Universidad de Oviedo

Grado en Matemáticas

Una introducción académica a la complejidad algorítmica

Trabajo Fin de Grado

Autor:
Jorge Rubio Suárez
Tutor:
César Luis Alonso González

Junio 2023

Índice

1. Introducción

Desde su inicio, el objetivo de las matemáticas ha sido trabajar con problemas en busca de una solución. No siempre el resultado ha sido completamente satisfactorio, ya que algunos siguen abiertos. Pese a esto, los avances realizados eran inimaginables hace unos siglos y ello ha hecho que nuestro modo de vida cambie por completo. Hoy en día, problemas que hace un tiempo parecían imposibles los solucionamos en segundos y gran culpa de esto lo tiene la rama de las matemáticas sobre la que vamos a tratar: la matemática computacional.

Este área cada vez tiene más importancia en nuestro día a día. Su objetivo es la búsqueda de medios mediante los cuales conseguir una resolución automática de un problema. Para hacerlo, necesitaremos dispositivos que nos permitan hacer este proceso: las máquinas. Una máquina (al menos las que conocemos) no dispone de capacidad de pensar, sólo dispone de memoria y la capacidad de ejecutar pasos previamente descritos. Por tanto, sólo podremos solucionar un problema si tenemos una serie de pasos sencillos que solucionan el problema, que formarán un algoritmo.

Es por esto que lo que aquí tratamos tiene una especial relevancia en nuestro modo de vida actualmente. En estos días, los algoritmos aparecen en cualquier campo que nos podamos imaginar. En la inteligencia artificial, tan de moda ahora con el ChatGPT, en la economía, la meteorología, la ciencia. Es un sinfín de utilidades las que tienen ahora mismo en nuestro mundo. Cualquier avance en el estudio de ellos supondría un cambio en nuestro modo de vida, como ya lo ha hecho en los últimos años el espectacular progreso de la informática. Es por ello que nos parece un tema clave que nos interesa analizar y estudiar aunque sea desde una perspectiva muy básica.

Creemos que el futuro de nuestras vidas tendrá que ver en gran parte en cómo evolucione el campo de la complejidad algorítmica. Nuestro papel aquí va a ser el de explicar de una manera básica cómo funcionan los algoritmos según el tipo de problema al que nos enfrentamos y clasificarlos en diferentes

clases de complejidad detallando qué características tiene cada una.

Aunque el título sea Una introducción académica a la complejidad algorítmica, nos centraremos especialmente en la complejidad del tiempo, es decir, en cuántos pasos nos requiere un algoritmo para solucionar un problema. Comenzaremos en el capítulo 1 explicando una serie de conceptos sobre los que nos basaremos durante el resto del trabajo. Utilizaremos las máquinas de Turing como modelo de computación, las cuales explicaremos con detalle en el capítulo 2.

Los capítulos 3 y 4 los dedicaremos a explicar de qué trata la complejidad algorítmica y definiendo también algún que otro concepto, como puede ser la reducibilidad de problemas, hasta llegar a las clases P y NP; claves a la hora de hablar de complejidad algorítmica. Finalmente, en el capítulo 5, pondremos la guinda con la conjetura de Cook, ¿P=NP?, intentando descifrar qué supone y viendo cuál de las respuestas es más probable.

El objetivo de cara al lector es el de explicar y mostrar la gran importancia que tiene la teoría de la complejidad algorítmica en los problemas que a día de hoy todavía no están solucionados y que pueden tener gran relevancia en el futuro.

Antes de comenzar, vamos a ver dos ejemplos que nos van a empezar a mostrar de lo que vamos a tratar:

Ejemplo 1 Consideramos el problema de multiplicar dos números n y m. Como vimos en la escuela, hay dos formas fáciles de hacerlo: sumando n m veces, o aplicando el algoritmo clásico. Por ejemplo, si queremos multiplicar 13 y 15, de la primera manera haríamos 13 sumas como mínimo:

$$15 \times 13 = 15 + 15 + 15 + 15 + 15 + 15 + 15 + 15 + 15 + 15 + 15 + 15 + 15$$

Con el algoritmo clásico haríamos dos multiplicaciones y una suma:

$$
\begin{array}{rcc}
 & 1 & 5 \\
\times & 1 & 3 \\
\hline
 & 4 & 5 \\
1 & 5 & \\
\hline
1 & 9 & 5
\end{array}
$$

Parece que hemos encontrado un algoritmo que nos ha hecho hallar una solución de una manera más rápida, más eficiente.

Ejemplo 2 (Problema del viajante) Consideremos ahora una persona que tiene que viajar por 35 ciudades de España peninsular y quiere intentar recorrer el mínimo número de distancia para poder ahorrar costes. Supongamos que puede viajar en línea recta de ciudad a ciudad para simplificar el problema pero sólo puede para una vez por cada ciudad. El algoritmo que necesitaríamos para poder solucionar este problema sería en visitar las 35 ciudades y sumar sus distancias entre cada una y la siguiente. Por tanto, tendríamos 35 posibilidades para la primera ciudad, 34 para la segunda, 33 para la tercera y así sucesivamente. Es decir, tendríamos un total de 35! itinerarios. Supongamos en este caso que la ciudad de partida y de llegada sea la misma, entonces tenemos 34! = 295232799039604140847618609643520000000 itinerarios. Suponiendo que tuviésemos un ordenador que pudiese procesar una ruta en 10^{-15} segundos, tardaría $9 * 10^{15}$ años en procesarlas todas. Es decir, la humanidad no sobreviviría para ver la solución al problema.

Con estos dos ejemplos podemos ver la eficiencia con la que dos algoritmos resuelven un mismo problema y la dificultad en algunos problemas para encontrar un algoritmo que sea eficaz. Como veremos, algunos problemas son más 'sencillos' que otros y y tendremos que estudiarlos y clasificarlos en clases de complejidad. Esto es lo que haremos a lo largo del texto.

2. Conceptos preliminares

En este capítulo vamos a definir una serie de conceptos y notaciones que serán necesarias durante el resto del documento.

2.1. Conjuntos, relaciones y funciones

Conjuntos Un conjunto es una serie de elementos. Ningún elemento tiene más de una copia y el orden de los elementos no importa. Si los elementos a, b y c pertenecen a un conjunto A se representa $A = \{a, b, c\}$. Dos conjuntos A y B son iguales si todo elemento de A está en B y viceversa. Lo denotamos como $A = B$.

Los símbolos $\in$ y $\notin$ denotan, repectivamente, si un elemento pertenece o no pertenece a un conjunto. Utilizando el ejemplo anterior, $a \in A$ pero $d \notin A$.

Un conjunto unitario es un conjunto que tiene un solo elemento. Un conjunto es vacío si no tiene elementos y lo representamos $A = \emptyset$. Si un conjunto tiene un número finito de elementos lo llamamos conjunto finito; en otro caso, lo llamamos conjunto infinito. Algunos ejemplos de conjuntos infinitos son los números naturales $\mathbb{N}$, los números enteros $\mathbb{Z}$ y los números reales $\mathbb{R}$.

En los conjuntos infinitos como no podemos escribir todos los elementos del conjuntos podemos expresar el conjunto en base a otro conjunto ya conocido y una propiedad. Dado un conjunto X y una propiedad π, podemos definir un conjunto Y como $Y = \{y \in Y : y \in X, y \ satisface \ \pi\}$.

Un conjunto B es un subconjunto de un conjunto A si cada elemento de B está en A. Se representa $A \subseteq B$.

Conectores y cuantificadores Utilizaremos la siguiente notación para los conectores: El 'y' se representa con $\wedge$, el 'o' se representa con $\vee$, el 'solo si' $\rightarrow$ y el 'no' con $\neg$.

$\exists$ y $\forall$ son cuantificadores que significan 'existe' y 'para todo', respectivamente.

Operaciones en conjuntos Dados dos conjuntos A y B, definimos las siguientes operaciones:

La unión es el conjunto de elementos que pertenecen tanto a A como a B, lo denotamos: $A \cup B = \{x : x \in A \vee x \in B\}$. La intersección es el conjunto de elementos comunes a A y B y lo denotamos $A \cap B = \{x : x \in A \wedge x \in B\}$. El conjunto diferencia de A menos B se define como los elementos de A quitando los de B: $A - B = \{x : x \in A \wedge x \notin B\}$. Por último, el conjunto de diferencia simétrica de A y B es el conjunto de elementos que están en A pero no en B o que están en B pero no en A: $A \triangle B = (A - B) \cup (B - A)$.

Dos conjuntos son disjuntos si su intersección es vacía, es decir, $A \cap B = \emptyset$.

El conjunto potencia de A lo denotamos $P(A)$ y es un conjunto cuyos elementos son todos los posibles subconjuntos de A, incluyendo el vacío y el propio conjunto A. Si A tiene n elementos, el conjunto potencia tiene 2^n.

Dado un conjunto no vacío A, una partición de A es un subconjunto P de $P(A)$ tal que

1. todo elemento de P es no vacío
2. los elementos de P son disjuntos 2 a 2
3. la unión de los elementos de P es igual a A

Pares, n-tuplas y relaciones Dados dos elementos x e y, denotamos el par ordenado de x e y como $\langle x, y \rangle$. No es lo mismo que el conjunto de dos elementos $\{x, y\}$ ya que en este no importa el orden y en el par ordenado sí.

El producto cartesiano de dos conjuntos A y B ($A \times B$) es el conjunto de todos los pares ordenados $\langle x, y \rangle$ con $x \in A$ y $y \in B$. Cualquier subconjunto R de $A \times B$ se llama una relación binaria entre A y B. Dada una relación R,

podemos asociarle un predicado $R(x, y)$ que asume el valor $true$ si y sólo si $\langle x, y\rangle \in R$. En caso contrario, será $false$.

El dominio de R es el conjunto de todos los x tal que $\langle x, y\rangle \in R$ para algún y. El codominio de R es el conjunto de todos los y tal que $\langle x, y\rangle \in R$ para algún x.

Podemos extender el concepto de par ordenado a secuencias de n elementos con n finito. Estas secuencias, llamadas n-tuplas ordenadas, las denotamos $\langle x_1, ..., x_n\rangle$.

El producto cartesiano de n conjuntos $A_1, ..., A_n$ se define $A_1 \times A_2 \times ... \times A_n = \{\langle a_1, a_2, ..., a_n\rangle : (a_1 \in A_1) \wedge (a_2 \in A_2) \wedge ... \wedge (a_n \in A_n)\}$. A veces, para productos cartesianos de n conjuntos siendo todos el mismo lo denotaremos de la siguiente manera: $A^n = A \times ... \times A$.

Cualquier subconjunto R de $A_1 \times A_2 \times ... \times A_n$ se llama una relación n-aria entre los conjuntos $A_1, ..., A_n$. Al igual que en el caso de los pares, dada una relación n-aria R podemos asociarle un predicado $R(x_1, ..., x_n)$ que será $true$ si y solo si $\langle x_1, ..., x_n\rangle \in R$. En caso contrario, será $false$.

Dado un conjunto A, un número natural $n > 0$ y una relación $(n + 1)$-aria $R \subseteq A^{n+1}$, un conjunto $B \subseteq A$ es cerrado respecto de R si, para toda $(n+1)$-tupla $\langle b_1, ..., b_{n+1}\rangle$ de R, $(b_1 \in B \wedge b_2 \in B \wedge ... \wedge b_n \in B) \rightarrow b_{n+1} \in B$.

