Las matemáticas de la organización industrial

¿Cómo abordamos en la práctica problemas que en teoría son intratables?

Pablo Angulo[1]

M^{2}ASAI, Universidad Politécnica de Madrid,
ETSI Navales, Avda. de la Memoria, 4, 28040 Madrid.

Resumen

La **investigación operativa** es la rama de las matemáticas que se ocupa de resolver problemas de toma de decisiones, para optimizar procesos de cualquier tipo, aunque especialmente asociada a problemas de gestión y organización, como la creación de turnos de trabajo, problemas de logística o de gestión de inventarios. En este artículo pretendemos asomarnos a la pregunta: *¿cómo de difíciles son los problemas típicos de la investigación operativa?*... para ello tendremos que reflexionar sobre *qué significa* que un problema sea *difícil* `;-)`.

Presentaremos dos herramientas en apariencia muy parecidas para resolver problemas muy diversos de investigación operativa: la programación lineal con variables reales (*Linear Programming*: **LP**), y la programación lineal con variables mixtas (*Mixed Integer Linear Programming*: **MILP**).

Después definiremos los conceptos de **problema** y **algoritmo**, y las nociones de **complejidad** de un algoritmo y de un problema. Definiremos después unas pocas **clases de complejidad** de problemas, en concreto las clases `P`, `NP` y `NP-duro`. Este enfoque nos ayudará a entender que estas dos técnicas de resolución son radicalmente diferentes en cuanto al tipo de problemas que pueden acometer. Para terminar, nos preguntaremos qué podemos hacer en la práctica cuando la teoría nos dice que *¡nos enfrentamos a gigantes :-0!*

1. Programación lineal. El problema de la dieta.

Ya antes de la Segunda Guerra Mundial, varios matemáticos se habían planteado problemas de gestión y organización, usando enfoques muy diversos. Pero comenzaremos este artículo en ese momento de conflicto, que es cuando se perfila el enfoque que queremos abordar hoy.

Durante la guerra, la investigación de operaciones permitió optimizar parámetros como el tamaño de los convoyes de barcos, la profundidad a la que debían detonar las cargas de profundidad anti-submarino, el color del que se debían pintar los aviones, el tamaño de los escuadrones de bombarderos y la distancia a la que debían volar unos de otros, y muchas otras, para reducir las bajas aliadas y aumentar los daños infligidos al enemigo [**?**]. Y, lo más importante para el tema que nos ocupa, se descubrió que una gran cantidad de problemas se podían plantear como problemas de **programación lineal**. Los indiscutibles

[1] pablo.angulo@upm.es

éxitos obtenidos sugerían extender el enfoque a otros ámbitos. Acabada la guerra, el matemático George Dantzig se planteó el problema de alimentar a los soldados de forma que se satisficieran todos los requisitos nutricionales conocidos en el momento, pero de la forma más barata posible.

"La programación lineal puede verse como parte de un gran desarrollo revolucionario que ha dado a la humanidad la capacidad de establecer objetivos generales y trazar un camino de decisiones detalladas a tomar para lograr "mejor" sus objetivos cuando se enfrentan a situaciones prácticas de gran complejidad."

George Dantzig . 1936

Dantzig obtuvo su doctorado resolviendo dos problemas abiertos en Estadística que un profesor de George había dejado escritos en la pizarra. Corría el año 1939 cuando George llegó tarde a la clase y copió los problemas pensando que eran ejercicios para los estudiantes. Unos días después entregó las soluciones al profesor pidiendo disculpas por haber tardado tanto `:-)`. Dantzig planteó el problema de la dieta como un problema de optimización donde *todas las funciones que definen el problema son lineales*. Veámoslo más despacio:

Un problema de **optimización** es un problema en el que buscamos encontrar el conjunto de valores reales para ciertas incógnitas (que llamaremos *variables de decisión*) donde se alcanza el *valor óptimo* (puede ser el máximo global o el mínimo global) de una función f, pero respetando una cantidad arbitraria de *restricciones*. Todas las variables pueden tomar valores reales. Escribimos el problema en la forma:

$$\text{argmax}_{\mathbf{x}\in \text{RF}} f(x),$$

donde:

- El argmax de una función es el punto donde la función alcanza su valor *máximo*. Por ejemplo: $\text{argmax}_x(3-x^2) = 0$, mientras que $\max_x(3-x^2) = 3$. En el caso de la dieta, no nos interesa exclusivamente saber cuánto nos va a costar (el máximo de la función objetivo), sino qué cantidad tenemos que poner de cada alimento en la dieta propuesta.º
- El conjunto *RF* es la **región factible**: : el conjunto de posibles valores para las variables de decisión que satisfacen *todas* las restricciones.

Un problema de **programación lineal (LP)** es un problema de optimización en el que la función objetivo f es *lineal* [2], y la región factible está definida por una cantidad arbitraria de restricciones, que tienen que ser *todas*, o bien *ecuaciones lineales* de la variables de decisión (como por ejemplo $x - y = 1$), o bien *desigualdades lineales* de las variables de decisión (como por ejemplo $x + y \leq 1$). Todas las variables pueden tomar valores reales.

[2] es decir, f es un polinomio de grado 1 de sus argumentos, como por ejemplo $f(x,y) = 1 + x - 2y$

Un típico problema podría ser:

Una línea de ferry ofrece pasajes de clase turista y preferente. El operador obtiene 5 euros de beneficio por cada pasaje de preferente que vende y 2 euros por cada pasaje de turista, pero cada pasajero de clase preferente ocupa $3m^2$, mientras que un pasajero de turista ocupa sólo $2m^2$. Además, el número de pasajes de preferente vendidos debe ser menor que un tercio de los pasajes de turista.

- *¿Cuál es el beneficio máximo que puede obtener en un viaje un ferry, si la superficie total es de $36m^2$?*

He elegido un problema muy sencillo, que podrías resolver sin ninguna herramienta sofisticada, pero quiero que entiendas cómo se puede expresar como un problema LP.

Para plantear problemas de programación lineal, pasamos siempre por las mismas fases:

Definir las variables de decisión. Debemos elegir qué variables codifican la decisión que debemos tomar.

- p es la cantidad de plazas de clase *preferente*.
- t es la cantidad de plazas de clase *turista*.

