	A	B	C	D	E	F	G	H
1	employee	day0	day1	day2	day3	day4	day5	day6
2	enp0	9:00-17:00 Enfermera		8:00-16:00 Enfermera	8:00-16:00 Enfermera	9:00-17:00 Enfermera		
3	enp1	8:00-15:00 Enfermera	8:00-15:00 Enfermera		8:00-15:00 Enfermera	9:00-16:00 Enfermera	7:00-14:00 Enfermera	
4	enp2			12:00-18:00 Enfermera			10:00-16:00 Enfermera	
5	enp3		11:00-19:00 Enfermera	10:00-18:00 Enfermera	10:00-18:00 Enfermera		8:00-16:00 Enfermera	10:00-18:00 Enfermera
6	enp4	11:00-18:00 Enfermera	7:00-14:00 Enfermera	10:00-17:00 Enfermera	10:00-17:00 Enfermera	8:00-15:00 Enfermera		8:00-15:00 Enfermera
7	enp5	8:00-15:00 Enfermera			8:00-15:00 Enfermera	8:00-15:00 Enfermera	9:00-16:00 Enfermera	8:00-15:00 Enfermera
8	enp6	15:00-22:00 Enfermera	14:00-21:00 Enfermera	13:00-20:00 Enfermera		14:00-21:00 Enfermera	13:00-20:00 Enfermera	15:00-22:00 Enfermera
9	enp7		11:00-19:00 Medico	10:00-18:00 Medico	10:00-18:00 Medico	8:00-16:00 Medico	8:00-16:00 Medico	10:00-18:00 Medico
10	enp8	9:00-17:00 Medico		9:00-17:00 Medico	8:00-16:00 Medico	9:00-17:00 Medico		
11	enp9	7:00-13:00 Medico		12:00-18:00 Medico		15:00-21:00 Medico	10:00-16:00 Medico	7:00-13:00 Medico
12	enp10	8:00-13:00 Medico	8:00-13:00 Medico		8:00-13:00 Medico	7:00-12:00 Medico	7:00-12:00 Medico	8:00-13:00 Medico
13	enp11	8:00-15:00 Medico	8:00-15:00 Medico		8:00-15:00 Medico	9:00-16:00 Medico	7:00-14:00 Medico	
14	enp12	8:00-14:00 Medico			8:00-14:00 Medico	8:00-14:00 Medico	9:00-15:00 Medico	8:00-14:00 Medico
15	enp13	9:00-17:00 Auxiliar	11:00-19:00 Auxiliar		10:00-18:00 Auxiliar	8:00-16:00 Auxiliar	8:00-16:00 Auxiliar	10:00-18:00 Auxiliar
16	enp14			7:00-14:00 Auxiliar				8:00-15:00 Auxiliar
17	enp15	8:00-15:00 Auxiliar	8:00-15:00 Auxiliar	10:00-17:00 Auxiliar	8:00-15:00 Auxiliar	9:00-16:00 Auxiliar	7:00-14:00 Auxiliar	
18	enp16	11:00-18:00 Secretario	7:00-14:00 Secretario		10:00-17:00 Secretario	8:00-15:00 Secretario	7:00-14:00 Secretario	8:00-15:00 Secretario
19	enp17	9:00-17:00 Secretario	8:00-16:00 Secretario	9:00-17:00 Secretario	8:00-16:00 Secretario	9:00-17:00 Secretario		
20	enp18	9:00-17:00 Celador		9:00-17:00 Celador	8:00-16:00 Celador	9:00-17:00 Celador	9:00-17:00 Celador	
21	enp19	7:00-13:00 Celador	9:00-15:00 Celador	12:00-18:00 Celador		8:00-14:00 Celador	10:00-16:00 Celador	7:00-13:00 Celador

