Teoría de autómatas y lenguajes formales 2

TEMA 1 CALCULABILIDAD

Tema 1
Calculabilidad

1. Conceptos básicos
2. Enumerabilidad
3. Funciones y conjuntos recursivos
4. Conjuntos enumerables recursivamente
5. Reducciones
6. Ejemplos de problemas indecidibles

1. CONCEPTOS BÁSICOS

Como queremos poder determinar qué problemas admiten una solución algorítmica y cuales no, es conveniente ponernos de acuerdo previamente en los diferentes términos a utilizar:

1.1. Lenguajes

- **Alfabeto:** Conjunto finito y no vacío cuyos elementos se denominan **símbolos**. Para designarlo se utilizan letras mayúsculas del alfabeto griego, especialmente Σ y Γ. Como ejemplos de alfabetos, contamos con {0,1} alfabeto binario; {a,b,c…x,y,z} alfabeto latino de letras minúsculas…
- **Palabras:** Secuencia finita de símbolos de un alfabeto, cuando además queremos dejar claro que los símbolos que se utilizan son los de un alfabeto determinado, se habla de **palabras sobre un alfabeto**. Por ejemplo, según el alfabeto {a,b} son palabras de este alfabeto aba, aab, aaabbaaaa, a, b y λ (ésta última es una secuencia vacía de símbolos y representa la **palabra vacía**, por lo cual es una palabra sobre cualquier alfabeto).
- **Lenguajes, conjunto de palabras:** Un lenguaje es un conjunto de palabras sobre un alfabeto determinado, para designarlos se usa la letra L con subíndices si es necesario, y otras letras mayúsculas del alfabeto latino. Algunos ejemplos de lenguajes sobre el alfabeto {a,b} son L_1={a, aa, aaa}; L_2= {a, b, aa, bb, ab, abababab}; L_3={λ}. No obstante, un lenguaje no debe ser necesariamente finito, por ejemplo, lenguaje de todas las palabras sobre Σ={0,1} que comiencen por el símbolo 1.

1.2. Operaciones sobre lenguajes

- **Concatenación:** Concatenar dos palabras significa construir una palabra nueva añadiendo los símbolos de la segunda tras los símbolos de la primera, el operador de concatenación es el símbolo ·. Así aaa·bbb=aaabbbb, aba· λ=aba. La concatenación **no es conmutativa**, porque en general w1·w2<>w2·w1; sí tiene en cambio la **propiedad asociativa** y la palabra vacía λ constituye el **elemento neutro** de la concatenación. La concatenación de una palabra consigo misma se suele representar con notación exponencial, de manera que w^2=w·w; y la concatenación de una palabra y un símbolo se denota de la misma manera que la concatenación de dos palabras.
- **Longitud:** el número de símbolos de una palabra se desgina por |w| y es interesante reconocer que: |w1·w2|=|w1|+|w2| y |w|=0 ➔ w= λ.
- **Número de ocurrencias de un símbolo:** la notación $|w|_a$ denota el número de apariciones del símbolo a en la palabra w, así $|aabbabaa|_a = 5$.
- **Inversión:** Consiste en escribir al revés una palabra dada, la palabra resultante de la inversión se denomina inversa, si w es una palabra cualquiera w^R denota su inversa; así $(ab)^R$=ba, $(011101)^R$=101110, $λ^R$= λ. Fijémonos que $(w1 \cdot w2)^R = (w2)^R \cdot (w1)^R$.
- **Operaciones conjuntistas:** Estas operaciones son las siguientes: la unión, intersección, complementación y diferencia.
- **Concatenación:** de dos lenguajes es otro lenguaje que contiene todas las palabras que se pueden construir concatenando una palabra del primer lenguaje con una palabra del segundo lenguaje. Como ejemplos tenemos, si L1={a, aa} y L2={b, bb} entonces L1·L2={ab, abb, aab, aabb}. Entre las propiedades de la concatenación de lenguajes destacan:

  - La concatenación no es conmutativa, así L1·L2<>L2·L1
  - La concatenación sí es asociativa (L1L2)L3=L1L2L3
  - El elemento neutro de la concatenación es λ.
  - La concatenación es distributiva respecto de la unión.
  - Pero no lo es respecto de la intersección.
- **Concatenación de un lenguaje consigo mismo:** También se puede representar con la notación exponencial, así L^2=L·L.
- **Concatenación de una palabra y un lenguaje:** El concatenar una palabra y un lenguaje {w}L y L{w} puede abreviarse como wL y Lw y es el resultado de añadir una palabra w delante y detrás de cada una de las palabras de L. Misma aplicación para los símbolos aL y La.
- **Inversión:** El lenguje inverso L^R de otro L dado no es más que el lenguaje formado por los inversos de las palabras de L.

1.3. Funciones

Una relación (AxB) es una función cuando todo elemento de A está relacionado, como máximo, con un elemento de B. En cambio, un elemento de B puede estar relacionado con ninguno, uno o varios elementos de A.

Definimos **función característica** como aquella función asociada al conjunto A, que determina si un elemento pertenece al conjunto o no. La expresión formal la podemos ver en la figura lateral.

Función característica

$$\chi_a(x) = \begin{cases} 1 si x \in A \\ 0 si x \notin A \end{cases}$$

Definimos **función de codificación** como aquella que nos permite codificar un par de palabras de un lenguaje en una sola palabra; además debe permitir la descodificación posterior de manera unívoca; es decir, dos parejas diferentes de palabras no deben dar lugar al ser codificadas a una misma palabra y además este cambio es reversible.

