

Máquinas de Turing y Teoría de Decidibilidad

Pedro O. Pérez M., PhD.

Implementación de métodos computacionales
Tecnológico de Monterrey

pperezm@tec.mx

06-2022

① Problemas que las computadoras no pueden resolver

② Máquinas de Turing

¿Porqué estudiamos este tipo de problemas?

Notación de una máquina de Turing

Ejemplos

③ Extensiones de Máquinas de Turing

Técnicas de programación para las MT's

- Almacenamiento de estado

- Pistas múltiples

- Subrutinas

Extensiones de la máquina de Turing básica

- Máquina de Turing de varias cintas

- Tiempo de ejecución

4 Problemas intratables

Introducción

Clases P

Problemas resolubles en tiempo polinómico

Clases NP

Tiempo polinómico no determinista

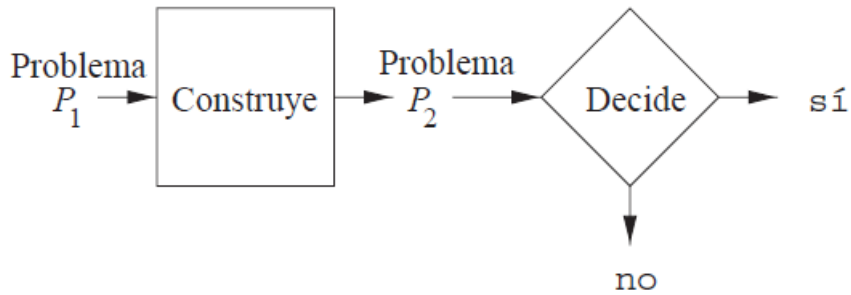
Problemas NP-completos

Otros problemas NP — *Completo*s

Problemas que las computadoras no pueden resolver

- Recuerdan “The Halting Problem: The Unsolvable Problem”. Con este problema, Dr. Turing demostró que ningún programa H puede informar si un determinado programa P con entrada I se detiene o se ejecuta por siempre.
- Pero, ¿cómo podemos demostrar que un nuevo problema, P_2 es indecidible? ¿Tenemos que usar una estrategia similar?

Basta con demostrar que si pudiéramos resolver el nuevo problema, P_2 , entonces utilizaríamos dicha solución para resolver un problema que ya sabemos que es indecidible, P_1 . Esta técnica se conoce como *reducción* de P_1 a P_2 .



El propósito de la teoría de los problemas indecidibles no es sólo establecer la existencia de tales problemas sino proporcionar también una guía a los desarrolladores sobre lo que se puede o no conseguir a través de la programación.

Existe otra razón.:

- La teoría también tiene un gran impacto práctico al tratar problemas que aunque sean decidibles, requieren mucho tiempo para ser resueltos. Estos problemas, conocidos como “problemas intratables”, suelen plantear una mayor dificultad al programador y el diseñador de sistemas que los problemas indecibles.
- Los problemas intratables son más comunes. Y, a menudo dan lugar a pequeñas modificaciones de los requisitos o soluciones heurísticas.

Veamos porqué decimos esto. Revisemos un segmentos del vídeo: “P vs. NP - The Biggest Unsolved Problem in Computer Science”.

Entonces, necesitamos herramientas que nos permitan determinar cuestiones acerca de la indecidibilidad o intratabilidad todos los días. Como resultado, necesitamos reconstruir nuestra teoría sobre indecidibilidad, basándonos en un modelo de computadora muy simple: la máquina de Turing.

Con la notación de la máquina de Turing, demostraremos que ciertos problemas, que aparentemente no están relacionados con la programación, son indecidibles. Por ejemplo, demostraremos que el “problema de la correspondencia de Post”, una cuestión simple que implica dos listas de cadenas, es indecidible y que este problema facilita la demostración de que algunas cuestiones acerca de las gramáticas, como por ejemplo la ambigüedad, son indecidibles.

