Modelos Avanzados de Computación: Tema 3: Clases de Complejidad

Serafín Moral

smc@decsai.ugr.es
Departamento de Ciencias de la Computación e IA
ETSI Informática

Abril, 2020

Contenido

- El problema de la complejidad estructural.
- Introducción histórica.
- Un problema sencillo: el flujo máximo.
- Un problema difícil: mínimo número de colores.
- Reducción de problemas.
- Clases de complejidad deterministas
  - Tiempo
  - Espacio
- Clases No Deterministas
- Relaciones entre Clases de Complejidad

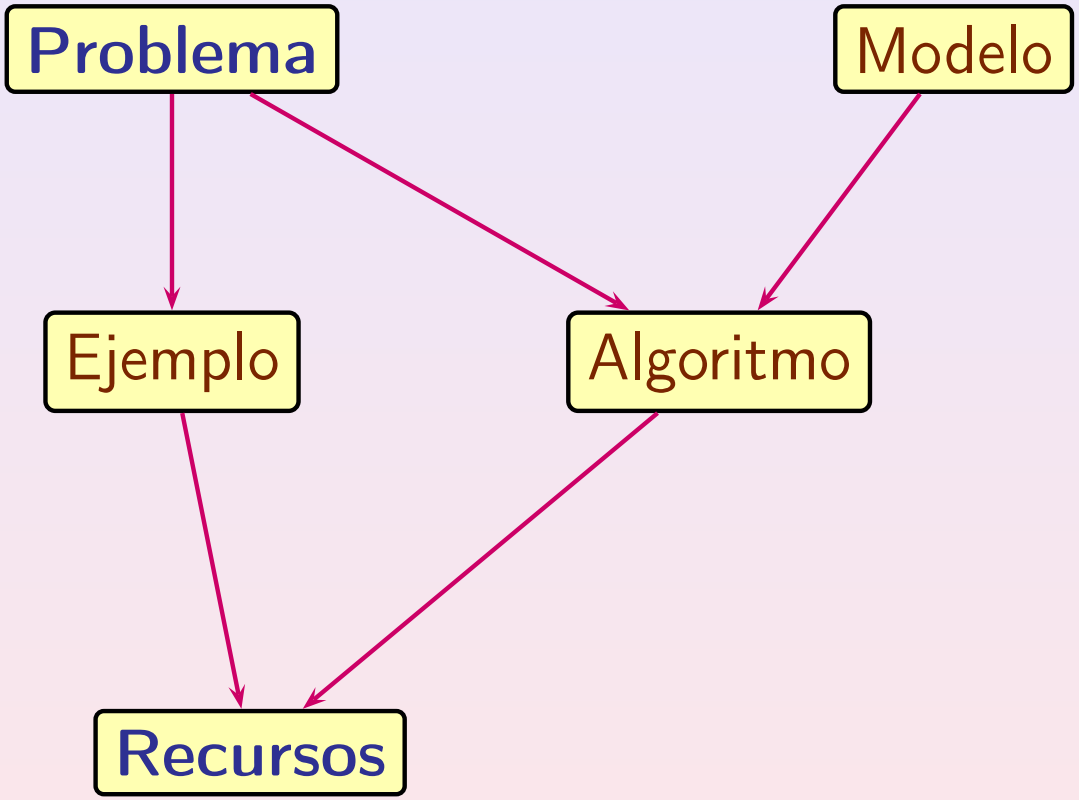
Objetivo Básico

Clasificar los problemas de acuerdo a su dificultad.

Nuesto objetivo último: comprender mejor (de manera profunda) la naturaleza de los problemas y los algoritmos que los resuelven.

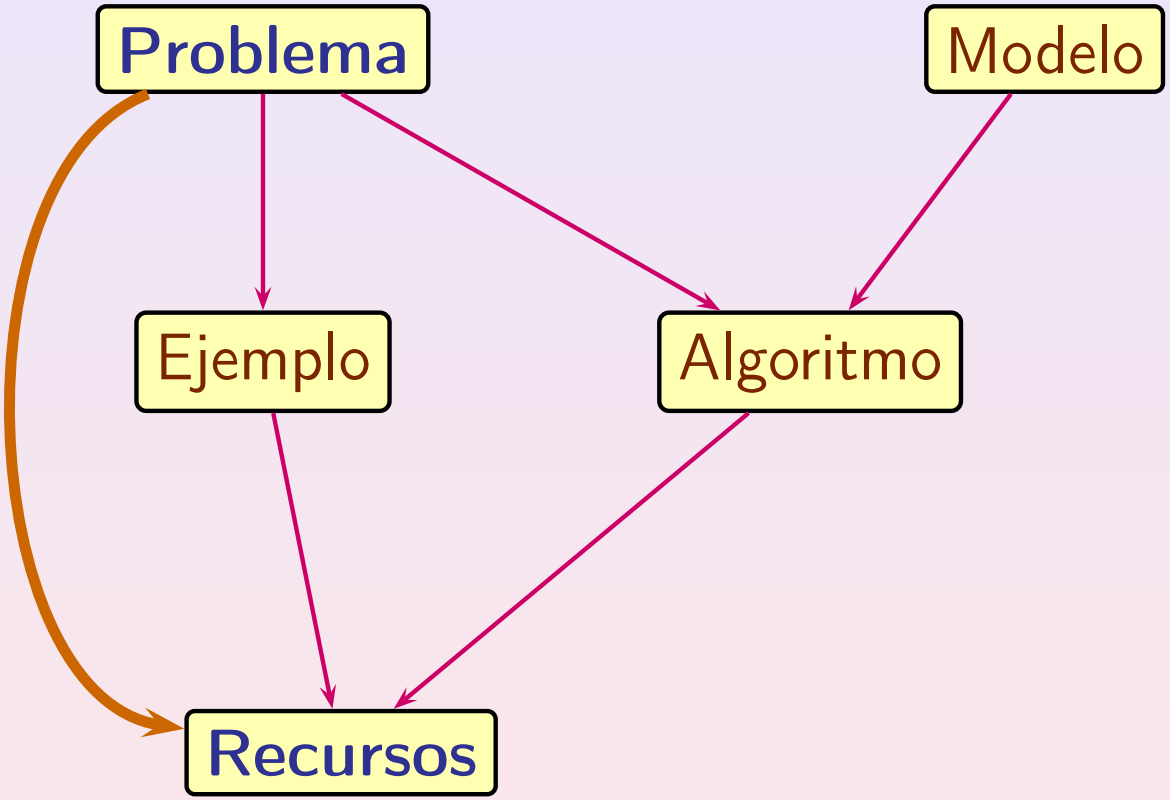
Recursos

Recursos que consume (espacio, tiempo, etc.)
Dificultad absoluta o relativa.



Recursos

Recursos que consume (espacio, tiempo, etc.)
Dificultad absoluta o relativa.



Metodología

- Abstracción de ejemplos: complejidad en función del tamaño de la entrada y en el peor de los casos.
- Abstracción del Modelo: Máquina de Turing. Inicialmente se trabaja con lenguajes (se abstrae la codificación).
- Recursos: Tiempo, espacio
- Algoritmos: El mejor algoritmo
- Clases de complejidad: clases muy amplias.

Tasas de Crecimiento Comparadas

Crecimiento Polinómico vs Crecimiento Exponencial

	$n = 10$	$n = 20$	$n = 30$	$n = 40$	$n = 50$	$n = 60$
n	0.00001 sg	0.00002 sg.	0.00003 sg.	0.00004 sg.	0.00005 sg.	0.00006 sg.
n^2	0.0001 sg.	0.0004 sg.	0.0009 sg.	0.0016 sg.	0.0025 sg.	0.0036 sg.
n^3	0.001 sg.	0.008 sg.	0.027 sg.	0.064 sg.	0.125 sg.	0.216 sg.
n^5	0.1 sg.	3.2 sg.	24.3 sg.	1.7 min.	5.2 min.	13.0 min.
2^n	0.001 sg.	1.0 sg.	17.9 min.	12.7 días	35.7 años	366 siglos
3^n	0.059 sg.	58 min.	6.5 años	3855 siglos	2×10^8 siglos	$1{,}3 \times 10^{13}$ siglos

La complejidad se mide en función de la longitud de la entrada

Importante Pregunta

Si tengo un número x, ¿cuantos caracteres necesito para escribir x?

La complejidad se mide en función de la longitud de la entrada

Importante Pregunta

Si tengo un número x, ¿cuantos caracteres necesito para escribir x?

Respuesta

Ya sea en binario, decimal, o con la codificación del tema anterior, si el número es x el número de caracteres que necesito es de orden $n = \log(x)$.

La complejidad se mide en función de la longitud de la entrada

Importante Pregunta

Si tengo un número x, ¿cuantos caracteres necesito para escribir x?

Respuesta

Ya sea en binario, decimal, o con la codificación del tema anterior, si el número es x el número de caracteres que necesito es de orden $n = \log(x)$.

Si quiero escribir un número en decimal, por ejemplo 1365, necesito 4 caracteres (dígitos).

La complejidad se mide en función de la longitud de la entrada

Importante Pregunta

Si tengo un número x, ¿cuantos caracteres necesito para escribir x?

Respuesta

Ya sea en binario, decimal, o con la codificación del tema anterior, si el número es x el número de caracteres que necesito es de orden $n = \log(x)$.

Si quiero escribir un número en decimal, por ejemplo 1365, necesito 4 caracteres (dígitos).
La complejidad de un problema en el que aparezca este número 1365 no hay que medirla en función de su valor (1365), sino en función del número de caracteres necesario para escribirlo (4)

La complejidad se mide en función de la longitud de la entrada

Importante Pregunta

Si tengo un número x, ¿cuantos caracteres necesito para escribir x?

Respuesta

Ya sea en binario, decimal, o con la codificación del tema anterior, si el número es x el número de caracteres que necesito es de orden $n = \log(x)$.

Si quiero escribir un número en decimal, por ejemplo 1365, necesito 4 caracteres (dígitos).
La complejidad de un problema en el que aparezca este número 1365 no hay que medirla en función de su valor (1365), sino en función del número de caracteres necesario para escribirlo (4)
Esta medida se puede aplicar a cualquier problema, sea numérico, de grafos, de conjuntos, o de cualquier tipo.

La complejidad se mide en función de la longitud de la entrada

Importante Pregunta

Si tengo un número x, ¿cuantos caracteres necesito para escribir x?

Respuesta

Ya sea en binario, decimal, o con la codificación del tema anterior, si el número es x el número de caracteres que necesito es de orden $n = \log(x)$.

Si quiero escribir un número en decimal, por ejemplo 1365, necesito 4 caracteres (dígitos).
La complejidad de un problema en el que aparezca este número 1365 no hay que medirla en función de su valor (1365), sino en función del número de caracteres necesario para escribirlo (4)
Esta medida se puede aplicar a cualquier problema, sea numérico, de grafos, de conjuntos, o de cualquier tipo.
Si n es la longitud del número en binario, entonces su valor x es de orden 2^n (si es en decimal, entonces cambia la base por 10).

Primalidad

Algoritmo de Primalidad

El algoritmo que para comprobar la primalidad de un número natural x va dividiendo ese número por todos los números y menores que x para ver si una división es exacta es exponencial.

El número de divisiones es $O(x)$, pero esa no es una medida correcta de la complejidad.

Primalidad

Algoritmo de Primalidad

El algoritmo que para comprobar la primalidad de un número natural x va dividiendo ese número por todos los números y menores que x para ver si una división es exacta es exponencial.

El número de divisiones es $O(x)$, pero esa no es una medida correcta de la complejidad.
Hay que tener en cuenta que la representación en binario de un número x ocupa del orden de $n = \log(x)$ caracteres.

Primalidad

Algoritmo de Primalidad

El algoritmo que para comprobar la primalidad de un número natural x va dividiendo ese número por todos los números y menores que x para ver si una división es exacta es exponencial.

El número de divisiones es $O(x)$, pero esa no es una medida correcta de la complejidad.
Hay que tener en cuenta que la representación en binario de un número x ocupa del orden de $n = \log(x)$ caracteres.
Un número y menor que x ocupa también del orden de n casillas.

Primalidad

Algoritmo de Primalidad

El algoritmo que para comprobar la primalidad de un número natural x va dividiendo ese número por todos los números y menores que x para ver si una división es exacta es exponencial.

El número de divisiones es $O(x)$, pero esa no es una medida correcta de la complejidad.
Hay que tener en cuenta que la representación en binario de un número x ocupa del orden de $n = \log(x)$ caracteres.
Un número y menor que x ocupa también del orden de n casillas.
Una división de un número de longitud n por otro de longitud n se hace con un orden $O(n^2)$.

Primalidad

Algoritmo de Primalidad

El algoritmo que para comprobar la primalidad de un número natural x va dividiendo ese número por todos los números y menores que x para ver si una división es exacta es exponencial.

El número de divisiones es $O(x)$, pero esa no es una medida correcta de la complejidad.
Hay que tener en cuenta que la representación en binario de un número x ocupa del orden de $n = \log(x)$ caracteres.
Un número y menor que x ocupa también del orden de n casillas.
Una división de un número de longitud n por otro de longitud n se hace con un orden $O(n^2)$.
Como el número $n = \log(x)$, entonces x es de orden 2^n, y el número de divisiones es de orden $O(2^n)$.

Primalidad

Algoritmo de Primalidad

El algoritmo que para comprobar la primalidad de un número natural x va dividiendo ese número por todos los números y menores que x para ver si una división es exacta es exponencial.

El número de divisiones es $O(x)$, pero esa no es una medida correcta de la complejidad.
Hay que tener en cuenta que la representación en binario de un número x ocupa del orden de $n = \log(x)$ caracteres.
Un número y menor que x ocupa también del orden de n casillas.
Una división de un número de longitud n por otro de longitud n se hace con un orden $O(n^2)$.
Como el número $n = \log(x)$, entonces x es de orden 2^n, y el número de divisiones es de orden $O(2^n)$.
Multiplicando una división por el número de divisiones, nos sale $O(n^2 2^n)$, que es una complejidad exponencial.

Inicios de la Complejidad

- *G. Lamé (1884).-*Número de divisiones para el máximo común divisor de dos números.
- *Años 50 y 60.-* Algoritmos Polinómicos = Algoritmos Buenos
  J. Von Neumann, M.D. Rubin, J. Edmons
- *Hartamis, Stearns (1965).-*Análisis sistemático de medidas de complejidad específicas. Inclusión de las clases de complejidad.
- *M. Blum (1967).-*Axiomas para una medida de complejidad.

Referencias Históricas

- *Cook (1971) The Complexity of Proving Procedures - Lavine .-*
  - Reducción Polinómica de Problemas
  - Problemas NP
  - NP-completitud
  - Demostración de que el problema de la consistencia en Lógica Proposicional es NP-completo
- *Karp (1972).-*Dió una amplia lista de problemas NP-completos.
- *Meyer(1970) Stockmeyer (1976).-* Definieron la jerarquían polinómica, muy útil para clasificar problemas difíciles.

Referencias Históricas

- *Baker, Hill and Solovay (1975).-* Resultados sobre complejidad relativos a oráculos.
- *Berman y Hartmanis (1977)* propusieron la conjetura del isomorfismo de los problemas NP-completos.
- *Solovay y Strassen (1977).-* Consideraron algoritmos probabilistas.
- *Valiant (1979).-* Definió la clase #P de las funciones que cuentan el número de soluciones.
- *Yao (1979).-* Propuso estudiar la complejidad de la comunicación (problemas distribuidos).
- Finales de los 70 y años 80.- Comenzó interés por la complejidad en función de circuitos booleanos y modelos de computación paralela en general.