Una relación binaria entre A y A se llama relación binaria en A. Sea R relación binaria en A. R se dice:

- reflexiva si $R(x, x) = true$ para todo x.
- antireflexiva si $R(x, x) = false$ para todo x.
- simétrica si para cada par de elementos x e y, $R(x, y)$ implica $R(y, x)$.
- antisimétrica si para cada par de elementos x e y, $R(x, y)$ y $R(y, x)$ implican $x = y$.
- transitiva si para tres elementos x, y, z, $R(x, y)$ y $R(y, z)$ implican $R(x, z)$

Si una relación binaria es reflexiva, simétrica y transitiva entonces es una relación de equivalencia.

Dada una relación $R \subseteq A \times B$, la relación inversa de R se define $R^{-1} = \{\langle y, x\rangle : \langle x, y\rangle \in R$. Cumple la propiedad: $(R^{-1})^{-1} = R$.

Funciones Dados dos conjuntos A y B, una función f de A a B ($f : A \to B$) es una relación binaria entre A y B que incluye, como mucho, una pareja $\langle a, b\rangle$ para cada $a \in A$. Las definiciones de dominio y codominio vistas previamente se pueden extender a funciones. En funciones preferimos decir que el valor de f en a es b ($f(a) = b$) en lugar de decir que la pareja $\langle a, b\rangle$ pertenece a f.

Una función $f : A \to B$ es total si su dominio coincide con A y es parcial en caso contrario. Decimos que es inyectiva si, para todo $a, a' \in A$ con $a \neq a', f(a) \neq f(a')$. Una función es suprayectiva si B coincide con el codominio de f. Es biyectiva si es inyectiva y suprayectiva al mismo tiempo. Una función biyectiva se dice también biyección.

Dada una función $f : A \to B$, decimos que admite función inversa $f^{-1} : B \to A$ si se cumple que $f(a) = b \leftrightarrow f^{-1}(b) = a$. Notar que solo las funciones inyectivas admiten inversas.

Cardinalidad de conjuntos Dado un conjunto finito X, su cardinal es igual a su número de elementos. Lo representamos $|X|$.

Dos conjuntos son equivalentes ($X \equiv Y$) si existe una biyección entre los dos conjuntos. Dado que si hay una biyección tienen el mismo número de elementos, se cumple que $|X| = |Y| \leftrightarrow X \equiv Y$.

En el caso de los conjuntos infinitos, no todos son equivalentes. Decimos que un conjunto es numerable si es equivalente al conjunto de los naturales. En caso contrario, es incontable.

2.2. Alfabetos, palabras y lenguajes

Un alfabeto es cualquier conjunto finito no vacío $\Sigma = \{\sigma_1, ..., \sigma_k\}$. Un símbolo es un elemento de un alfabeto. Una palabra es una tupla finita $x = \langle \sigma_{i_1}, ..., \sigma_{i_n} \rangle$ de símbolos de Σ; la palabra vacía la denotaremos por e. Para trabajar con menos notación, la palabra $\langle \sigma_{i_1}, ..., \sigma_{i_n} \rangle$ la expresaremos $\sigma_{i_1}...\sigma_{i_n}$. El conjunto infinito de todas las palabras lo denotaremos Σ^*.

La longitud $|x|$ de una palabra $x = \sigma_{i_1}, ..., \sigma_{i_n}$ es el número n de símbolos que contiene x. La palabra vacía tiene longitud 0. El número de palabras de longitud n es k^n. Dadas dos palabras x e y, la concatenación de x e y se define como una palabra z que consiste en los símbolos de x seguidos de los de y. Por tanto, $|z| = |x| + |y|$. La concatenación de una palabra por sí misma k veces se denota como x^k.

Dado un alfabeto Σ, un lenguaje sobre Σ es un subconjunto de Σ^*. El complemento de un lenguaje L, en símbolos L^c, se define como $L^c = \Sigma^* - L$.

Dado un alfabeto Σ, cualquier orden dentro de Σ induce un orden en Σ^* de la siguiente manera:

1. Para cualquier n, las palabras de longitud n precederán a las de longitud $n + 1$.

2. Para cada longitud, se ordenará en orden alfabético.

Este orden se llama orden lexicográfico. Como consecuencia, cualquier lenguaje sobre Σ es un conjunto contable.

Dado un lenguaje L, denotaremos como L_n el conjunto de todas las palabras de L que tienen longitud n, y $L_{\leq n}$ al conjunto de las palabras cuya longitud no es mayor que n.

La función censal de un lenguaje L (c_L) es una función que dice, para cada n, cuántas palabras de longitud no mayor que n hay en L, es decir, $c_L(n) = |L_{\leq n}|$.

La densidad de un lenguaje viene determinada por el ratio de crecimiento de su función censal según la siguiente clasificación:

1. Los lenguajes finitos son menos densos ya que su función censal es constante para un n lo suficientemente grande.

2. Lenguajes cuya función censal está acotada por un polinomio en n, es decir, cumplen $|L_n| \leq p(n)$, son llamados dispersos. Los lenguajes incluidos en $\{\sigma\}^*$, es decir, lenguajes sobre conjuntos unitarios, son llamados lenguajes de conteo. Claramente, la función censal de un lenguaje de conteo crece como mucho hasta n, así que los lenguajes de conteo son dispersos.

3. Los lenguajes más densos son aquellos cuya función censal crece exponencialmente con n. Se puede ver fácilmente que la función censal de Σ^* es igual a $\Sigma_{i=0}^{n} k^i = (k^{n+1}-1)/(k-1)$ donde k denota la cardinalidad de Σ.

2.3. Fórmulas booleanas

Definición 1 Una variable booleana es cualquier símbolo al que le podemos asociar los valores 0 y 1.

Definición 2 Sea X un conjunto contable de variables booleanas. La clase de las fórmulas booleanas sobre X es la clase más pequeña definida por:

1. Las constantes booleanas 0 y 1 son fórmulas booleanas.

2. Para cada variable booleana x en X, x es una fórmula booleana.

3. Si F_1 y F_2 son fórmulas booleanas entonces ($F_1 \vee F_2$, $F_1 \wedge F_2$ y $\neg F_1$ son fórmulas booleanas.

   Además, la clase de las fórmulas booleanas cuantificadas sobre X se obtiene de las reglas:

4. Toda fórmula booleana es una fórmula booleana cuantificada.

5. Si x está en X y F es una fórmula booleana cuantificada, entonces $\exists x F$ y $\forall x F$ son fórmulas booleanas cuantificadas.

Definición 3 Una asignación booleana es una aplicación de X a $\{0,1\}$. Dada una fórmula booleana cuantificada F y una asignación V, el valor de F sobre V es el valor booleano definido como sigue:

1. $V(F) = 0$ si $F = 0$
2. $V(F) = 1$ si $F = 1$
3. $V(F) = V(x)$ si $F = x$ para $x \in X$
4. $V(F) = V(F_1) \vee V(F_2)$ si $F = (F_1 \vee F_2)$
5. $V(F) = V(F_1) \wedge V(F_2)$ si $F = (F_1 \wedge F_2)$
6. $V(F) = \neg V(F_1)$ si $F = \neg F_1$
7. $V(F) = V(F_1|_{x:=0}) \vee V(F_1|_{x:=1})$ si $F = \exists x F_1$
8. $V(F) = V(F_1|_{x:=0}) \wedge V(F_1|_{x:=1})$ si $F = \forall x F_1$

Definición 4 Dada una fórmula booleana F, decimos que una asignación V satisface F si $V(F) = 1$. Una fórmula F es satisfactible si y sólo si existe una asignación V que satisface F; en caso contrario, decimos que F es insatisfactible.

Definición 5 Dados dos fórmulas booleanas cuantificadas F_1 y F_2, decimos que son equivalentes, y se denota por $F_1 \equiv F_2$, si y sólo si, para toda asignación V, $V(F_1) = V(F_2)$.

Definición 6 Un literal es una fórmula booleana que es una variable booleana o la negación de una variable booleana. Una claúsula es una disyunción de literales.

Definición 7 Una fórmula booleana está en forma normal conjuntiva (CNF) si y sólo si es una conjunción de un conjunto de claúsulas.

Ejemplo 3 Sean l_i literales, definimos una claúsula de la siguiente forma:

$$C = l_1 \vee l_2 \vee ... \vee l_n$$

Sean C_i claúsulas entonces la fórmula booleana F está en forma CNF si

$$F = C_1 \wedge C_2 \wedge ... \wedge C_n$$

Definición 8 Una fórmula booleana está en 3-CNF si está en (CNF) y además cada claúsula tiene exactamente 3 elementos.

2.4. Grafos

Grafos no dirigidos Un grafo no dirigido G es un par de conjuntos finitos (N, E) tal que E es una relación binaria simétrica en N. El conjunto N es el conjunto de nodos y E es el conjunto de aristas. Si $\langle x, y\rangle \in E$ decimos que x e y son adyacentes y que son los extremos de la arista. el número de nodos adyacentes de un nodo dado x se le llama grado de x. El grado de G es el máximo grado de entre todos los nodos.

Decimos que $G' = (N', E')$ es un subgrafo de $G = (N, E)$ si $N' \subseteq N$ y $E' \subseteq \{\langle n_i, n_j\rangle : n_i, n_j \in N' \wedge \langle n_i, n_j\rangle \in E\}$.

A los grafos se les puede añadir una función $c : E \to N$ que asocia un coste a cada arista tal que $c(n_i, n_j) = c(n_j, n_i)$.

Un grafo no dirigido $G = (N, E)$ se dice completo o clique si $E = N \times N$, es decir, si todos los nodos son adyacentes dos a dos.

Dado un grafo no dirigido G y dos nodos n_0 y n_k, un camino de n_0 a n_k de longitud k es una secuencia de aristas $\langle n_0, n_1\rangle$, $\langle n_1, n_2\rangle$,...,$\langle n_{k-1}, n_k\rangle$ tal que para $0 \leq i < j \leq k$, $n_i \neq n_j$. Si $n_0 = n_k$ se dice un ciclo. Llamamos

camino hamiltoniano a un camino que pasa por todos los vértices de un grafo exactamente una vez. Decimos ciclo hamiltoniano a un ciclo que pasa por todos los vértices de un grafo exactamente una vez.

Un grafo no dirigido es conexo si, para cualquier par de nodos x e y, existe un camino de x a y. Un árbol es un grafo conexo acíclico. Uno de los nodos es la raíz, que crea estructura jerárquica entre los nodos. Un nodo por sí mismo es un árbol. Llamamos subárboles de la raíz a los árboles que tienen una raíz diferente $n_1, ...n_k$. Estos nodos son llamados los hijos de la raíz.

La altura de un nodo en un árbol es la longitud del camino de la raíz al nodo. La altura de un árbol es la máxima altura de un nodo.

En un árbol, un nodo sin hijos se dice hoja. Un árbol es un árbol binario si todos los nodos que no sean hojas tienen dos hijos como mucho. Un árbol binario completo es perfecto si todas las hojas tienen la misma altura. El número de nodos de un árbol binario perfecto que tienen una altura h es igual a $2^{h+1} - 1$.