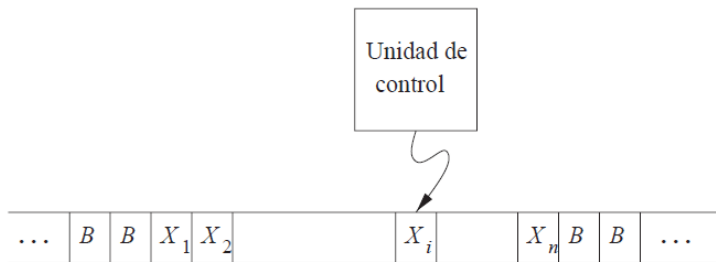
El intento de decidir todas las cuestiones matemáticas

- A finales del siglo XX, el matemático D. Hilbert se preguntó si era posible encontrar un algoritmo para determinar la verdad o falsedad de cualquier proposición matemática. En particular, se preguntó si existía una forma de determinar si cualquier fórmula del cálculo de predicados de primer orden, aplicada a enteros, era verdadera.
- Sin embargo, en 1931, K. Gödel publicó su famoso teorema de la incompletitud. Construyó una fórmula para el cálculo de predicados aplicada a los enteros, que afirmaba que la propia fórmula no podía ser ni demostrada ni refutada dentro del cálculo de predicados.

El cálculo de predicados no era la única idea que los matemáticos tenían para “cualquier computación posible”. De hecho, el cálculo de predicados, al ser declarativo más que computacional, entraba en competencia con una variedad de notaciones, incluyendo las “funciones recursivas parciales”, una notación similar a un lenguaje de programación, y otras notaciones similares. En 1936, A. M. Turing propuso la máquina de Turing como modelo de “cualquier posible computación”.

Lo interesante es que todas las propuestas serias de modelos de computación tienen el mismo potencial; es decir, calculan las mismas funciones o reconocen los mismos lenguajes. La suposición no demostrada de que cualquier forma general de computación no permite calcular sólo las funciones recursivas parciales (o, lo que es lo mismo, que las máquinas de Turing o las computadoras actuales pueden calcular) se conoce como *hipótesis de Church* (por el experto en lógica A. Church) o *tesis de Church*.

Básicamente, una máquina de Turing es un autómata finito que dispone de una única cinta de longitud infinita en la que se pueden leer y escribir datos. Una ventaja de la máquina de Turing sobre los problemas como representación de lo que se puede calcular es que la máquina de Turing es lo suficientemente simple como para que podemos representar su configuración de manera precisa, utilizando una notación sencilla muy similar a las descripciones de un autómata de pila.



- Inicialmente, la *entrada*, que es una cadena de símbolos de longitud finita elegidos del *alfabeto de entrada*, se coloca en la cinta. Las restantes casillas de la cinta, que se extiende infinitamente hacia la izquierda y la derecha, inicialmente almacenan un símbolo especial denominado *espacio en blanco*.
- Existe una *cabeza de la cinta* que siempre está situada en una de las casillas de la cinta. Se dice que la máquina de Turing *señala* dicha casilla. Inicialmente, la cabeza de la cinta está en la casilla más a la izquierda que contiene la entrada.
- Un *movimiento* de la máquina de Turing es una función del estado de la unidad de control y el símbolo de cinta al que señala la cabeza.
 - Cambiará de estado.
 - Escribirá un símbolo de cinta en la casilla que señala la cabeza.
 - Moverá la cabeza de la cinta hacia la izquierda o hacia la derecha.

