Fundamentos de la Computación

TC-4001

Enero 2010

Finalidad

- Predecir los recursos que requiere la ejecución de un algoritmo
  - Espaciales
  - Temporales
- El análisis se enfocará primordialmente al segundo tipo

Modelo

- Suposición: un solo procesador

Problema

- Componentes
  - Instancia: Datos
  - Pregunta

Algoritmo

- Procedimiento para resolver un problema
- Número finito de pasos

Relación

- Tamaño del Input

- Dado un algoritmo
- Tiempo de procesamiento

Tamaño del input

- Cantidad de espacio requerida para especificar la instancia de un problema

- Ejemplos
  - Ordenar un vector
    - número de componentes
  - Problema de la mochila
    - número y tamaño de los objetos

Tiempo de ejecución

- Número de operaciones básicas
- Debe ser independiente de:
  - procesador
  - lenguaje
  - compilador

Tiempo de ejecución

- Número de operaciones básicas ejecutadas
- Suposición: se requiere una cantidad constante de tiempo para ejecutar cada línea en el pseudo código
- Esto es cada ejecución de la *i-ésima* línea toma un tiempo c_i donde c_i es una constante

- INSERTION
- I NSERTION
- IN SERTION
- INS ERTION
- EINS RTION
- EINRS TION
- EINRST ION
- EIINRST ON
- EIINORST N
- EIINNORST

Insertion Sort

- Input: n, A
- Output: A ordenado
- 1. **for** $j = 2$ **to** n
- 2. $k := A[j]$
- 3. $i := j - 1$
- 4. **while** $i > 0$ **and** $A[i] > k$
- 5. $A[i+1] := A[i]$
- 6. $i := i-1$
- 7. $A[i+1] := k$

Cálculo del tiempo

$$T(n) = c_1 n + c_2 (n-1) +$$

$$c_3 (n-1) + c_4 \sum_{j=2}^{n} t_j +$$

$$c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) +$$

$$c_7 (n-1)$$

Casos

- 1. Mejor: los números ya están ordenados, $t_j = 1$
- $T(n) = a\ n + b$

- 2. Peor: los números están ordenados en sentido inverso
- $t_j = j$
- $T(n) = a\ n^2 + bn + c$