1.4. Problemas

Entendemos por problemas un conjunto de preguntas relacionadas, tales que para cada una existe una respuesta finita. Un **problema de decisión** es aquel cuya respuesta es sí o no. Un **problema de cálculo** consiste en calcular el valor que toma una función dado un elemento de su dominio. Afinando un poco podríamos incluso restringirnos solo a problemas de decisión, dado que siempre podemos modelar el problema de cálculo diciendo ¿su solución es 5? Y seguir así hasta encontrar la verdadera, como vemos convertimos el problema de cálculo en problema de decisión; aunque a nosotros en esta asignatura lo que nos interesa es saber si un problema es calculable o no, pero no su resultado.

Una característica esencial es que todo problema puede codificarse en forma de lenguaje. Por ejemplo, el problema de decisión P consistente en determinar si un número es primo o no, se puede codificar con f(n)=SI si n es primo y f(n)=NO si n no es primo. Eso sí, el hecho de poder codificar un problema en forma de lenguaje no significa que lo hayamos resuelto.

1.5. Algoritmos

Entendemos por algoritmo (o **procedimiento efectivo**) un conjunto explícito y finito de reglas para resolver un problema. Podemos ver desde el exterior un algoritmo como una "caja negra" que dada una entrada produce una salida, de manera que calcula una función. Ojo, no confundir algoritmo con función; una función matemática es un conjunto, mientras que un algoritmo es un texto que da instrucciones sobre cómo hay que proceder desde las entradas para obtener los resultados; así un algoritmo siempre calcula una función, pero no toda función es calculable.

1.6. Cálculos

Cálculo es aquello que un ordenador hace. ¿y qué es un ordenador? Pues un ordenador para definirlo fuera de tecnologías y evoluciones actuales es una máquina de Turing: una máquina capaz de leer un símbolo de una cinta muy larga, con posibilidad de cambiar el símbolo leído por otro, posibilidad de hacer correr la cinta en los dos sentidos una posición cada vez.

1.7. Demostraciones

En esta asignatura veremos muchas demostraciones; estas demostraciones tienen como objetivo convencer al lector de que el resultado es cierto, pero también ayuda a entender porqué es cierto. Una mala demostración, aun siendo correcta no le da ninguna otra visión al lector que no tuviera antes y por tanto pierde validez. En toda demostración cada paso derivará de un hecho verdadero y ya probado usando principios generales de razonamiento.

Podemos utilizar dos tipos de demostraciones: la **demostración directa** que consiste en asumir en p→q que si p es cierto, q también lo es. La **demostración indirecta** es equivalente a la expresión, si p→q, entonces ¬q→¬p. Una variante de lo anterior es la **demostración por reducción al absurdo**, y consiste en demostrar que si determinado enunciado no es cierto, se desprende de él una contradicción.

No obstante, siempre que sea posible intentaremos realizar una demostración directa antes que una indirecta

1.8. Programas

A partir de ahora en lugar de utilizar el término algoritmo, utilizaremos el término programa, y nos servirá como modelo abstracto de cálculo. El lenguaje de programación elemental debe cumplir:

- Existen variables enteras y variables booleanas
- Existen expresiones aritméticas sencillas, constituidas por variables, números y/o operaciones (por ejemplo E = X + (y + 23)/2).
- Existen expresiones booleanas, formadas for operadores mayor que, menor que, igual, negación, etc y dos expresiones (Ejemplo: B = (x+4) >(y div 2).).
- La lista de instrucciones que tenemos es:
    - La instrucción skip se ejecuta sin modificar el resto del programa.
    - La instrucción de asignación "x:=E" hace que x tome el valor E.
    - La instrucción de concatenación "S1;S2 hace que primero se ejecute S1 y si termina correctamente, se ejecuta S2.
    - La instrucción alternativa "si B entonces S1, si no S2" empieza evaluando B y si es cierto se ejecuta S1 y si no lo es S2.
    - La instrucción iterativa "mientras B hacer S", hace que mientras B sea ciertos se ejecuta S, de lo contrario se ejecuta skip.

Para ser efectivos en esta asignatura solo consideraremos programas escritos en lenguaje de alto nivel, solo evaluaremos un solo parámetro de entrada que devuelva un valor; si este valor es booleano, resolveremos problemas de decisión, si tiene otro tipo, resolveremos con él problemas de cálculo.

---

2. ENUMERABILIDAD

Formalmente, un conjunto S es **enumerable** si es finito o si existe una función total y biyectiva f: N→S. Por ejemplo, el conjunto N es enumerable, una secuencia obvia es 0, 1, 2… El conjunto de todos los enteros también lo es 0, 1, -1, 2, -2…

La técnica más adecuada para demostrar la enumerabilidad o no enumerabilidad de una función es la **técnica de la diagnozalización.** Con esta técnica pretendemos construir una matriz; por ejemplo si pretendemos enumerar todo el conjunto de los números reales, buscamos una función tal que f: N →R. Así, por ejemplo a cada dígito del número decimal le aplicaremos la función de que valga 0 si ese número es distinto de 0 y valga 1 si es igual a 0. Lo que sucede es que cuando obtengamos por ejemplo el decimal en la diagonal 0,010; resulta que este mismo número debe ocupar un lugar en la matriz que estamos desarrollando, pero ese número no podrá existir; es decir ¿Cuál es el siguiente dígito que le deberemos añadir? Por propia definición si es un 1 o un 0 no cumplirá el requisito inicial que le hemos impuesto.