Plantear las restricciones. Debemos traducir las restricciones técnicas a desigualdades lineales en función de las variables de decisión:

- *...la superficie total es de $36m^2$*: $3p + 2t \leq 36$
- *...El número de pasajes de preferente debe ser menor que un tercio de los pasajes de turista*: $p \leq t/3$
- por sentido común: $p \geq 0$, $t \geq 0$

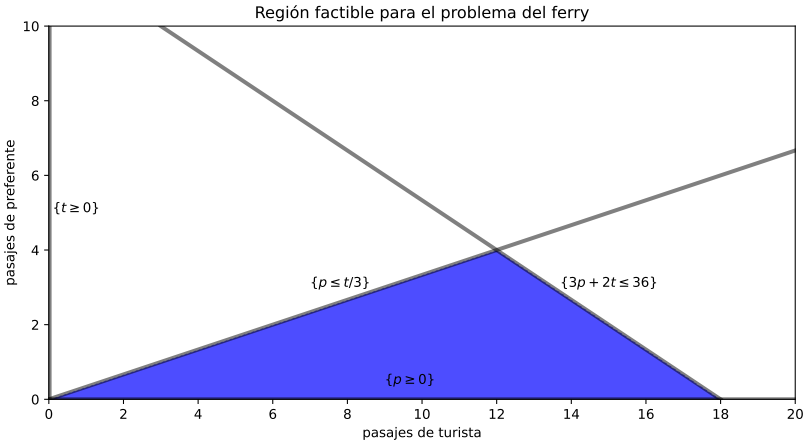


Figura 1: La región factible es la intersección de los semiplanos definidos por las restricciones.

Observa que cada desigualdad lineal define un *semiplano* del plano que contiene todos los posibles valores (p, t) para las variables de decisión, luego al exigir que se cumplan *todas* las restricciones, definimos siempre un conjunto *convexo* [3].

Elegir la función objetivo. En este caso, la empresa quiere *maximizar* el *beneficio*

$$\text{Objetivo:} \quad \text{argmax}_{p,t} 5\,p + 2\,t$$

Al ser un problema con sólo dos variables de decisión, podemos visualizar el resultado fácilmente en la figura **??**.

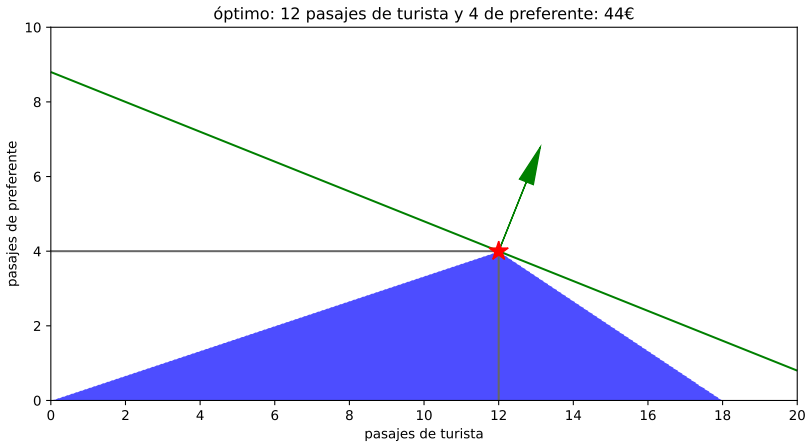


Figura 2: Solución gráfica del problema del Ferry

- La región azul es la **región factible**.
- La estrella roja marca el **óptimo** de la función objetivo.
- La línea verde representa el conjunto de todos los puntos del plano donde la función objetivo toma el mismo valor que en el óptimo de nuestro problema (que es $12 \cdot 2€ + 4 \cdot 5€ = 44€$). [4]
- La flecha verde es la dirección en la que la función objetivo aumenta más rápidamente. A esta dirección la llamamos el **gradiente** de la función.

Es fácil ver en la gráfica que no podemos mejorar la función objetivo sin salir de la región factible. La solución de 4 pasajes de clase preferente y 12 de turista es *óptima*.

El *problema de la dieta* de Dantzig es muy parecido:

Variables de decisión: La cantidad a incluir de cada uno de 70 posibles alimentos: $\{x_i\}_{i=1\ldots70}$

Restricciones: Se debe alcanzar la cantidad mínima recomendada de cada uno de los 7 nutrientes que un artículo de la época había identificado como esenciales.

[3] Un conjunto $C \subset \mathbf{R}^d$ es **convexo** si para dos puntos x, y cualesquiera de C, el segmento que une a esos dos puntos está contenido en C.

[4] Los conjuntos de puntos $N_c = \{x : f(x) = c\}$ donde la función f toma un mismo valor c se denominan **curvas de nivel** de la función.

Función objetivo: Minimizar el coste del menú.

Al tener 70 variables de decisión, ni la región factible ni el resultado óptimo se pueden visualizar, pero un ordenador puede resolver el problema igual que si fueran sólo 3 variables.

Si sólo tuviéramos 3 variables, x: coste de 1kg de brócoli cocido, y: coste de un litro de leche de vaca, z coste de un litro de zumo de naranja, con costes respectivos de $p_{\text{brocoli}}, p_{\text{leche}}, p_{\text{zumo}}$, la función objetivo, la que queremos minimizar, sería $p_{\text{brocoli}}x + p_{\text{leche}}y + p_{\text{zumo}}z$. El precio unitario de cada alimento se multiplica por la cantidad total de ese alimento en la dieta para obtener la cantidad de euros gastada en cada alimento, y se suma todo para obtener el coste total.

Con 70 alimentos, usamos la notación de *sumatorio* para escribir la función objetivo: $\sum_{i=1}^{70} p_i \cdot x_i$, donde p_i es el precio de una unidad del alimento i-ésimo [5]. La restricción de la vitamina C, por ejemplo, también es una desigualdad que involucra una *función lineal*:

$$\sum_{i=1}^{70} c_i \cdot x_i \geq \text{min vitamina C recomendado}$$

donde c_i es la cantidad de vitamina C por unidad del alimento i-ésimo.

Cada una de las restricciones dietéticas (proteínas, vitamina A, hierro...) tiene un aspecto similar, y corresponde a un *semiespacio* del espacio $\mathbb{R}^{70}$, así como las restricciones implícitas de que la cantidad de cualquier alimento debe ser *positiva*. Cuando imponemos *todas* las restricciones a la vez, la región factible resulta ser una intersección de semiespacios y es por tanto un subconjunto *convexo* del espacio de decisión, y la existencia del valor óptimo no sólo está garantizada de forma teórica, sino que no es difícil de encontrar si contamos con la ayuda de un ordenador programable.

Como este tipo de problemas de optimización era útil en muchos ámbitos, se despertó mucho interés en la búsqueda de algoritmos generales para resolver este tipo de problemas y de resultados teóricos sobre la cantidad de *recursos computacionales* que serían necesarios para resolver un problema de programación lineal dado.

Pero en el caso del problema de la dieta fue al revés: en un fabuloso ejemplo de que *"Cuando tienes un buen martillo, todos los problemas parecen clavos"*, G. Dantzig abordó el problema de la dieta para poner a prueba el **algoritmo del Simplex** [**?**] que acababa de desarrollar. No entraremos a explicar este algoritmo, ya que no es parte de la tarea que hemos propuesto para este artículo.