Grafos dirigidos Un grafo se dice grafo dirigido o digrafo si las aristas tienen un sentido definido. Es un par de conjuntos finitos $G = (N, E)$ donde N es el conjunto de nodos y E es un conjunto de pares ordenados de elementos de N denominados arcos. Un arco $e = (x, y)$ se considera dirigido de x hacia y. El arco (y, x) se le denomina el arco invertido de $(x, y$.

Decimos que $G' = (N', E')$ es un subgrafo de $G = (N, E)$ si $N' \subseteq N$ y $E' \subseteq \{(n_i, n_j) : n_i, n_j \in N' \wedge (n_i, n_j) \in E\}$.

A los grafos dirigidos se les puede añadir una función $c : E \to N$ que asocia un coste a cada arco pero en este caso no se cumple que $c(n_i, n_j) = c(n_j, n_i)$.

Dado un grafo dirigido G y dos nodos n_0 y n_k, un camino de n_0 a n_k de longitud k es una secuencia de arcos (n_0, n_1), (n_1, n_2),...,(n_{k-1}, n_k) tal que para $0 \leq i < j \leq k$, $n_i \neq n_j$. Si $n_0 = n_k$ se dice un ciclo. Llamamos camino hamiltoniano a un camino que pasa por todos los vértices de un

grafo exactamente una vez. Decimos ciclo hamiltoniano a un ciclo que pasa por todos los vértices de un grafo exactamente una vez salvo por el que empieza y acaba.

2.5. Notación de Landau

Cuando tenemos dos funciones, necesitamos tener herramientas para poder compararlas. Una de esas herramientas es la denominada notación de Landau, que compara asintóticamente dos funciones.

Definición 9 Sean f y g dos funciones definidas en un entorno de un punto x_0, entonces:

1. $f = O(g)$ cuando $x \to x_0$ si y sólo si existe un $\varepsilon > 0$ tal que $|f(x)| \leq \varepsilon|g(x)|$ para todo x en un entorno de x_0. En este caso decimos que f es O grande de g cuando $x \to x_0$. También se puede decir que f y g tienen el mismo orden.

2. $f = o(g)$ cuando $x \to x_0$ si y sólo si existe un $\varepsilon > 0$ tal que $|f(x)| < \varepsilon|g(x)|$ para todo x en un entorno de x_0.En este caso decimos que f es o pequeña de g cuando $x \to x_0$.

3. Modelo de computación

Nuestro objetivo va a ser estudiar la complejidad de resolver ciertos problemas para los cuales hay un algoritmo (conjunto ordenado y autocontenido de operaciones que resuelven un problema en una cantidad finita de tiempo y espacio) que los resuelve. Pero antes de ponernos a analizar la complejidad que supone resolver un problema mediante un determinado método, tenemos que encontrar un modelo computacional por el cual podamos resolver cada problema que se nos plantea, ya que no parece muy eficiente desarrollar uno cada vez.

Afortunadamente, para esta pregunta tenemos una respuesta satisfactoria, existe un modelo computacional mediante el cual podremos simular todos los demás: la Máquina de Turing. En este apartado nos dedicaremos a explicar qué son las máquinas de Turing y ver los dos tipos que hay, ya que nos serán necesarias para luego podernos adentrar en el estudio de la complejidad algorítmica.

3.1. Una introducción a las máquinas de Turing

Como ya hemos dicho, necesitamos un modelo computacional para resolver problemas y ese modelo va a ser las máquinas de Turing. En esta parte veremos una explicación de manera muy sencilla de cómo funciona una máquina de Turing.

Una máquina de Turing es un conjunto de cintas (una o varias) cada una con un cabezal que se va moviendo por la cinta y que va leyendo la información que tiene cada cinta y escribiendo y moviéndose en función de la información que recibe. Estas cintas se llaman semi-infinitas ya que sabemos su inicio pero no su final ya que depende de la longitud de la entrada que le pongamos.

A esto hay que sumarle la información de cada uno de los "pasos"de la máquina a los cuales llamamos estados. Cada vez que la máquina lee, escribe

y se mueve tenemos un estado. Todas estas operaciones están definidas por la función de transición que dice a la máquina qué hacer en cada estado con la información recibida.

Una máquina empieza a operar cuando le metemos una palabra poniendo una letra en cada celda de la cinta y dejando el resto de celdas en blanco. Este estado es el estado inicial en el que el cabezal está justo a la izquierda de donde comienza la palabra. La máquina opera la función de transición mientras es posible. Si en un momento la función de transición no está definida la máquina para. Si después de una secuencia de pasos la máquina para en un estado que llamamos aceptado, entonces decimos que la máquina acepta la palabra. En caso contrario, decimos que la máquina rechaza la palabra.

3.2. Máquinas de Turing deterministas

Definición 10 Una máquina de Turing determinista de k cintas con $k \geq 1$ es una tupla $M = \langle Q, \Sigma, \delta, q_0, F \rangle$ donde:

1. Q es el conjunto finito de estados internos
2. Σ es el alfabeto de la cinta
3. $q_0 \in Q$ es el estado inicial
4. F es el conjunto de estados finales aceptados
5. $\delta : Q \times \Sigma^k \to Q \times \Sigma^k \times \{R, N, L\}^k$ es una función llamada función de trasición de M.

Si la función de transición está indefinida indica que la computación debe parar. La tercera componente de la imagen de la función significa: R moverse una celda a la derecha, N no moverse y L moverse una celda a la izquierda. Esta función también se puede ver como una quíntupla

$$\langle q, s, s', m, q' \rangle$$

con $q, q' \in Q$, $s, s' \in \Sigma^k$ y $m \in \{R, N, L\}^k$

Definición 11 Dada una máquina M, una configuración de M es una descripción del estado de la computación: incluye los contenidos de las cintas, la posición del cabezal de la cinta, y el estado de la máquina de Turing. Si M tiene k cinta, una configuración de M es una $k+1$ tupla

$$(q, x_1, x_2, ..., x_{k-1}, x_k)$$

donde q es el estado actual de M y cada $x_j \in \Sigma^* \# \Sigma^*$ representa los contenidos actuales de la cinta j-ésima. El símbolo "#"marca la posición del cabezal. Todos los símbolos que no aparecen de cada cinta se suponen como espacios en blanco.

Definición 12 Llamamos configuración inicial de la máquina M a la palabra x que consiste de n elementos colocada en la primera cinta en los primeros n espacios y la máquina en el estado q_0 con el cabezal en la celda 0.

Definición 13 Una computación determinista para una máquina M de una entrada x, en símbolos $M(x)$, es una secuencia de configuraciones empezando por la configuración inicial y tal que cada configuración te lleva a la siguiente.

Ahora que hemos definido lo que es una configuración y una computación determinista nos falta por definir las computaciones con parada. Decimos que una computación $M(x)$ para si es finita y su última configuración es una configuración final. Tenemos dos tipos de máquinas de Turing según la definición de sus configuraciones finales:

- Máquinas de aceptación: tienen dos estados finales llamados estado aceptado y estado rechazado. Una entrada x es aceptada por M si la computación $M(x)$ termina en una configuración aceptada(configuración cuyo estado final es aceptado). El conjunto de entradas aceptadas de M se llama lenguaje aceptado de M y se representa $L(M)$.

- Máquinas transductoras: este tipo de máquinas recibe una palabra x de entrada y produce una palabra de salida y. Incluye un único estado final en el que las $|y|$ primeras celdas de la cinta en la que está la palabra de salida contienen la salida de la computación. Dada una función f, una máquina transductora M computa una función si la computación $M(x)$ alcanza una configuración final conteniendo $f(x)$ como salida siempre que $f(x)$ esté definido. En caso contrario, $M(x)$ es infinito y no hay configuración final. Una función es computable si existe una máquina de Turing capaz de computarla.

Vamos a probar ahora un teorema que nos servirá para probar la equivalencia de cualquier máquina de Turing a una con una única cinta.

Teorema 1 Sea M una máquina de Turing determinista con k cintas y alfabeto Σ. Entonces existe M' una máquina de Turing determinista de una cinta tal que $M(w) = M'(w)$ para cada entrada w.

Demostración Codificaremos las k cintas de M en una sola cinta en M' de la siguiente manera: para los símbolos de la primera cinta utilizaremos las posiciones $1, k+1, 2k+1, ...$, para los símbolos de la segunda cinta utilizaremos $2, k+2, 2k+2, ..$ y así sucesivamente. El alfabeto Σ' de M' contendrá por cada símbolo $\sigma \in \Sigma$, dos símbolos σ, σ^*. Así, los símbolos que lleven $*$ serán en donde se encuentre el cabezal de cada cinta.

Para simular un paso de M, la máquina M' realizará las siguientes operaciones:

1. Recorre de izquierda a derecha la cinta guardando los k símbolos que llevan $*$ en los estados.

2. Utiliza la función de transición de M para determinar el nuevo estado, los nuevos símbolos y los movimientos de los cabezales.

3. Recorre la cinta de derecha a izquierda actualizando la información.

Parece obvio que el resultado de la máquina M para cualquier entrada w será el mismo que para la máquina M'.

Definición 14 Un lenguaje L_1 es aceptado si existe una máquina de Turing determinista M tal que $L(M) = L_1$.

Definición 15 Un lenguaje L es decidible si existe una máquina de Turing determinista de aceptación M tal que para todo $x \in \Sigma^*$, $M(x)$ es una configuración aceptada si y sólo si $x \in L$. Se dice también que M decide L.

3.3. Máquinas de Turing no deterministas

En el apartado anterior, cada movimiento venía determinado por la situación actual. Tanto el estado de la máquina como los símbolos escaneados por el cabezal determinaban el próximo estado y los movimientos del cabezal. En esta sección, relajamos esta condición obteniendo máquinas no deterministas, cuyo siguiente movimiento pueden ser varios. Vamos a dar una definición formal:

Definición 16 Una máquina de Turing no determinista se define como una tupla

$$M = \langle Q, \Sigma, \delta, q_0, F \rangle$$

donde:

1. Q es el conjunto finito de estados internos
2. Σ es el alfabeto de la cinta
3. $q_0 \in Q$ es el estado inicial
4. F es el conjunto de estados finales aceptados

5. $\delta : Q \times \Sigma^k \to P(Q \times \Sigma^k \times \{R, N, L\}^k)$ es una función llamada función de trasición de M y donde para cualquier conjunto A, $P(A)$ es el conjunto potencia de A.

Definición 17 Llamamos grado de no determinismo al número máximo de posibles imágenes que puede tener la función de transición para una tupla (q, s).

Todos los conceptos definidos para el caso determinista aplican también para el no determinista. Sin embargo, dada una entrada ya no tenemos solo una computación, sino un conjunto de posibles computaciones. Por tanto, definimos la aceptación para máquinas no deterministas como sigue:

Definición 18 Una palabra de entrada w es aceptada para una máquina no determinista M si y sólo si existe una computación de M que termine en una configuración aceptada. Denotamos $L(M)$ el lenguaje aceptado de M, es decir, el conjunto de palabras aceptadas de M.

Dada la complicación para imaginarnos una máquina de Turing no determinista transductora, a partir de ahora consideraremos solo máquinas de aceptación.