Referencias Históricas

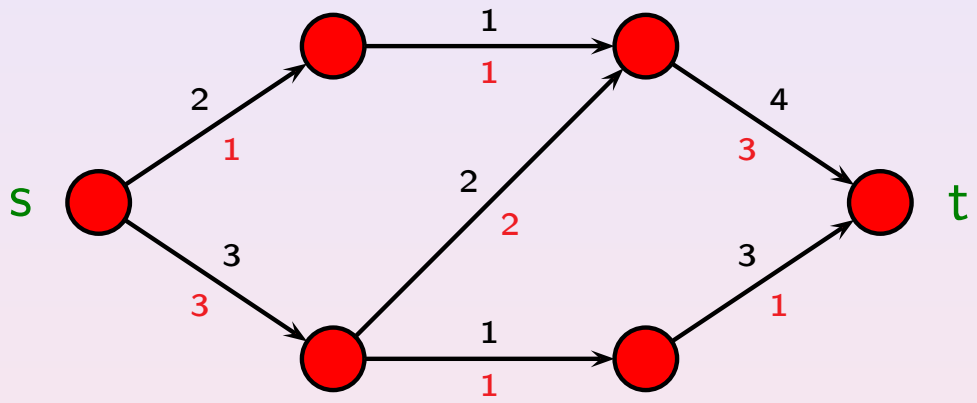
- *Babai (1985).-* Sistemas interactivos de demostración.
- *Papadimitriou y Yannakakis (1988).-* Definieron clases de complejidad para la resolución aproximada de problemas.
- *Bernstein y Vazirani (1997).-* Complejidad de la computación cuántica.
- Los problemas matemáticos del milenio: P frente a NP
  http://www.claymath.org/prize_problems/index.html
- *Agrawal, Kayal y Saxena (2002).-* Han demostrado que la primalidad está en P.
- *Reingold (2005).-* Ha demostrado que la conectividad en grafos no dirigidos se puede resolver en espacio logarítmico.
- *Hartmanis (31 de diciembre de 1962).-* "Este ha sido un buen año", En realidad están siendo unos buenos casi 60 años de complejidad computacional.
  El zoo de la complejidad:
  `https://complexityzoo.net/Complexity_Zoo`
  ¡545 clases en abril de 2021! y subiendo.

El problema del Flujo Máximo

- Tenemos un grafo dirigido en el que los arcos están etiquetados por su capacidad. Hay un origen (s) y un destino (t).
- Un flujo es una asignación de valor a cada arco que no supere su capacidad y de forma que la suma de lo que entra a cada nodo intermedio es igual a la suma de lo que sale.
- El valor de un flujo es la suma de lo que sale del origen (que es igual a la suma de lo que llega al destino).
- Se trata de calcular el flujo de valor máximo.

Ejemplo

En rojo está representado un flujo de valor 4

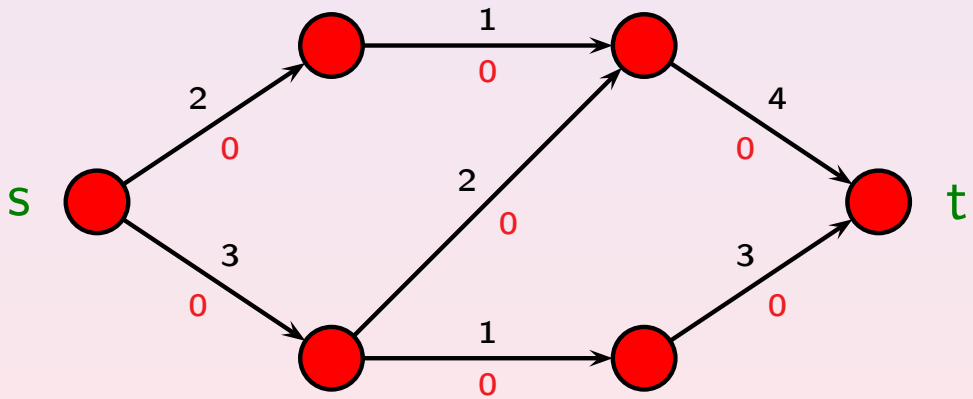


El Algoritmo de Ford-Fulkenson

Partimos del problema (V, E, s, t, c) donde V es el conjunto de vértices, E el conjunto de aristas, s el nodo inicial, t el nodo final y c es una función que asigna a cada pareja de nodos (x, y) su capacidad $c(x, y)$.

Un flujo se representará por una función f

1. Se comienza con un flujo cualquiera, por ejemplo el flujo cero: $f(x, y) = 0$

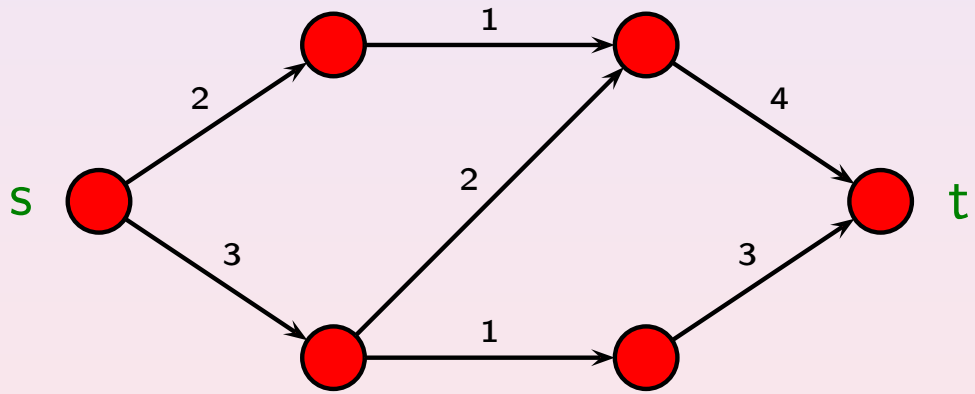


El Algoritmo de Ford-Fulkenson

2. Se calcula el grafo diferencia (v, E', s, t, c') donde

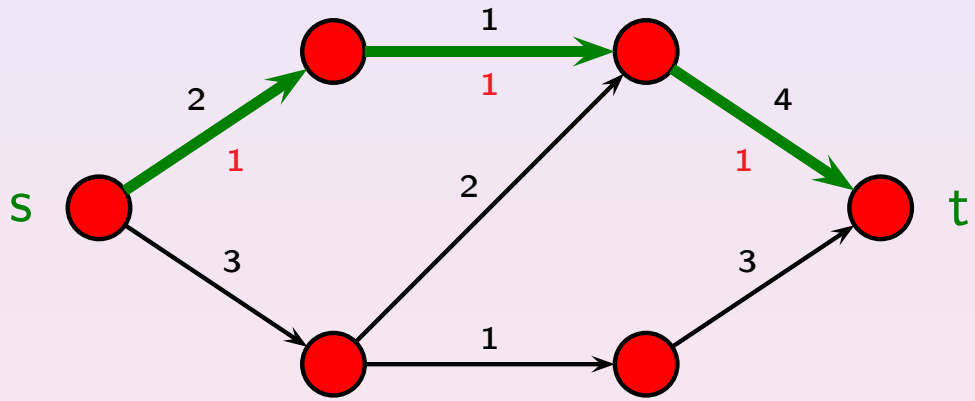
$$E' = E - \{(x,y) : f(x,y) = c(x,y)\} \cup \{(x,y) : f(y,x) > 0\}$$
$$c'(x,y) = c(x,y) - f(x,y), \quad \text{si } (x,y) \in E$$
$$c'(x,y) = f(y,x), \quad \text{si } (x,y) \notin E$$

En nuestro caso, el grafo es el mismo del principio.



El Algoritmo de Ford-Fulkenson

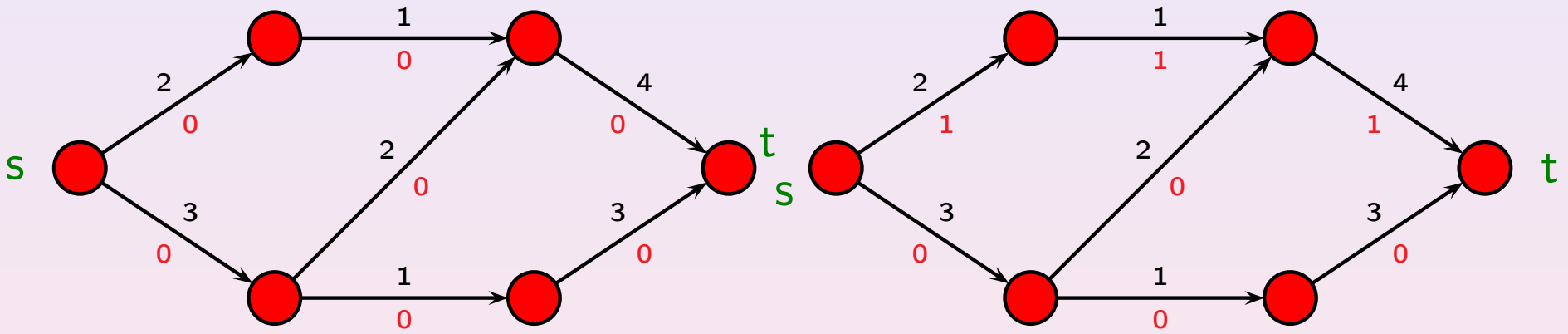
3. Se calcula un camino de s a t en el nuevo grafo. Se le asigna un flujo al nuevo camino que es valor mínimo de c' en todas las aristas que lo componen.



Si no existe el camino se termina y f contiene el flujo máximo.

El Algoritmo de Ford-Fulkenson

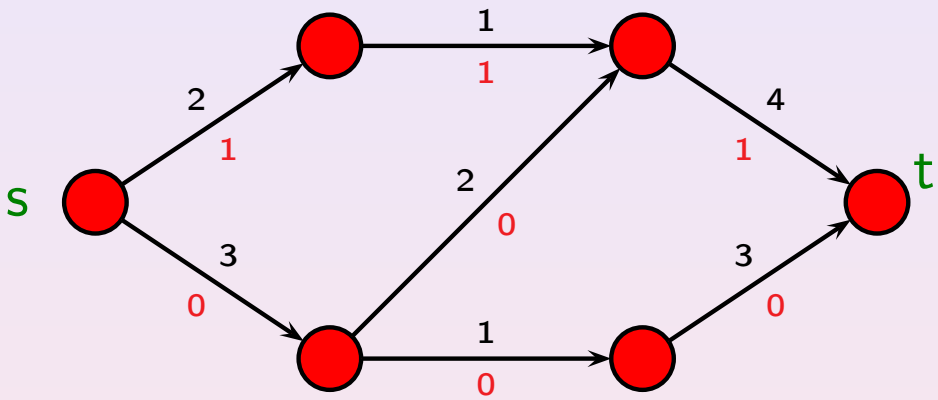
4. Se añade el flujo del camino al flujo original. Si una arista está en el camino en sentido inverso, entonces el valor de esta arista se resta:



5. Se vuelve al paso 2.

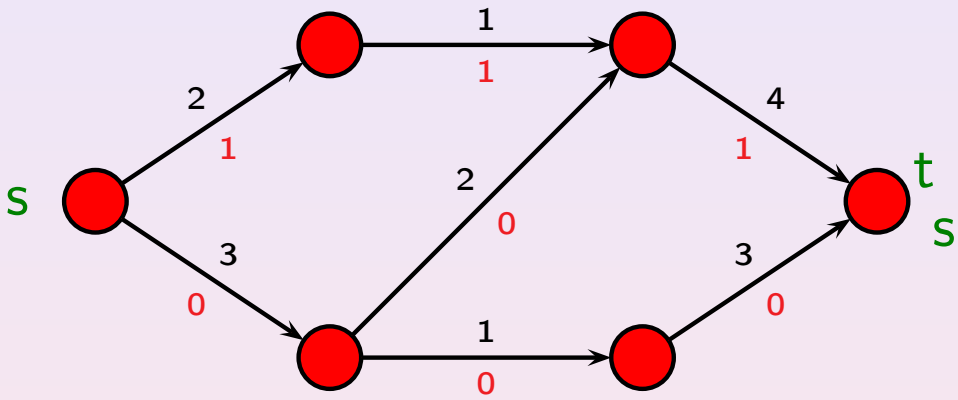
Ejemplo (cont.)

Flujo Inicial

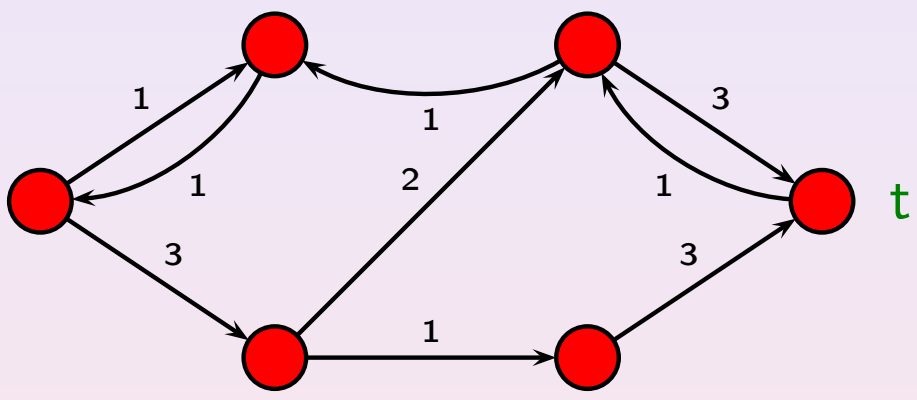


Ejemplo (cont.)

Flujo Inicial

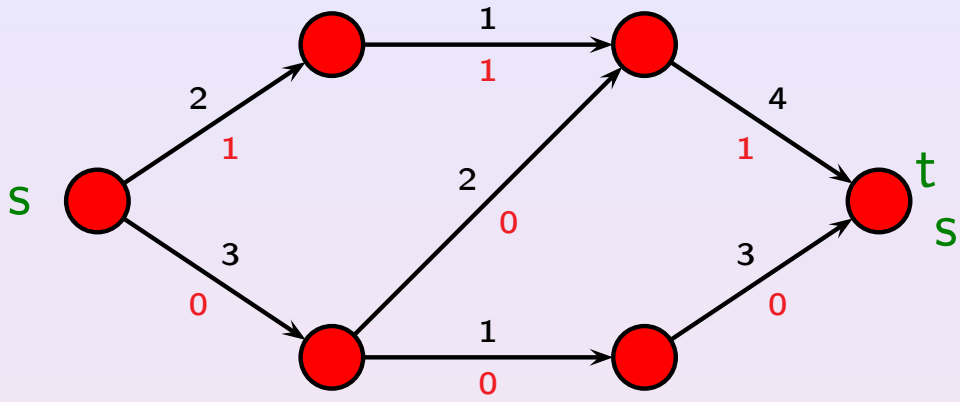


Grafo Diferencia

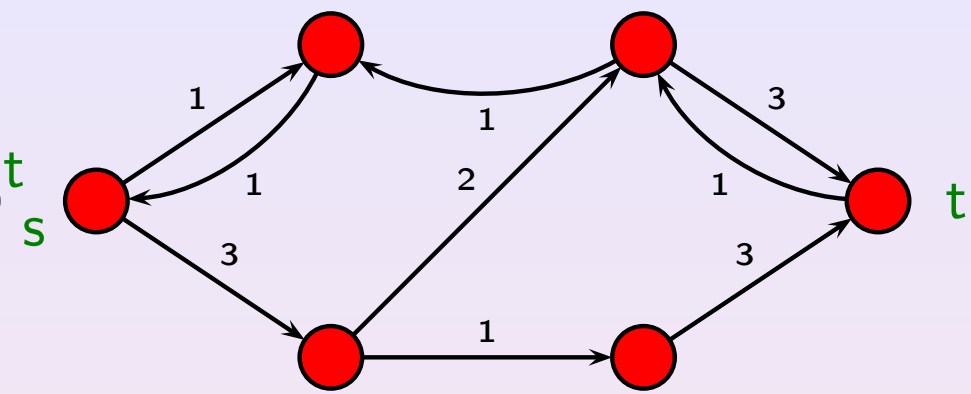


Ejemplo (cont.)