2. Y ahora con variables enteras

Si repetimos el problema del ferry, pero con un espacio de $34m^2$ en vez de $36m^2$, topamos con un inconveniente: la solución óptima es $34/9$ pasajeros de preferente y $34/3$ de turista, y las leyes marítimas impiden embarcar fracciones de pasajero ;-)

En la práctica, la solución es sencilla. Cualquier biblioteca de programación lineal permite declarar que ciertas variables sólo pueden tomar valores enteros, de modo que un valor de $34/9$ para el número de pasajeros de preferente sería inaceptable. Podemos ver gráficamente la solución al problema del ferry para esta versión del problema, pero esta vez con variables enteras, en la figura **??**.

Aunque pueda parecer un añadido inofensivo, exigir que algunas variables sean enteras es un cambio *muy* profundo. La cantidad de problemas de importancia real que se pueden

[5] Observa que el sumatorio $\sum_{i=1}^{70} p_i \cdot x_i = p_1x_1 + \cdots + p_{70}x_{70}$ recorre todos los alimentos posibles, pero si no incluimos en la dieta el alimento i-ésimo, entonces x_i es 0, y p_ix_i también es 0, así que su coste no contribuye al coste total de la dieta.

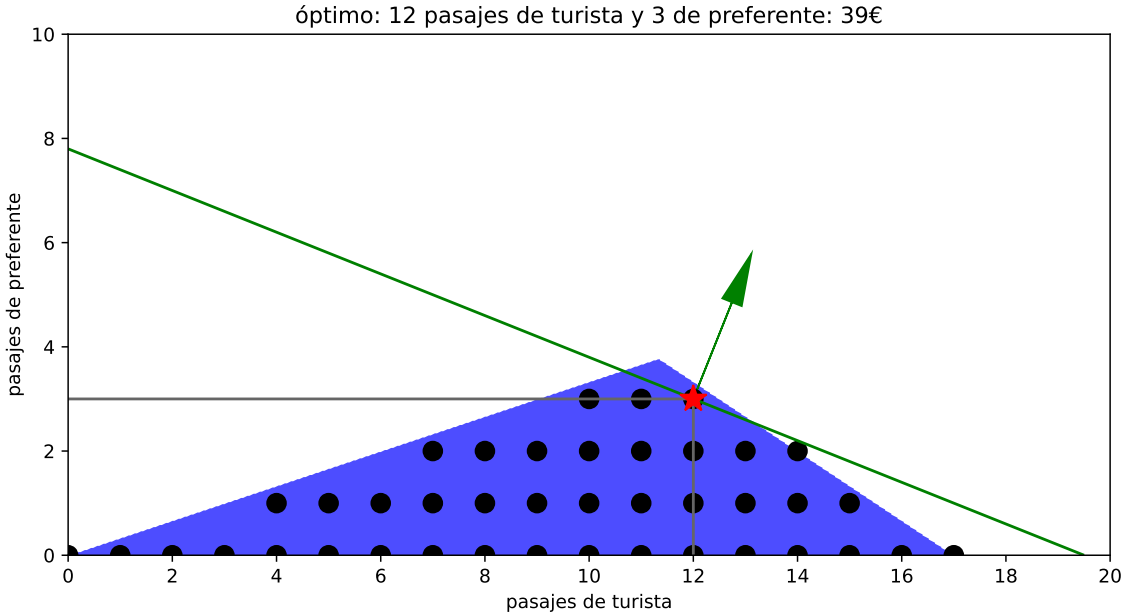


Figura 3: Solución gráfica del problema del Ferry con $34m^2$ de superficie

plantear como problemas **LP**, con variables de decisión reales, es grande, pero limitada. Pero si permitimos declarar que *algunas* variables *sólo* puedan tomar valores enteros (lo que en inglés se denomina **Mixed Integer Linear Programming**, o **MILP**), entonces sentimos la tentación de parafrasear la expresión anterior, y decir que *"Cuando tu martillo es **realmente bueno**, **muchos** problemas **son** clavos"*.

Figura 4: *Si Thor hubiera conocido el poder de MILP...*

Veamos otro ejemplo de problema que no se puede resolver como un **LP**, pero sí como un **MILP**: el sudoku. Debemos completar una cuadrícula 9×9 con dígitos entre 1 y 9 de forma que no se repita el mismo número en ninguna fila, en ninguna columna, y en ninguno de los cuadrados 3×3 marcados (figura **??**). Si alguna vez has intentado resolver uno, espero que estés de acuerdo en que algunos son *bastante* difíciles. Si una persona resuelve rápidamente un sudoku complicado, admiraremos su inteligencia. Simplemente probar todas las combinaciones y descartar las que violan alguna restricción no es práctico, ni siquiera para un ordenador.

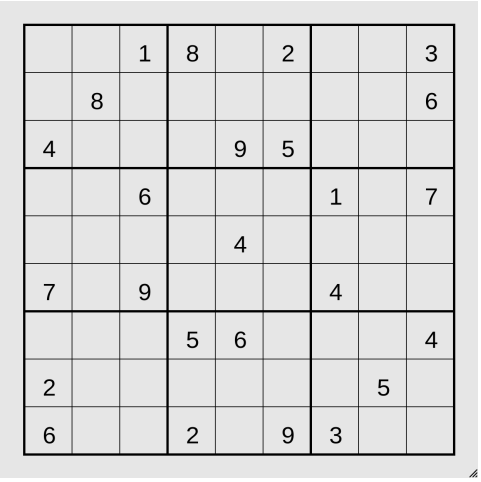


Figura 5: Conoces el Sudoku, ¿verdad?

Aunque en concreto para el sudoku hay algunos algoritmos sencillos que son bastante eficaces, vamos a plantearlo como un problema MILP, porque esos otros algoritmos no son tan generales como la metodología MILP, que puede resolver variantes del sudoku de relevancia práctica. En muchos institutos de secundaria, por ejemplo, la persona encargada de confeccionar los horarios del centro llama a esta tarea cariñosamente *hacer el sudoku*. No es difícil ver la analogía, ya que los horarios del instituto deben garantizar que ninguna persona está en dos sitios a la vez, que un aula no puede acoger a dos grupos a la vez, etcétera. Sin embargo, muchos algoritmos específicos para resolver sudokus no son fáciles de adaptar a estas variantes.