A partir de la explicación anterior sobre diagonalización, demostramos que si un conjunto es numerable, cualquier subconjunto suyo también lo es. Por ejemplo si A es un subconjunto de B, y B es enumerable entonces:

- A es enumerable si es finito
- A es infinito, en ese caso el conjunto del que es subconjunto B, también lo es; pero como B es numerable y existe la función f: N→ B que establece un orden de los elementos de B, también puede, por ello existir la función g: N→ A

Existencia de funciones no calculables algorítmicamente

El conjunto de las funciones calculables en la vida real es muy pequeño; hay innumerablemente más funciones no calculables que calculables. Este aspecto vamos a estudiarlo a través del **problema de la parada**. Este problema consiste en decidir si un programa se detendrá o no, por un valor de entrada determinado. Vamos a ver como este problema no puede resolverse algorítimicamente. Supongamos que tenemos un programa r cualquiera y un valor de entrada x para ese programa r. Con este valor de entrada x, el programa r puede detenerse o no. Ahora bien, tenemos otro programa p que sabe si el programa r se detiene con el valor de entrada x o no. Dado que x puede ser cualquier valor, x puede ser incluso el propio programa r (recordemos que cualquier función la podiamos codificar en el lenguaje que usa el ordenador y, por tanto, también se lo podemos pasar por parámetro.

Consideramos ahora otro programa q que vemos en el lateral. ¿Qué ocurre cuando q recibe q como entrada? ¿Se parara o no? Vamos a ver los dos supuestos:

Programa q

```
Programa q
  entrada r;
  si intérprete (p,<r,r>)=cierto
    entonces
      mientras 1=1 hacer
        buclear
      fmientras
    sino
      {detener el cálculo}
  fsi
salida x {valor indeterminado}
fprograma
```

- Si q(q) se para, entonces, p, que todo lo sabe, dira que q(q) se ha parado, pero en este caso q(q) ejecuta el bucle mientras, y por tanto q(q) no se para. Hemos llegado a una contradicción; si repasamos las dos suposiciones que hemos hecho es que p existe (el programa que sabe si un programa se para o no según un valor de entrada) o que q(q) se para; por tanto veamos lo que ocurre si q(q) no se para
- Si q(q) no se para, p que todo lo sabe, dirá que q(q) no se ha parado, pero realmente q(q) va al sino y se detiene. Por tanto se ha parado. Como hemos vuelto a obtener una contradicción, no queda más remedio que definir como errónea la suposición de que p existe. Por ello, el problema de la parada es indecidible.

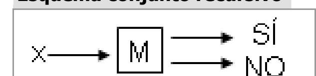
Esto implica que no existe ningún "programa buscador de bucles" que examine programas de ordenador y conjuntos de datos posibles y determine si el programa podrá entrar o no en un bucle infinito.

3. FUNCIONES Y CONJUNTOS RECURSIVOS

Una función es **recursiva parcial** si hay algún programa que la calcula con un lenguaje de programación que cumpla las condiciones ya especificadas; y es **recursiva total** si además de ser recursiva parcial, está definida para todas las entradas posibles. Así, la función "producto de 2 naturales" es recursiva total; mientras que la función "división entera, div" es recursiva parcial pero no total.

Decimos que L es un **conjunto recursivo** si su función característica es recursiva total; es decir, si existe un algoritmo (como el del esquema del margen) que pueda decidir si pertenece a él o no. La clase de conjuntos recursivos se denomina REC. Los **problemas decidibles** son aquellos que se dan cuando, una vez enunciados representando sus soluciones en forma de conjunto, este conjunto es recursivo. Son ejemplos de conjuntos (lenguajes) recursivos los números impares, los números primos, los finitos, los regulares y los incontextuales.

Esquema conjunto recursivo



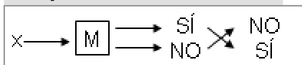
Veamos dos ejemplos:

- F(x)=1 si el teorema de Fermat es cierto y f(x)=0 si es falso; es recursiva total porque para toda x tiene un valor constante 0 ó 1; ahora bien, que no sepamos

cual de los dos valores según Fermat debamos aplicar no significa que el algoritmo no exista, sino que no sabemos cual es.
- G(n)=1 si existen n números 7 seguidos en el desarrollo decimal de Pi; 0 en caso contrario. G es una función legítima pero ¿es calculable? Pues el problema está en que a pesar de poner existir n números 7 seguidos en el desarrollo de Pi, pueden llegar muy tarde y haber decidido nosotros parar el algoritmo antes. Es por ello que la idea de **procedimiento efectivo** implica que el procedimiento debe llevarse a cabo en una serie de pasos cada uno de los cuales se completa en un tiempo finito, y que cualquier salida debe emerger después de un número finito de pasos.

Las **propiedades de los conjuntos recursivos** son las siguientes:

Complementariedad



- Dado un conjunto recursivo A, su complementario es también un conjunto recursivo. La idea es muy simple. Si tenemos un programa que determina si un elemento pertenece o no a A, podemos construir otro que determine si un elemento pertenece o no al complementario, simplemente invirtiendo los valores, tal como se ve en la figura lateral.
- Dados dos conjuntos recursivos, su unión y su intersección también lo es; y será recursivo total si A y B también lo eran.

Parametrización

```
Función f(y:entero) devuelve entero es
  Const x:=3 fconst;
  Var r: entero fvar;
  R:= x+y;
Devuelve r;

Función f(x,y:entero) devuelve entero es
  Var r:entero fvar;
  R:= x+y;
Devuelve r
```

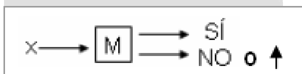
Un **programa enumerador** es un programa que, sin entrada, genera una lista (posiblemente infinita) de palabras. Un lenguaje A es recursivo si, y solo si, A es generado en orden y sin repeticiones por un programa enumerador.

El **teorema de parametrización** establece que siempre podemos particularizar un programa (función) eliminando un parámetro de entrada y estableciéndolo como un valor constante. Así el primer programa y el segundo, harán lo mismo para el valor de x=3.