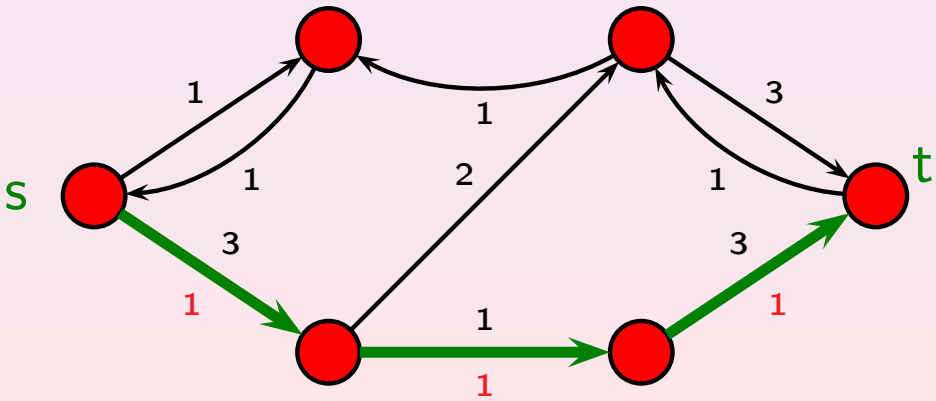
Flujo Inicial



Grafo Diferencia

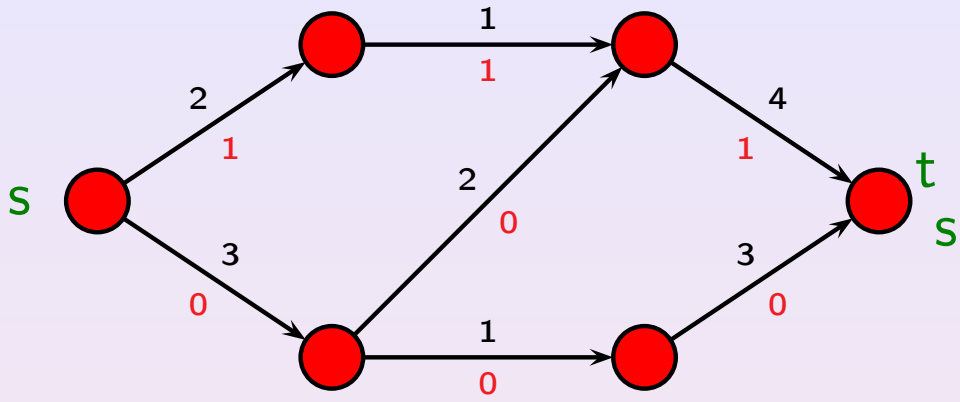


Nuevo Camino

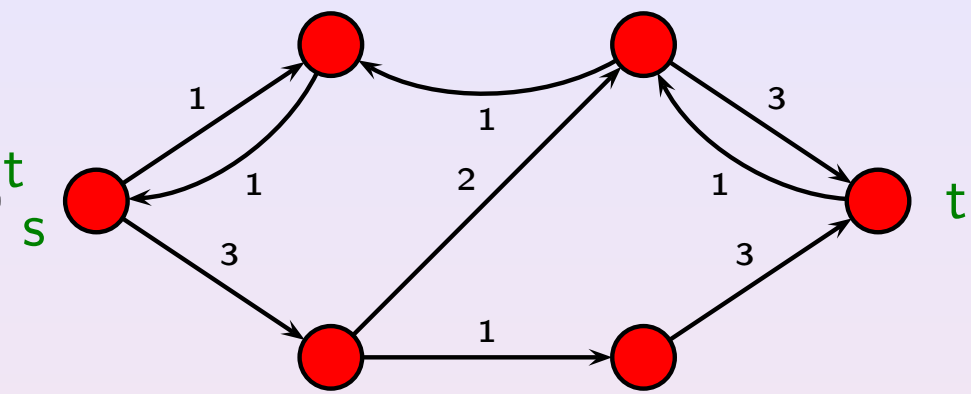


Ejemplo (cont.)

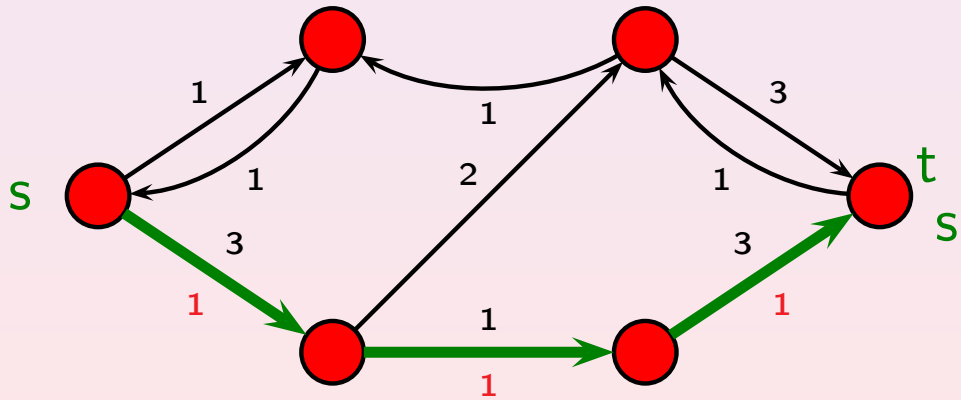
Flujo Inicial



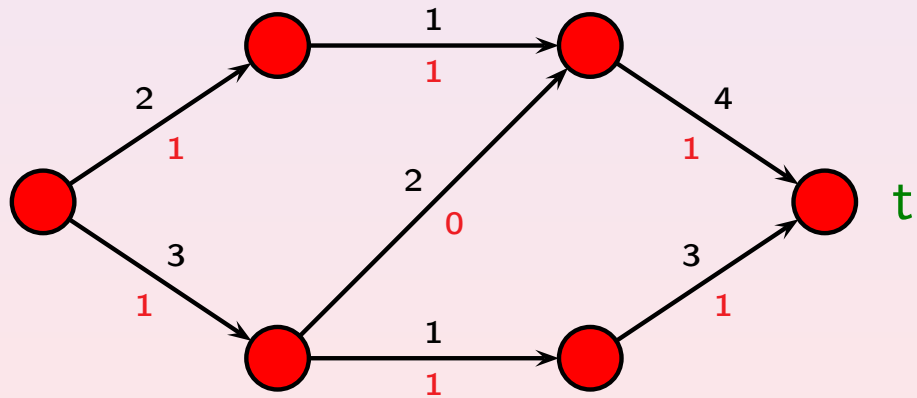
Grafo Diferencia



Nuevo Camino

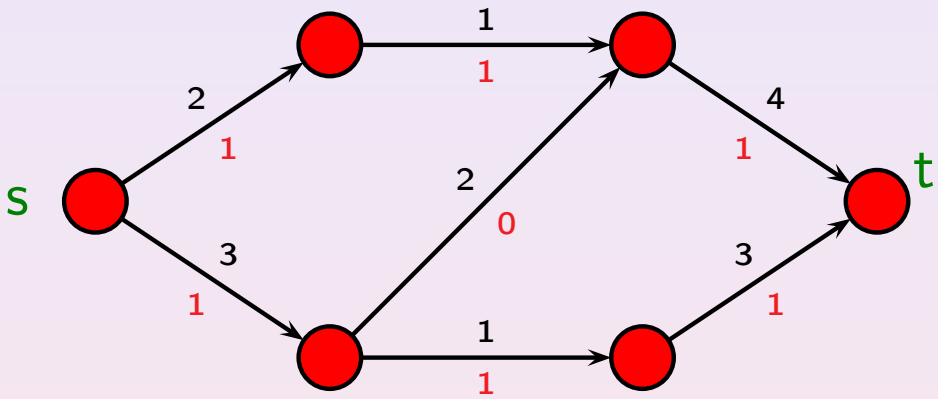


Flujo Suma



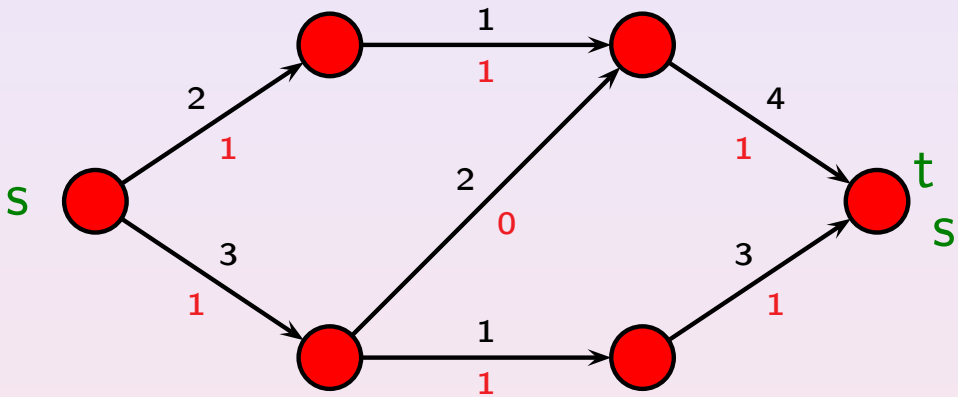
Ejemplo (cont.)

Flujo Inicial

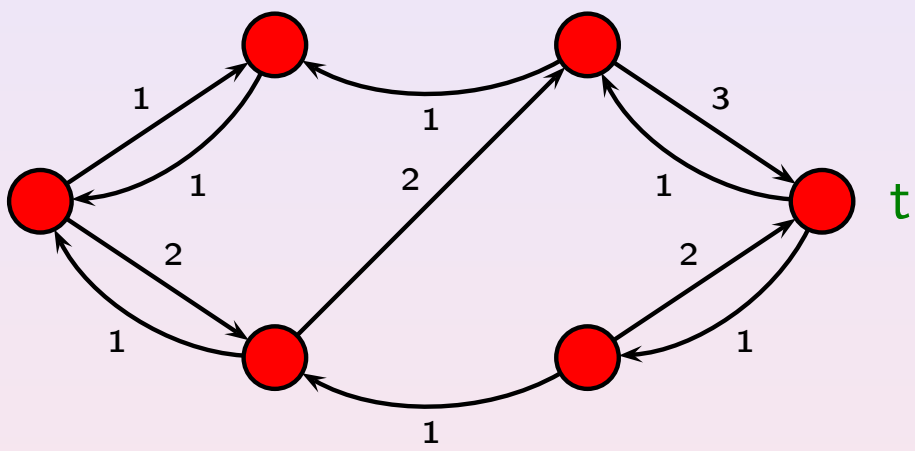


Ejemplo (cont.)

Flujo Inicial

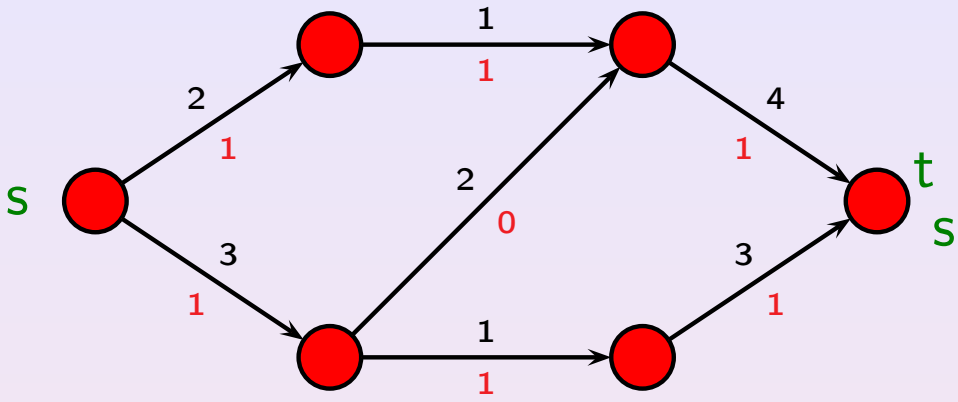


Grafo Diferencia

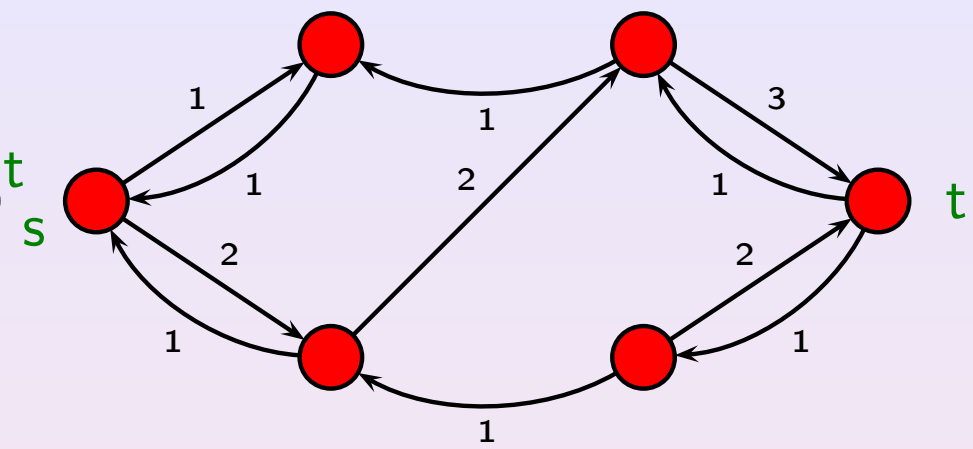


Ejemplo (cont.)

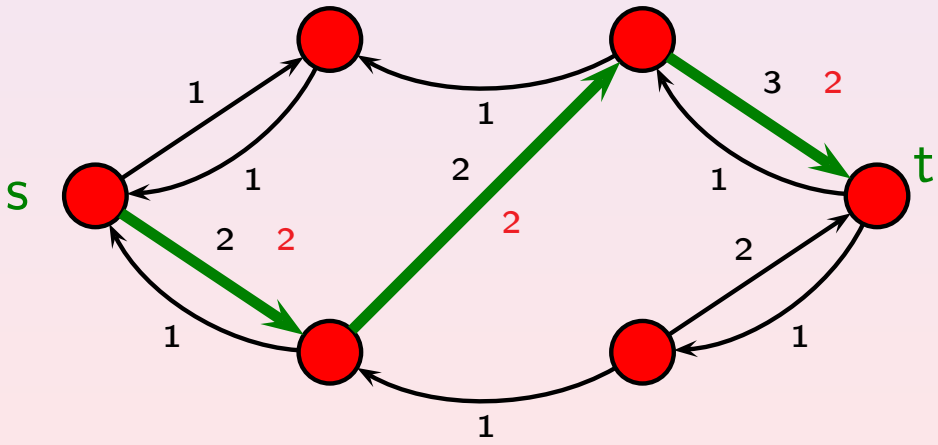
Flujo Inicial



Grafo Diferencia

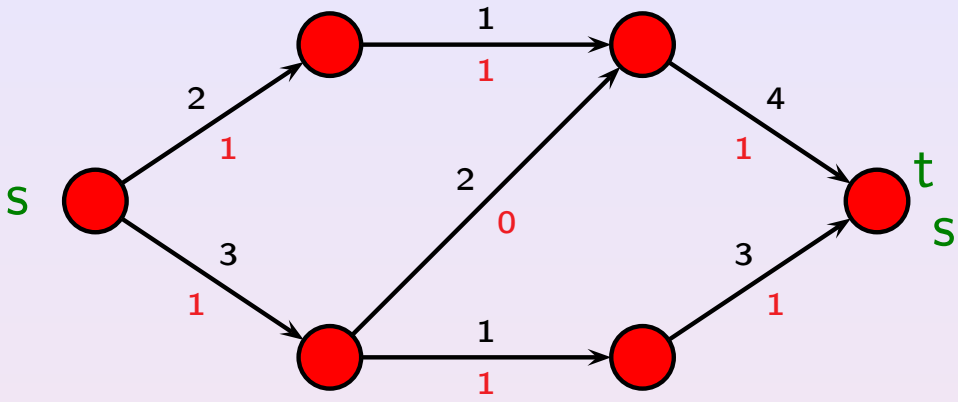


Nuevo Camino

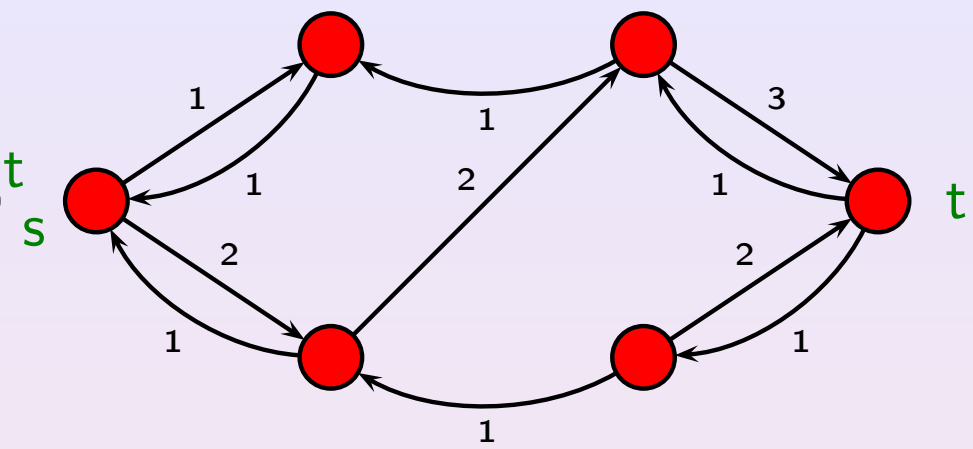


Ejemplo (cont.)