Una manera natural de almacenar un sudoku en la memoria del ordenador sería colocando números del 1 al 9 en una matriz 9×9, ¿verdad? Toda la información se almacena en la matriz $A = (a_{ij})_{i=1\ldots9, j=1\ldots9}$ de 9 filas y 9 columnas, donde cada número a_{ij} es un número del 1 al 9. Es una forma útil que sirve para algunos algoritmos, pero nos obliga a representar las restricciones como funciones no lineales. Por ejemplo, para decir que el dígito en la casilla $(1,1)$ debe ser diferente del dígito en la casilla $(1,2)$, decimos que $a_{11} \neq a_{12}$. Pero esta relación *no es ni una igualdad, ni una comparación entre funciones lineales* [6], así que no es aceptable para un problema MILP.

En vez de eso, en programas MILP es habitual usar *variables de decisión binarias*, que sólo pueden tomar los valores 0 y 1. Para el sudoku, definimos:

$$x_{ijk} = \begin{cases} 1 & \text{si en la fila i y columna j encontramos el dígito k} \\ 0 & \text{en otro caso} \end{cases}$$

Ahora es fácil incorporar las restricciones usando sólo igualdades y funciones lineales. La restricción de que, por ejemplo, en la primera fila el dígito 7 debe aparecer exactamente una vez, se escribe así:

$$x_{117} + x_{127} + x_{137} + x_{147} + x_{157} + x_{167} + x_{177} + x_{187} + x_{197} = 1$$

Como todas las variables sólo pueden tomar los valores 0 o 1, la identidad anterior garantiza que todas las variables de decisión x_{1j7} tomarán el valor 0, menos una, que tomará el valor 1.

[6] Observa que el subconjunto $\{x \neq y\}$ de $\mathbf{R}^2$ no es convexo, luego no se puede describir apilando desigualdades lineales...

Te animamos a intentar plantear de forma similar el resto de las restricciones del sudoku [7].

3. Algoritmos

Para poder responder a las preguntas que nos hemos planteado, necesitamos definir con cuidado qué es un algoritmo. Veamos primero unos ejemplos con los que a lo mejor ya te has encontrado antes...

Dados dos números n y m, podemos calcular su **máximo común divisor (mcd)** con la fórmula que dice:

$$mcd = p_1^{d_1} \cdot \dots \cdot p_f^{d_f}$$

donde $p_1, \dots, p_f$ son todos los números primos que dividen tanto a n como a m, y d_j es el mayor exponente posible tal que $p_j^{d_j}$ divide tanto a n como a m... Pero es una fórmula muy ineficiente, que nos obliga a factorizar completamente ambos números, cosa que, aunque no tiene ningún misterio, sí requiere realizar muchos cálculos.

No hay ninguna fórmula que permita escribir el mcd de dos números de forma compacta y eficiente, pero sí que existe un *algoritmo* capaz de obtener el **mcd** de dos números de forma muy eficaz, y sin necesidad de factorizarlos.

Algoritmo de Euclides. Dados dos números naturales $m \geq 1$ y $n \geq 1$:

- Lo cambiamos de orden, si fuera necesario, para que $n < m$
- Dividimos m entre n, y anotamos el resto d.
  - Si el resto es 0, entonces el **mcd** de m y n es n.
  - Si es distinto de 0, el **mcd** de m y n es igual al **mcd** de n y d.

Se trata de un algoritmo **recursivo**, ya que reduce el problema de calcular *un* mcd, el de m y n, al de calcular *otro* mcd, pero de dos números más pequeños, n y d. Es fácil probar que el algoritmo siempre termina, aunque no tanto estudiar cuántas iteraciones son necesarias `:-/`... Si no conocías el algoritmo, te animamos a probar con varios pares de números n y m. Mientras lo haces, recuerda que este algoritmo tan ingenioso, y que se sigue usando profusamente en los ordenadores hoy en día, tiene más de 2300 años.

Otro ejemplo que se enseña en las clases de 2º curso de Bachillerato es el **algoritmo de Gauss** para calcular determinantes. Si todavía no has estudiado este algoritmo puedes saltar esta parte y continuar desde la definición de algoritmo, más abajo. No entraremos ahora a describirlo en detalle, dado que se estudia en Bachillerato y no queremos alejarnos de nuestro propósito. Pero contrastemos dos formas de calcular un mismo determinante 4×4:

Aplicando la *fórmula* de Leibniz, es necesario sumar $4! = 24$ productos de cuatro elementos de la matriz:

$$\det(A) = |A| = \begin{array}{cccc} +a_{11}a_{22}a_{33}a_{44} & -a_{11}a_{22}a_{34}a_{43} & -a_{11}a_{23}a_{32}a_{44} & +a_{11}a_{23}a_{34}a_{42} \\ +a_{11}a_{24}a_{32}a_{43} & -a_{11}a_{24}a_{33}a_{42} & -a_{12}a_{21}a_{33}a_{44} & +a_{12}a_{21}a_{34}a_{43} \\ +a_{12}a_{23}a_{31}a_{44} & -a_{12}a_{23}a_{34}a_{41} & -a_{12}a_{24}a_{31}a_{43} & +a_{12}a_{24}a_{33}a_{41} \\ +a_{13}a_{21}a_{32}a_{44} & -a_{13}a_{21}a_{34}a_{42} & -a_{13}a_{22}a_{31}a_{44} & +a_{13}a_{22}a_{34}a_{41} \\ +a_{13}a_{24}a_{31}a_{42} & -a_{13}a_{24}a_{32}a_{41} & -a_{14}a_{21}a_{32}a_{43} & +a_{14}a_{21}a_{33}a_{42} \\ +a_{14}a_{22}a_{31}a_{43} & -a_{14}a_{22}a_{33}a_{41} & -a_{14}a_{23}a_{31}a_{42} & +a_{14}a_{23}a_{32}a_{41} \end{array}$$

[7] ¿Quieres probar la solución en directo? launch binder

El *algoritmo de Gauss* aplica *operaciones elementales* a las filas de la matriz de forma sistemática, hasta alcanzar siempre una matriz con el mismo determinante, pero triangular:

$$\begin{vmatrix} 1 & 0 & 0 & 2 \\ 1 & 2 & 0 & 7 \\ 0 & 0 & 3 & 7 \\ 4 & 0 & -3 & 0 \end{vmatrix} \xrightarrow[(4)-4\cdot(1)\to(4)]{(2)-(1)\to(2)} \begin{vmatrix} 1 & 0 & 0 & 2 \\ 0 & 2 & 0 & 5 \\ 0 & 0 & 3 & 7 \\ 0 & 0 & -3 & -8 \end{vmatrix} \xrightarrow[(4)+(3)\to(4)]{} \begin{vmatrix} 1 & 0 & 0 & 2 \\ 0 & 2 & 0 & 5 \\ 0 & 0 & 3 & 7 \\ 0 & 0 & 0 & -1 \end{vmatrix} \to 1\cdot 2\cdot 3\cdot(-1) = -6$$

Ambas formas de calcular un determinante tienen una gran relevancia teórica y desde ese punto de vista, ninguna es mejor que la otra: es importante conocer *ambas*. El algoritmo de Gauss tiene la "desventaja" de no tener una expresión cerrada en la que simplemente sustituir cada término por su valor, pero tiene otras ventajas. La más importante es que, como veremos, es *mucho* más eficiente.