---

4. CONJUNTOS ENUMERABLES RECURSIVAMENTE

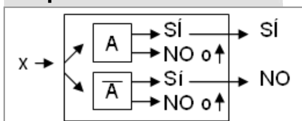
Conjunto enumerable recursivamente



Decimos que un programa p acepta L cuando se cumple que:
- x pertenece a L → p(x) es cierto.
- X no pertenece a L → p(x) es falso o p(x) no acaba

Igualmente decimos que L es un conjunto **enumerable recursivamente** si hay un programa que acepta L. Observamos que un conjunto recursivo siempre es enumerable recursivamente. La clase de los conjuntos enumerables recursivamente se denomina ER y se cumple que REC $\subset$ ER. Los **problemas semidecidibles** son aquellos que se dan cuando, una vez enunciados de manera que representen sus soluciones en forma de conjunto, este conjunto resulta ser enumerable recursivamente.

Teorema del complementario



El **teorema del complementario** establece que si un problema es semidecidible y su complementario también lo es, entonces el problema es decidible; y dice que A es recursivo si, y solo si, A es enumerable recursivamente y su complementario también es enumerable recursivamente.

La **propiedad de clausura** afirma que dados dos conjuntos A y B tales que A, B pertenecen a ER, entonces, la unión de ambos pertenece a ER y la intersección también.

La caracterización de los conjuntos enumerables en términos de **programas enumeradores** es posible gracias a la propiedad siguiente: un lenguaje (conjunto de palabras) A es enumerable recursivamente si, y sólo si, A es generado por un programa enumerador (no necesariamente en orden y quizá con repeticiones). La demostración es sencilla: si A es enumerable recursivamente, entonces por definición, existe un programa p que lo acepta; construiremos un

nuevo programa sobre la base de p que en lugar de aceptar las palabras, las vaya listando de una en una; así si se pregunta a p si una palabra pertenece al conjunto dirá que sí y en ese caso la listaremos; en caso contrario no la listaremos. Lo único que puede suceder es que haya repeticiones porque probablemente probemos una misma palabra varias veces y también saldrá desordenado porque el tiempo que tarda p en aceptar puede ser distinto para cada palabra. Así tenemos las siguientes caracterizaciones todas ellas equivalentes:

- A es enumerable recursivamente.
- A es la imagen de una función recursiva.
- A es conjunto vacío o bien A es la imagen de una función recursiva total.
- A es el dominio de una función recursiva.
- A es generado por un programa, no necesariamente en orden y tal vez con repeticiones.

Diremos que P es un **predicado recursivo** si este conjunto asociado que lo define lo es. Un predicado es una sentencia con variables que podemos clasificar como cierta o falsa, por ejemplo $x^2=4$. El conjunto de soluciones se puede expresar así, P = { x | $x^2=4$ }.

La siguiente función se denomina **función reloj**:

R(x,y,t)= Cierto si $P_x(y)$ se para en t o menos pasos
Falso si $P_x(y)$ no se para en t o menos pasos

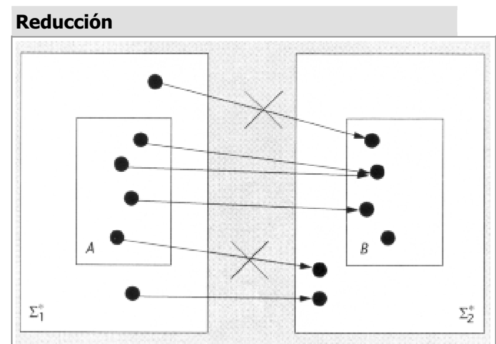
Esta función reloj es claramente recursiva total; solo hay que simular P_x con entrada y durante t pasos, al cabo de los cuales nos paramos tanto si P_x se había parado como si no, y contestamos cierto o falso respectivamente. Es decir, esta función nos dice si el programa P_x se detiene con una determinada entrada y en t o menos pasos.

5. REDUCCIONES

Un conjunto **A es reducible a otro conjunto B** si podemos decidir la pertenencia a A a partir de saber decidir la pertenencia a B y, por tanto, el problema de decisión para A no es más complicado que el de B. Entendemos que A es un caso especial de B, y que para resolver A solo hay que codificar sus enunciados como enunciados de B. Esta reducción debe cumplir las condiciones siguientes:

- Todo enunciado de A se transforma en un enunciado de B. Puede ocurrir que varios enunciados de A den el mismo enunciado de B; pero no puede ocurrir que un enunciado de A no dé un enunciado de B.
- No puede ocurrir que una cosa que no es un enunciado de A se convierta en un enunciado de B. En la imagen lateral se reflejan gráficamente estas condiciones.

Reducción



Un ejemplo de reducción puede ser por ejemplo que si ya tenemos un programa A que es capaz de calcular cuando un número es par, bastará una pequeña modificación para construir otro programa B que me indique cuando es impar f(x):= x+1.

Propiedades

$$\left.\begin{array}{l} A \leq_m B \\ B \in REC \end{array}\right\} \Rightarrow A \in REC$$

$$\left.\begin{array}{l} A \leq_m B \\ A \notin REC \end{array}\right\} \Rightarrow B \notin REC$$

Propiedades interesantes de la reducción son:

- Dado un conjunto A recursivo y un conjunto B tal que no es conjunto vacío y no es Σ^*, entonces A $\leq_m$ B (A es reducible a B).
- Si el conjunto A es reducible a B y B pertenece al conjunto de los recursivos, entonces a también pertenece al conjunto de los recursivos. Esta propiedad la podemos usar para probar la indecidbilidad de ciertos problemas si conocemos

la indecibilidad de otros; para ello usaremos la propiedad al revés, es decir, Si A no es más dificil que B (A es reducible a B) y es indecidible, B no puede ser decidible, porque sino A lo sería.

- $A \leq_m A$. Es decir, la m-reductibilidad es reflexiva.
- $A \leq_m B$ y $B \leq_m C$, entonces $A \leq_m C$. La m-reductibilidad es transitiva.
- La m-reductibilidad no es ni simétrica ni antisimétrica.