La notación formal de una *máquina de Turing* (MT) es similar a la que hemos empleado para los autómatas finitos o los autómatas de pila. Describimos un MT mediante la siguiente séptula: $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ cuyos componentes tienen el siguiente significado:

- Q : El conjunto finito de *estados* de la unidad de control.
- Σ : El conjunto finito de *símbolos de entrada*.
- Γ : El conjunto completo de *símbolos de cinta*; Σ siempre es un subconjunto de Γ .
- δ : La *función de transición*. Los argumentos $\delta(q, X)$ son un estado q y un símbolo X . La función regresa, si está definido, la tripleta (p, Y, D) , donde p es el nuevo estado, Y es el símbolo de Γ que se escribe en la casilla y D es una *dirección* y puede ser L o R .
- q_0 : El *estado inicial*, un elemento de Q .
- B : El símbolo *espacio en blanco*.
- F : El conjunto de estados *de aceptación*.

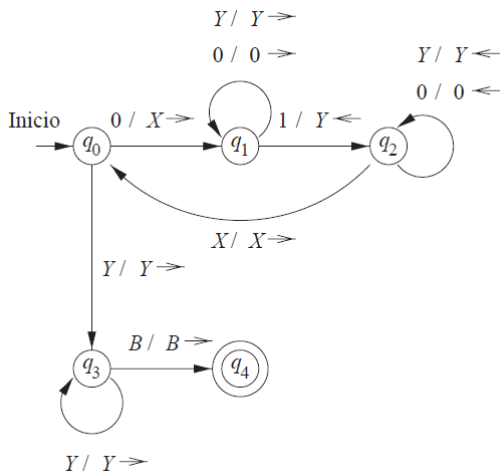
Vamos a diseñar una máquina de Turing y a ver cómo se comporta una entrada típica. La MT que vamos a construir aceptará el lenguaje $\{0^n 1^n \mid n \geq 1\}$. Inicialmente, se proporciona a la cinta una secuencia infinita de ceros y unos, precedida y seguida por secuencias de infinitas de espacios en blanco.

Comenzando por el extremo izquierdo de la entrada, se cambia sucesivamente un 0 por una X y se mueve hacia la derecha pasando por encima de todos los ceros y las letras Y que ve, hasta encontrar un 1. Cambia el 1 por una Y y se mueve hacia la izquierda pasando sobre todas las letras Y y ceros hasta encontrar una X . En esta situación, busca un 0 colocado inmediatamente la derecha y, si lo encuentra, lo cambia por una X y repite el proceso, cambiando el 1 correspondiente por una Y .

$$M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1\}, \{0, 1, X, Y, B\}, \delta, q_0, B, \{q_4\})$$

Estado	Símbolo				
	0	1	X	Y	B
q_0	(q_1, X, R)	—	—	(q_3, Y, R)	—
q_1	$(q_1, 0, R)$	(q_2, Y, L)	—	(q_1, Y, R)	—
q_2	$(q_2, 0, L)$	—	(q_0, X, R)	(q_2, Y, L)	—
q_3	—	—	—	(q_3, Y, R)	(q_4, B, R)
q_4	—	—	—	—	—

$$M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1\}, \{0, 1, X, Y, B\}, \delta, q_0, B, \{q_4\})$$



Ahora, veamos cómo una máquina de Turing puede calcular la función $\dot{-}$, que recibe el nombre de *sustracción propia* y se define mediante $m \dot{-} n = \max(m - n, 0)$. Es decir, $m \dot{-} n$ es $m - n$ si $m \geq n$ y 0 si $m < n$.

La MT empezará a trabajar con una cinta que conste de $0^m 10^n$ rodeada de espacios en blanco. La MT se parará cuando el contenido de la cinta sea $0^{m \dot{-} n}$ rodeado de espacios en blanco.

La MT encuentra repetidamente el 0 más a la izquierda que queda y lo reemplaza por un espacio en blanco. A continuación, se mueve hacia la derecha, en busca de un 1. Después de encontrar un 1, continúa moviéndose hacia la derecha hasta que llega a un 0, que sustituye por un 1. M vuelve entonces a moverse hacia la izquierda, buscando el 0 más a la izquierda, el cual identifica cuando encuentra un espacio en blanco a su izquierda, y luego se mueve una casilla hacia la derecha.