Con estos ejemplos en mente, podemos dar una definición general:

Algoritmo: una secuencia de instrucciones, que puede incluir operaciones matemáticas, lógicas y de control de flujo (lo que serían las bifurcaciones `if...else` o los bucles `while`), que a partir de un *input* genera un *output*.

Tanto el input como el output pueden ser cualquier combinación de información numérica, textual y/o combinatoria (incluyendo matrices, grafos, textos...). Si tienes experiencia programando en cualquier *lenguaje de programación imperativa*, como `python`, `javascript` o `C`, reconocerás que un *algoritmo* se corresponde con una *función*, y tanto el *input* como el *output* pueden ser cualquier tipo de *estructura de datos* [8].

4. Complejidad de un problema computacional

Un **problema computacional** consiste en encontrar un algoritmo que construya el **output** deseado a partir de ciertos **inputs**. Por ejemplo, el algoritmo de Euclides resuelve el problema de: *"calcular el mcd de dos números naturales"*. Los inputs son esos dos números naturales y el output debe ser otro número natural: su *mcd*.

¿Cuándo decimos que un algoritmo tiene **complejidad alta**? Cuando se necesitan muchas operaciones para resolverlo o, más exactamente, cuando el número de operaciones que el algoritmo ejecuta crece muy rápido cuando el tamaño del input crece.

Por ejemplo, para calcular un determinante aplicando la regla de Leibniz se necesitan del orden de $n \cdot n!$ operaciones, donde n es la dimensión de la matriz cuadrada $n \times n$ [9]. En la asignatura de **Cálculo Numérico I** del Grado en Matemáticas de la UPM (**UPM-GeM**) estudiarás, que para calcular un determinante con el algoritmo de Gauss, el número de operaciones crece como un polinomio de grado 3, que es un crecimiento mucho más lento que $n\,n!$. Esta función, el número de operaciones en función del tamaño del problema, es la **complejidad del algoritmo**.

Para un mismo problema puede haber más de un algoritmo que lo resuelva. En esos casos decimos que la **complejidad del problema** es la *complejidad* del algoritmo que resuelve el problema con la menor complejidad. Se dice que un problema está en la **clase de complejidad** `P` si su complejidad está acotada por un polinomio.

[8] En otros paradigmas de programación, un programa de ordenador no se corresponde con un *algoritmo*, sino con un *problema*...

[9] El número exacto depende de varios detalles, como el mecanismo que usamos para recorrer las permutaciones de n números de forma eficiente...

La complejidad de un algoritmo no tiene mucho que ver con si es fácil de implementar en un ordenador, y la complejidad de un problema no tiene mucho que ver con si es fácil o difícil entender el problema, formularlo como un problema matemático, ni con si es fácil encontrar un algoritmo para resolverlo. En la asignatura de **Álgebra Lineal** del **UPM-GeM** verás que la fórmula de Leibniz, por ejemplo, es muy fácil de derivar, y que es una gran ayuda teórica, que sirve para demostrar propiedades importantes de los determinantes y los volúmenes, pero, como algoritmo para calcular determinantes, tiene una gran complejidad.

Calcular la complejidad de un algoritmo no es especialmente complicado: se trata de contar el número de operaciones y anotar el número de operaciones para la medida adecuada del tamaño del input. Pero calcular la complejidad de un problema puede ser muy difícil: hay que pensar en *todos los posibles algoritmos que lo resuelven*. Hay veces que no conseguimos demostrar cuál es la complejidad de un problema, pero a pesar de ello conseguimos encontrar una relación con la complejidad de otro. Por ejemplo, sabemos que el problema "LI" de *"dados n vectores de $\mathbb{R}^n$, determinar si son linealmente independientes"* está en P, y tiene complejidad a lo sumo n^3. Lo sabemos porque podemos **reducir** el problema LI al problema "D" de calcular determinantes: n vectores de $\mathbb{R}^n$ son linealmente independientes si y sólo si el determinante de esos n vectores es distinto de 0. El coste de esa reducción es ínfimo: sólo hay que componer la matriz con las coordenadas de los vectores. Después llamamos al algoritmo de Gauss para determinantes, y después con una única comparación contra el valor fijo 0, obtenemos la respuesta a la pregunta original.

- ¡Ey, tú! ¿Te gustan las matemáticas? Dime entonces: si tienes una olla vacía, un grifo, y un fogón ¿cómo consigues agua hirviendo?
- ¡Fácil! Lleno la olla en el grifo y la caliento en el fogón.
- ok, pero ¿y si la olla ya está llena de agua?
- ¡Fácil también! Tiro el agua, y ...
  ¡lo he **reducido** al problema anterior! [**?**]

5. El problema de *verificación*. Problemas P y NP.

Para algunos problemas, se necesitan muchas operaciones para encontrar la solución, pero es fácil evaluar si un candidato es realmente una solución. Dado un número n grande, encontrar sus factores primos es muy laborioso. Dados varios factores, es fácil multiplicarlos y comprobar si su producto es n.

Dado un problema F, definimos el **problema de verificación** NF como el problema cuyo input es doble:

- Un input x válido para F
- Un candidato y a solución para $F(x)$

y cuyo output es

- 1 si $y = F(x)$
- 0 en otro caso

Veamos un ejemplo: el **Problema del Viajante**, en su variante de *optimización*, pregunta cuál es la forma de recorrer n ciudades en un viaje, de forma que la distancia total recorrida sea la menor posible [**?**].

Tomemos como F la variante de *decisión* de este problema: nos dan una lista de n ciudades $c_1, \ldots, c_n$, las distancias $\{d_{ij}\}$ entre cualquier par de ciudades c_i y c_j, y una cantidad total de kilómetros D. Nos piden encontrar una ordenación de las ciudades de modo que si las recorremos en ese orden, la distancia total recorrida sea menor que C. No parece fácil, ¿verdad? Hay $n!$ posibles ordenaciones de las ciudades, y ningún algoritmo obvio que nos garantice encontrar la mejor de estas ordenaciones `:-/`.

¿Cómo es el problema de verificación VF para el problema anterior? Nos dan la misma información que antes, pero además nos dan una ordenación de las ciudades, y *sólo* tenemos que decir si esa ordenación concreta consigue una distancia total menor que D. Basta con sumar las distancias entre cada par de ciudades de la ordenación[10]...