El propósito de una máquina de aceptación, ya sea determinista o no determinista, es el de aceptar un lenguaje. Por tanto, consideramos dos máquinas equivalentes si aceptan el mismo lenguaje. Vamos a ver un resultado que nos relaciona las máquinas deterministas y no deterministas:

Teorema 2 Dada una máquina de Turing no determinista NT, siempre es posible derivarla en una máquina de Turing determinista T. Además, existe una constante c tal que, para cualquier palabra de entrada x, si NT acepta x en t pasos, T acepta x en c^t pasos.

Nota 1 Para la siguiente demostración vamos a introducir una serie de definiciones que nos resultarán muy útiles a la hora de probar este teorema.

Veremos las máquinas no deterministas como árboles llamados árboles de computación. Los nodos corresponden a las configuraciones mientras que las aristas corresponden a las transiciones entre configuraciones causadas por un paso de la máquina de Turing. Llamamos camino de la computación a la ruta que sigue la computación en el árbol. Un argumento de entrada x se dice que está aceptado si sigue al menos un camino de computación finito cuyo último nodo es un estado aceptado.

Demostración Para una entrada x, el objetivo de T es sistemáticamente visitar el árbol de computación asociado a $NT(x)$ hasta que encuentre un camino de computación aceptado. El problema que tenemos es el hecho de que existan caminos de computación infinitos que no sabemos detectar.

Para solucionar esto, vamos a simular $NT(x)$ mediante un recorrido en anchura. Es decir, en primer lugar, vamos a simular todas las configuraciones obtenidas al ejecutar un paso. Después, lo haremos con dos pasos y, así, sucesivamente. Si NT acepta x, T aceptará x. Si denotamos por r el grado de no determinismo de NT, la computación de paso i de $NT(x)$ incluirá, como mucho, r^i caminos de computación.

Podemos asociar una palabra del alfabeto $\Gamma = \{1, 2, ..., r\}$ con una máxima longitud i para cada uno de los caminos. Estas palabras las vamos a llamar direcciones y nos dicen el camino que debe seguir la computación. Cada nodo tiene asociado una dirección unívocamente y si una dirección no existe diremos que es no válida. Las vamos a ordenar lexicográficamente según su longitud.

Ahora para simular la máquina determinista T vamos a utilizar tres cintas:

- Cinta 1: de entrada
- Cinta 2: de simulaciones
- Cinta 3: de direcciones

El algoritmo que implementa T sería el siguiente:

1. Inicialmente, la cinta 1 contiene a la palabra x y las cintas 2 y 3 están vacías.
2. Colocar en la cinta 3 la primera cadena de Γ^i.
3. Borrar la cinta 2 y copiar la cinta 1 en la 2.
4. Usar la cinta 2 para simular la computación correspondiente a la cadena que hay en la cinta 3.
   - Se analiza el primer símbolo de la cinta 3.
   - Si es válido, realizamos la correspondiente elección no determinista. Si es una configuración de aceptación, aceptamos y terminamos. Si no, analizamos el siguiente símbolo de la cinta 3 y volvemos a empezar el paso. En caso de que no haya otro símbolo sin analizar, ir al paso 5.
   - Si no es válido, ir al paso 5.
5. Reemplazar la cadena de la cinta 3 por la siguiente cadena. Si no hay siguiente cadena, entonces terminar.
6. Ir al paso 3.

Para demostrar la segunda parte del teorema, tenemos que observar que para cada camino de computación visitamos como mucho i niveles del árbol, por tanto, como mucho ejecutamos c_1^i pasos con c_1 una constante. Entonces si NT ejecuta x en t pasos, el tiempo total simulando está acotado por $\Sigma_{i=1}^{t} c_1^i \leq c^t$ con c una constante apropiada.

4. Conceptos básicos de complejidad

Después de presentar en los primeros capítulos algunos ingredientes necesarios, presentamos aquí los conceptos y teoremas básicos de complejidad algorítmica. Comenzaremos definiendo qué es, presentaremos sus clases y explicaremos cómo funciona la reducibilidad de problemas.

Antes de definir lo que es la complejidad algorítmica, vamos a ver el tipo de problemas con el que trabajaremos.

4.1. Problemas de decisión

Los problemas de decisión son un tipo especial de problema computacional en los que la respuesta puede ser únicamente 'sí' o 'no' (1 o 0, de manera más formal).

Podemos verlos también como un lenguaje formal donde los elementos que pertenecen al lenguaje son los argumentos de entrada cuya respuesta es 'sí' y los que no pertenecen al lenguaje son los que tienen como respuesta 'no'. Decimos que un algoritmo acepta una entrada si tiene como respuesta 'sí' y que lo rechaza si tiene como respuesta 'no'.

Este tipo de problemas son muy interesantes ya que prácticamente todos los problemas se pueden reducir a problemas de decisión.

4.2. ¿Qué es?

La complejidad algorítmica es el análisis de la eficiencia de un algoritmo a la hora de resolver un problema. Este análisis nos ayuda a entender cómo un algoritmo escala a medida que el tamaño del argumento de entrada aumenta, y nos permite comparar la eficiencia relativa de diferentes algoritmos para resolver un mismo problema. También nos permite diseñar algoritmos más eficientes y mejorar el rendimiento de nuestros programas.

En general, hay dos clases principales de complejidad: la complejidad temporal y la complejidad espacial. La complejidad temporal se refiere al

tiempo que tarda un algoritmo en resolver un problema, mientras que la complejidad espacial se refiere a la cantidad de memoria que requiere un algoritmo para resolver un problema.

Posteriormente, nos centraremos en la complejidad temporal pero en el próximo apartado veremos una breve explicación de ambas.

4.3. Clases de complejidad: temporal y espacial

Definición 19 Una clase de complejidad C es un conjunto de problemas que poseen la misma complejidad computacional, es decir, que pueden ser resueltos utilizando un número de recursos limitados.

Este recurso generalmente puede ser el tiempo necesario para resolverlo o el espacio de memoria utilizado. Vamos a definir las clases más importantes:

Definición 20 La clase de complejidad $DTIME(f(n))$ es el conjunto de problemas de decisión que pueden ser resueltos en una máquina de Turing determinista en un tiempo $O(f(n))$.

Definición 21 La clase de complejidad $NTIME(f(n))$ es el conjunto de problemas de decisión que pueden ser resueltos en una máquina de Turing no determinista en un tiempo $O(f(n))$.

Definición 22 La clase de complejidad $DSPACE(f(n))$ es el conjunto de problemas de decisión que pueden ser resueltos en una máquina de Turing determinista en espacio $O(f(n))$.

Definición 23 La clase de complejidad $NSPACE(f(n))$ es el conjunto de problemas de decisión que pueden ser resueltos en una máquina de Turing no determinista en espacio $O(f(n))$.

Estas cuatro clases de complejidad son las clases temporales y espaciales tanto deterministas como no deterministas. Son las clases básicas y nos

servirán para definir al resto.

Definición 24 Una clase de complejidad de lenguajes es un conjunto de lenguajes cuyos correspondientes problemas de decisión están es una clase de complejidad C. Denotaremos a esta clase de complejidad de lenguajes también por C.

A partir de ahora consideraremos las clases de complejidad como clases de lenguajes.

Nota 2 Podríamos definir la clase $DTIME(f(n))$ de una manera alternativa para lenguajes:

$$DTIME(f(n)) = \{L \subseteq \Sigma^* | existe\ una\ máquina\ de\ Turing\ determinista$$

$$que\ decide\ L\ y\ tiene\ tiempo\ de\ ejecución\ O(f(n))\}$$

En el caso de $NTIME(f(n))$ se haría de forma análoga:

$$NTIME(f(n)) = \{L \subseteq \Sigma^* | existe\ una\ máquina\ de\ Turing\ no$$

$$determinista\ que\ decide\ L\ y\ tiene\ tiempo\ de\ ejecución\ O(f(n))\}$$

4.4. Reducibilidad de problemas

Definición 25 Dados dos lenguajes A y B, decimos que el problema de decisión asociado a A se reduce o es reducible a B si existe una transformación R, llamada reducción, tal que, para toda palabra $x \in \Sigma^*$, $x \in A \Leftrightarrow R(x) \in B$.

Veamos el siguiente ejemplo para dejar clara la idea de reducción de problemas:

Ejemplo 4 Consideremos el problema del viajante (ejemplo 2). Recordemos que queríamos recorrer 35 ciudades recorriendo la mínima distancia po-

sible. El algoritmo que utilizábamos era recorrer todos los itinerarios posibles haciendo 34! itinerarios. Vamos a realizar la siguiente reducción: se calcula la distancia de cada uno de los posibles itinerarios. Entonces el problema se reduciría a, dada la lista de distancias, devolver el mínimo. Aún así, aunque hayamos reducido el problema, es fácil ver que realizar dicha reducción requiere tiempo exponencial, por lo que tampoco hemos llegado a una solución eficiente.

Definición 26 Llamamos reducción de Karp o en tiempo polinómico a toda reducción f que sea una aplicación computable en tiempo polinómico en la longitud de la entrada. Dados dos lenguajes A y B decimos que A es Karp-reducible o reducible en tiempo polinómico a B y lo denotamos por $A \leq_p B$.

Proposición 1 Sean A, B y C tres lenguajes. Si $A \leq_p B$ y $B \leq_p C$ entonces $A \leq_p C$.

Demostración Si tenemos dos funciones $f(n) \leq n^c$ y $g(n) \leq n^d$ entonces su composición $g \circ f(n) \leq n^{cd}$, luego la composición polinómica de funciones es polinómica. Por tanto, si f_1 es una reducción de Karp de A a B y f_2 es una reducción polinómica de B a C, la aplicación $f_2 \circ f_1$ es computable en tiempo polinómico, pues las funciones de tiempo de las máquinas que las implementan son polinómicas; por lo que lo será su composición. Además, dada una palabra $x \in \Sigma^*$, $f_2 \circ f_1(x) = f_2(f_1(x)) \in C \Leftrightarrow f_1(x) \in B \Leftrightarrow x \in A$. Luego $f_2 \circ f_1$ es una reducción de Karp de A a C, por tanto, $A \leq_p C$.

Introduciremos a continuación la definición de reducción de clases:

Definición 28 Decimos que una clase de complejidad C es reducible a otra clase C' si todos los problemas de la primera son reducibles a problemas en la segunda.

Para los próximos apartados necesitamos definir los conceptos de dureza y completitud de clases:

Definición 29 Sea C una clase de complejidad, decimos que un lenguaje L es C-duro para un cierto tipo de reducciones si todos los lenguajes de la clase C son reducibles a L.

Definición 30 Sea C una clase de complejidad, decimos que un lenguaje L es C-completo para un cierto tipo de reducciones si $L \in C$ y L es C-duro.

Proposición 2 Sean C y C' dos clases de complejidad con $C' \subseteq C$ y consideremos un tipo de reducciones.

1. Si un lenguaje L es C-duro y $L \in C'$ entonces $C' = C$.
2. Si un lenguaje L es C-completo entonces $L \in C'$ si y solo si $C = C'$.

Demostración

1. Sea $A \in C$ un lenguaje, puesto que L es C-duro A es reducible a L, y como $L \in C'$ y el cómputo de la reducción f está en la clase C', entonces la máquina M que, sobre una entrada x, calcula $f(x)$ y simula la ejecución de la máquina que decide L, decide A con los recursos acotados correspondientes a la clase C', luego $A \in C'$ y por lo tanto $C = C'$.
2. Por ser L un lenguaje C-completo entonces es C-duro, y del apartado anterior se deduce una implicación. La otra implicación es inmediata, ya que si L es C- completo entonces $L \in C$ y suponemos por hipótesis que $C = C'$ por lo que $L \in C'$.