Un conjunto es más completo si es el más "difícil" de su clase. Es decir, la **completitud** indica que A es m-completo para L si se cumple que $A \in L$ y que para todo $B \in L$, $B \leq_m A$.ç

El conjunto de para K es completo en la clase ER. La idea fundamental de todo esto es que es posible ver que un conjunto A no es recursivo si reducimos K a A. También para ver que un conjunto no es siguiera enumerable recursivamente podemos reducir $\overline{K}$ a A.

---

6. EJEMPLOS DE PROBLEMAS INDECIDIBLES

En este apartado demostraremos la indecibilidad de algunos problemas aplicando los teoremas del apartado anterior. Muchas demostraciones son parecidas y un teorema reúne estos parecidos en un caso general; si una vez hemos demostrado un teorema y ahora lo podemos aplicar particularizándolo para nuestro caso concreto, podremos sustituir un fragmento de nuestra demostración por el teorema.

El **teorema de Rice** explica que sea P el conjunto de todas las funciones recursivas (totale sy no totales), si S es un subconjunto de índice de las funciones de cierto subconjunto $S \subseteq P$, entonces se cumple que:

S es recursivo $\Leftrightarrow$ S = Conjunto vacío o S = P

Por ejemplo, consideremos el conjunto iny = {x|Im_x es inyectiva}; que una función sea inyectiva significa que nunca devuelve un mismo valor de salida para dos valores de entrada diferentes. Consideremos ahora f(x) = X y g(x)=1. Las dos son claramente recursivas (calculables con un algoritmo). Además, f es inyectiv ay g no; por tanto iny no es ni el conjunto vació (porque existe f) ni el conjunto de todas las funciones recursivas (porque existe la función recursiva g que no pertenece al conjunto de las inyectivas); así pues, por el teorema de Rice, iny no es recursiva.

TEMA 2 COMPLEJIDAD

Tema 2
Complejidad
1. Medidas de complejidad
2. Reducción y Completitud
3. La clase NP-completo

1. MEDIDAS DE COMPLEJIDAD

En esta parte del módulo y visto que ya sabemos de la existencia de problemas indecidibles, estudiaremos los decidibles. Dentro de ellos, encontraremos dos grandes tipos: los tratables (que pueden ser resueltos en un tiempo/espacio "aceptable") y los intratables (no son resueltos de manera aceptable).

El concepto de intratabilidad

En este momento estudiaremos la complejidad de un problema como la complejidad del mejor algoritmo para la peor entrada posible de se problema. Hemos de tener en cuenta que todo problema decidible tiene un algoritmo que lo resuelve, pero algunos de ellos lo harían en un tiempo prohibitivo, desaforado. Es decir todos los problemas decidibles, no son igual de decidibles. Tenemos problemas intratables (aquellos cuyo mejor algoritmo termina en un tiempo prohibitivo) y los problemas tratables (terminan en un tiempo aceptable). Ahora bien ¿Dónde se fija la frontera entre unos y otros? Pues generalmente se fija en el coste polinómico y en el exponencial. Generalmente para valores más o menos grandes un coste polinómico puede ser aceptable, pero un coste exponencial nos dispara el coste de resolución del problema Por ello **un problema intratable** (clase NP) es aquel para el que no existe un algoritmo de tiempo polinómico que lo resuelva, es decir, su complejidad es exponencial; y un **problema es tratable** si existe un algoritmo te tiempo polinómico que lo resuelve (clase P).

No obstante todo lo anterior, existen algoritmos de tiempo exponencial que son aplicables en la práctica, debido a que nosotros definimos la complejidad a partir del peor caso, lo que significa que como mínimo una instancia del problema tiene este coste, pero muchas otras instancias pueden requerir un tiempo bastante menor, y si se ha estudiado mucho el problema, se pueden conocer las características que debe cumplir la entrada para que el algoritmo sea aplicable en la práctica.

No determinismo

La complejidad de los problemas viene especificada por 3 parámetros: el modelo de cálculo (máquina de Turing, lenguaje de programación…), el recurso (tiempo o espacio principalmente) y la cota o función correspondiente. Pero nos falta introducir un cuarto parámetro: el **modo de computación**, y solo se conocerán dos: el modelo determinista y el modo no determinista.

La idea principal de las máquinas **indeterministas** es la siguiente: una entrada o lenguaje si acepta si existe alguna secuencia de opciones no deterministas que nos responde "sí" (o mejor dicho, nos lleva a un estado de aceptación). Puede haber otras opciones que nos lleven a rechazar la entrada, pero con que solo una de las posibles, la acepte, es suficiente para aceptarla. La entrada es rechazada o no pertenece al lenguaje si no existe ninguna secuencia de opciones que nos lleve a la aceptación. Esto es muy importante, pues se utiliza mucho en la lógica (una sentencia debe intentar todas las posibles demostraciones y solo con que una lo acepte, quedará demostrado), la inteligencia artificial, etc.

La clase de todos los problemas que tienen una solución algorítmica no determinista con tiempo de ejecución polinómico respecto a la entrada, recibe el nombre de clase NP y la estudiaremos posteriormente.

Clases de complejidad

Dada una función de complejidad f(n), se consideran las clases de complejidad siguientes:

- TIME (f(n)): complejidad temporal f(n)

- SPACE (f(n)) Completedidad en espacio f(n)
- NTIME (f(n)): Complejidad no determinista en tiempo f(n)
- NSPACE (f(n)): Complejidad no determinista en espacio f(n).