Hay bastantes problemas complicados cuyo problema de verificación, sin embargo, es sencillo: Se dice que un problema F es NP si el problema VF de *verificación* de F se puede resolver con un algoritmo de complejidad polinómica.

Es obvio que si un problema F está en la clase P, entonces VF también lo está, porque podemos resolver el problema de verificación con una complejidad polinómica apoyándonos en un algoritmo A con complejidad polinómica para resolver F (es decir, **reducimos** el problema VF al problema F):

- Dados x e y, calcula $F(x)$ usando el algoritmo A.
- En una única operación de comparación, decide si y es igual a $F(x)$.

En términos de las clases de complejidad, este argumento demuestra que P$\subset$NP.

6. ¿P=NP?

A finales de los 60 y principios de los 70 del siglo pasado se sentaron las bases de la moderna teoría de la complejidad, en paralelo en Estados Unidos y la Unión Soviética. Stephen Cook, en Toronto, y Leonid Levin, en Moscú, plantearon en su forma moderna el problema de si todos los problemas cuyo problema de verificación asociado se puede resolver con un algoritmo de complejidad polinómica se pueden resolver con un algoritmo de complejidad polinómica. En términos de las clases de complejidad, se preguntaron si P=NP.

El problema sigue abierto, después de más 50 años, un premio de un millón de dólares a quien lo resuelva y mucho, mucho esfuerzo. Además, actualmente la mayoría de expertos piensan que estamos muy lejos de encontrar una demostración [**?**]. Sin embargo, ya en los años 70, se acumuló una gran evidencia a favor de la conjetura de que P$\neq$NP.

El primer avance fue la demostración de que cualquier problema NP se puede reducir a un problema NP muy específico, el SAT (*problema de satisfacibilidad booleana* [**?**]). Además, el número de operaciones necesarias para hacer esta reducción está acotado por un polinomio. Por lo tanto, si alguien encontrase un algoritmo con complejidad polinómica capaz de resolver este problema, quedaría demostrado que P=NP, ya que *todos* los problemas de la clase NP se podrían resolver con complejidad polinómica via reducción a SAT.

[10] Un *detalle* importante: sólo podemos verificar una ordenación de ciudades. Si alguien afirma que es imposible recorrer las n ciudades en menos de D kilómetros, no podremos verificarlo fácilmente. Pero la definición de NP acepta esta posibilidad: sólo exige que se puedan verificar en tiempo polinómico las soluciones *positivas*.

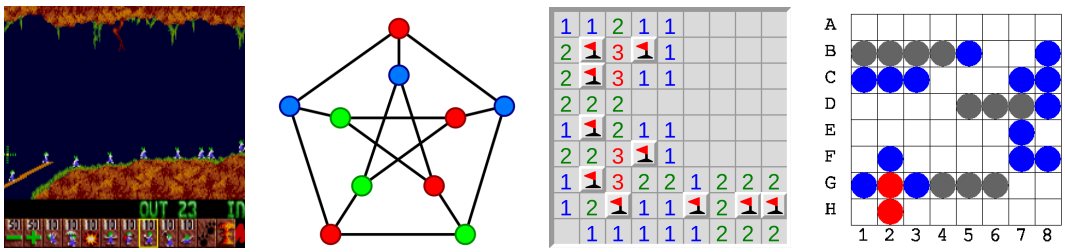
Figura 6: Stephen Cook (1968) y Leonid Levin (1993)

Además, el problema SAT es un problema NP, luego la pregunta de si P=NP es *completamente equivalente* a la pregunta de si SAT (un problema NP concreto) está en P. Aunque este resultado parece progreso en la dirección de demostrar que P=NP, veremos pronto que es *justo al revés* :-O.

El teorema de Cook y Levin motiva dos definiciones:

- Si cualquier problema NP se puede reducir a un mismo problema F en tiempo polinómico, decimos que F es **NP-duro**.
- Si F es a la vez NP y NP-duro, decimos que es NP-completo.

Al problema SAT le siguieron *muchos* otros. En 1972, Richard Karp publicó una lista con 21 problemas bastante conocidos, demostrando que todos ellos eran también NP-completos. ¡Y a esa lista siguieron otras muchas! La lista de problemas NP-completos incluye problemas derivados del Tetris, el juego de Lemmings, el sudoku de tamaño $n^2 \times n^2$, el juego de los barquitos, el problema de colorear un grafo con el mínimo número posible de colores y muchos otros puzzles y problemas matemáticos de todo tipo [**?**].



7. El Problema de Mochila

Personalmente, el problema que creo que mejor ejemplifica la sorprendente ubicuidad de los problemas NP-duros es el **problema de mochila**. Hay varias versiones, y no todas son problemas NP, aunque todas son problemas NP-duros. En el artículo *Criptografía*, en el número 1 de esta misma revista [**?**], Jorge J. Urroz mencionó una variante NP-completa, y mencionó que dilucidar si P=NP es uno de los **"Problemas del Milenio"** [**?**].

Presentamos otra variante NP-completa, que llamaremos *problema de decisión de mochila*. Nos preguntamos si podemos llenar una mochila de objetos de un bazar de forma que el precio de los objetos sea al menos un número O dado:

- Inputs
  - Una lista $\{v_1, \ldots, v_n\}$, con el volumen de cada objeto de una lista de n objetos.
  - Una lista con el precio $\{p_1, \ldots, p_n\}$ de una unidad de cada uno de los objetos de la lista.
  - El volumen V de nuestra mochila.
  - Un objetivo total O.
- Las variables de decisión son números naturales: x_i es la cantidad de unidades del objeto i-ésimo que tomamos.
- Nos preguntamos si es posible llenar la mochila con objetos de tal forma que
  - los objetos elegidos quepan en la mochila: $\sum_i v_i x_i \leq V$.
  - su precio combinado sea al menos O: $\sum p_i x_i \geq O$
- En caso afirmativo, devolvemos `True`, y en otro caso, `False`.

Este problema es claramente `NP`: si nos dan una elección o, lo que es equivalente, una lista de n valores $\{x_1, \ldots, x_n\}$ enteros, se puede *verificar* si se cumplen las dos condiciones con un algoritmo obvio (aplicar la fórmula) que tiene complejidad *lineal*.

Desde el punto de vista de la programación lineal con la que comenzamos el artículo, quizá te preguntes por qué no intentamos *maximizar* el precio combinado en vez de preguntarnos si podemos alcanzar la cota O. El único motivo es que esa versión del problema de mochila, que llamaremos *problema de la mochila óptima* no es `NP`, y por tanto no es `NP-completo`.

Figura 7: Un experto en el Problema de Mochila

¿Por qué me sorprende tanto que este problema sea `NP-completo`? Porque la versión **"relajada"** del problema, en la que permitimos que las cantidades x_i sean números reales, es increíblemente sencilla [11].