5. Las clases P y NP

Como ya dijimos, la complejidad algorítmica se dedica a la clasificación de problemas según la cantidad de recursos que necesitan. En este apartado, nos centraremos ya en el estudio de la complejidad temporal, es decir, del número de pasos de cálculo que debe efectuar una máquina de Turing para solucionar un problema. Tendremos que diferenciar entre dos clases principales relacionadas con la complejidad de tiempo: la clase P y la clase NP.

5.1. La clase P

La clase P es el conjunto de todos los problemas que son resolubles en un tiempo polinomial por una máquina de Turing determinista. La importancia de esta clase reside en que incluye todos los problemas tratables computacionalmente. Esto nos ayudará a clasificar todos los problemas a los que podemos dar solución en un intervalo de tiempo razonable. Vamos a ver una definición formal:

Definición 31 Definimos la clase P como

$$P = \cup_{k \geq 0} DTIME(n^k)$$

Diremos que un lenguaje $L \in P$ si existe una máquina de Turing determinista M que decide L en tiempo polinomial.

Proposición 3 Sean A y B lenguajes. Si $A \leq_p B$ y $B \in$P entonces $A \in$P.

Demostración Sea $x \in \Sigma^*$, $x \in A \Leftrightarrow f(x) \in B$, donde f es una reducción de Karp y, por tanto, computable en tiempo polinómico. Además, sabemos que existe una máquina de Turing determinista M que decide B en un tiempo polinómico. Por tanto, existe una máquina M' que, dada una entrada x, calcula $f(x)$ y ejecuta M y es determinista.

Veamos un ejemplo de lenguaje en P:

Ejemplo 5 Sea el lenguaje $RELPRIME = \{\langle n, m\rangle | mcd(n, m) = 1\}$ (mcd es el máximo común divisor de ambos números). Podríamos intentar resolver el problema dividiendo ambos números entre todos los factores primos que les preceden. Es decir, si queremos ver si $\langle 15, 14\rangle \in RELPRIME$ dividiríamos ambos números por 2,3,5,7,11 y 13. Si algunas de estas divisiones tuviese resto 0 no pertenecerían a $RELPRIME$ y, en caso contrario, sí lo harían. El problema de esta forma de solucionar el problema es que requiere tiempo exponencial. Es decir, si cogiésemos dos números muy grandes requeriría un tiempo que no podemos abarcar. Entonces, tenemos que buscar un algoritmo que nos permita solucionar el problema de una manera más eficiente. Ese algoritmo existe y nos permite probar que $RELPRIME \in$P. Es el algoritmo de Euclides y se basa en la siguiente idea: sea $a > b$ ambos enteros, al dividir a entre b, se obtiene un cociente q y un resto r; entonces el máximo común divisor de a y b es el mismo que el de b y r. Sea $c = mcd(a, b)$, como $a = bq+r$ y c divide a a y a b, divide también a r. Si existiera un número mayor que c dividiese b y a r, también dividiría a a, por lo que c no sería el máximo común divisor de a y b. Una vez probado esto, el algoritmo sería el siguiente:

1. Sea $r_0 = a$ y $r_1 = b$
2. Creamos un contador $i = 1$
3. Si $r_i = 0$ pasar al paso 4. Si $r_i \neq 0$:
   - $r_{i+1} = r_{i-1} \ mod \ r_i$
   - $i = i + 1$
4. El resultado es r_{i-1}. Terminar

5.2. Clase NP

La clase NP es el conjunto de todos los problemas que podemos resolver de un modo no determinístico en un tiempo polinomial. Viene a significar que son los problemas que no podemos resolver en un tiempo polinomial pero que sí podemos comprobar en un tiempo polinomial si tenemos una solución dada. Es decir, si nos dan una solución, podemos comprobar fácilmente si se trata de la solución que buscaba, pero si no nos la dan, no podemos encontrarla.

Nos interesa definir bien esta clase para saber cuáles son los problemas no tratables computacionalmente hablando. Vamos a ver una definición formal:

Definición 32 Definimos la clase NP como

$$NP = \cup_{k \geq 0} NTIME(n^k)$$

Diremos que un lenguaje $L \in NP$ si existe una máquina de Turing no determinista NM que decide L en tiempo polinomial.

Proposición 4 Sean A y B lenguajes. Si $A \leq_p B$ y $B \in$NP entonces $A \in$NP.

Demostración La demostración sería análoga al caso de P.

Vamos a ver una definición de la clase NP mediante verificadores que nos interesará para futuras pruebas:

Definición 33 Un verificador para un lenguaje $L \subseteq \Sigma^*$ es una máquina de Turing determinista V tal que:

$$L = \{w | \exists c \in \Sigma^*, V(\langle w, c \rangle) = acepta\}$$

Es decir, a la palabra w la acompañamos de un certificado c que nos ayuda a ver si $w \in L$ o no.

Observación 1

- Medimos el tiempo de ejecución de un verificador V en función del tamaño de w, ignorando el certificado.
- Un verificador V es de tiempo polinomial si su tiempo de ejecución es polinomial.
- Un lenguaje $L \subseteq \Sigma^*$ es polinomialmente verificable si tiene un verificador de tiempo polinomial.

Ejemplo 6 Supongamos el lenguaje

$$HAMPATH = \{\langle G, v, w\rangle | G \ grafo \ dirigido \ con \ un$$

$$camino \ hamiltoniano \ de \ v \ a \ w\}$$

. Podemos ver que este lenguaje es polinomialmente verificable. Si existe un camino hamiltoniano en un grafo G, la lista de vértices $x_1, x_2, ..., x_n$ por las que pasa el camino se puede usar como certificado:

$$V(\langle G, v, w\rangle, \langle x_1, x_2, ..., x_n\rangle)$$

Para verificar deberíamos ver si $x_1 = v$ y $x_n = w$, que no haya vértices repetidos, que todos los vértices de G estén en $x_1, x_2, ..., x_n$ y que $x_1, x_2, ..., x_n$ sean vértices de G. Si se cumple todo esto el verificador V debería aceptar y, en caso contrario, rechazar.

Definición 34 $NP = \{L \subseteq \Sigma^* | L$ es polinomialmente verificable$\}$.

Observación 2 Si un lenguaje está en P, también está en NP (P$\subseteq$NP).

Demostración Supongamos un lenguaje $L \in$P. Es decir, existe una máquina de Turing determinista M que decide L. Podemos construir un verificador polinomial para L: $V(w, c)$ tal que ignora el certificado c y simula $M(w)$. Es de tiempo polinomial y verifica el lenguaje L.

Ejemplo 7 $HAMPATH \in$NP. Ya vimos que el lenguaje era polinomialmente verificable por lo que ya está probado.

Vamos a ver otro ejemplo de lenguaje NP:

Ejemplo 8 $COMPUESTOS = \{\langle n \rangle | n \in \mathbb{N}, \exists p, q \in \mathbb{N}, p, q > 1, n = pq\}$. Podemos utilizar como verificador el propio número n y un factor p de n. Así, el verificador $V = (\langle n \rangle, \langle n, p \rangle)$ aceptará si n y p son naturales, si $p > 1$ y si $p < n$. Es obvio que el verificador requiere tiempo polinomial.

6. La conjetura de Cook

¿P=NP?

Esta pregunta llevan haciéndose los matemáticos durante más de 50 años sin haber conseguido llegar a una solución. Fue propuesto como uno de los problemas del milenio en el año 2000 por el Clay Mathematics Institute y su solución tiene un premio de un millón de dólares. El estudio de este problema está planteado hoy en día como la búsqueda de los límites de la computación.

Ya vimos que P⊆NP, ya que todo problema resoluble en tiempo polinomial por una máquina determinista lo será también por un máquina no determinista. La gran pregunta aquí es si ese contenido es estricto o no. Para acercarnos a una solución, estudiaremos diferentes tipos de problemas NP que hay y veremos qué supondría el hecho de que la igualdad entre P y NP fuese cierta.

6.1. Problemas NP-completos

Dentro de los problemas NP, tenemos un tipo de problemas llamados NP-completos que son considerados los más difíciles que estén en NP. Nos interesa el estudio de este tipo de problemas ya que en cuanto encontremos uno que pertenezca o no a P, con todos los demás problemas NP ocurriría lo mismo. Definiremos de una manera formal la NP-Completitud:

Definición 35 Sea $A \subseteq \Sigma^*$ un lenguaje. Decimos que A es NP-completo si:

1. $A \in NP$
2. $\forall B \subseteq \Sigma^*$, $B \in NP \Rightarrow B \leq_p A$

Teorema 3 Si A es NP-completo y $A \in$P entonces P=NP.

Demostración Supongamos que A es NP-completo y $A \in$P. Veamos P=NP. Que P$\subseteq$NP ya lo vimos en la observación 2. Veamos ahora que NP$\subseteq$P. Sea $B \in$NP. Como A es NP-completo, $B \leq_p A$. Por tanto, $B \in$P. Ya está probado.

Teorema 4 Si A es NP-completo, $B \in$NP y $A \leq_p B$. Entonces B es NP-completo.

Demostración Supongamos que A es NP-completo, $B \in$NP y $A \leq_p B$. Veamos que B es NP-completo.

1. $B \in$NP. Se cumple.

2. Veamos que $\forall C \in$NP, $C \leq B$. Sea $C \in$NP, como A es NP-completo, $C \leq_p A$. Como sabemos que $A \leq_p B$, se cumple que $C \leq_p B$.

En el Ejemplo 2 vimos uno de los mayores problemas no resueltos relacionados con la conjetura de Hook: el problema del viajante (TSP). Vamos a generalizar el problema a un número no determinado de ciudades y vamos a intentar probar que es un problema NP-completo. Para ello, tendremos que hacer otras pruebas previas que también son de gran interés.

Definición 36 Se define el lenguaje SAT como el conjunto de las fórmulas booleanas satisfactibles:

$$SAT := \{F \ \textit{fórmula booleana} : \exists V \ \textit{asignación tal que} \ V(F) = 1\}$$

Teorema 5(de Cook-Levin) SAT es NP-completo.

Demostración Veamos que SAT cumple con la definición de NP-completo.

1. $SAT \in$NP.

   Veamos que SAT se puede verificar en tiempo polinomial. Sea $V(F, c)$ un verificador. El certificado c debe ser una lista de variables $[x_1, ...x_n]$.

Evaluamos la fórmula F y le damos valor verdadero (1) a las variables que están en la lista y falso (0) a las demás. Si la fórmula tiene valor 1 aceptamos. En caso contrario, rechazamos. Este verificador es obviamente de tiempo polinomial ya que solo hay que recorrer la lista de variables.