Conviene saber, sobre todo, las relaciones que se dan entre estas clases:

- SPACE (f(n)) $\subseteq$ NSPACE (f(n))
- TIME (f(n)) $\subseteq$ NTIME (f(n))
- TIME (f(n)) $\subseteq$ SPACE (f(n)), ya que en t pasos como mucho se pueden examiar t+1 casillas, y el peor de los casos es que todas las casillas visitadas sean diferentes.
- Si f(n)>=$\log_2$(n) entonces SPACE (f(n)) $\subseteq$ TIME ($C^{f(n)}$)
- NTIME (f(n)) $\subseteq$ TIME ($C^{f(n)}$); esto expresa algo muy importante y que ya sabíamos, y es que la falta de no determinismo se paga con una complejidad exponencial.

Existen dos clases parametrizadas lo suficientemente importantes como para tener un nombre propio:

- TIME (n^k) conocida normalmente como clase P, es la clase de los problemas tratables, para los que existe un algoritmo determinista de tiempo polinómico.
- NTIME(n^k) conocida como clase NP para los que existe un algoritmo no determinista de complejidad temporal polinómica.

No obstante, quedan todavía muchos frentes abiertos y que tendríamos que ser capaces de demostrar, queda claro que P$\subseteq$NP, pero nadie ha conseguido aun demostrar ni que P=NP ni que P!=NP. Todo hace pensar esto último, pero de momento, solo puede considerarse una conjetura. Sí se ha llegado a demostrar que PSPACE = NSPACE, es decir, el no determinismo no aporta ninguna potencia adicional desde el punto de vista de la complejidad en espacio polinómica. A modo de resumen:

$$L \subseteq P \subseteq NP \subseteq PSPACE \subseteq NPSPACE$$

Algunos problemas intratables

Generalmente vamos a intentar tratar dos tipos de problemas: los problemas de decisión y los problemas de optimización. Un **problema de decisión** es aquel que contiene una instancia genérica en términos de algoritmos, grafos… y una segunda parte a la que hay que responder Si ó No. Un **problema de optimización** es aquel en el que a partir del problema de decisión, se quita la cota asociada a ese problema y se quiere conocer el valor mínimo que puede llegar a tener esa cota. Existen así varios tipos de problemas a tratar:

- **El problema del viajante de comercio**: Consiste en dada unas ciudades y una distancias entre ellas, ¿existe una ruta que visite todas ellas saliendo y llegando a una misma ciudad y pasando por cada ciudad una sola vez con una distancia total no superior a B? Si consideramos esta pregunta como un grafo completo etiquetado positivamente podemos preguntarnos entonces si existe un circuito hamiltoniano es ese grafo de coste acumulado no mayor que B. Es un problema NP dado que hay que pensar en tener una instrucción de opción no determinista de una entre todas las posibles permutaciones y posteriormente comprobar que tenga un coste menor o igual a B. El cálculo tiene un coste lineal respecto al número de ciudades n (Clase NP). Si no disponemos de la opción no determinista, no queda más remedio que realizar el cálculo para cada una de las permutaciones posibles hasta encontrar una con un coste no superior a B. En el peor de los casos debemos hacer todos los cálculos (coste n) para todas las permutaciones (exponencial). Por ello, el algoritmo determinista tiene complejidad exponencial. Hablando con propiedad

debemos decir que es un problema intratable ya que no se ha encontrado una solución algorítmica determinista de coste polinómico, aunque también es cierto que nadie ha demostrado que no exista.

A partir de aquí tenemos el problema de optimización asociado consistente en preguntar cual es el valor mínimo de un ciclo hamiltoniano; es decir, prescindimos de la cota B y pretendemos encontrar la mínima distancia que nos permita recorrer todas las ciudades. El coste mínimo del problema de optimización asociado tiene, como mínimo, la misma complejidad que el problema de decisión.

- **Problema del coloreado de un grafo**: Dado un grafo, se trata de asignar un valor de 1 a K a cada número de color distinto de los vértices del grafo sin que dos vértices adyacentes tengan el mismo color. A menudo se puede tratar del problema de un mapa para colorear los países (nos seguimos encontrando en el mismo caso que el grafo). Existe con claridad un algoritmo no determinista de complejidad polinómica que lo resuelve. El coste de dicha función que comprueba que dos vértices adyacentes no tienen el mismo color tiene un coste cuadrático (es decir, polinómico); pero no existe un algoritmo determinista de complejidad polinómica que lo resuelva, por tanto también es de clase NP.

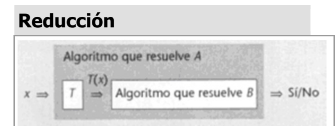
---

2. REDUCCIÓN Y COMPLETITUD

La teoría de la NP-completitud se centran en demostrar resultados de la forma "si P!=NP, el problema pertenece a NP-P" y este formato condicional no le quitado fuerza a la teoría. Necesitamos, no obstante, una noción precisa de qué significa para un problema que sea "como mínimo tan difícil como otro"; así conociendo la complejidad de algunos problemas, definiremos como de complejidad equivalente la de otros.

La **Reducción de tipo polinómico** es una herramienta muy utilizada en TALF; vamos a reducir un problema A (con entrada x) a otro B con una entrada equivalente T(x), si para resolver A por la entrada x solo hay que calcular T(x) y resolver B por ella; tal y como se puede observar en el esquema lateral. Parece razonable pensar que si el problema A se reduce al problema B, entonces B es como mínimo tan difícil como A. Así podremos decir que si existe una reducción polinómica de A a B diremos que A es reducible polinómicamente a B y lo señalaremos como $A \leq_p B$. La importancia de esta reducción polinómica proviene de la propiedad siguiente: si $A \leq_p B$ entonces $B \in P$ implica que $A \in P$ (y también para NP).

Reducción



O sea, si A es reducible polinómicamente a B y existe un algoritmo en tiempo polinómico que resuelve B, entonces existe un algoritmo en tiempo polinómico que resuelve A.