Resolvamos primero el problema de la mochila óptima: una unidad del objeto i-ésimo ocupa un volumen v_i y tiene un precio p_i, lo que da lugar a un *precio unitario* $r_i = p_i/v_i$. Llamemos i^* al índice del objeto con el precio unitario más alto [12]:

$$i^* = \text{argmax}_i \, r_i$$

[11] La versión relajada del problema de mochila óptima resuelve el problema de llenar la mochila consiguiendo el valor máximo posible en una tienda de productos *a granel*.

[12] Asumimos por simplicidad que el máximo se alcanza en un sólo objeto, pero el resultado es *muy* similar si se alcanza en más de uno. Te proponemos como ejercicio que lo pienses, y que te asomes a la cuestión, mucho más interesante, de encontrar *todas* las soluciones óptimas en el caso de que dos o más objetos alcancen el mismo ratio óptimo.

Si cualquier x_i es positivo, para $i \neq i^*$, podemos encontrar una solución factible con un valor mayor de la función objetivo cambiando x_i por 0, y sumando $x_i v_i / v_{i^*}$ a x_{i^*}. Es decir, quitamos todo lo que habíamos cogido del objeto i-ésimo, y ocupamos el espacio liberado con más cantidad del objeto i^*-ésimo:

- El volumen de la mochila disminuye en $v_i x_i$ por quitar x_i unidades del objeto i, y aumenta en $v_{i^*} x_i v_i / v_{i*} = x_i v_i$, luego no varía.
- El precio total de los objetos de la mochila disminuye en $p_i x_i$ y aumenta en $p_{i^*} x_i v_i / v_{i^*} = p_i x_i r_{i^*} / r_i$, luego mejora, porque $r_{i^*} / r_i > 1$.

También es obvio que si la mochila no está llena, podemos conseguir un precio mayor llenando la fracción de mochila disponible, lo que siempre es posible cuando se permite que los x_i tomen valores reales.

De aquí se deduce que la única solución que no puede ser mejorada (la solución *óptima*) es aquella en la que todos los x_i son 0 cuando $i \neq i^*$, y x_{i^*} toma el valor máximo posible de V / v_{i^*}.

Con este algoritmo para el problema de mochila óptima es muy fácil resolver el problema de decisión de la mochila... Te lo dejamos como ejercicio `;-)`.

Detengámonos a contrastar la diferencia entre el problema de mochila relajado, que se puede resolver con un algoritmo de complejidad *lineal*, con el problema de mochila, que es `NP-duro` en cualquiera de sus variantes. Si existiera un algoritmo de complejidad no ya lineal, sino polinómica, de cualquier grado, que pudiera resolver el problema de mochila, entonces sería posible resolver *todos* los problemas `NP` con complejidad polinómica. No hay ninguna contradicción lógica en pensar que esto sea posible, pero todos los problemas de la larga lista de problemas `NP-duros` se han intentado resolver durante muchas décadas, de muchas formas distintas, por muchas personas distintas, y nadie, nunca, ni con búsqueda asistida por ordenador, ni con los modernos modelos de lenguaje, ha podido conseguir un algoritmo de complejidad polinómica para *ninguno* de ellos. De hecho, el problema de mochila se ha planteado como base de un sistema de cifrado, y si cualquier persona encontrase un algoritmo para resolverlo *con complejidad baja*, el cifrado no funcionaría.

Al igual que los problemas de horarios son variantes del sudoku, los problemas que vimos al principio del ferry, o de la dieta, y muchos otros problemas de gestión de almacenes son muy parecidos al problema de mochila, y es fácil representarlos como problemas MILP.

8. Problemas `NP-duros` como MILP

Algo similar a lo que ocurre con el problema de la mochila ocurre con la programación lineal: hay un algoritmo capaz de resolver *cualquier* problema LP con complejidad polinómica, pero la mayoría de los expertos piensan que no hay ningún algoritmo capaz de resolver *cualquier* problema MILP en tiempo polinómico [13]. Ya hemos visto cómo reducir el problema de mochila a un problema de programación lineal (LP o MILP, según permitamos a las variables tomar valores reales, o no). Por tanto, el hecho de que *LP está en P, pero MILP es `NP-duro`* no debería sorprendernos.

Ya hemos visto que es fácil plantear problemas *tipo sudoku* o *tipo mochila* como MILP. Aunque es un poco más engorroso, también se puede plantear el Problema del Viajante [**?**]. Y lo más importante, todos estos problemas se pueden combinar, para preguntarnos por

[13] Curiosamente, no los algoritmos más populares, como el simplex de G. Dantzig, que no ofrecen garantías teóricas de éxito, pero sí funcionan bien en la práctica.

ejemplo cómo orquestar una flota de camiones de reparto para que todos los paquetes lleguen a su destino en la fecha requerida, con el mínimo coste, y respetando las limitaciones de tamaño de cada camión.

A modo de ejemplo, puedes probar varios puzzles lógicos de la excelente colección de Simon Tatham [**?**], codificados como problemas MILP: launch binder .

9. ¿Cómo podemos afrontar los problemas típicos de la investigación operativa?

Resumamos la situación:

- Casi todos los problemas de investigación operativa (gestión de flotas de vehículos, de almacenes, de centros de trabajo...) contienen como subproblemas problemas de tipo sudoku, mochila o viajante.
- Por tanto, no hay ninguna garantía teórica de que esos problemas se puedan resolver en un tiempo razonable.
- Hay una gran evidencia empírica que apoya la conjetura P≠NP y, si es cierta, no habrá nunca ninguna garantía de ese tipo.

No parece una buena situación, ¿verdad?
Pero en la práctica la situación no es tan mala como parece:

- No sólo es *posible*, sino *fácil*, reducir muchos problemas NP-duros a problemas MILP, como aprenderás en la asignatura de **Investigación Operativa** de **UPM-GeM**.
- Si planteas un problema como $MILP$, puedes intentar resolverlo con el software para $MILP$ ya existente. Es decir: ¡*no necesitas desarrollar un algoritmo para tu problema computacional concreto* :-D!
- Aunque no tengamos garantías teóricas, muchos problemas de gran tamaño se pueden resolver en la práctica con variantes del algoritmo del simplex que mencionamos antes [14].
- El software para resolver problemas $MILP$ ha acumulado *décadas* de experiencia tratando de buscar estrategias que resuelven *en un tiempo razonable* muchos patrones habituales en la práctica.
- Una vez planteado un problema como $MILP$, es *muy fácil* cambiar entre varias bibliotecas de software para $MILP$ que usan estrategias completamente diferentes.
- Con ese mismo esfuerzo de modelización, también puedes probar software que ni siquiera es para problemas $MILP$, sino para problemas más generales como programación cuadrática (**Quadratic Programming: QP**) o programación con restricciones (**Constraint Programming: CP**), o de **búsqueda heurística**, que no aspira a encontrar la solución óptima, o al menos no aspira a demostrar que es óptima, sino únicamente a buscar una solución *suficientemente buena*. Para problemas especialmente grandes, este último enfoque es muy habitual en la industria: aunque se pierden las garantías de que la solución alcance el valor máximo para la función objetivo, se siguen obteniendo soluciones que verifican *todas las restricciones*.