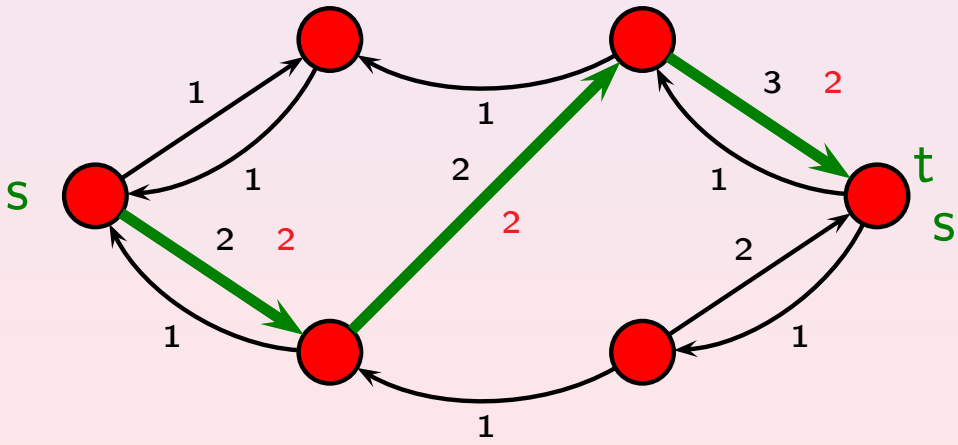
Flujo Inicial



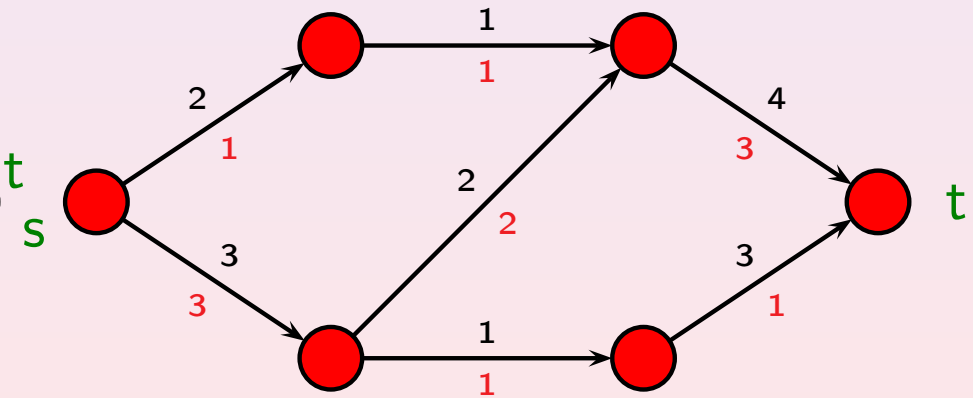
Grafo Diferencia



Nuevo Camino

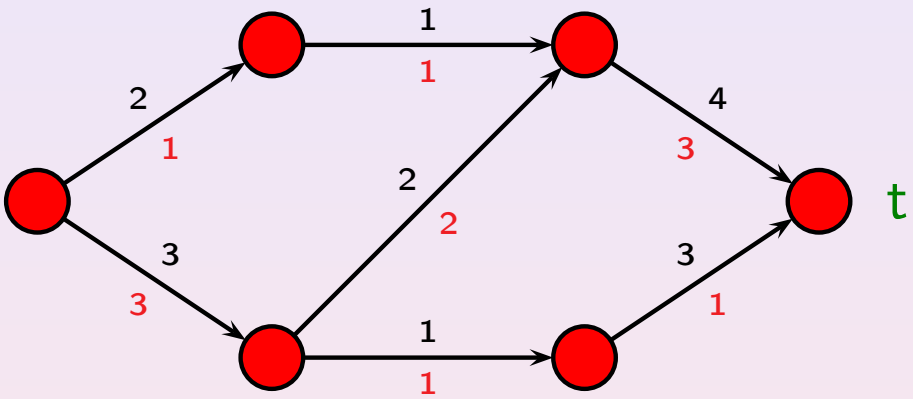


Flujo Suma



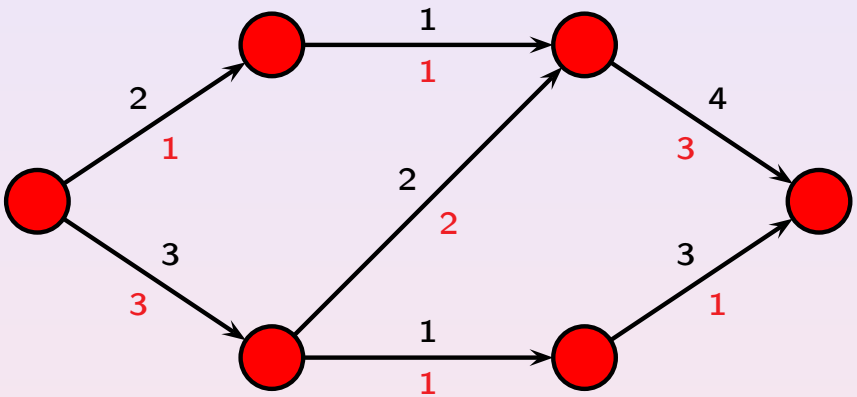
Ejemplo (cont.)

Flujo Inicial

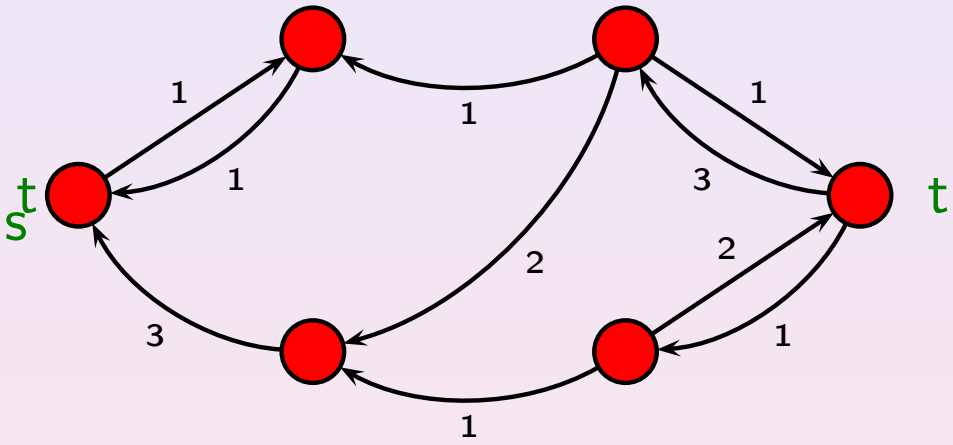


Ejemplo (cont.)

Flujo Inicial

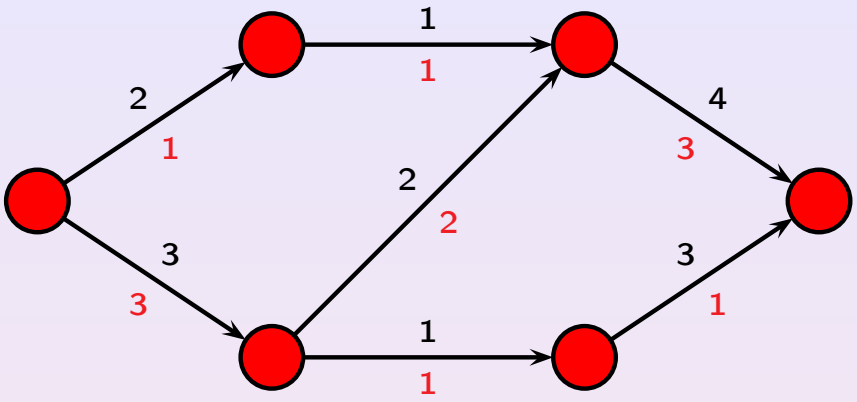


Grafo Diferencia

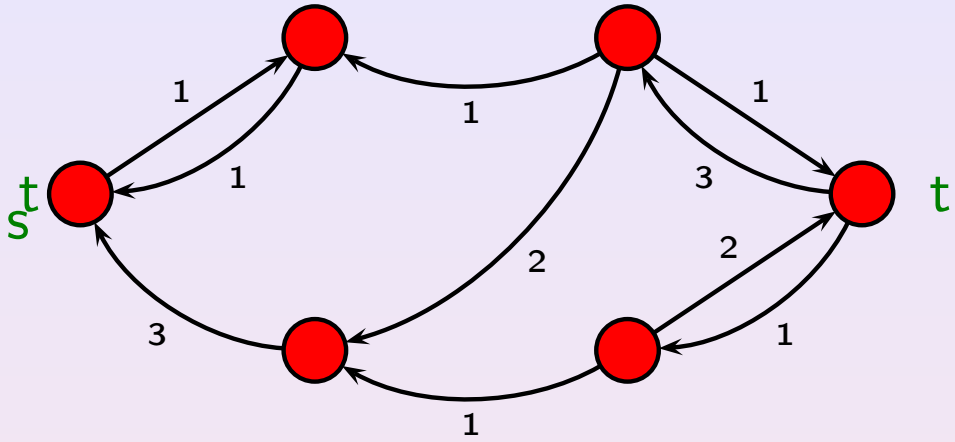


Ejemplo (cont.)

Flujo Inicial



Grafo Diferencia

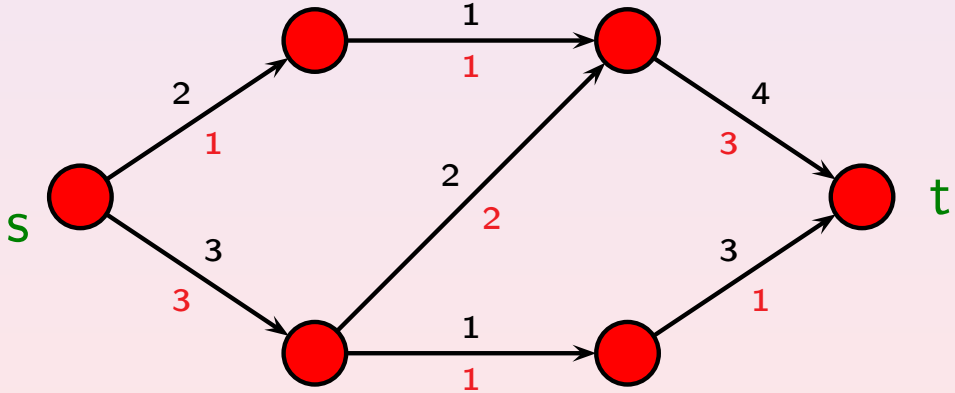


No Existe Nuevo Camino.

Solución Óptima: Flujo 4

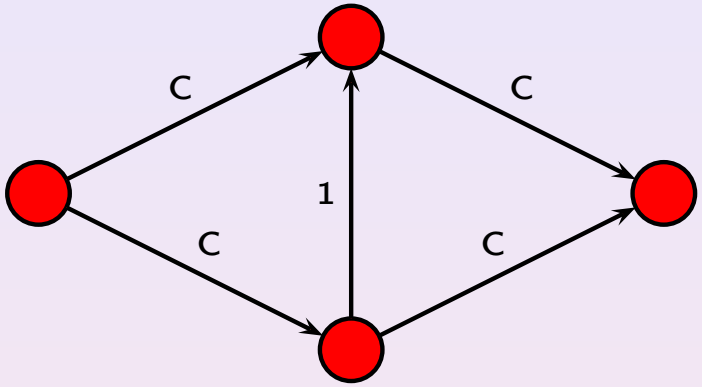
Estamos

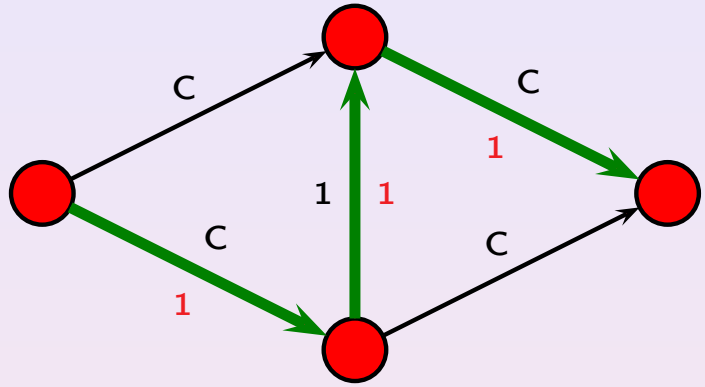
en la solución óptima:



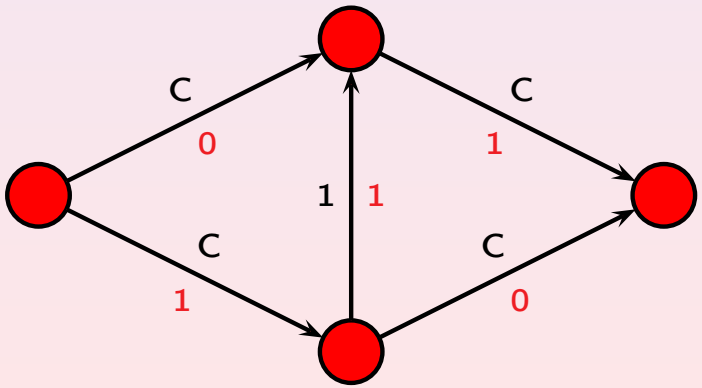
Flujo Máximo: Un ejemplo con problemas

Supongamos el siguiente ejemplo, con un óptimo de $2C$.