Este proceso se termina si:

- Buscando hacia la derecha un 0, encuentra un espacio en blanco. Esto significa que los n ceros de $0^m 10^n$ han sido todos ellos sustituidos por unos, y $n + 1$ de los m ceros han sido sustituidos por B . Se reemplazan los $n + 1$ unos por un 0 y n símbolos B , dejando $m - n$ ceros en la cinta. Dado que $m \geq n$ en este caso, $m - n = m - n$.
- Si al comenzar un ciclo, la MT no puede encontrar un 0 para sustituirlo por un espacio en blanco, porque los m primeros ceros ya han sido sustituidos por símbolo B . Entonces $n \geq m$, por lo que $m - n = 0$. Se reemplazan todos los unos y ceros restantes por símbolos B y termina con una cinta completamente en blanco.

Estado	Símbolo		
	0	1	B
q_0	(q_1, B, R)	(q_5, B, R)	—
q_1	$(q_1, 0, R)$	$(q_2, 1, R)$	—
q_2	$(q_3, 1, L)$	$(q_2, 1, R)$	(q_4, B, L)
q_3	$(q_3, 0, L)$	$(q_3, 1, L)$	(q_0, B, R)
q_4	$(q_4, 0, L)$	(q_4, B, L)	$(q_6, 0, R)$
q_5	(q_5, B, R)	(q_5, B, R)	(q_6, B, R)
q_6	—	—	—

Podemos usar una máquina de Turing para hacer cálculos de una manera muy diferente a como lo hace una computadora convencional. En particular, veremos que las máquinas de Turing pueden realizar, sobre otras máquinas de Turing, el 'tipo de cálculos que puede realizar mediante un programa al examen otros problemas.

Podemos emplear la unidad de control no sólo para representar una posición en el “programa” de la máquina de Turing, sino también para almacenar una cantidad finita de estados.

Por ejemplo, diseñemos una máquina de Turing que recuerde en su unidad de control el primer símbolo (0 o 1) que vea y compruebe que no aparece en ningún otro lugar de su entrada. Por tanto, la MT acepta el lenguaje $\{01^*|10^*\}$.

Para resolver este problema, consideraremos que la unidad de control no sólo consta de un estado de “control” (estado actual), sino que, además, puede almacenar tres elementos de datos A , B y C . Es decir, podríamos ver el estado como la tripleta $[q, A, B, C]$.

$$M = (Q, \{0, 1\}, \{0, 1, B\}, \delta, [q_0, B], \{[q_1, B]\})$$

El conjunto de estado Q es $\{q_0, q_1\} \times \{0, 1, B\}$. Es decir, puede pensarse en los estados como en pares de dos componentes:

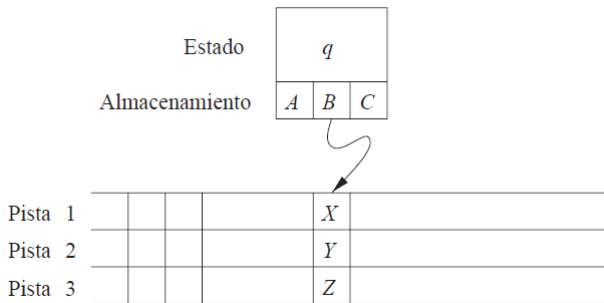
- a) Una parte de control, q_0 o q_1 , que recuerda lo que está haciendo la MT. El estado de control q_0 indica que todavía no se ha leído el primer símbolo, mientras que q_1 indica que ya se ha leído, y está comprobando que no aparece en ningún otro lugar, moviéndose hacia la derecha con la esperanza de alcanzar una casilla con un espacio en blanco.
- b) Una parte de datos, recuerda el primer símbolo que se ha visto, que tiene que ser 0 o 1. El símbolo B en este componente indica que no se ha leído ningún símbolo.