Con todo lo anterior, podemos imaginarnos que la mayor parte de la dificultad ahora estriba en encontrar ese problema B del que ya conocemos su coste y al que queremos reducir polinómicamente. Encontrar la función de transformación es lo más difícil

Otra noción importante es la **condición de transitividad**, que se expresa formalmente: si $A \leq_p B$ y $B \leq_p C$, entonces $A \leq_p C$. Por funciones. Si f es la función de reducción del problema A al problema B y f´es la dunción de reducción del problema B al problema C, entonces la composición f·f´ es la función de reducción del problema A al C. También existe **reciprocidad** es decir, dos problemas de decisión sistema operativo son problemas polinómicamente equivalentes cuando se de que $A \leq_p B$ y $B \leq_p A$.

Completitud: La transitividad de la reducción polinómica establece una relación de equivalencia $\leq_p$ y ello impone un orden parcial en las clases de equivalencia, donde la clase P forma la "menor" clase de equivalencia y puede verse como la clase que contiene los problemas de decisión "más fáciles" de

resolver. La clase de problemas NP-comletos formará otra clase de equivalencia, caracterizada por contar con los problemas de decisión más difíciles de la clase NP.

Sea C una clase de complejidad y sea P un problema incluido en la clase, diremos que P es un problema C-completo si todo problema P´$\in$ C se puede reducir a P. El hecho de que existan problemas naturales (no rebuscados) importantes que sean completas para una clase, da significado a esa clase; por el contrario, si no los hay, podemos pensar que se trata de una clase supérflua.

La propiedad más importante de los problemas completos es que si un problema completo de una cierta clase C pertenece a una clase menor C´$\subseteq$ C cerrada respecto a la reducción, entonces toda la clase C coincide con la clase menor C´. Ya sabemos que P es cerrada respecto a la reducción, por tanto, si algún problema NP-completo se demuestra que pertenece a P (se encuentra una solución algorítmica de tiempo polinómico para él), habremos demostrado que P=NP y seremos mundialmente famosos.

3. LA CLASE NP-COMPLETO

Nos situamos ya de lleno en la **clase NP**, que son la clase de problemas que pueden resolverse con un algoritmo no determinista de tiempo polinómico. En los problemas anteriores hemos observado como una vez conjeturada una solución, su verificación se hace en tiempo polinómico, pero si no disponemos de la posibilidad de conjeturar hay que generar todas las soluciones posibles, que en el peor de los casos es de orden exponencial, y verficarlas una a una.

Por ello, a alguien se le podría ocurrir utilizar la clase EXP en lugar de la NP, porque está claro que NP$\subseteq$EXP; pero es que hay problemas en EXP que no están en NP, la inclusión no es estricta. En NP hay problemas completos y además muchos comunes e importantes. Dentro de NP a nosotros nos interesa especialmente la clase NP-completo, porque incluye los problemas más difíciles de NP (y si alguno de ellos estuviese en P supondría la desaparición de la clase), y también porque hay muchos y de gran aplicabilidad; debemos tener en cuenta que la mayoría de los problemas interesantes de la clase NP son NP-completos.

No obstante, todavía podemos definir un tipo de problema más el **problema NP-difícil**, Se dice que A es un problema NP-difícil si todo problema en NP es reducible polinómicamente a él, es decir si para todo B $\forall B \in NP,$ B $\leq_p$ A.

Fijémonos en que NP-difícil no es lo mismo que NP-completo, ya que el problema no tiene que pertenecer necesariamente a NP. UN problema es NP-difícil si, como mínimo es tan difícil como cualquier problema de la clase NP, pero puede ser peor, más difícil y, por ello, estar fuera de la clase NP. Se dice por ello que A es un problema NP-completo si cumple que A $\in$ NP y que A es NP-difícil.

Hay que tener cuidado también con las inclusiones de unas clases en otras. Como P$\subseteq$NP, si un problema es P, también es NP y si es NP también puede ser P. Ahora bien, normalmente se dice que es NP sólo cuando no es también P, es decir, utilizamos NP como equivalente de NP-P, y para el resto reservamos el nombre P.

Teorema de Cook

El honor de ser el primer problema NP-completo, recayó en un problema de decisión del terreno de la lógica el **problema de la satisfactibilidad**. Tiene el planteamiento siguiente. Un conjunto de variables V y una fórmula F sobre V en forma normal conjuntiva, ¿existe pues una asignación de valores a las variables tal que evalúe F a cierto?

Básicamente consiste en responder si hay una combinación de 0 y 1 asignados a las variables que hagan que la fórmula se evalúe a cierto. Un algoritmo no determinista debe conjeturar una asignación de valores a las variables y posteriormente verificar que esta asignación satisface todas las cláusulas de F. La verificación consiste en evaluar F aplicando las tablas de verdad y eso puede

hacerse en tiempo polinómico. Ahora bien, debemos demostrar que todo problema NP puede reducirse al problema SAT, esto ya no es tan trivial y no lo veremos en esta asignatura.. Si no contamos con la opción no determinista, al algoritmo no le queda más remedio que probar todas las posibles combinaciones (para n variables será 2^n) y el coste por tanto será exponencial.

Pues bien, la importancia del teorema de Cook radica en dar este primer problema NP-completo que cumple que pertenece a NP y como sabemos que $\forall B \in NP$, B $\leq_p$ A.; entonces B también es NP-completo.