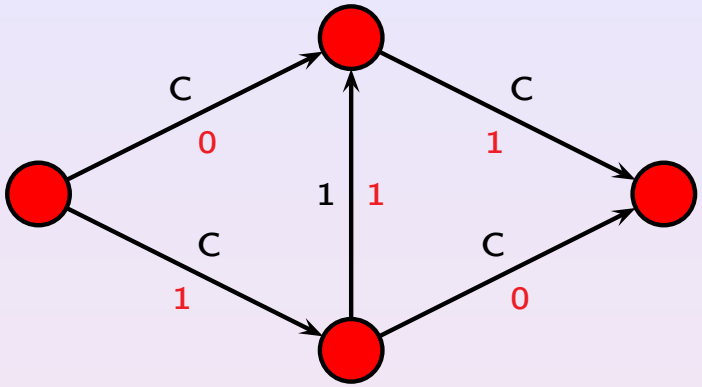


Flujo Obtenido

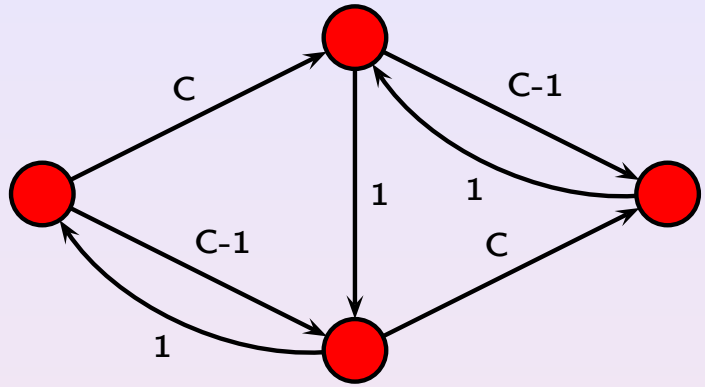


Flujo Máximo: Un ejemplo con problemas

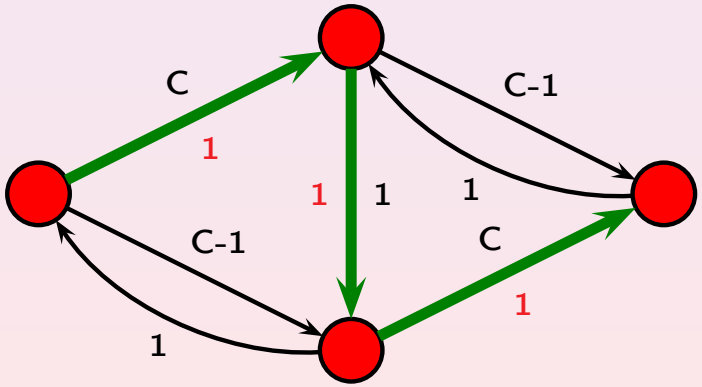
Flujo Actual



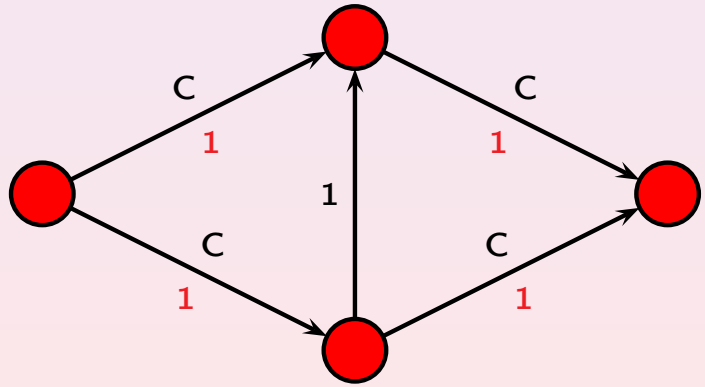
Grafo Diferencia



Nuevo Camino

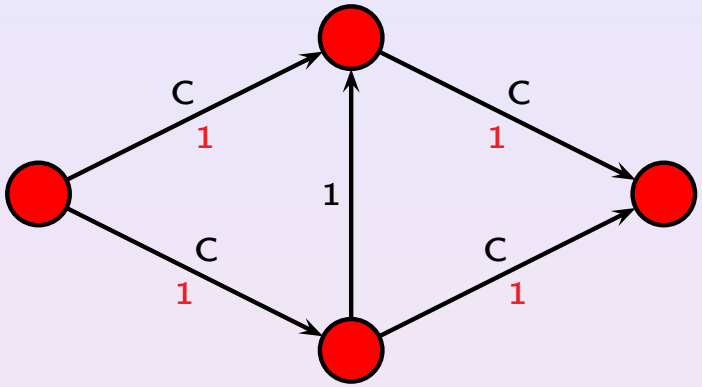


Nuevo Flujo

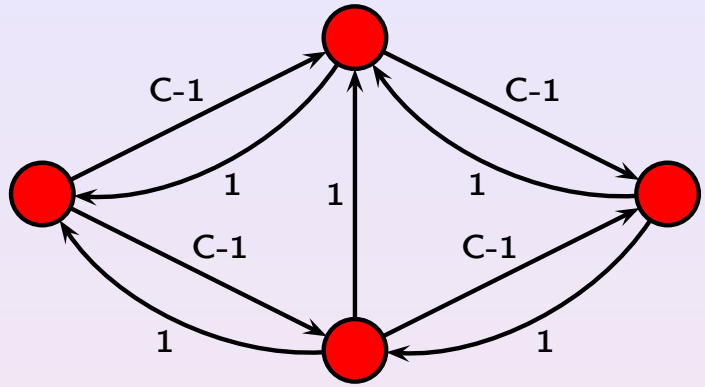


Flujo Máximo: Un ejemplo con problemas

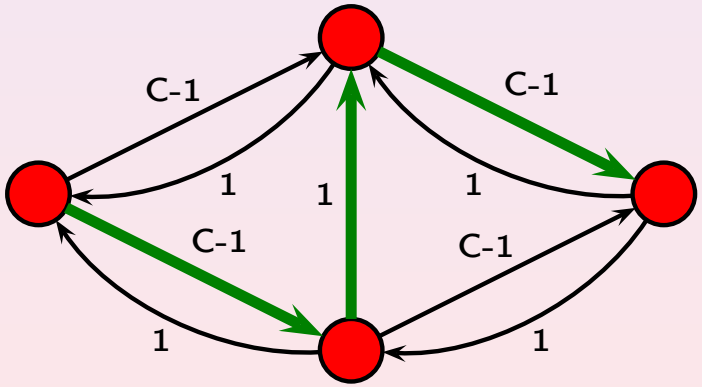
Flujo Actual



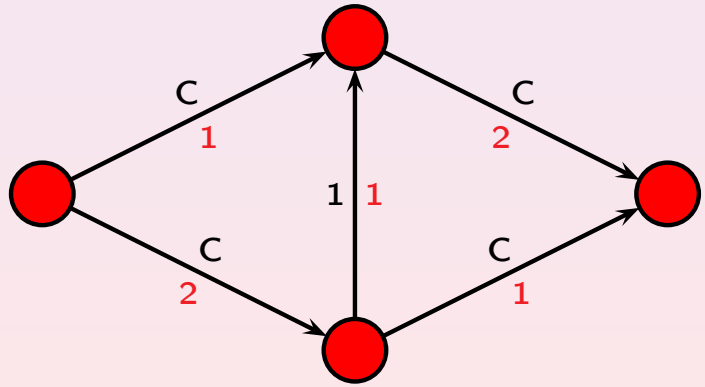
Grafo Diferencia



Nuevo Camino

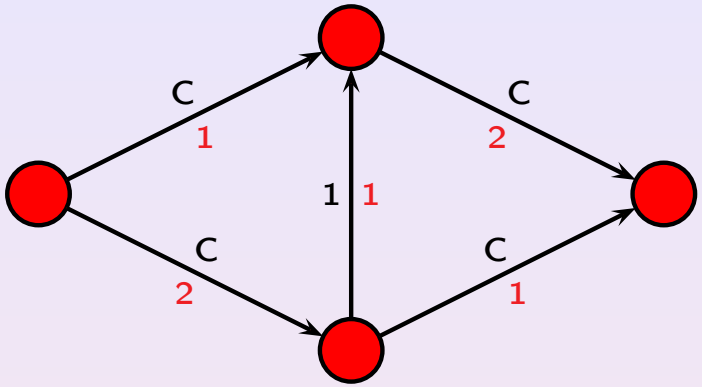


Nuevo Flujo

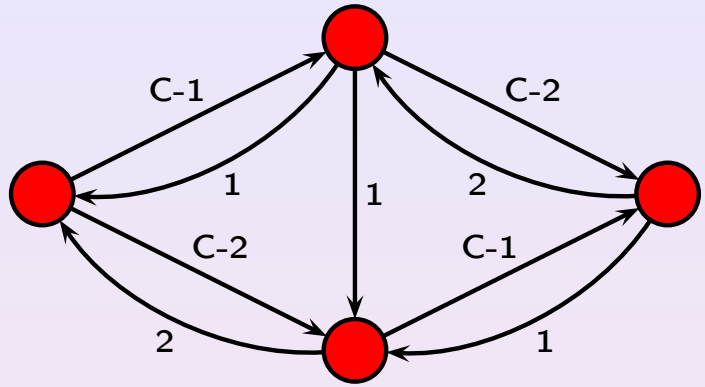


Flujo Máximo: Un ejemplo con problemas

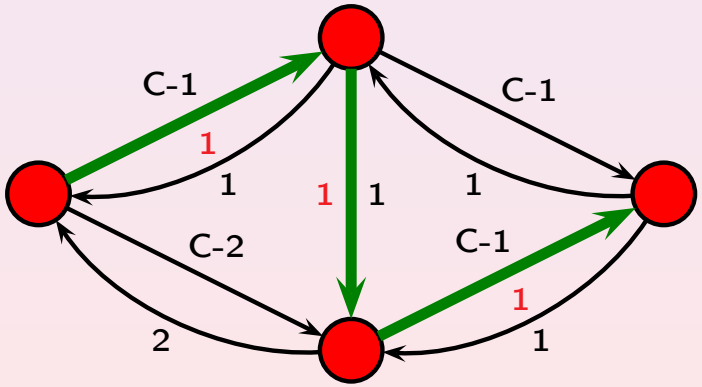
Flujo Actual



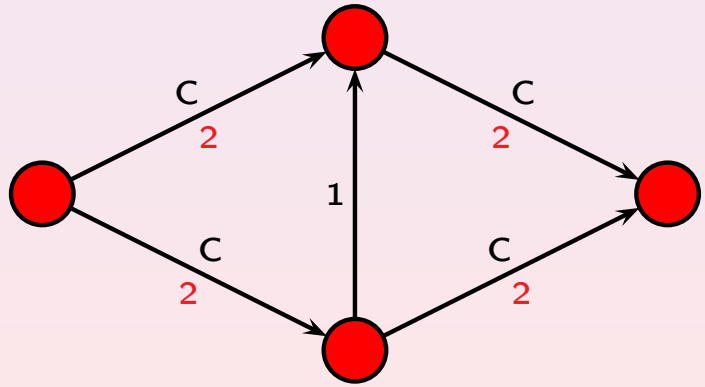
Grafo Diferencia



Nuevo Camino



Nuevo Flujo



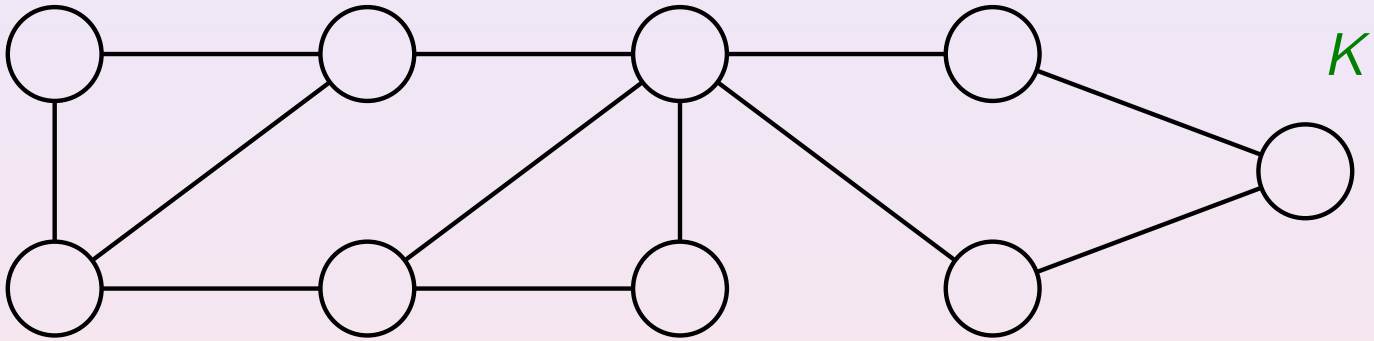
Flujo Máximo: Un ejemplo con problemas

Así el flujo se va siempre mejorando, pero de una manera muy lenta: una unidad cada vez. Si el número C es muy grande, esto da a muchas iteraciones: la complejidad es exponencial en función del número de dígitos de C.

Afortunadamente existe una forma de elegir los caminos que garantiza que este crecimiento tan lento no ocurre y que la complejidad es realmente polinómica: basta con elegir en cada caso el camino más corto entre el origen y el destino. En ese caso, se puede demostrar que el número máximo de iteraciones en $O(m^3)$ donde m es el número de nodos. La complejidad total es de orden $O(n^5)$ donde n es la longitud de la entrada.

Problema de colorear un grafo (COLOR)

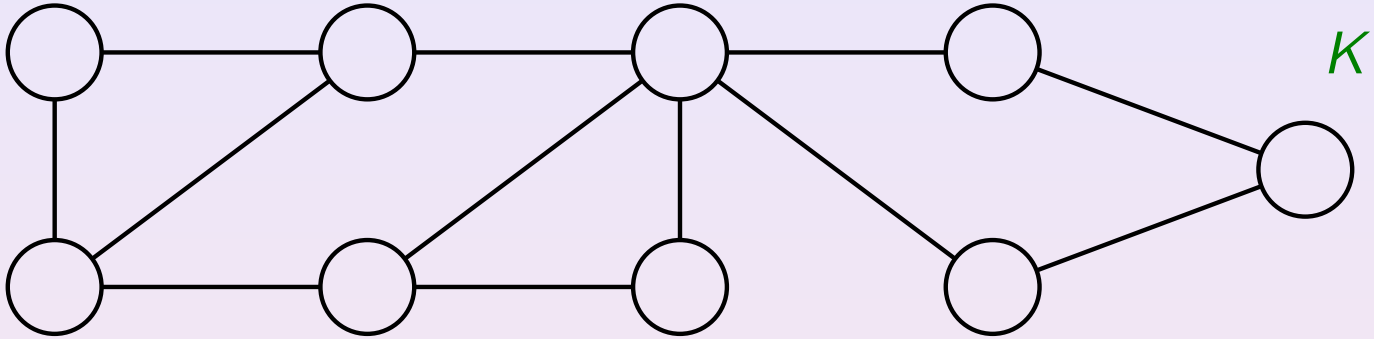
Dado un grafo (G, V) y un número entero K



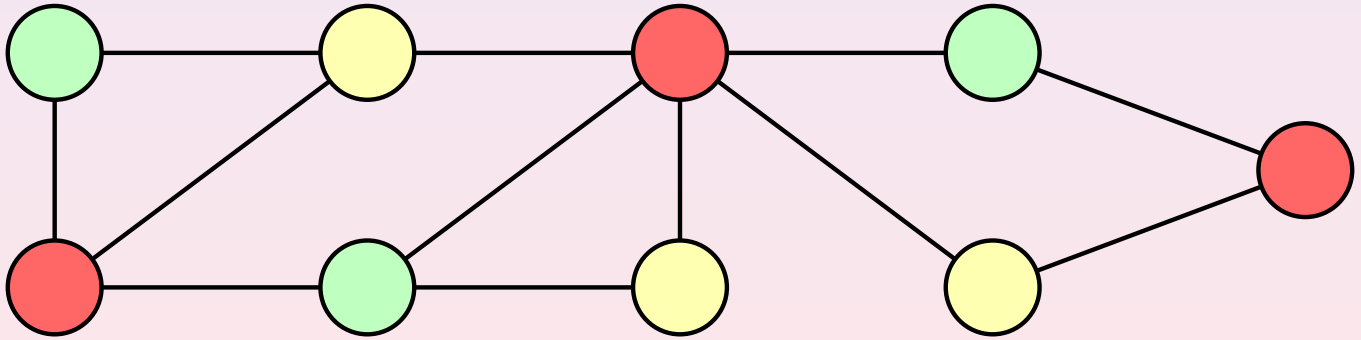
$K = 3$

Problema de colorear un grafo (COLOR)

Dado un grafo (G, V) y un número entero K



$K = 3$



SI

La gran diferencia

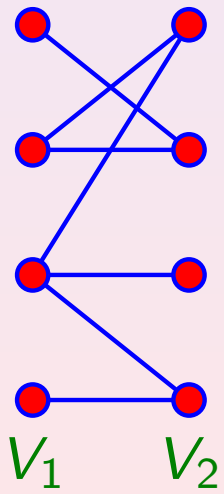
- En el problema del flujo máximo se conoce un procedimiento que necesita hacer un cálculo polinómico y que va acercándose al óptimo de forma creciente.
- En el problema de colorear grafos nadie conoce nada que se puede calcular en el grafo en tiempo polinómico y que evite la fuerza bruta: la búsqueda en el espacio de todas las opciones.

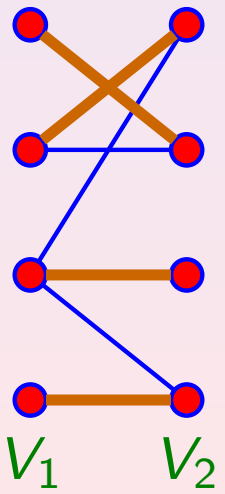
Reducción de Problemas

Hay otros problemas cuya resolución se puede reducir a la resolución del flujo máximo (FM). Este es el caso del problema de las pareja (PAR).