a

	A	B	C	D	E	F	G	H
1	employee	day0	day1	day2	day3	day4	day5	day6
2	enp0	9:00-17:00 Enfermera		8:00-16:00 Enfermera	10:00-18:00 Enfermera	10:00-18:00 Enfermera		
3	enp1	8:00-15:00 Enfermera	9:00-16:00 Enfermera		9:00-16:00 Enfermera	9:00-16:00 Enfermera	8:00-15:00 Enfermera	
4	enp2			13:00-19:00 Enfermera			10:00-16:00 Enfermera	
5	enp3		12:00-20:00 Enfermera	10:00-18:00 Enfermera	11:00-19:00 Enfermera		9:00-17:00 Enfermera	10:00-18:00 Enfermera
6	enp4	13:00-20:00 Enfermera	8:00-14:00 Enfermera	11:00-18:00 Enfermera	12:00-19:00 Enfermera	9:00-16:00 Enfermera		8:00-15:00 Enfermera
7	enp5	9:00-16:00 Enfermera			9:00-16:00 Enfermera	9:00-16:00 Enfermera	9:00-16:00 Enfermera	9:00-16:00 Enfermera
8	enp6	15:00-21:00 Enfermera	14:00-21:00 Enfermera	14:00-21:00 Enfermera		14:00-21:00 Enfermera	13:00-20:00 Enfermera	15:00-21:00 Enfermera
9	enp7		11:00-19:00 Medico	11:00-19:00 Medico	11:00-19:00 Medico	9:00-17:00 Medico	8:00-16:00 Medico	10:00-18:00 Medico
10	enp8	9:00-17:00 Medico		9:00-17:00 Medico	10:00-18:00 Medico	9:00-17:00 Medico		
11	enp9	8:00-14:00 Medico		12:00-18:00 Medico		15:00-21:00 Medico	11:00-17:00 Medico	8:00-14:00 Medico
12	enp10	9:00-14:00 Medico	11:00-16:00 Medico		9:00-14:00 Medico	8:00-13:00 Medico	9:00-14:00 Medico	8:00-13:00 Medico
13	enp11	8:00-15:00 Medico	8:00-15:00 Medico		11:00-18:00 Medico	9:00-16:00 Medico	8:00-15:00 Medico	
14	enp12	12:00-18:00 Medico			10:00-16:00 Medico	9:00-15:00 Medico	9:00-15:00 Medico	9:00-15:00 Medico
15	enp13	10:00-18:00 Auxiliar	11:00-19:00 Auxiliar		10:00-18:00 Auxiliar	9:00-17:00 Auxiliar	9:00-17:00 Auxiliar	10:00-18:00 Auxiliar
16	enp14			9:00-16:00 Auxiliar				8:00-15:00 Auxiliar
17	enp15	8:00-15:00 Auxiliar	8:00-15:00 Auxiliar	10:00-17:00 Auxiliar	11:00-18:00 Auxiliar	9:00-16:00 Auxiliar	8:00-15:00 Auxiliar	
18	enp16	12:00-19:00 Secretario	8:00-14:00 Secretario		10:00-17:00 Secretario	9:00-16:00 Secretario	8:00-15:00 Secretario	9:00-16:00 Secretario
19	enp17	9:00-17:00 Secretario	10:00-18:00 Secretario	9:00-17:00 Secretario	10:00-18:00 Secretario	10:00-18:00 Secretario		
20	enp18	9:00-17:00 Celador		9:00-17:00 Celador	10:00-18:00 Celador	9:00-17:00 Celador	10:00-18:00 Celador	
21	enp19	8:00-14:00 Celador	9:00-15:00 Celador	13:00-19:00 Celador		8:00-14:00 Celador	10:00-16:00 Celador	8:00-14:00 Celador

b

Figura 4.6: *En (a) se muestra los resultados obtenidos con* `solve()` *mientras que en (b) se muestra la solución calculada por* `pulp_solve()` *usando el módulo de Gurobi.*

más temprano que los que elige `pulp_solve()`. Esto podría estar causado por la naturaleza voraz de `solve()`, dado que si un empleado declara un intervalo de disponibilidad que es mayor que su valor de $MAXHR$, ese intervalo de disponibilidad contendrá más de un bloque válido para ese empleado. Las posiciones de los bloques que están dentro de ese intervalo de disponibilidad tendrán el mismo valor de preferencia ya que tendrán la misma longitud, $MAXHR$. También habrá bloques con longitud menor a $MAXHR$, pero solo serán elegidos antes que los de mayor longitud en el caso en que hubiera un solapamiento entre turnos y se excediese el número máximo de trabajadores $MAXNUM$. Entonces, como los bloques que empiezan antes aparecen antes en el tensor T, estos prevalecerán sobre los que empiezan más tarde, a pesar de que tengan el mismo valor de preferencia. Sin embargo, eso no explica por qué la puntuación obtenida por `solve()` es mejor que la que se consigue usando algoritmos que tienen en cuenta más casos y que exploran más valores óptimos como el algoritmo Simplex.

Capítulo 5

Conclusiones

El propósito de este trabajo es principalmente académico. El problema NSP se ha resuelto numerosas veces y se han empleando todo tipo de técnicas y algoritmos. Los algoritmos voraces no son una excepción, que cuentan con la ventaja de ser muy versátiles a la hora de su programación dada su alta simplicidad. En este trabajo, se ha hecho una adaptación del clásico problema NSP para conseguir dos problemas lineales y ambos se han resuelto utilizando los métodos de optimización que se habían elegido.

Por un lado, se han podido comparar los resultados calculados por el algoritmo voraz con los obtenidos con algoritmos más complejos y se ha visto que existen diferencias en su forma de actuar. La naturaleza voraz de la función `solve()`, no resuelve subproblemas ni explora más soluciones como lo hacen otros algoritmos, lo que le lleva a obviar posibilidades que pueden ser igualmente buenas. Debido a esto, aparecen diferencias en las asignaciones de turnos que escoge cada función de resolución. Esto indica que, a pesar de que una implementación con voraces funcione, un programa basado en algoritmos más complejos puede aportar más flexibilidad y probablemente obtenga soluciones que satisfagan más a los empleados.