2. Si $A \in$NP entonces $A \leq_p SAT$.

   - Supongamos que $A \in$NP. Es decir, existe una máquina de Turing no determinista N tal que N decide A en tiempo polinomial.
   - Suponemos que la máquina N va a tener una sola cinta porque por el teorema 1 a partir cualquier máquina de Turing de k cintas podemos transformarla en una de una cinta.
   - Supongamos que dada una cadena $w \in \Sigma^*$, la máquina N termina su ejecución en tiempo $|w|^k = n^k$ siendo n el número de caracteres de la cadena.
   - Una cadena $x \in \Sigma^*$ es aceptada por la máquina N si existe una secuencia de configuraciones: $\alpha_0, \alpha_1, \alpha_2, ..., \alpha_m$ $m \leq n^k$ tal que α_0 es la configuración inicial, α_m es una configuración aceptada y existen transiciones de α_i a $\alpha_{i+1} \forall i \in 0...(m-1)$.
   - Transformaremos una cadena $w \in \Sigma^*$ en una fórmula F que sea satisfactible si y sólo si existe una secuencia de configuraciones de N que acepta x.
   - Vamos a crear una tabla de ejecución de la máquina N de $n^k \times n^k$. En cada fila hay una configuración de la máquina pero, en este caso, al haber solo una cinta vamos a representarla de una manera diferente. Tendremos los caracteres de la palabra puestos cada uno en una celda por orden pero en la columna de la izquierda de donde se encuentra el cabezal estará el estado en el que nos encontramos.

Vease un ejemplo para aclarar: (recordamos que e es la palabra vacía)

#	q_0	w_1	w_2	w_3	w_4	w_5	e	...	e	#
#	z	q_3	w_2	w_3	w_4	w_5	e	...	e	#
.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.

- Vamos a tener variables de la forma x_{ijs} donde
  - $1 \leq i \leq n^k$ es un número de fila
  - $1 \leq j \leq n^k$ es un número de columna
  - $s \in C = \{\Sigma \cup Q \cup \#\}$ es un símbolo (# es simplemente un separador para marcar los límites de la tabla)
- La variable x_{ijs} representa el hecho de que en la fila i, columna j de la tabla se encuentra el símbolo s.
- Vamos a construir una fórmula:

$$F = F_{celda} \wedge F_{inicial} \wedge F_{mover} \wedge F_{aceptar}$$

- Vamos a definir el primer elemento de la fórmula. Esta se encarga de que cada celda tenga un único elemento de C.

$$F_{celda} = \wedge_{i=1...n^k} \wedge_{j=1...n^k} (\vee_{s \in C} x_{ijs}) \wedge (\wedge_{s \in C} \wedge_{t \in C(t \neq s)} \neg x_{ijs} \vee \neg x_{ijt})$$

- El segundo elemento se encarga que la primera fila sea la configuración inicial de la máquina N para la palabra w:

$$F_{inicial} = x_{11\#} \wedge x_{12q_0} \wedge x_{13w_1} \wedge x_{14w_2} \wedge ... \wedge x_{1(n+2)w_n} \wedge$$

$$\wedge x_{1(n+3)e} \wedge ... \wedge x_{1(n^k-1)e} \wedge x_{1n^k\#}$$

- El tercer elemento es el más complicado de notar. Queremos que

este elemento sea satisfactible cuando las sucesivas filas de la tabla correspondan a transiciones de la máquina N. Hay 4 tipos posibles de transiciones que vienen explicadas en la fórmula:

$$F_{mover} = (\wedge_{i=1\ldots(n^k-1)} \wedge_{j=3\ldots(n^k-2)} (x_{i(j-1)a} \wedge x_{ijb} \wedge x_{i(j+1)c}) \rightarrow x_{(i+1)jb}) \wedge$$

$$\wedge (\wedge_{i=1\ldots(n^k-1)} (x_{i1\#} \wedge x_{i2b} \wedge x_{i2c}) \rightarrow (x_{(i+1)1\#} \wedge x_{(i+1)2b})) \wedge$$

$$\wedge (\wedge_{i=1\ldots(n^k-1)} \wedge_{j=3\ldots(n^k-2)} (x_{i(j-1)q_l} \wedge x_{ijb} \wedge x_{i(j+1)c}) \rightarrow$$

$$\rightarrow ((x_{(i+1)jq_m} \wedge x_{(i+1)jq_m}) \vee x_{(i+1)jb}))$$

Dado que no quedan demasiado claras con la fórmula vamos a ilustras cuáles son las cuatro transiciones posibles de la tabla en subtablas de tamaño 2×3:

(Notar que $a, b, c \in \Sigma$, $q_x, q_y \in Q$)

a	b	c
$?$	b	$?$

$\#$	b	c
$\#$	b	$?$

q_x	b	c
z	q_y	$?$

q_x	b	c
$?$	b	$?$

- El cuarto elemento es satisfactible si la máquina acepta la cadena de entrada:

$$F_{aceptar} = \vee_{i=1\ldots n^k} \vee_{j=1\ldots n^k} x_{ijq_{aceptado}}$$

- Dada una palabra $w \in \Sigma^*$, hemos construido la codificación de una fórmula $F(w)$. Se cumple que $w \in A \iff F(w) \in SAT$
- Hemos reducido el lenguaje A a SAT y, además, la función es computable en tiempo polinomial ya que la parte que más requiere es F_{celda} que requiere n^{2k} pasos. Por tanto, $A \leq_p SAT$.

Definición 37

$$3SAT = \{F : F \ \textit{fórmula booleana satisfactible y está en}$$

$forma\ 3CNF\}$

Corolario 1 $3SAT$ es NP-completo.

Demostración Como $3SAT$ es un caso particular del problema SAT sabemos que está en NP. Tenemos que demostrar que es NP-completo. Para ello, por el teorema 3,es suficiente demostrar que existe una transformación polinómica de SAT a $3SAT$, es decir, que $SAT \leq_p 3SAT$.

Consideramos cualquier fórmula $F = C_1 \wedge ... \wedge C_m$ con C_i claúsulas y construimos una nueva fórmula F' que contenga únicamente 3 variables por claúsula, de tal forma que F' es satisfactible si y sólo si F lo es. Sea C_i una de las claúsulas de F, distinguiremos 3 casos:

1. Si C_i tiene 3 variables no hacemos nada.

2. Si $C_i = x_1 \vee x_2 \vee ... \vee x_k$ con $k > 3$, reemplazamos C_i por $k-2$ claúsulas como sigue:

   $$(x_1 \vee x_2 \vee y_1) \wedge (\neg y_1 \vee x_3 \vee y_2) \wedge (\neg y_2 \vee x_4 \vee y_3) \wedge ... \wedge (\neg y_{k-3} \vee x_{k-1} \vee x_k)$$

   donde $y_1, ..., y_{k-3}$ son nuevas variables. Es trivial ver que estas claúsulas son satisfactibles si y sólo si C_i lo es.

3. Si $C_i = (x)$, reemplazamos C_i por $(x \vee y \vee z$, y si $C_i = (x_1 \vee x_2)$, lo reemplazamos por $(x_1 \vee x_2 \vee y)$. Luego añadimos las claúsulas

   $$(\neg z \vee a \vee b) \wedge (\neg z \vee \neg a \vee b) \wedge (\neg z \vee a \vee \neg b) \wedge (\neg z \vee \neg a \vee \neg b) \wedge (\neg y \vee a \vee b) \wedge$$

   $$(\neg y \vee \neg a \vee b) \wedge (\neg y \vee a \vee \neg b) \wedge (\neg y \vee \neg a \vee \neg b)$$

   donde y, z, a y b son nuevas variables. Esta adición fuerza a que las variables y y z sean no satisfactibles en cualquier asignación que satisfaga F'.

Hemos reemplazado todas las claúsulas C_i por claúsulas equivalentes de 3 variables. Además se cumple que F' es satisfactible si y sólo si F lo es y la construcción se puede hacer en tiempo polinomial por lo que hemos descrito una transformación polinomial de SAT a $3SAT$. Por tanto, $3SAT$ es NP-completo.

Teorema 6 $HAMPATH$ es NP-completo.

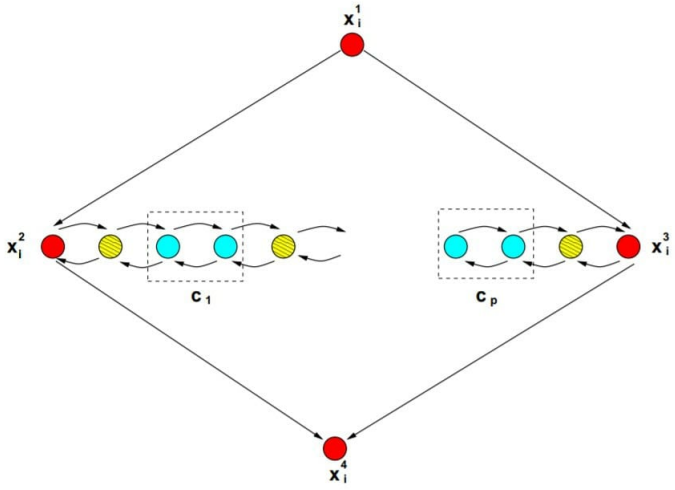
Demostración

1. En el ejemplo 6 ya probamos que $HAMPATH \in$NP.

2. Veamos que $3SAT \leq_p HAMPATH$ y por el teorema 3 ya tendremos que $HAMPATH$ es NP-completo. Es decir, dada una fórmula F en 3-CNF, vamos a querer construir un grafo G, y vértices v y w tales que F es satisfactible si y sólo si G tiene un camino hamiltoniano desde v a w.

   Supongamos que $F = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge ... \wedge (a_k \vee b_k \vee c_k)$ con k claúsulas y l variables x_1, x_2,..., x_l.

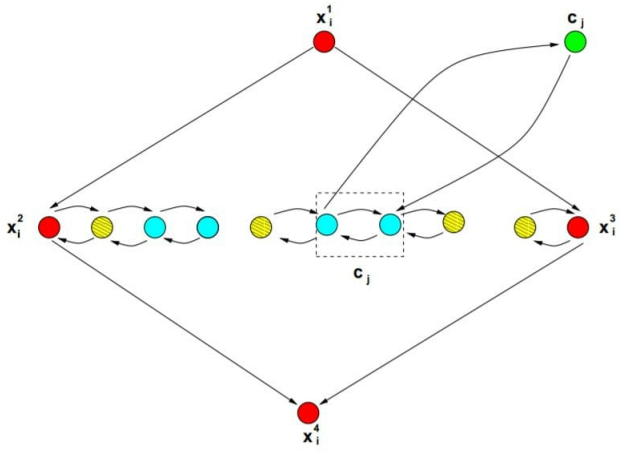
   Para cada variable x_i vamos a construir un grafo con forma de diamante con una fila central de vértices tal que:



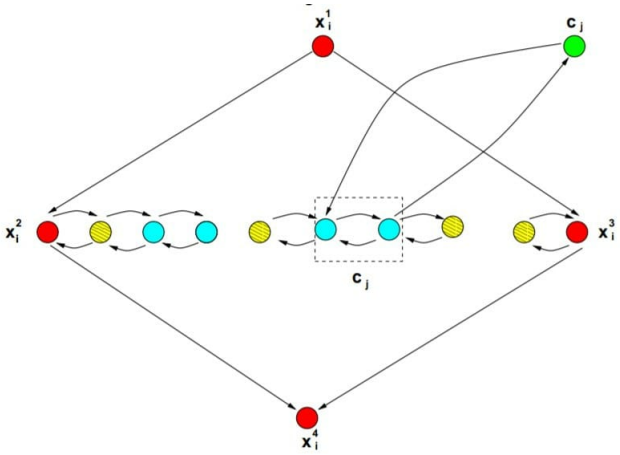
En el diamante de cada variable x_i, la fila de vértices en el centro contiene 2 vértices asociados a cada claúsula (los azules) y un separador entre una claúsula y otra (los amarillos). Por tanto, habrá $3k+1$ vértices en la fila central del diamante sin contar los extremos.