El problema de las parejas $PAR(V_1, V_2, A)$

Tenemos dos conjuntos del mismo tamaño V_1 y V_2 y un subconjunto $A \subseteq V_1 \times V_2$ (representa las compatibilidades de elementos de V_1 con elementos de V_2. El problema consiste en decidir si existe un subconjunto $A' \subseteq A$ tal que cada elemento de $v_1 \in V_1$ aparece en uno y sólo en uno de los pares $(v_1, y) \in A'$ y cada elemento de $v_2 \in V_2$ aparece en uno y sólo en uno de los pares $(x, v_2) \in A'$.





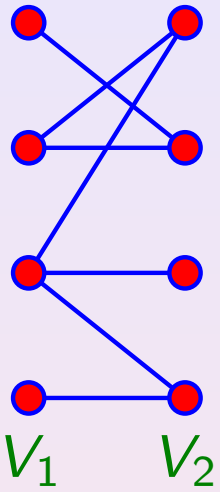
Reducción

Para cada ejemplo de las parejas $PAR(V_1, V_2, A)$ podemos construir un problema del flujo máximo equivalente $FM(G, c, s, t)$:

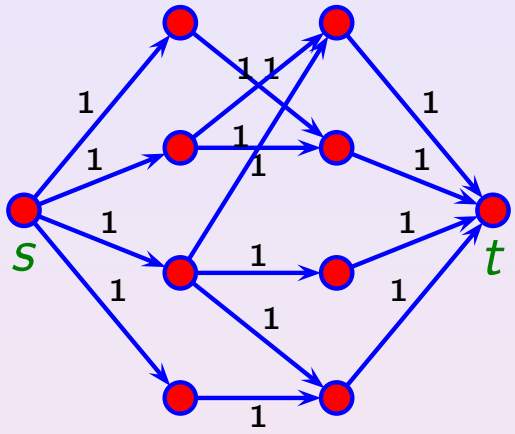
1. Añadir un nodo s y arcos dirigidos desde este nodo a todos los de V_1.
2. Dirigir los arcos originales desde V_1 a V_2.
3. Añadir un nodo final t y arcos dirigidos desde los nodos de V_2 a este nodo.
4. Asignar una capacidad de 1 a todos los arcos

Todo esto se puede hacer mediante un algoritmo que además es rápido.

Reducción



PAR



¿FM$\geq$ 4?

Si m es el número de elementos en V_1 y V_2, la pregunta equivalente al problema de las parejas es:

¿Existe un flujo máximo de tamaño mayor o igual a m?

Reducción

- Supongamos que $FM(G, c, s, t, m)$ es una función que resuelve el problema del flujo máximo en su versión decisión, es decir responde a la pregunta: ¿Existe un flujo de valor mayor o igual a m?
- Supongamos que $REDUCE(V_1, V_2, A)$ es el algoritmo que implementa la reducción, es decir calcula $(G, c, s, t, m) = REDUCE(V_1, V_2, A)$.
- Entonces podemos hacer un algoritmo para resolver el problema de las parejas:
  - Calcula $(G, c, s, t, m) = REDUCE(V_1, V_2)$
  - Return $FM(G, c, s, t, m)$
- Con eso podemos usar un algoritmo del FM para resolver PAR, pero de forma más importante, como REDUCE es rápido, nos compara la dificultad de FM y PAR: FM es más difícil o igual que PAR, ya que cualquier algoritmo de FM se puede usar para PAR.

CLASES DE COMPLEJIDAD

Complejidad de una Máquina de Turing

Una Máquina de Turing (u otro dispositivo de cálculo) es de complejidad $f(n)$ si y solo si para toda entrada $x \in A^*$ de longitud $|x| = n$, la máquina acepta esta entrada o la rechaza consumiento menos de $f(n)$ unidades.

Complejidad de un Lenguaje o Problema de Decisión

Un lenguaje se dice de complejidad $f(n)$ si existe una Máquina de Turing que acepta el lenguaje y tiene complejidad $f(n)$.

- Unidades pasos de cálculo $\rightarrow$ complejidad en tiempo
- Unidades casillas de la cinta $\rightarrow$ complejidad en espacio

Existen otras medidas de complejidad

Órdenes de Complejidad

Se dice que una medida $g(n)$ es de orden $O(f(n))$ si existe n_0 y $c > 0$ tal que $\forall n \geq n_0$, $g(n) \leq c.f(n)$.

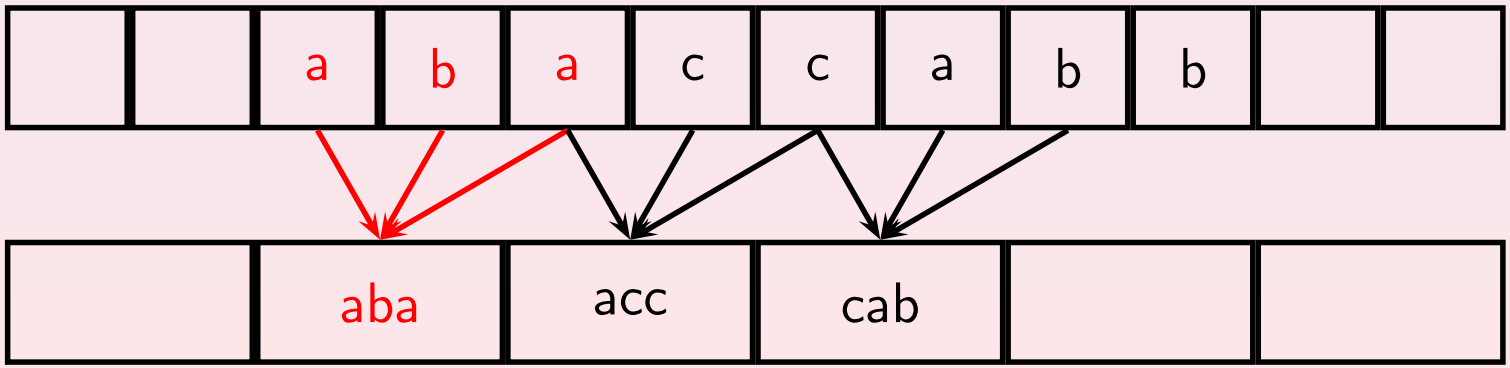
Teorema

Si L es aceptado en tiempo $t(n)$ por una Máquina de Turing con k cintas, entonces $\forall m \geq 0$ existe una Máquina de Turing con $k+1$ cintas que acepta el mismo lenguaje en tiempo

$$\frac{1}{2^m} t(n) + n$$

Idea de la Demostración

- Vamos a ver cómo se reduce el tiempo a la mitad (aplicándolo varias veces se puede obtener el resultado deseado
- Si M es la máquina que acepta con alfabeto A, construimos una máquina de Turing M' en la que hay un símbolo nuevo w por cada 3 símbolos abc de A.
- La Máquina es tal que codifica la cinta de entrada en otra cinta de la nueva máquina de forma que la casilla i de la nueva cinta va a representar las casillas $2i-1, 2i, 2i+1$ de la cinta de entrada de M.



Idea de la Demostración

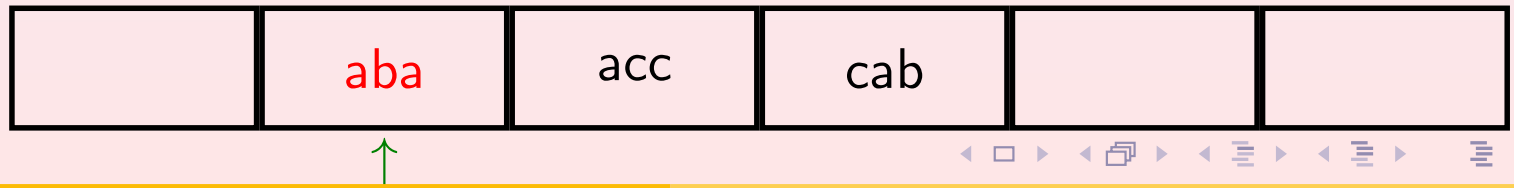
- El programa para M' se escribe, simulando para w lo que haría M para los símbolos abc hasta que sale de estos símbolos (o cicla en ellos). Esto siempre se puede calcular ya que son sólo 3 casillas.

  Esto conlleva resumir, al menos, dos pasos, por cada paso de la original.

- Para llevar cuenta de los símbolos escritos en las casillas comunes a dos celdas consecutivas de M' se supone que ese símbolo se guarda en memoria (añadiendo estados).

- Con esto, cada dos pasos se hacen en 1 y se dividen los pasos por la mitad (hace falta n para cambiar la entrada y codificarla en el nuevo alfabeto).

- Repitiendo varias veces el mismo procedimiento, se obtiene el resultado deseado.



Idea de la Demostración

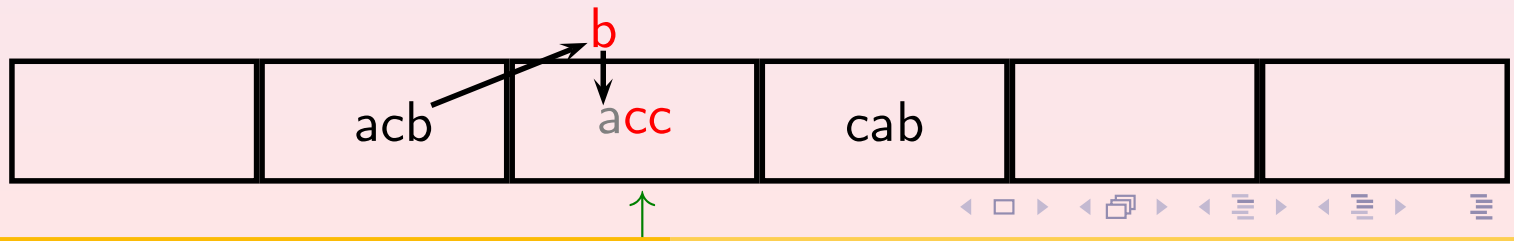
- El programa para M' se escribe, simulando para w lo que haría M para los símbolos abc hasta que sale de estos símbolos (o cicla en ellos). Esto siempre se puede calcular ya que son sólo 3 casillas.

  Esto conlleva resumir, al menos, dos pasos, por cada paso de la original.

- Para llevar cuenta de los símbolos escritos en las casillas comunes a dos celdas consecutivas de M' se supone que ese símbolo se guarda en memoria (añadiendo estados).

- Con esto, cada dos pasos se hacen en 1 y se dividen los pasos por la mitad (hace falta n para cambiar la entrada y codificarla en el nuevo alfabeto).

- Repitiendo varias veces el mismo procedimiento, se obtiene el resultado deseado.



Codificando Problemas

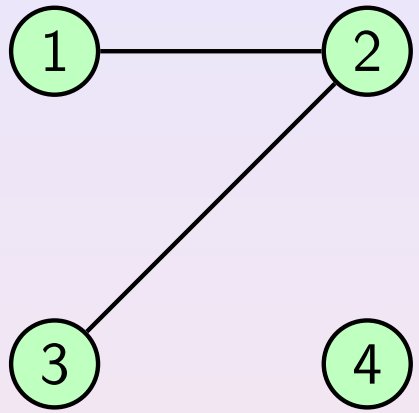
Codificando Números

Los números se pueden codificar en cualquier base menos unario (es muy poco eficiente y necesita mucha longitud). En teoría supondremos binario, pero en la práctica usaremos decimal.

Codificando Objetos de un Conjunto

- Supongamos que tenemos que codificar un conjunto de objetos en un problema, por ejemplo, en un grafo un conjunto de vértices.
- Podemos suponer que cada objeto se codifica con un nombre en un cierto alfabeto, por ejemplo $\{a, b\}$.
- Si tenemos m objetos, ¿cual será lo longitud del nombre de un objeto?
- Con palabras de longitud k tenemos para darle nombre a $m = 2^k$ objetos (hay 2^k palabras distintas de longitud k).
- Luego, si el número de objetos es $m = 2^k$, la longitud del nombre k es del orden de $\log(m)$ donde m es el número de objetos.

Distintas Codificaciones de un Grafo



Para representar un grafo podemos usar distintos procedimientos:

a) Listas los vértices y las aristas

b) Dar una lista de vecinos para cada vértice

c) Dar una matriz de adyacencia del grafo

Mét.	Representación	lg.	Cota Inf.	Cota Superior
a)	v[1]v[2]v[3]v[4](v[1]v[2])(v[2]v[3])	36	$4v+10a$	$4v+10a+(v+2a)\log_{10} v$
b)	(v[2])(v[1]v[3])(v[2])()	24	$2v+8a$	$2v+8a+2a\log_{10} v$
c)	0100/1010/0010/0000	19	v^2+v-1	v^2+v-1

Complejidad de problemas de grafos

- En la complejidad de problemas sobre grafos, si lo que estamos interesados es en saber si es polinómica, da igual la representación que usemos y si la medimos en función del número de vértices.
- Como la longitud de la entrada n verifica que $v \leq n \leq v^3$, una función es de orden polinómico como función de n si y solo si lo es en función de v.
- Si es de orden $O(n^k)$, entonces será a lo más $O(v^{3k})$.
- Si es de orden $O(v^k)$, entonces será a lo más $O(n^k)$.
- También ocurre lo mismo si queremos saber si la complejidad es de orden logarítmico $O(\log(n))$: es independiente de la representación o si lo medimos como una función del número de vértices: el exponente en un logaritmo se transforma en una constante.

Medidas de Complejidad en Espacio

En general para medir el número de unidades que se consumen se siguen las siguiente reglas:

- Se cuentan las casillas que se usan (se escribe o se pasa sobre ellas).
- Si nunca se escribe sobre la cinta de entrada, entonces las casillas de esta cinta no se cuentan.
- Si las casillas de la cinta de salida se escriben de izquierda a derecha, sin volver nunca hacia atrás, tampoco se cuentan.

En un algoritmo que se cuenta el espacio, pueden existir cálculos que nunca acaben, pero siempre se puede transformar el algoritmo en uno que no cicle.

Reconocer palíndromos en espacio $O(\log(n))$

Se hace con una Máquina de Turing con las siguiente estructura de cintas:

1	2	3	3	2	1	□	Entrada
1	0	1	□	□	Posición que se está comprobando (binario) $N2$		
1	1	□	□	□	Contador en binario para encontrar posiciones $N3$		

- Ponemos 1 en la segunda cinta ($N2$), Ponemos 1 en la tercera cinta ($N3$)
- Nos ponemos al principio de la primera cinta.
- Repetir:
    - Repetir: Incrementar $N3$ en 1, mover a la derecha en la primera , hasta $N2 = N3$
    - Copiar el símbolo de la primera cinta en memoria
        - Si el símbolo en memoria es blanco **Aceptar**
        - en otro caso
            - Poner 1 en la tercera cinta ($N3 = 1$), ir al final de la primera cinta
            - Repetir: Incrementar $N3$ en 1, mover izquierda en la primera, hasta $N2 = N3$
            - Si símbolo en primera cinta es distinto al de memoria **Rechazar**
            - Incrementar $N2$ en 1

Los números $N2$ y $N3$ necesitan $\log(n)$ casillas donde n es la longitud del número en la cinta 1.