La función de transición δ es la siguiente:

- ① $\delta([q_0, B], a) = ([q_1, a], a, R)$ para $a = 0$ o $a = 1$. Inicialmente, q_0 es el estado de control y la parte de los datos es B . El símbolo que está siendo señalada se copia al segundo componente del estado y la MT se mueve hacia la derecha entrando en el estado de control q_1 .
- ② $\delta([q_1, a], \bar{a}) = ([q_1, a], \bar{a}, R)$ donde \bar{a} es el “complementario” de a , es decir, 0 si $a = 1$ y 1 si $a = 0$. En el estado q_1 , la MT salta por encima de cada símbolo 0 o 1 que es diferente del que se tiene almacenado en su estado y continúa moviéndose hacia la derecha.
- ③ $\delta([q_1, a], B) = ([q_1, B], B, R)$ para $a = 0$ o $a = 1$. Si llega al primer espacio en blanco, entra en el estado de aceptación $[q_1, B]$.

Observa que la MT no tiene definida una transición $\delta([q_1, a], a)$ para $a = 0$ o $a = 1$. Es decir, si encuentra una segunda aparición del símbolo almacenado inicialmente en su unidad de control, se para sin entrar en el estado de aceptación.

Otro “truco” práctico es pensar que la cinta de una máquina de Turing está compuesta por varias pistas. Cada pista puede almacenar un símbolo, y el alfabeto de la cinta consta de tuplas, con una componente para cada “pista”.

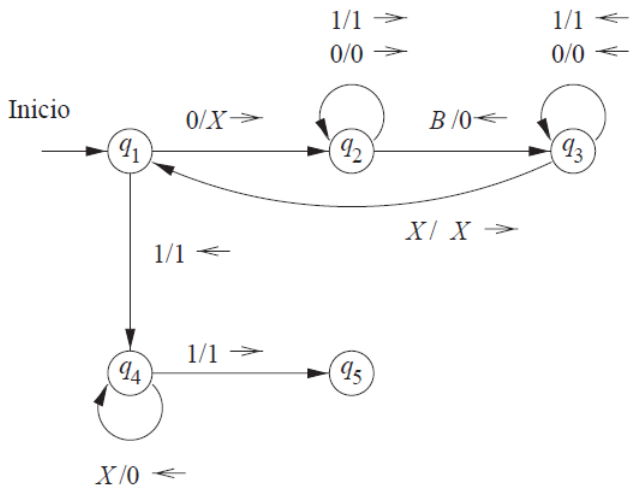


Un uso común de las pistas múltiples consiste en considerar una pista que almacena los datos y una segunda pista que almacena una marca. Podemos marcar cada uno de los símbolos como “utilizado”, o podemos seguir la pista de un número pequeño de posiciones dentro de los datos marcando sólo dichas posiciones.

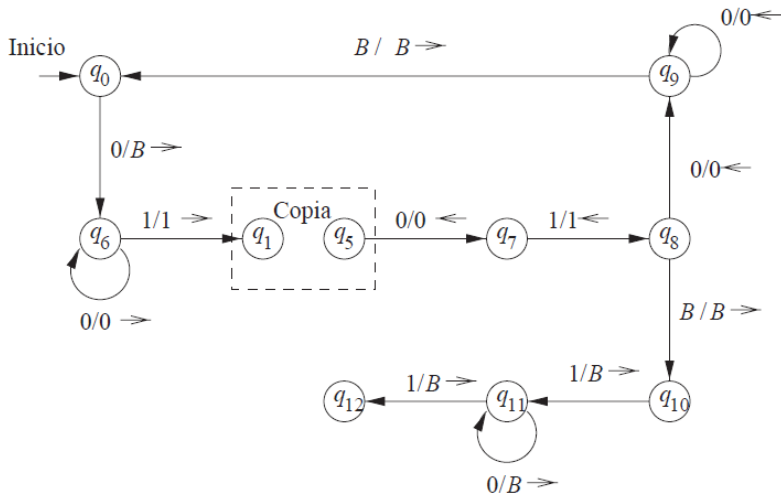
- Una subrutina de una máquina de Turing es un conjunto de estados que realizar un determinado procesos útil. Este conjunto de estados incluye un estado inicial y otro estado en el que, temporalmente, no existen movimientos, y que sirve como estado de “retorno”.
- La “llamada” a una subrutina se produce cuando existe una transición a su estado inicial.
- Dado que la MT no tiene ningún mecanismo que le permita recordar una “dirección de retorno”; es decir, un estado al que volver una vez que haya terminado, si la llamada a una MT que actúa como subrutina tiene que hacerse desde varios estados, se pueden hacer copias de la subrutina utilizando un nuevo conjunto de estados para cada copia.