Por otro lado, la diferencia entre los resultados de los dos métodos de resolución en el problema de la cafetería es algo que podría cambiarse conociendo mejor las necesidades de cada establecimiento. Como ya se explicó en la Sección 4.4.1, respetar o no la disponibilidad de los empleados se hace a costa de cumplir con el mínimo de trabajadores que debe haber en el turno. Puede que para algunos establecimientos sea más adecuado respetar la disponibilidad, por lo que en esos casos habría que adaptar el programa para que indicase que no se ha cumplido la condición de mínimo de trabajadores. En cambio, puede que en otros establecimientos se le de más importancia a que se cubra el mínimo, en cuyo caso, simplemente se cambiaría el código de `solve()` para que actuase como lo hace `pulp_solve()`. Lo más sencillo para conseguirlo sería cambiar el $-\infty$ que se introduce en $P_{d,b,r,e}$ cuando un empleado no esta disponible por un valor de penalización como $-100\,000$. Esto conllevaría la modificación de la construcción de T para evitar conflictos en los casos en que un empleado no tiene experiencia en el rol, $S_{r,e} = 0$ y no tiene disponibilidad ($P_{d,b,r,e} = -100\,000$) o sí la tiene ($P_{d,b,r,e} = m,\ m \in \mathbb{R}$).

Esta solución demuestra que todavía hay mucho margen de mejora en cuanto a optimización se refiere. Especialmente, el problema mencionado en el párrafo anterior quizás pudiera ser resuelto modificando la función $f(S, P)$ de la función objetivo. Como ya se

dijo, la elección de f es completamente arbitraria y una elección de esta que fuera más selectiva podría solucionarlo. Por otra parte, otro aspecto a mejorar sería la forma de introducir los datos. Utilizar un programa de hojas de cálculo no es una mala opción pero puede llegar a ser algo engorroso cuando se trata de introducir una gran cantidad de datos. Desde un principio la idea de incorporar una interfaz gráfica al programa estaba sobre la mesa, pero finalmente, por falta de tiempo, no se ha podido añadir.

En cualquier caso, queda claro que resolver el problema de asignación de turnos no es tarea fácil porque siempre podrán encontrarse casos especiales en los que actuar de forma distinta.

Bibliografía

[1] R. Lagatie, S. Haspeslagh y P. De Causmaecker, «Negotiation Protocols for Distributed Nurse Rostering,» *Belgian/Netherlands Artificial Intelligence Conference*, ene. de 2009.

[2] R. Otero-Caicedo, C. E. M. Casas, C. B. Jaimes, C. F. G. Garzón, E. A. Y. Vergel y J. C. Z. Valdés, «A preventive–reactive approach for nurse scheduling considering absenteeism and nurses' preferences,» *Operations Research for Health Care*, vol. 38, pág. 100 389, 2023, ISSN: 2211-6923. DOI: https://doi.org/10.1016/j.orhc.2023.100389. dirección: https://www.sciencedirect.com/science/article/pii/S2211692323000127.

[3] A. A. El Adoly, M. Gheith y M. Nashat Fors, «A new formulation and solution for the nurse scheduling problem: A case study in Egypt,» *Alexandria Engineering Journal*, vol. 57, n.º 4, págs. 2289-2298, 2018, ISSN: 1110-0168. DOI: https://doi.org/10.1016/j.aej.2017.09.007. dirección: https://www.sciencedirect.com/science/article/pii/S111001681730282X.

[4] W. B. Richard Hoshino Aaron Slobodin, «An Automated Employee Timetabling System for Small Businesses,» en *Proceedings of the AAAI Conference on Artificial Intelligence,*, vol. 1, 2018. DOI: 10.1609/aaai.v32i1.11383.

[5] G. van Rossum, *python*, ver. 3.10.10, Python Software Foundation, 1991. dirección: https://www.python.org.

[6] C. R. Harris, K. J. Millman, S. J. van der Walt et al., «Array programming with NumPy,» *Nature*, vol. 585, n.º 7825, págs. 357-362, sep. de 2020. DOI: 10.1038/s41586-020-2649-2. dirección: https://doi.org/10.1038/s41586-020-2649-2.

[7] J. M. X.-J. Lu A. Ruschhaupt, «Fast shuttling of a particle under weak spring-constant noise of the moving trap,» *American Physical Society*, vol. 97, 2018. DOI: 10.1103/PhysRevA.97.053402.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest y C. Stein, *Introduction to Algorithms*, English, 3.ª ed. MIT Press, 2009, ISBN: 978-0-262-53305-8.

[9] H. P. Williams, *Logic and Integer Programming*, English, 1.ª ed. Springer, 2009, ISBN: 978-0-387-92279-9.

[10] S. Mitchell, A. Kean, A. Mason, M. O'Sullivan y et al. «PuLP.» (2009), dirección: https://coin-or.github.io/pulp/index.html.

[11] A. O. Makhorin, *GNU Linear Programming Kit*, ver. 5.0, GNU Project, 2000. dirección: https://www.gnu.org/software/glpk/glpk.html.

[12] I. I. Cplex, «V12. 1: User's Manual for CPLEX,» *International Business Machines Corporation*, vol. 46, n.º 53, pág. 157, 2009.

[13] Gurobi Optimization, LLC, *Gurobi Optimizer Reference Manual*, 2023. dirección: https://www.gurobi.com.