Existencia de Caminos

Tenemos un grafo dirigido G y dos nodos v_1 y v_2. ¿Existe un camino entre estos dos nodos? Problema $CAMINO(G, v_1, v_2)$
Por ejemplo la MT con tres cintas que tiene como entrada una cinta con las aristas (v, v') del grafo, y después $v1$ y $v2$ con un separados y que funciona de la siguiente forma:

- Colocamos $v1$ en la segunda y tercera cinta
- Repetir hasta que la segunda cinta esté vacía:
    - Cogemos el último elemento de la segunda cinta v
    - Buscamos todos los pares (v, v') en la primera cinta
    - Si $v' = v2$ **Aceptar**
    - Si v' no está en la tercera cinta, se añade a la segunda y la tercera cinta
    - Se borra v de la segunda cinta
- **Rechazar**

G		v_1	v_2	Entrada	
u	z	v	Nodos por analizar		
x	y	u	z	v	Nodos visitados

Existencia de Caminos

G		v_1	v_2	Entrada	
u	z	v		Nodos por analizar	
x	y	u	z	v	Nodos visitados

Supongamos n el tamaño de la entrada en la primera cinta.
Las cintas 2 y 3 siempre son más cortas que la entrada, luego es de $O(n)$ en espacio.
El número de pasos sobre las tres cintas es:

- La cinta 1 se recorre un número de veces menor o igual al número de vértices v que a su vez es menor o igual a n y como su tamaño es n, el número de pasos es de orden $O(n^2)$.
- En la cinta 2 cada nodo se recorre varias veces: para escribirlo, buscarlo en la primera cinta y borrarlo. Buscarlo en la entrada, a lo más una vez por arista. El número de aristas es menor o igual a n, y recorrer todos los nodos una vez es, a lo más n, luego la complejidad total es de orden $O(n^2)$.
- La cinta 3, hay que recorrerla cada vez que analizamos un nodo v y encontramos la arista (v, v'). Como cada nodo se analiza solo una vez, entonces a lo más se recorre una vez por arista. Como el número de aristas es $O(n)$ y el tamaño de la cinta es $O(n)$, obtenemos $O(n^2)$ en total.

En **total**, la complejidad en tiempo es $O(n^2)$ y en espacio $O(n)$
También se puede hacer en espacio $O(v)$ y en tiempo $O(v^3)$, donde v es el número de vértices (por ejemplo codificando como matriz 0-1).

CAMINO: Espacio Determinista

Teorema de Savitch: *Existencia de Caminos* La complejidad en espacio de existencia de caminos en grafos dirigidos es del orden $O(\log^2(v))$, donde v es el número de nodos del grafo y, por tanto, también en función del tamaño de la entrada n.

La demostración se basa en una resolución ordenada del problema CAMINO(x, y, i): Existencia de un camino de longitud $\leq 2^i$, y la relación

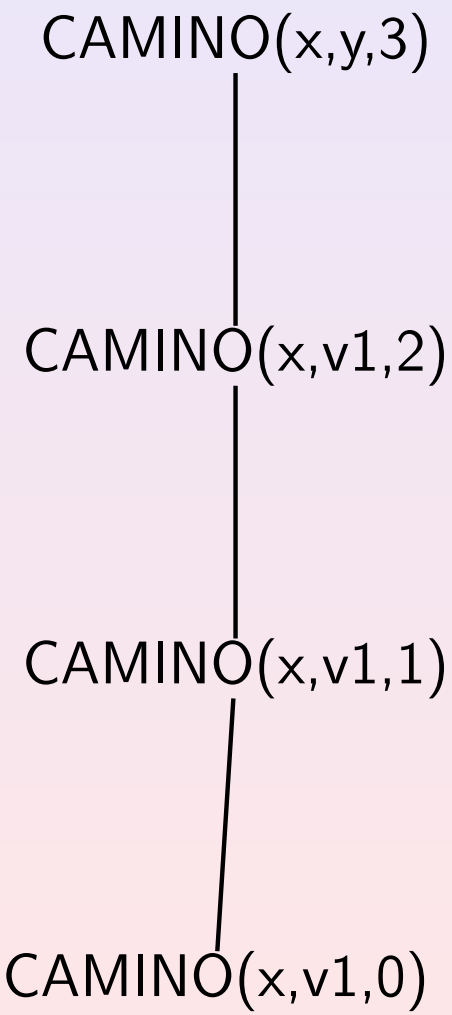
$$\text{CAMINO}(x, y, i) \Leftrightarrow \exists z, \text{CAMINO}(x, z, i-1) \text{ y } \text{CAMINO}(z, y, i-1)$$

CAMINO: Espacio Determinista

Como si ejecutásemos el siguiente algoritmo recursivo al que hay que llamar con $N > \log(v)$:

CAMINO(x,y,N)
- Si x=y Return TRUE
- Si N=0
    - Return TRUE si hay un enlace entre x e y en G
    - Return FALSE si no hay un enlace entre x e y en G
- Para cada nodo z:
    - Si CAMINO(x,z,N-1)
        - Si CAMINO(z,y,N-1)
            - Return TRUE
- Return FALSE

CAMINO: Espacio Determinista

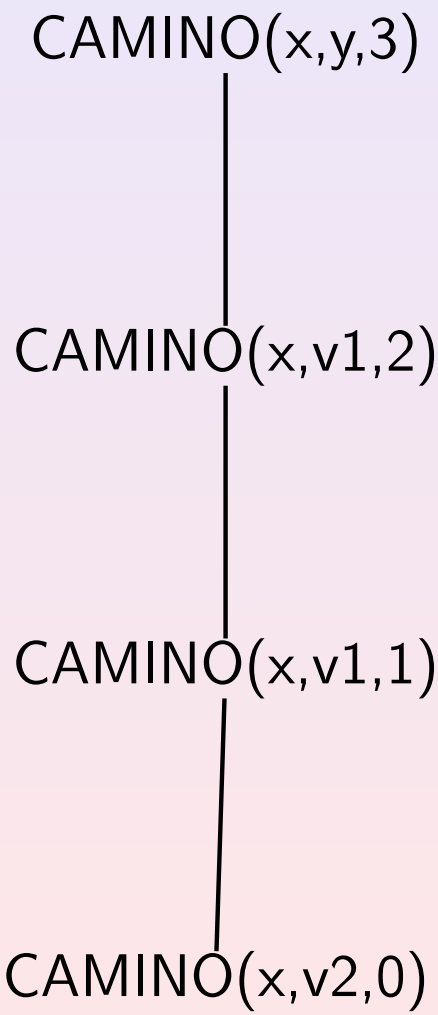


- Si los nodos son iguales, devuelve TRUE
- Si $N = 0$ comprueba en el grafo si existe un enlace
- Haz primera llamada con el primer nodo
- Si recibe FALSE del nivel anterior, prueba siguiente nodo
- Si es el último devuelve FALSE
- Si recibe TRUE de la primera: realiza segunda mismo nodo
- Si recibe TRUE de la segunda, devuelve TRUE

La MT va colocando las tripletas necesarias para resolver el problema de forma recursiva en una cinta separadas por un 0 ó un 1 si es la primera llamada o la segunda llamada. En cada llamada recursiva sólo hay que colocar una tripleta.

Cada tripleta incluyendo el bit de primera o segunda llamada es de longitud $O(\log(v))$ y el número de tripletas es de orden $O(\log(v))$: TOTAL $O(\log^2(v))$ en espacio.

CAMINO: Espacio Determinista

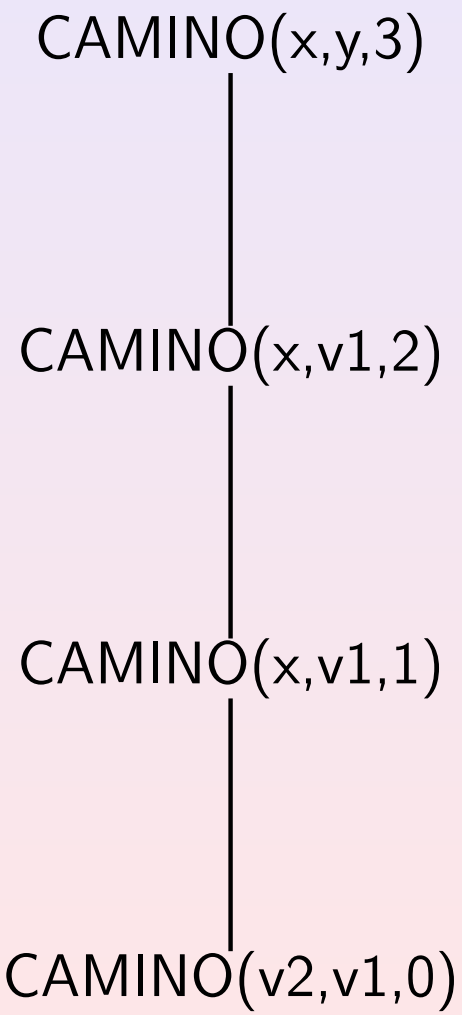


- Si los nodos son iguales, devuelve TRUE
- Si $N = 0$ comprueba en el grafo si existe un enlace
- Haz primera llamada con el primer nodo
- Si recibe FALSE del nivel anterior, prueba siguiente nodo
- Si es el último devuelve FALSE
- Si recibe TRUE de la primera: realiza segunda mismo nodo
- Si recibe TRUE de la segunda, devuelve TRUE

La MT va colocando las tripletas necesarias para resolver el problema de forma recursiva en una cinta separadas por un 0 ó un 1 si es la primera llamada o la segunda llamada. En cada llamada recursiva sólo hay que colocar una tripleta.

Cada tripleta incluyendo el bit de primera o segunda llamada es de longitud $O(\log(v))$ y el número de tripletas es de orden $O(\log(v))$: TOTAL $O(\log^2(v))$ en espacio.

CAMINO: Espacio Determinista



- Si los nodos son iguales, devuelve TRUE
- Si $N = 0$ comprueba en el grafo si existe un enlace
- Haz primera llamada con el primer nodo
- Si recibe FALSE del nivel anterior, prueba siguiente nodo
- Si es el último devuelve FALSE
- Si recibe TRUE de la primera: realiza segunda mismo nodo
- Si recibe TRUE de la segunda, devuelve TRUE

La MT va colocando las tripletas necesarias para resolver el problema de forma recursiva en una cinta separadas por un 0 ó un 1 si es la primera llamada o la segunda llamada. En cada llamada recursiva sólo hay que colocar una tripleta.

Cada tripleta incluyendo el bit de primera o segunda llamada es de longitud $O(\log(v))$ y el número de tripletas es de orden $O(\log(v))$: TOTAL $O(\log^2(v))$ en espacio.

Clases No-Deterministas

La función de complejidad se puede medir en una máquina de Turing no-determinista.

Clases no-deterministas

Una máquina de Turing no determinista tiene complejidad $f(n)$ en tiempo (espacio) si y solo si para una entrada x de longitud n **todas** las posibles opciones de cálculo de la máquina terminan en $f(n)$ pasos (terminan y no usan más de $f(n)$ casillas).

Ejemplo (en términos de algoritmos)

Un algoritmo no-determinista que resuelve el problema de los colores en un grafo donde m máximo de colores.

1. Asignar un color posible a cada nodo: mediante una serie de pasos que asignan de forma no determinista un número mediante una serie de 0s o 1s de longitud menor o igual a la longitud de m.
2. Si para un nodo hemos asignado un número mayor que m rechazar.
3. Comprobar si no hay dos nodos conectados con el mismo color, entonces aceptar.
4. En caso contrario rechazar.

Para que el algoritmo resuelva el problema, si la respuesta es SI, entonces debe de existir la posibilidad de que acepte; si la respuesta es NO, necesariamente ha de rechazar.
La complejidad en tiempo (no-determinista) sería proporcionar al número de arcos.

Búsqueda de Caminos en Grafos: Espacio no-determinista

En espacio no-determinista el problema se resuelve en espacio $O(\log(v))$ donde v es el número de vértices, (la misma complejidad en función de la entrada $O(\log(n))$).

Supongamos que queremos determinar si existe un camino entre el nodo x_i y el nodo x_j:

Llamamos a $Camino(x_i, x_j, m)$ con $m = v$ (el número de nodos).

Si $m = 0$ solo acepta si $x_i = x_k$ y rechaza si $x_i \neq x_k$.

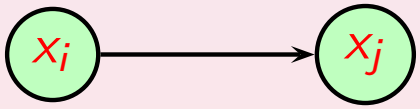
Si $m > 0$:

El algoritmo funciona escribiendo el identificador de un nodo x_k conectado con x_i de forma no-determinista (si no existe ninguno rechaza).

a) Si x_k es igual a x_j, entonces para y acepta.

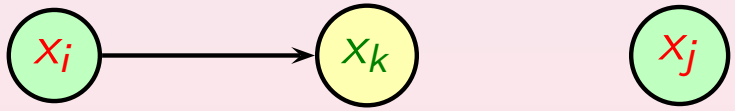
b) Si x_k es distinto de x_j, vuelve a ejecutar el algoritmo con x_k en el lugar de x_i y m decrementado en uno.

$m > 0$:

a) 

SI

$k = j$

b) 

Llamada recursiva a $Camino(x_k, x_j, m-1)$

Clases de Complejidad

TIEMPO(f)	Todos los lenguajes aceptados por una máquina de Turing determinista en tiempo $O(f(n))$.
ESPACIO(f)	Todos los lenguajes aceptados por una máquina de Turing determinista en espacio $O(f(n))$.
NTIEMPO(f)	Todos los lenguajes aceptados por una máquina de Turing n determinista en tiempo $O(f(n))$.
NESPACIO(f)	Todos los lenguajes aceptados por una máquina de Turing n determinista en espacio $O(f(n))$.

Clases de Complejidad

- Clase polinómica (tiempo): $\mathrm{P} = \bigcup_{j>0} \mathrm{TIEMPO}(n^j)$
- Clase polinómica no determinista (tiempo): $\mathrm{NP} = \bigcup_{j>0} \mathrm{NTIEMPO}(n^j)$
- Clase polinómica (espacio): $\mathrm{PESPACIO} = \bigcup_{j>0} \mathrm{ESPACIO}(n^j)$
- Clase polinómica no determinista (espacio): $\mathrm{NPESPACIO} = \bigcup_{j>0} \mathrm{NESPACIO}(n^j)$
- Clase de espacio logarítmico determinista: $\mathrm{L} = \mathrm{ESPACIO}(\log(n))$
- Clase de espacio logarítmico no determinista: $\mathrm{NL} = \mathrm{NESPACIO}(\log(n))$
- Clase exponencial en tiempo: $\mathrm{EXP} = \bigcup_{j>0}(2^{n^j})$

Dependencia del Modelo

Tesis de Church-Turing

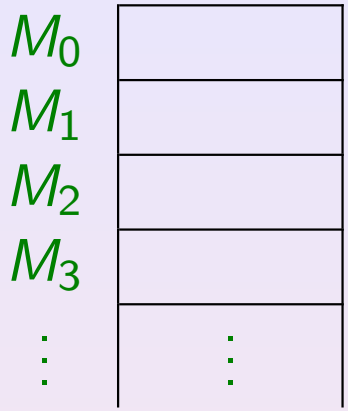
Todo procedimiento de cálculo físicamente realizable se puede simular por una máquina de Turing.

Tesis de Church-Turing Fuerte

Todo procedimiento de cálculo físicamente realizable se puede simular por una máquina de Turing, con una sobrecarga polinómica en el número de pasos (si el mecanismo da $f(n)$ pasos, entonces la máquina de Turing puede dar $f^k(n)$ pasos).

La tesis de Church-Turing fuerte es más controvertida, ya que no se piensa que no se verifica para ordernadores cuánticos.

Máquina RAM



Tipos de Instrucciones:

$M_i \leftarrow 1$
$M_i \leftarrow M_j + M_k$
$M_i \leftarrow M_j - M_k$
$M_i \leftarrow [M_1/2]$
$M_i \leftarrow M_{M_j}$ (poner en M_i el valor contenido en la celda número M_j)

$M_{M_i} \leftarrow M_j$ (poner en la celda número M_i el valor de M_j)
Goto m if $M_i > 0$
Halt

Cada celda contiene un entero de cualquier longitud.
Una RAM se controla por un programa que se guarda en la unidad de control.
El estado de la Máquina RAM es el número de instrucción que se está ejecutando.

Máquina RAM

Entrada: m enteros en las celdas $M_1, \ldots, M_m$
Tamaño de la Entrada: Suma del tamaño de los enteros de entrada
RAM Aceptadoras: Escriben 0 en M_0 si rechazan y 1 si aceptan. Si no paran rechazan
RAM calculadoras de $f(x)$: Escriben $f(x)$ en M_0

Tabla de Simulaciones

	Máquina Simuladora		
Máquina Simulada	1TM	kTM	RAM
Máquina Turing 1 cinta: 1TM	—	$O(T(n))$	$O(T(n)\log T(n))$
Máquina Turing k cintas: kTM	$O(T^2(n))$	—	$O(T(n)\log T(n))$
Máquina RAM: RAM	$O(T^3(n))$	$O(T^2(n))$	—

(Del libro de Garey, Johnson. Distintos autores pueden dar distintas relaciones. Lo importante es que no cambiemos de clase cambiando de modelo.)