Vamos a diseñar una MT para implementar la función “multiplicación”. La MT tendrá inicialmente en la cinta $0^m 10^n$ y al final deberá tener 0^{mn} . Un esquema de la estrategia es la siguiente:

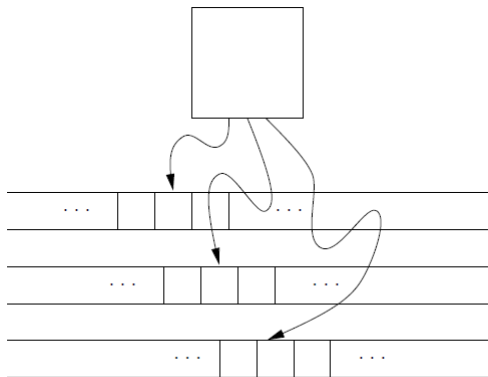
- 1 En general, la cinta contendrá una cadena sin ningún espacio en blanco de la forma $0^m 10^n 10^{kn}$ para un cierto k .
- 2 En el paso básico, cambiamos un 0 del primer grupo por un espacio en blanco B y añadimos n ceros al último grupo, obteniendo una cadena de la forma $0^{m-1} 10^n 10^{(k+1)n}$.
- 3 Como resultado, se habrá copiado m veces el grupo de n ceros al final, una por cada vez que hayamos cambiado un 0 por un espacio en blanco B . Cuando todo el primer grupo de ceros se haya cambiado por espacios en blanco, habrá mn ceros en el último grupo.
- 4 El último paso consiste en cambiar la cadena $10^n 1$ del principio por espacios en blanco.



MT Multiplicación



Máquina de Turing de varias cintas



El dispositivo tiene una unidad de control (estado) y un número finito de cintas. Cada cinta está dividida en casillas, y cada casilla puede contener cualquier símbolo del alfabeto finito. Al igual que en la MT de una sola cinta, el conjunto de símbolos de la cinta incluye el espacio en blanco y también dispone de un subconjunto de símbolos de entrada, al que no pertenece el espacio en blanco. Con este tipo de máquinas puede simularse computadoras reales.

El conjunto de estados incluye un estado inicial y varios estados de aceptación.
Inicialmente:

- 1 La entrada, una secuencia finita de símbolos de entrada, se coloca en la primera cinta.
- 2 Todas las casillas de las demás cintas contienen espacios en blanco.
- 3 La unidad de control se encuentra en el estado inicial.
- 4 La cabeza de la primera cinta apunta al extremo izquierdo de la entrada.
- 5 Las cabezas de las restantes cintas apunta a una casilla arbitraria.

El *tiempo de ejecución* de la máquina de Turing M para la entrada w es el número de pasos que M realiza antes de pararse. Si M no se para con la entrada w , entonces el tiempo de ejecución de M es w es infinito.

- Una de las malas interpretaciones que solemos tener con respecto al tema de indecidibilidad es que, a medida que pase el tiempo, las prestaciones de las computadoras seguirán creciendo indefinidamente en aspectos como el tamaño del espacio de direccionamiento, el tamaño de la memoria principal, etc.
- Si nos centramos en la máquina de Turing, en la que no existen estas limitaciones, podremos entender mejor la idea básica de que algún dispositivo de computación será capaz de hacerlo, si no hoy día, sí en un futuro.
- Entonces, dividiremos los problemas que puede resolver una máquina de Turing en dos categorías: aquellos que tienen un algoritmo (es decir, una máquina de Turing que se detiene en función de si acepta o no su entrada), y aquéllos que sólo son resueltos por máquinas de Turing que pueden seguir ejecutándose para entradas que no aceptan.