Vamos a unir todas las x_i de tal manera que cada $x_i^4 = x_{i+1}^1$. Así tendremos l grafos unidos como el anterior y tendríamos que $v = x_1^1$ y $w = x_l^4$. Junto a ellos cada claúsula aporta un nodo $c_1, ..., c_k$. Cada uno de estos vértices de cada claúsula los vamos a unir al grafo de la siguiente manera:

- Si la variable x_i aparece en la claúsula c_j, lo uniremos al grafo así:



- Si la variable $\neg x_i$ aparece en la claúsula c_j, de la siguiente manera:

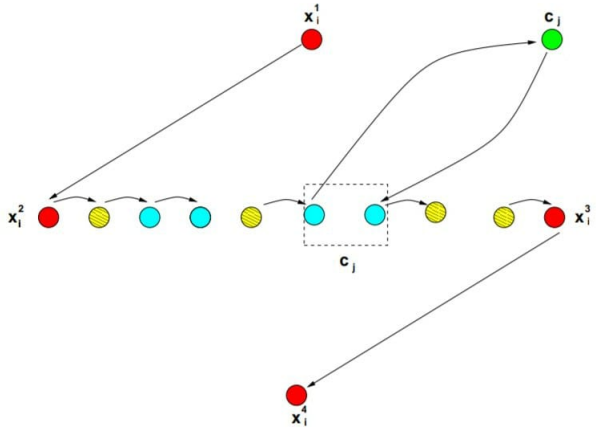


Vamos a probar ahora que F es satisfactible si y sólo si $\langle G, v, w\rangle \in HAMPATH$.

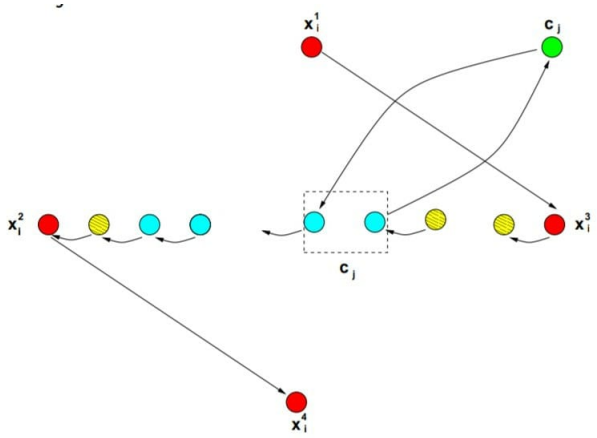
($\Rightarrow$) Supongamos F satisfactible. Esto quiere decir que para una asignación V de F, $V(F) = 1$. Construimos un camino hamiltoniano como sigue:

- Si $V(x_i) = 1$, cuando entramos al diamante de x_i entramos por la izquierda y recorremos la fila de vértices de izquierda a derecha.

- Si $V(x_i) = 0$, cuando entramos al diamante de x_i entramos por la derecha y recorremos la fila de vértices de derecha a izquierda.
- Sea c_j una claúsula y $r_0 = min\{r : F(c_j^r) = 1\}$. Si $c_j^{r_0} = x_i$, hacemos el siguiente desvío:



Si $c_j^{r_0} = \neg x_i$, hacemos el siguiente desvío:



($\Leftarrow$) Supongamos ahora que $\langle G, v, w\rangle \in HAMPATH$. Sea $\langle G, v, w\rangle \in HAMPATH$. Este camino recorre cada diamante de izquierda a derecha o de derecha izquierda salvo cuando visite uno de los vértices de las claúsulas. Definimos V asignación tal que $V(x_i) = 1$ si el diamante asociado a x_i se recorre de izquierda a derecha y $V(x_i) = 0$ si el diamante asociado a x_i se recorre de derecha a izquierda. Se tiene claramente que

$V(F) = 1$.

Para ver que esta transformación se hace en tiempo polinomial basta con ver que el número de pasos a realizar depende del número de claúsulas, de hecho, el orden es $O(k^2)$. Por tanto, tenemos que $3SAT \leq_p HAMPATH$.

Definición 38 $UHAMPATH = \{\langle G, v, w\rangle | G \ grafo \ no \ dirigido \ con \ un \ camino \ hamiltoniano \ de \ v \ a \ w\}$

Corolario 2 $UHAMPATH$ es NP-completo.

Demostración

1. Para demostrar que $UHAMPATH \in$NP, podemos utilizar un verificador como el del ejemplo 6. Si existe un camino hamiltoniano en un grafo G, la lista de vértices $x_1, x_2, ..., x_n$ por las que pasa el camino se puede usar como certificado:

   $$V(\langle G, v, w\rangle, \langle x_1, x_2, ..., x_n\rangle)$$

   Para verificar comprobamos si $x_1 = v$ y $x_n = w$, que no haya vértices repetidos, que todos los vértices de G estén en $x_1, x_2, ..., x_n$ y que $x_1, x_2, ..., x_n$ sean vértices de G.

2. Veamos que $HAMPATH \leq_p UHAMPATH$.

   Sea $\langle G = (N, E), v, w\rangle \in HAMPATH$. Construimos un grafo no dirigido $G' = (N', E')$ tal que:

   - Cada nodo $u \in N - v, w$ proporciona un nodo de G': u'.
   - El nodo v proporciona el nodo v' de G'.
   - El nodo w proporciona el nodo w' de G'.

- Para cada arco $(x, y) \in E$, se tiene que $\langle x', y' \rangle \in E'$.

Definimos una función $f : HAMPATH \to UHAMPATH$:

$$F(G, v, w) = (G', v', w')$$

- f es computable en tiempo polinomial (el tiempo que tardamos depende del número de nodos)
- Para cada $(G, v, w) \in HAMPATH$, G posee un camino hamiltoniano de v a w si y sólo si G' posee un camino hamiltoniano de v' a w'.

  ($\Rightarrow$) Sea $c = (v, x_1, ..., x_r, w)$ un camino hamiltoniano de v a w en G. Entonces $c' = (v', x'_1, ..., x'_r, w')$ es un camino hamltoniano en G'.

  ($\Leftarrow$) Sea $c' = (v', y'_1, ..., y'_r, w')$ un camino hamiltoniano de v' a w' en G'. Entonces $c = (v, y_1, ..., y_r, w)$ es un camino hamltoniano en G.

Definición 39

$UHAMCYCLE = \{G | G \ grafo \ no \ dirigido \ con \ un \ ciclo \ hamiltoniano\}$

Corolario 3 $UHAMCYCLE$ es NP-completo.

Demostración

1. Para demostrar que $UHAMCYCLE \in$NP, podemos utilizar un verificador similar al de la demostración anterior. Si existe un ciclo hamiltoniano en un grafo G, la lista de vértices $x_1, x_2, ..., x_n$ por las que pasa el camino se puede usar como certificado:

   $$V(G, \langle x_1, x_2, ..., x_n \rangle)$$

Para verificar comprobamos que no haya vértices repetidos, que todos los vértices de G estén en $x_1, x_2, ..., x_n$ y que $x_1, x_2, ..., x_n$ sean vértices de G.

2. Veamos que $UHAMPATH \leq_p UHAMCYCLE$.

   Sea $\langle G = (N, E), v, w\rangle \in HAMPATH$. Construimos un grafo no dirigido $G' = (N', E')$ tal que:

   - $N' = N \cup \{v^*, w^*\}$, con $v^*, w^* \notin N$
   - $E' = E \cup \{\{v^*, v\}, \{w^*, w\}, \{v^*, w^*\}\}$

   Definimos una función $f : UHAMPATH \to UHAMCYCLE$:

   $$F(G, v, w) = G'$$

   - f es computable en tiempo polinomial (el tiempo que tardamos depende del número de nodos)
   - Para cada $(G, v, w) \in UHAMPATH$, G posee un camino hamiltoniano de v a w si y sólo si G' posee un ciclo hamiltoniano.

     ($\Rightarrow$) Sea $c = (v, x_1, ..., x_r, w)$ un camino hamiltoniano de v a w en G. Entonces $c' = (v^*, v, x_1, ..., x_r, w, w^*, v^*)$ es un ciclo hamltoniano en G'.

     ($\Leftarrow$) Sea c un ciclo hamiltoniano en G'. Debido a las conexiones definidas en G', los nodos w, w^*, v^*, v son consecutivos en c. Por tanto, $c = (w, w^*, v^*, v, y_1, ..., y_r, w)$ Entonces $c' = (v, y_1, ..., y_r, w)$ es un camino hamltoniano de v a w en G.

Paso previo a realizar la demostración que el TSP es NP-completo, vamos a definir de forma informal y formal este problema.

Definición 40 (informal) Dado un conjunto de ciudades y el coste (distancia) de viajar entre cada par posible de estas ciudades, el problema del viajante (TSP) consiste en encontrar el mejor camino posible que visite todas las ciudades una única vez saliendo y llegando de la misma ciudad minimizando el coste de viaje.

Definición 41 (formal) El problema del viajante (TSP) en un grafo no dirigido G con costes c consiste en encontrar un ciclo hamiltoniano de coste menor que un presupuesto L.

Nota 4 Tomamos un grafo no dirigido porque suponemos que el coste de ir a una ciudad A a una ciudad B es el mismo que el del viaje contrario.

Teorema 7 El TSP es NP-completo.

Demostración El problema de encontrar un ciclo hamiltoniano, que acabamos de demostrar que es un problema NP-completo (Corolario 3), es un caso particular del problema del viajante de comercio. Sea $G = (N, E)$ un grafo cualquiera, construimos un ejemplo del TSP con $|V|$ ciudades. Sean $n_i, n_j \in N$, llamamos $c(n_i, n_j)$ al coste de viajar de n_i a n_j y viceversa (grafo no dirigido). Sea $c(n_i, n_j) = 1$ si $\langle n_i, n_j \rangle \in E$ y $c(n_i, n_j) = 2$ en otro caso. Supongamos que el presupuesto L es igual a $|V|$. Es inmediato ver que existe un ciclo de coste igual a L si y sólo si existe un ciclo hamiltoniano en G. Hemos reducido el problema del viajante a $HAMCYCLE$. Por tanto, el TSP es NP-completo.

6.2. Problemas NP-intermedios

Hemos estudiado ya los problemas con los que podemos tratar fácilmente; los problemas P, y también los problemas que en teoría son más difíciles de resolver; los problemas NP-completos. Si suponemos que los conjuntos P y

NP no son iguales, podemos demostrar ahora que existen también problemas que están en un punto medio de ambos.

Definición 42 Supongamos que P≠NP. Los lenguajes que se encuentran en el conjunto diferencia NP-(P∪NP-completos) se denominan lenguajes NP-intermedios.

Teorema 8(de Ladner) Si P≠NP entonces existe un lenguaje $L \in$NP-P que no es NP-completo.