Problemas NP-completos básicos

Estudiamos varios problemas NP-completos, en total 6. Lo que buscamos es demostrar que algún problema que ya sabemos que es NP-completo puede reducirse polinómicamente al nuevo problema. Los problemas que más se han usado para demostrar que otros también son NP-completos son los 6 que veremos a continuación:

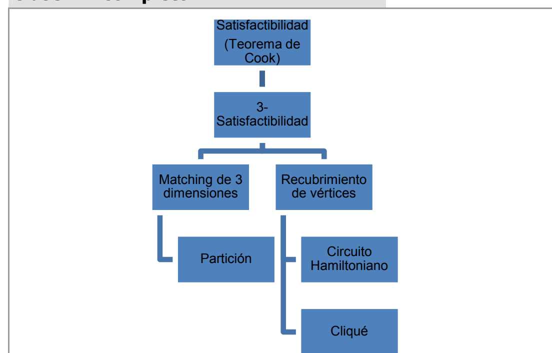
1. **Problema de la 3-satisfactibilidad**: Dado un conjunto de variables V y una fórmula F sobre V en forma normal conjuntiva tal que cada cláusula tiene exactamente 3 literales ¿existe alguna asignación de valores a las variables tal que evalúe F a cierto?

   Es una simple restricción del problema anterior de Cook que le da más sencillez y por tanto es NP-completo.

2. **Problema del matching en 3 dimensiones**: Dado un conjunto M que es subconjunto de AxBxC, donde estos 3 últimos son conjuntos disjuntos con n elementos cada uno ¿Contiene M un subconjunto tal que el número de elementos sea n y que contenga todos los elementos de $A \cup B \cup C$?

   Este problema puede interpretarse como el de 3 subconjuntos de hombres, mujeres y niños; cada uno de ellos está dispuesto a formar familia con 2 de los otros subconjuntos. ¿Podemos obtener familias formadas por un hombre, una mujer y un niño, que sea satisfactoria y en la que todos estén contentos, y no sobre ningún elemento? El problema del matching se demuestra a partir del problema de la 3-satisfactibilidad.

Clase NP-completo



3. **Problema de recubrimiento de vértices**: Dado un grafo con V vértices y E aristas y un entero positivo K <= V. ¿existe un recubrimiento de vértices de tamaño K o menor tal que cada arista de G tenga como mínimo uno de los vértices en V´. Este problema del recubirmiento también se demuestra mediante una reducción a partir del problema 3-satisfactibilidad.

4. **Problema del clique**: Al igual que antes dado un grafo V con v vértices y un entero positivo K <=V. Existe un subconjunto V´de V tal que V´>= K y en el que todo par de vértices de V´está unido por una arista? El problema del clique es NP-completo y se demuestra a partir del recubrimiento de vértices anterior.

5. **Problema del circuito hamiltoniano**: Dado un grafo con V vértices y E aristas ¿Contiene ese grafo un circuito hamiltoniano, es decir, una ordenación de todos los vértices de V tal que haya arista entre todos los vértices vecinos además de entre el último y el primero? Se demuestra gracias a una reducción a partir del problema del recubrimiento de vértices.

6. **Problema de la partición**: Dado un conjunto finito C y un tamaño t(c) entero positivo para todo c $\in$C ¿Existe un subconjunto C´ tal que la suma de los tamaños de los elementos que pertenecen a C´ sea igual a la suma de los tamaños de los elementos que no pertenecen? Se trata básicamente de ver si un conjunto de números puede dividirse en dos partes que sumen exactamente

los mismo. Si conjeturamos una división en dos del conjunto, comprobar que cada parte suma lo mismo tiene claramente un tiempo lineal, polinómico; pero sin la opción no determinista, debemos empezar a probar todos los posibles subconjuntos (2^n) hasta encontrar una solución, con lo cual tenemos un coste exponencial. Este problema se demuestra con una reducción a partir del problema del matching en 3 dimensiones.

Si cada demostración de NP-completitud fuese tan complicada como la del problema de la satisfactibilidad (Teorema de Cook), difícilmente existiría una lista tan larga de problemas NP-completos como la que hay actualmente. Así tenemos una nueva definición de NP-completo que nos servirá para "catalogar" los distintos problemas a los que nos enfrentemos, y constará de 4 pasos:

1. Demostrar que D (problema de decisión) pertenece a NP. Es facil viendo que si el problema se resuelve por conjetura no determinista, se verifica en tiempo polinómico.
2. Seleccionar un problema D´ que ya sepamos que es NP-completo. Este paso es un punto clave en la demostración; debemos identificar los parecidos de fondo entre los dos problemas D y D´, generalmente basándonos en los 6 problemas vistos en este mismo apartado.
3. Construir una reducción f de D´ a D. Es decir, concretamos la función de transformación que relaciona los dos problemas.
4. Demostrar que f es una reducción polinómica. Es en sí la demostración de NP-completitud, demostrando que es en sí una reducción y que el cálculo de f tiene un coste polinómico.

Existen 3 técnicas principales de demostración: por restricción, por reemplazo local y por diseño de componentes. En esta asignatura introductoria nos interesará solo la primera técnica: la **demostración por restricción**. Ésta es la técnica más simple y quizá la que puede aplicarse más a menudo. Consiste en ver que el problema D contiene un problema NP-completo D´ como caso especial. Vamos a ver un ejemplo con el problema de la mochila.

Recordemos que tenemos un conjunto finito U y para cada elemento y del conjunto un tamaño t y un beneficio b. Queremos saber si existe un suconjunto del conjunto U tal que la suma de los tamaños de sus elementos no supere el tamaño T total disponible y la suma de sus beneficios supere B. Ya sabemos que es un problema que pertenece a NP. Ahora debemos elegir un problema NP-completo conocido con el que se identifique, y, en este caso, lo hace con el problema de la partición. Restringimos en nuestro problema de la mochila que t=b para todo y del conjunto Y y en que T=B=1/2b para todo el conjunto U, se da una correspondencia unívoca entre las instancias del problema de la partición y el de la mochila.

Una vez dada y justificada la restricción que hace idénticos a estos dos problemas, prácticamente ya hemos acabado la demostración. La función de reducción f del paso 3 es la función identidad y la justificación de la restricción sirve como demostración de que f es realmente una reducción. No hace falta decir que la función identidad tiene un coste polinómico, con lo que ya tenemos el paso 4.