- Los problemas resolubles en un tiempo polinómico en una computadora típica son exactamente los mismos que los problemas resolubles en un tiempo polinómico en una máquina de Turing.
- La experiencia ha demostrado que existe una línea divisoria fundamental entre los problemas que se pueden resolver en un tiempo polinómico y los que requieren un tiempo exponencial o mayor. Los problemas prácticos que necesitan un tiempo polinómico casi siempre pueden solucionarse en un tiempo que podemos considerar tolerable, mientras que aquellos que precisan un tiempo exponencial, generalmente, no pueden resolverse excepto para casos sencillos.

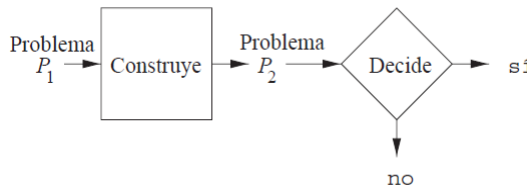
Problemas resolubles en tiempo polinómico

- Se dice que una máquina de Turing M tiene complejidad temporal $T(n)$ [o tiene un “tiempo de ejecución $T(n)$ ”] si siempre que M recibe una entrada w de longitud n , M se para después de realizar como máximo $T(n)$ movimientos, independientemente de si la acepta o no.
- Decimos que un lenguaje L pertenece a la clase P si existe alguna $T(n)$ polinómica tal que $L = L(M)$ para alguna máquina de Turing determinista M de complejidad temporal $T(n)$.

Una clase fundamental de problemas en el estudio de la intratabilidad es la de aquellos problemas que pueden resolverse mediante una MT no determinista que trabaja en tiempo polinómico. Formalmente, decimos que un lenguaje L pertenece a la clase NP (polinómico no determinista) si existe una MT no determinista M y una complejidad de tiempo polinómico $T(n)$ tal que $L = L(M)$, y cuando M recibe una entrada de longitud n , no existe ninguna secuencia de más de $T(n)$ movimientos de M .

Con el fin de tener una idea de la potencia de NP , vamos a ver un ejemplo de un problema que parece ser de clase NP pero no P : el problema del agente viajero (TSP, Traveling Salesman Problem). La entrada del TSP es la misma que la del árbol MWST, un grafo con pesos enteros en los arcos y un límite de peso W . La cuestión que se plantea es si el grafo contiene un “circuito hamiltoniano” de peso total como máximo igual a W . Un *circuito hamiltoniano* es un conjunto de arcos que conectan los nodos en un único ciclo, en el que cada nodo aparece exactamente una vez. Observe que el número de arcos de un circuito hamiltoniano tiene que ser igual al número de nodos del grafo.

Primero, debemos replantear el problema para que pueda ser resuelto por una *MT*:
“dado el grafo G y el límite W , ¿tiene ciclo hamiltoniano de peso W o menor?”. Por ejemplo, para este grafo, podríamos preguntar ¿existe un ciclo hamiltoniano de peso menor a 64?



Una forma de resolver esta pregunta sería probar todos los ciclos y calcular su peso total. Siendo inteligentes, podemos eliminar algunas de las opciones obviamente malas. Sin embargo, parece que da igual lo que hagamos, tendremos que examinar una cantidad exponencial de ciclos antes de poder concluir que no existe ninguno con el límite de peso deseado W , o de encontrar uno si no tenemos suerte al elegir el orden en el que examinemos los ciclos.