[14] ...aunque a veces hay que probar varias formulaciones para ver cuál funciona mejor :-/

Por ejemplo, veamos cómo introducir en un ordenador el problema del ferry del comienzo de este artículo [15]:

```
p = Variable("p",lb=0)
t = Variable("t",lb=0)
constraints = [
    Constraint(3*p+2*t, ub=36),
    Constraint(p-t/3, ub=0)
]
obj = Objective(5*p+2*t, direction="max")

model = Model(name="My model")
model.objective = obj
model.add(constraints)

status = model.optimize()
```

Después de cursar la asignatura de **Informática**, no te costará mucho aprender a escribir programas que resuelvan problemas de tamaño arbitrario.

10. Conclusiones: Matemáticas y realidad

En este artículo nos hemos acercado a una metodología que se usa mucho en la práctica para resolver problemas de lo más variopinto. Podría parecer que la *teoría*, en este caso, se queda corta, pero en realidad aporta conocimiento muy valioso.

Por ejemplo, cuando el software de MILP termina su ejecución y dice que ha encontrado el óptimo, tenemos la *certeza* de que es así, cosa que no ocurre si usamos técnicas de *machine learning*.

Si el algoritmo termina, pero el software para MILP reporta que un problema es **inviable**, está garantizado que *no hay* ninguna solución. No es sólo que no hayamos alcanzado el óptimo, sino que la región factible es vacía. Reflexionemos un momento sobre las consecuencias: si has buscado una solución mediante machine learning o heurísticas, y no has tenido éxito, podría ser que no hayas esperado suficiente tiempo, o que no hayas elegido el algoritmo adecuado. Cuando un software MILP, *cualquier* software MILP, dice que el problema es *inviable*, tenemos la *garantía* de que *no se puede resolver*, y hay que pensar si realmente *todas* las restricciones son innegociables...

Por último, si reconocemos que nuestro problema contiene un subproblema `NP-duro`, sabemos, *si somos humildes* `;-)`, que no debemos aspirar a encontrar un algoritmo determinista con garantía de éxito. Debemos usar nuestro tiempo de diseño y las capacidades de cómputo a nuestra disposición de la manera más *práctica* posible, quizá probando varias estrategias con versiones más pequeñas de nuestros problemas para evaluar cuáles tienen mayores probabilidades de éxito. Puedes pensar en esta teoría de los problemas `NP-duros` como en un análogo matemático de las leyes de la Termodinámica, que han evitado a la humanidad enormes gastos de tiempo y dinero en *búsquedas infructuosas* al demostrar que, por ejemplo, el móvil perpetuo es *imposible* [**?**].

Sin embargo, es posible que necesitemos otro enfoque para estudiar la complejidad. Por un lado, aunque en matemáticas hemos dedicado mucho tiempo a estudiar las clases de

[15] Hemos usado la biblioteca `optlang` de *modelado* de programación lineal para el lenguaje de programación `python`

complejidad `P`, `NP`, y `NP-duro`, reconocemos que saber que un problema está en `P` no implica necesariamente que se pueda resolver en poco tiempo. Un algoritmo polinómico podría crecer también muy rápido, hasta el punto de no ser usable en la práctica. Y, por otro lado, sí parece razonable pensar que para muchos problemas `NP-duros`, aunque no exista ningún algoritmo que pueda resolver el problema para *todos* los inputs en tiempo polinómico, sí sea posible resolver el problema para *la mayoría de los* inputs en tiempo polinómico. No cabe duda de que estudiar la complejidad de los problemas computacionales es uno de los **Problemas del Milenio** [**?**].

11. Epílogo: ¿y la IA qué opina de todo esto?

Quizá en algún momento leyendo este artículo te hayas preguntado qué pueden hacer algoritmos del siglo pasado en la era de los grandes **modelos de lenguaje**. Si has preguntado a algún modelo de lenguaje, sé que te ha dicho que la metodología sigue vigente [16]. Los modelos de lenguaje usan billones de parámetros para responder a prompts de texto con unos pocos elementos, y de vez en cuándo **alucinan** [**?**]. Plantear un problema de investigación operativa como un problema $MILP$ permite obtener una solución con garantías y de forma más eficiente o, si el problema es inviable, obtener un certificado de que no tiene solución. También son modelos más interpretables y permiten extraer conclusiones adicionales, como cuáles de las restricciones son las más limitantes. Y además, ¡sin que te hagan la pelota!

Aunque por supuesto, a día de hoy no es raro usar un modelo de lenguaje para que nos ayude a programar un programa $MILP$. Yo, por ejemplo, he usado una IA para migrar mis cuadernos de "Puzzles Lógicos con $MILP$" a otra biblioteca de software, para que los podáis probar con launch binder.

Agradecimientos

Muchas gracias a Alicia Cantón, Leonardo Fernández, Antonio Sevilla y David Sanz por sus comentarios.

Bibliografía

[1] `https://blogs.upm.es/retazos-matematicas/criptografia/`

[2] `https://blogs.upm.es/retazos-matematicas/como-aprende-a-hablar-una-maquina/`

[3] Folklore matemático `;-)`

[4] `https://es.wikipedia.org/wiki/Veinti%C3%BAn_problemas_NP-completos_de_Karp`

[5] `https://es.wikipedia.org/wiki/Problemas_del_milenio`

[6] Colección de Puzzles de Simon Tatham: `https://www.chiark.greenend.org.uk/~sgtatham/puzzles/`

[7] ¿Quieres aprender a resolver el problema del viajante como un problema MILP? launch binder.

[8] `https://es.wikipedia.org/wiki/Problema_de_satisfacibilidad_booleana`

[9] `https://es.wikipedia.org/wiki/Criptoaritmo`

[10] `https://en.wikipedia.org/wiki/List_of_NP-complete_problems`

[16] Lo sé porque si no, no habrías seguido leyendo `;-)`

[11] https://es.wikipedia.org/wiki/M%C3%B3vil_perpetuo

[12] https://en.wikipedia.org/wiki/Operations_research#History

[13] https://en.wikipedia.org/wiki/P_versus_NP_problem#Results_about_difficulty_of_proof

[14] https://en.wikipedia.org/wiki/Simplex_algorithm