Simulación de Máquinas No-Deterministas

Teorema.- *Supongamos que L es decidido por una máquina de Turing no-determinista en tiempo $f(n)$, entonces es decidido por una máquina de Turing determinística con 3 cintas en tiempo $O(d^{f(n)})$ donde $d > 1$ es una constante que depende de la máquina no determinística inicial.*

Supongamos que k es el número máximo de opciones de la MT no determinista y que $k > 1$ (en otro caso la Máquina es determinista y el resultado es trivial):

- Para $L = 0, 1, 2, \ldots$
  - Vamos colocando en la tercera cinta todas las secuencias $a_1 \ldots a_L$ de longitud L, donde cada símbolo se elige entre $\{1, \ldots, k\}$.
  - Para cada secuencia $a_1 \ldots a_L$, simulamos la MT no determinista L pasos en una segunda cinta donde en el paso i se elige la opción a_i.
  - Si para una secuencia la simulación acepta, termina y acepta
  - Si para todas las secuencias de longitud L la simulación rechaza, entonces termina y rechaza
  - Si no ocurre ninguna de las dos cosas, pasa al siguiente L

Simulación de Máquinas No-Deterministas

- La simulación termina seguro cuando $L = f(n)$ o antes.
- La cantidad de secuencias de longitud L es k^L. La cantidad total de secuencias es menor o igual a :

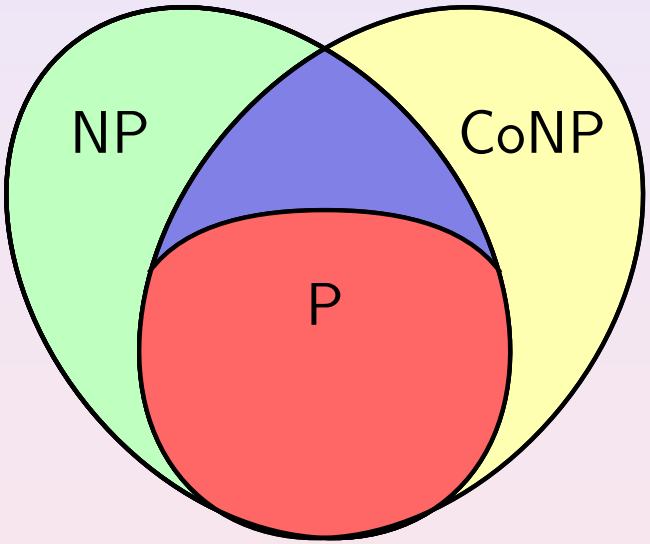
$$\sum_{L=0}^{f(n)} k^L = \frac{k^{f(n)+1} - 1}{k - 1}$$

  que es de orden $k^{f(n)}$.
- Cada simulación y cada cambio de secuencia (para pasar a la siguiente) se lleva del orden de $L \leq f(n)$ pasos.
- En total tenemos que la simulación es de orden $O(f(n)k^{f(n)})$ y como $f(n)$ es menor que $k^{f(n)}$, tenemos que la simulación es de orden $O(k^{f(n)}.k^{f(n)})$ y teniendo en cuenta que $k^{f(n)}.k^{f(n)} = k^{2f(n)} = (k^2)^{f(n)}$ y el resultado se obtiene para $d = k^2$.

Complementarios de Clases

La clase de los lenguajes complementarios de los lenguajes en la clase C se llama CoC.
Se verifica que $L \in C \Leftrightarrow \overline{L} \in CoC$.
El complementario de una clase determinista coincide con la propia clase: $\mathrm{CoP} = \mathrm{P}$.
No ocurre lo mismo con las clase no deterministas.
La clase CoNP es el conjunto de problemas cuyo complementario está en NP.

P, NP y CoNP



Relación en $A^* \times A^*$

Una relación binaria R es un subconjunto $R \subseteq A^* \times A^*$

Podemos considerar también que una relación binaria es una aplicación $R : A^* \times A^* \to \{0,1\}$ (su función característica) donde $R(x,y) = 1$ cuando $(x,y) \in R$.
$(x,y) \in R$ a veces se escribe también como xRy.

Relación calculable polinómicamente

R en $A^* \times A^*$ se dice calculable polinómicamente si existe una MT M que calcula la función característica de R en tiempo polinómico (o que acepta R en tiempo polinómico).

Es decir, para cada x, y de entrada calcula si xRy.

Definiciones Alternativas

NP

Un lenguaje $L \subseteq A^*$ está en NP si y solo si existe una relación R en $A^* \times A^*$ calculable en tiempo polinómico y un polinomio $p(n)$ tal que

$$L = \{x \in A^* : \exists y \in A^* \text{ con } |y| \leq p(|x|), R(x,y) = 1\}$$

Se dice que los lenguajes (problemas) de NP son los problemas que se pueden *verificar* en tiempo polinómio (*de forma eficiente*).
Al algoritmo que calcula R se le llama un verificador. A y se le llama un certificado.

CoNP

Un lenguaje $L \subseteq A^*$ está en CoNP si y solo si existe una relación R en $A^* \times A^*$ calculable en tiempo polinómico y un polinomio $p(n)$ tal que

$$L = \{x \in A^* : \forall y \in A^* \text{ con } |y| \leq p(|x|), R(x,y) = 1\}$$

Ejemplo

El problema del circuito hamiltoniano está en NP, porque se puede expresar como determinar los x (que representan grafos) para los que existe un y (que representa una sucesión de nodos) tal que todos los nodos aparecen una y una sola vez y existe un arco desde cada nodo al siguiente y del último al primero (relación R).

- La condición que se pide para x e y (la relación $R(x, y)$) se puede calcular en tiempo polinómico.
- La longitud del y que cumple la relación es menor que la de x, entonces la longitud de y está acotada por un polinomio de la longitud de x.

Problemas NP

Los que se pueden expresar como: dados unos datos x comprobar si existe un objeto y (con un tamaño limitado a un polinomio del tamaño de x) tal que se cumple una condición $R(x, y) = 1$ que es verificable en tiempo polinómico.

Ejemplo

Saber si un número x es compuesto: si existe $1 < y < x$ tal que la división entera de x entre y es exacta.

Problemas NP

Los que se pueden expresar como: dados unos datos x comprobar si existe un objeto y (con un tamaño limitado a un polinomio del tamaño de x) tal que se cumple una condición $R(x, y) = 1$ que es verificable en tiempo polinómico.

Ejemplo

Saber si un número x es compuesto: si existe $1 < y < x$ tal que la división entera de x entre y es exacta.

C. Moore, S. Mertens (2011) The Nature of Computation.

$$193707721 \times 761838257287 = 147573952588676412927$$

Frank Nelson Cole, American Mathematical Society, 1903 (trabajó en ello los domingos de 3 años).
El número de la derecha es $2^{67} - 1$ y se había conjeturado en el siglo XVII que era primo.

Problemas NP

Los que se pueden expresar como: dados unos datos x comprobar si existe un objeto y (con un tamaño limitado a un polinomio del tamaño de x) tal que se cumple una condición $R(x, y) = 1$ que es verificable en tiempo polinómico.

Ejemplo

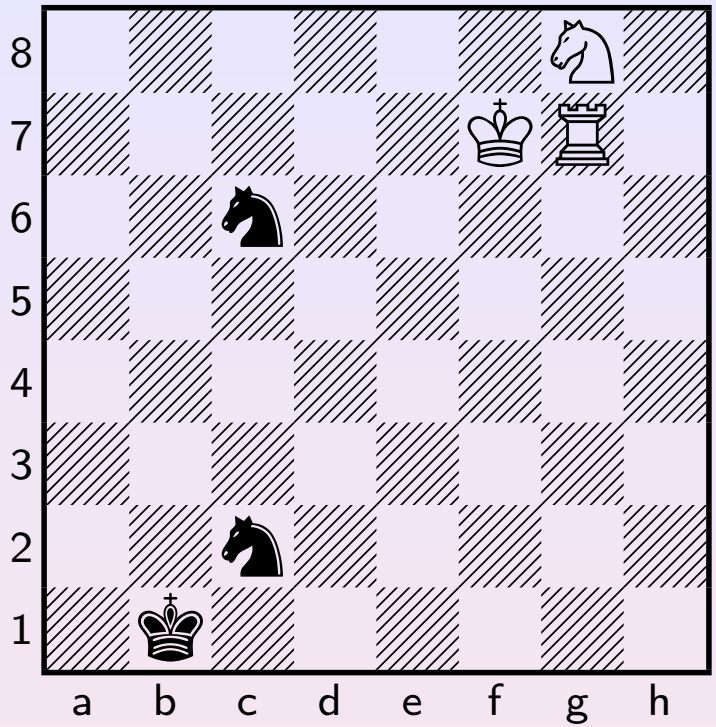
Saber si un número x es compuesto: si existe $1 < y < x$ tal que la división entera de x entre y es exacta.

C. Moore, S. Mertens (2011) The Nature of Computation.

$$193707721 \times 761838257287 = 147573952588676412927$$

Frank Nelson Cole, American Mathematical Society, 1903 (trabajó en ello los domingos de 3 años).
El número de la derecha es $2^{67} - 1$ y se había conjeturado en el siglo XVII que era primo.
¿Conocéis una caracterización parecida para saber si un número es primo? Existe, pero no es sencilla.

Ejemplo



Mate en 262

No es fácil convencer de que realmente se puede obtener un mate en ese número de jugadas. El tamaño del objeto necesario para convencer es el espacio de todas las posibles jugadas de blancas y negras, y ese objeto es de tamaño exponencial.

Sistemas Interactivos de Demostración

Condiciones

- Existe un demostrador con capacidad ilimitada de cálculo (siempre quiere convencer de que la respuesta es positiva)
- Existe un verificador con capacidad polinómica de cálculo (quiere saber la verdad)
- Se pueden intercambiar mensajes de un tamaño polinómico en función de una palabra inicial x

Clase NP

Clase de problemas que el verificador puede decidir, teniendo en cuenta que si la respuesta es positiva, el demostrador tratará de convencer al verificador de que lo es.

El escenario es distinto si se permite aleatorizar las preguntas y un error para el verificador (Computational Complexity: A Modern Approach. Sanjeev Arora, Boaz Barak)

Relaciones entre Clases de Complejidad

a) $\text{ESPACIO}(f(n)) \subseteq \text{NESPACIO}(f(n))$
$\text{TIEMPO}(f(n)) \subseteq \text{NTIEMPO}(f(n))$

b) $\text{NTIEMPO}(f(n)) \subseteq \text{ESPACIO}(f(n))$

c) $\text{NESPACIO}(f(n)) \subseteq \text{TIEMPO}(k^{f(n)})$ para un $k > 1$

a) La demostración de a) es trivial.

b) La demostración de b) se basa en la simulación de una MT no determinista mediante una determinista y que consistía en ir poniendo palabras $a_1 \dots a_L$ donde $a_i \in \{1, \dots, k\}$ y entonces ir simulando la MT por una determinista que en el paso i coge la opción número i. El espacio que hace falta es el espacio para escribir las palabras que es menor o igual a $f(n)$ (recordemos que $L \leq f(n)$) y el espacio para hacer la simulación de L pasos, que también es de orden $f(n)$. En total, el espacio necesario es de orden $f(n)$.

El Método de la Alcanzabilidad:

Se basa en considerar un grafo en el que los nodos son las posibles configuraciones de una Máquina de Turing, y los arcos conectan configuraciones tales que se puede llegar de una a otra en un paso de cálculo. Para simplificar vamos a suponer una MT con una sola cinta. Una configuración es una tripleta (q, u, v). Como no se ocupa más de $f(n)$ en espacio, la longitud de u y v es menor o igual a $f(n)$. El número de configuraciones es por tanto del orden de

$$|Q||B|^{2f(n)}$$

donde B es el alfabeto de trabajo. Es decir del orden de $t^{f(n)}$ donde $t = |B|^2$.

Saber si una palabra es aceptada es equivalente a saber si existe un camino desde el estado inicial a una configuración que contenga un estado final y eso se puede comprobar en tiempo $O(m^3)$ donde m es el número de nodos, en nuestro caso en orden $\left(t^{f(n)}\right)^3 = t^{2f(n)} = \left(t^3\right)^{f(n)}$. Lo que demuestra el teorema para $k = t^3$.

Teorema de la Jerarquía

Funciones de Complejidad Propias

Son aquellas que verifican: $f(n+1) \geq f(n)$ y tales que la función $g(u) = f(|u|)$ expresando $g(u)$ como una secuencia de longitud $g(u)$ (es decir usando unario) es calculable en espacio $O(f(n))$ y tiempo $O(f(n)+n)$.

Teorema de la jerarquía en Tiempo

Si $f(n) \geq n$ es propia, entonces
$\text{TIEMPO}((f(n)) \subset \text{TIEMPO}(f^2(n))$
$\text{TIEMPO}((f(n)) \neq \text{TIEMPO}(f^2(n))$

Corolario.

$\text{P} \neq \text{EXP}$

Dem. $\text{P} \subseteq \text{TIEMPO}(2^n) \subsetneq \text{TIEMPO}((2^n)^2) \subseteq \text{EXP}$

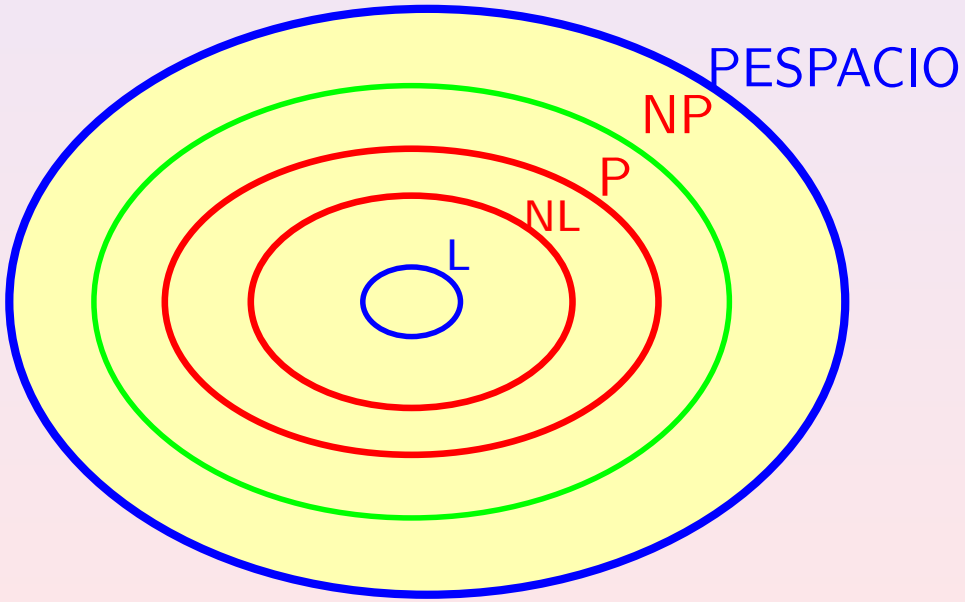
Jerarquía en Espacio

Si $f(n)$ es una función de complejidad propia, entonces
$\text{ESPACIO}(f(n)) \subset \text{ESPACIO}(f(n)\log f(n))$
$\text{ESPACIO}(f(n)) \neq \text{ESPACIO}(f(n)\log f(n))$

Inclusión de Clases

$$L \subseteq NL \subseteq P \subseteq NP \subseteq PESPACIO$$

No se sabe si alguna o varias de estas inclusiones son igualdades. Lo único que se sabe es que, **NL** está incluida escrictamente en **PESPACIO**.



Espacio No-Determinista

Teorema

Si $f(n) \geq \log(n)$ y es propia, entonces

$$\text{NESPACIO}(f(n)) \subseteq \text{ESPACIO}(f^2(n))$$

Corolario

$$\text{PESPACIO} = \text{NPESPACIO}$$

Teorema

Si $f(n) \geq \log(n)$ es propia entonces

$$\text{NESPACIO}(f(n)) = \text{CoNESPACIO}(f(n))$$