Si disponemos de una computadora no determinista, podremos conjeturar una permutación de los nodos y calcular el peso total para el ciclo de nodos en dicho orden. Si se tratara de una computadora real no determinista, ningún camino podría utilizar más de $O(n)$ pasos si la longitud de la entrada fuera n . En una *MT* de varias cintas, podemos elegir una permutación en $O(n^2)$ pasos y comprobar su peso total en una cantidad de tiempo similar. Por tanto, una *MTN* de una sola cinta puede resolver el problema *TSP* en un tiempo $O(n^4)$ como máximo. Concluimos que el problema *TSP* pertenece a *NP*.

A continuación vamos a abordar la familia de problemas que son candidatos bien conocidos para pertenecer a NP pero no a P . Sea L un lenguaje (problema) que pertenece a NP . Decimos que L es NP – *completo* si las siguientes afirmaciones sobre L son verdaderas:

- 1 L pertenece a NP .
- 2 Para todo lenguaje L' perteneciente a NP existe una reducción en tiempo polinómico de L' a L .

Es decir, un problema NP-completo es cualquier problema computacional para el que no se ha encontrado un algoritmo de solución eficiente.

El problema de satisfacibilidad (SAT)

Las *expresiones booleanas* se construyen a partir de:

- 1 Variables cuyos valores son booleanos; es decir, toman el valor 1 (verdadero) o el valor 0 (falso).
- 2 Los operadores binarios \wedge y \vee , que representan las operaciones lógicas *and* y *or* de dos expresiones.
- 3 El operador unitario \neg que representa la negación lógica.
- 4 Paréntesis para agrupar los operadores y los operandos, si fuera necesario para modificar la precedencia predeterminada de los operadores: \neg es el operador de mayor precedencia, le sigue \wedge y, finalmente, \vee .

Un ejemplo de una expresión booleana es $x \wedge \neg(y \vee z)$.

- Una *asignación de verdad* para una expresión booleana dada E asigna el valor verdadero o falso a cada una de las variables que aparecen en E . El valor de la expresión E para una asignación de verdad T se designa como $E(T)$, y es el resultado de evaluar E reemplazando cada variable x por el valor $T(x)$ (verdadero o falso) que T asigna a x .
- Una asignación de verdad T *satisface* la expresión booleana E si $E(T) = 1$; es decir, la asignación de verdad T hace que la expresión E sea verdadera. Se dice que una expresión booleana E es *satisfacible* si existe al menos una asignación de verdad T que satisface E .

El *problema de la satisfacibilidad* es: Dada una expresión booleana, ¿es satisfacible?

¿Qué sucede cuando descubrimos nuevos problemas *NP* — *Completo*s?

- Cuando descubrimos que un problema es *NP* — *completo*, sabemos que existen pocas posibilidades de que pueda desarrollarse un algoritmo eficiente para resolverlo. Es aconsejable y animamos a buscar soluciones heurísticas, soluciones parciales, aproximaciones u otras formas con el fin de evitar afrontar el problema directamente. Además, podemos hacerlo con la confianza de que no se nos está “escapando la solución adecuada”.

- Cada vez que añadimos un nuevo problema *NP – completo* a la lista, estamos reforzando la idea de que *todos* los problemas *NP – completos* requieren un tiempo exponencial. El esfuerzo que sin duda se ha dedicado a encontrar algoritmos en tiempo polinómico para un problema *P* ha sido, inconscientemente, un esfuerzo dedicado a demostrar que $P = NP$. Es el peso acumulado de los intentos sin éxito hechos por muchos matemáticos y científicos expertos para demostrar algo que es equivalente a $P = NP$ lo que fundamentalmente nos lleva al convencimiento de que es muy improbable que $P = NP$, el lugar de ello lo más probable es que todos los problemas *NP – completos* requieran un tiempo exponencial.

En grupos pequeños,

- 1 Revisa que otros problemas NP-completos existen.
- 2 Selecciona uno de estos problema (que no sea de los mencionados en clase) y revisa en que situaciones de la vida real podemos encontrar este problema.