Nota 5 Dado $i \in \mathbb{N}$, vamos a llamar M_i a la máquina de Turing determinista que nos da la representación binaria del string $[i]$. Si este string no representa ninguna máquina de Turing, mapeamos i a una máquina que no hace nada.

Demostración Para cada función $H : \mathbb{N} \to \mathbb{N}$ definimos el lenguaje SAT_H como el lenguaje de las fórmulas satisfactibles de longitud n que hemos rellenado con $n^{H(n)}$ unos:

$$SAT_H = \{F01^{n^{H(n)}} : F \in SATyn = |F|\}$$

, donde $|F|$ denota la longitud de la fórmula.

Definimos la función $H : \mathbb{N} \to \mathbb{N}$ de forma que: $H(0) = H(1) = 0$ y $H(n)$ sea el menor entero $i < log(logn)$ tal que para todo $x \in \{0,1\}^*$ con $x < logn$, M_i devuelve $SAT_H(x)$ en como mucho $i|x|^i$ pasos. Si este entero no existe definimos $H(n) = log(logn)$.

- H está bien definida: $H(n)$ sirve para definir los elementos de SAT_H de longitud mayor que n, mientras que la definición de H depende de chequear elementos de longitud $logn$ como máximo.

- SAT_H $\in$P si y sólo si $H(n) = O(1)$. Además, si SAT_H $\notin$P entonces $lim_{n\to\infty} H(n) = \infty$.

  ($\Rightarrow$) Veamos que si SAT_H $\in$P entonces $H(n) = O(1)$ (acotada por una constante). Supongamos que existe una máquina de Turing determinista M que decide SAT_H en a lo sumo cn^c pasos. Como M admite una infinidad de cadenas binarias que la representan, sea $i < c$ tal que $M = M_i$. Por definición de $H(n)$, $H(n) \leq i$.

  ($\Leftarrow$) Supongamos ahora que $H(n) = O(1)$, es decir, la imagen de H está en un conjunto finito y debe existir i tal que $H(n) = i$ para infinitos velores de $n \in \mathbb{N}$. Pero esto implica que la máquina de Turing determinista M_i decide SAT_H en tiempo in^i, pues en otro caso existiría una entrada x sobre la cual M_i no devuelve la respuesta correcta en ese tiempo, y para todo $n > 2^{|x|}$ tendríamos que $H(n) \neq i$, con lo que llegamos a una contradicción. Basta con suponer que $H(n)$ está acotado por una constante, con lo que se demuestra la segunda parte del lema.

Tenemos que probar ahora que si P$\neq$NP entonces SAT_H no es P ni NP-completo:

- Supongamos que SAT_H $\in$P, entonces hemos probado que $H(n) \leq C$ para alguna constante C, y por lo tanto SAT_H es simplemente SAT con un número de unos polinómico. Entonces una máquina de Turing determinista que resuelva SAT_H en tiempo polinómico puedo utilizarse para resolver SAT. Por el teorema de Cook-Levin, tendríamos que P=NP, que es una contradicción.

- Supongamos que SAT_H es NP-completo, entonces $SAT \leq SAT_H$. Es decir, existe una reducción de SAT a SAT_H en tiempo $O(n^i)$. Dado que SAT_H $\notin$P, sabemos que $lim_{n\to\infty} H(n) = \infty$. Puesto que la reducción funciona en tiempo $O(n^i)$, para n suficientemente grande aplicará fórmulas de SAT de longitud n en elementos de SAT_H de longitud

menor que $n^{H(n)}$. Entonces aplicará una fórmula suficientemente larga F en una cadena de la forma $F'01^{|F'|^{H(|F'|)}}$, con $F' \in SAT$ de longitud menor que n. Pero entonces esta reducción nos proporciona un algoritmo recursivo en tiempo polinómico para SAT, que implica que P=NP por el teorema de Cook-Levin, por lo que tenemos una contradicción.

Ejemplo 9 Hay muy pocos lenguajes candidatos a ser NP-intermedios. Conocemos dos: la factorización de enteros y el isomorfismo de grafos. Para ellos no se ha encontrado un algoritmo que los resuelva de forma eficiente y también hay bastante indicios de que no sean NP-completos.

6.3. Utopía P=NP

Hemos probado en el apartado anterior que basta con encontrar un algoritmo eficiente que resuelva cualquier problema NP-duro para que se tenga la igualdad P=NP. Con esto, todos los problemas en NP tendrían un algoritmo eficiente que los resolviese.

Esto implicaría que todo problema cuyas soluciones son verificables de forma eficiente puede ser resuelto también en tiempo polinómico. Nuestro mundo cambiaría completamente, ya que el software de inteligencia artificial sería prácticamente perfecto; podría efectuar búsquedas exhaustivas en un árbol de tamaño exponencial de forma eficiente. Todos los problemas de optimización podrían resolverse, la conducción autónoma sería posible disminuyendo prácticamente a cero el número de accidentes, los médicos sabrían qué medicamento y cuánta dosis sería la exacta para curarnos de una enfermedad, lo que aumentaría drásticamente nuestra esperanza de vida y un sinfín más de aplicaciones.

Afectaría también a la meteorología ya que se podría saber el tiempo que va a hacer con un año de antelación lo cual sería óptimo en cuestión de cosechas y turismo. En cuestión de economía, se podría saber predecir cómo va a comportarse el mercado con gran antelación y se podrían prevenir

las crisis. La ciencia avanzaría mucho más rápido, ya que a partir de datos experimentales podríamos crear una teoría más simple y exacta. Podríamos resolver ecuaciones diferenciales de forma eficiente, por lo que ya no serían necesarios los métodos numéricos de aproximación.

Hasta aquí la gran cantidad de ventajas que todo esto conllevaría, pero también acarrea algunas desventajas. Por ejemplo, la criptografía de las claves públicas sería descifrable fácilmente, por lo que tendríamos que inventarnos otro métodos de seguridad. Esto es muy peligroso, ya que si algún país consiguiese la solución de la conjetura P=NP, podría descifrar todo el sistema de seguridad del resto de países y podría llevar a grandes conflictos.

Generalmente, se ve este problema desde la perspectiva de que P=NP no es cierto, intentando probar lo contrario, ya que parece la solución más intuitiva. Pero si algo nos han demostrado las matemáticas es que la solución más lógica no siempre es la cierta. Por lo tanto, mientras no consigamos la prueba de una de las dos posibilidades, tendremos siempre que considerarlas ambas como posibles.

7. Conclusiones

Como comentábamos en la introducción, nuestro principal objetivo con esta memoria era explicar de una forma clara y concisa en qué consistía la complejidad algorítmica. Desde una perspectiva divulgativa, hemos intentado acercar un tema del que apenas se trata en el Grado de Matemáticas y que está obteniendo gran relevancia.

Desde un punto de vista académico, nuestros objetivos estan cubiertos: explicamos los conceptos básicos de complejidad y las clases principales; centrándonos en las clases P y NP, hemos indagado con algunos resultados sobre la conjetura P=NP, llegando a probar que el problema del viajante es un problema NP-completo, y hemos finalizado viendo qué supondría el hecho de que la conjetura fuese cierta.

En cuanto a los objetivos personales, creo haber crecido académicamente en varios ámbitos. Por una parte, me he adentrado en unos contenidos que no conocía y que me servirán para seguir conociendo el mundo de la computación. Y por otra parte, he aprendido a cómo gestionar una gran cantidad de información, resumiéndola y sabiendo qué extraer de cada parte. Hacer este trabajo también me ha enseñado a desarrollar y redactar mi propio contenido.

Para finalizar, quería agradecer a mi tutor César por guiarme en el desarrollo de este trabajo del que tanto aprendizaje he obtenido.

8. Referencias

[1] Balcázar, José L., Díaz, Josep y Gabarró, Joaquim. Structural Complexity I. 2ª edición. Editorial Springer. 1995. ISBN: 3-540-58384-X.

[2] Bovet, Daniel Pierre y Crescenzi, Luigi. Introduction to the theory of complexity. 1ª edición. Editorial Prentice Hall. 1993. ISBN-13: 978-0139153808.

[3] Gil Tolano, María Inés. Introducción a la teoría de las perturbaciones. Capítulo 1: Conceptos básicos. 2008. Universidad de Sonora. Tesis.

[4] Kiwi, Marcos. Problemas en P y NP. 2012. Universidad de Chile. Enlace: http://www.dim.uchile.cl/~mkiwi/ma50b/12/problemPandNP-handout.pdf

[5] Morán Cañón, Mario. Introducción a la Complejidad Computacional. Dirigido por: Philippe T. Giménez. Universidad de Valladolid. Trabajo de fin de carrera.

[6] Orellana Martín, David. Teoría de la Complejidad Computacional. Tema 4: Clases de complejidad computacional. 2022. Grupo de Investigación en Computación Natural. Departamento de Ciencias de la Computación e Inteligencia Artificial. Universidad de Sevilla. Enlace: https://www.cs.us.es/cursos/tcc-2022/temas/tcc-tema-4.pdf

[7] Orellana Martín, David. Teoría de la Complejidad Computacional. Tema 5: Problemas NP completos. 2022. Grupo de Investigación en Computación Natural. Departamento de Ciencias de la Computación e Inteligencia Artificial. Universidad de Sevilla. Enlace: https://www.cs.us.es/cursos/tcc-2022/temas/tcc-tema-5.pdf

[8] Peña Marí, Ricardo. El problema que los informáticos no han podido resolver en 45 años. 2017. Artículo, periódico El País. Enlace: https://elpais.com/tecnologia/2017/05/19/actualidad/1495202801_698394.html?event=go&event_log=go&prod=REGCRARTTEC&o=cerrtec.

[9] Pérez Jiménez, Mario. Teoría de la Complejidad Computacional. Tema 2: Modelos de Computación. 2020. Grupo de Investigación en Computación Natural. Departamento de Ciencias de la Computación e Inteligencia Artifi-

cial. Universidad de Sevilla. Enlace: http://www.cs.us.es/~marper/docencia/TCC-2019-2020/temas/tema-2-trans.pdf

[10] Rich, Elaine. Automata, Computability and Complexity: Theory and Applications. 1ª Edición. Editorial Pearson. 2008. ISBN-13: 978-0132288064.

[11] Veiga Losada, Francisco José. El problema del viajante. Dirigido por: Julio González Díaz. Santiago de Compostela: 2013. Universidad de Santiago de Compostela. Trabajo de fin de máster.

Fuentes web sin autor indicado:

[12] Complejidad Computacional Semana 3: P versus NP, Problemas NP intermedios. Artículo de la Universidad Nacional de Córdoba. Enlace: https://cs.famaf.unc.edu.ar/~hoffmann/icc12/slides/Slides030.html

[13] El problema del viajante de comercio. Artículo de la Universidad Nacional de la Plata. Enlace: http://sedici.unlp.edu.ar/bitstream/handle/10915/4059/1__El_problema_del_viajante_de_comercio.pdf?sequence=4&isAllowed=y

[14] Reducciones. Completitud en NP. Teorema de Cook-Levin. Artículo de la Universidad Nacional de Córdoba. Enlace: https://cs.famaf.unc.edu.ar/~hoffmann/md18/06.html#teorema-de-cook-levin

[15] Teorema de Cook Levin. Artículo basado en Wikipedia. Enlace: https://hmong.es/wiki/Cook-Levin_theorem