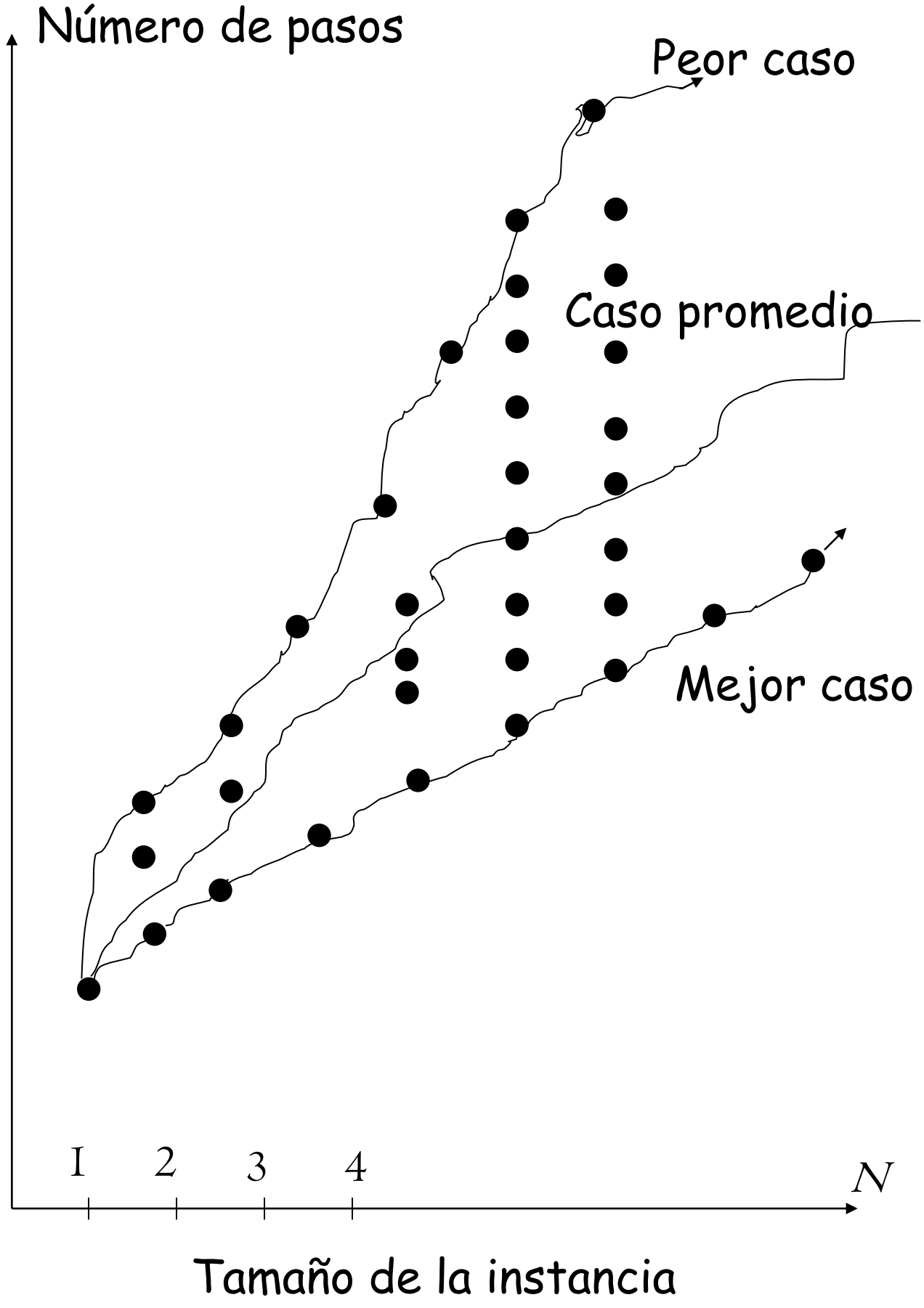
Peor vs. Promedio

- 1. El peor de los casos es una cota superior sobre el tiempo de ejecución de un algoritmo para cualquier instancia. Nos da una garantía de que algoritmo nunca empleará más tiempo.

Peor vs. Promedio

- 2. Para ciertos algoritmos el peor de los casos sucede muy a menudo. Ej: algoritmos de búsqueda.

- 3. Muchas veces el caso promedio es tan malo como el peor de los casos. Ej. Insertion sort $t_j = j / 2$


Número de pasos
Peor caso
Caso promedio
Mejor caso
1
2
3
4
N
Tamaño de la instancia

- SELECTIONSORT
- C SELETIONSORT
- CE SLETIONSORT
- CEE SLTIONSORT
- CEEI SLTONSORT
- CEEIL STONSORT
- CEEILN STOSORT
- CEEILNO STSORT
- CEEILNOO STSRT
- CEEILNOOR STST
- CEEILNOORS TST
- CEEILNOORSS TT
- CEEILNOORSST T
- CEEILNOORSSTT

Selection Sort

- Input: n, A
- Output: A ordenado
- 1. **for** $i = 1$ **to** n
- 2. $min = i$
- 3. **for** $j := i+1$ **to** n
- 4. **if** $(A[j] < A[min])$ $min = j$
- 5. $t = A[min]$
- 6. $A[min] = A[i]$
- 7. $A[i] := t$

Crecimiento asintótico

- $f,g : \mathbf{N} \dashrightarrow \mathbf{R}^*$
- $O(f) = \{g \mid \exists\, c \in \mathbf{R}^+, n_0 \in \mathbf{N}, g(n) \leq c\, f(n)\}$

  $\Omega(f) = \{g \mid \exists\, c \in \mathbf{R}^+, n_0 \in \mathbf{N}, g(n) \geq c\, f(n)\}$

  $\Theta(f) = O(f) \cap \Omega(f)$

Propiedades

- 1. Transitividad

  $f \in O(g), g \in O(h) \Rightarrow f \in O(h)$
- 2. $f \in O(g) \Leftrightarrow g \in \Omega(f)$
- 3. $f \in \Theta(g) \Leftrightarrow g \in \Theta(f)$
- 4. Θ induce una relación de equivalencia. $\Theta(f)$ es una clase de equivalencia: clase de complejidad.
- 5. $O(f+g) = O(\max\{f, g\})$

Algunos Ejemplos

- $\log n \in O(n^{\alpha}),\ \alpha > 0$
- $n^{k} \in O(c^{n}),\ c > 1$

n	lg n	n	n lgn	n**2	2**n	n!
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 años
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 seg	8.4 x 10 ** 15
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 días	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4 x 10 ** 13	
1000	0.010 μs	1.0 μs	9.966 μs	1 ms		
10000	0.013 μs	10 μs	130 μs	100 ms		
100000	0.017 μs	0.1 ms	1.67 ms	10 seg		
1000000	0.020 μs	1 ms	19.93 ms	16.7 seg		
10000000	0.023 μs	0.01 seg	0.23 seg	1.16 días		
100000000	0.027 μs	0.10 seg	2.66 seg	115.7 días		
1000000000	0.030 μs	1 seg	29.9 seg	31.7 años		

Algunas funciones comunes

Pisos y techos

Para cualquier número real x, definimos :

$\lceil x \rceil$ el “techo” de x, el menor entero mayor o igual que x.

$\lfloor x \rfloor$ el “piso” de x, el mayor entero menor o igual que x.

Algunas propiedades

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil \leq x + 1$$

Para cualquier entero n,

$$\lceil n / 2 \rceil + \lfloor n/2 \rfloor = n$$

y para cualquier entero n, y enteros a, $b \neq 0$.

$$\lceil \lceil n / a \rceil / b \rceil = \lceil n / ab \rceil$$

Exponenciales

Para cualquier número real a, m y n enteros, tenemos las siguientes identidades:

$$a^0 = 1,$$

$$a^1 = a,$$

$$a^{-1} = 1/a,$$

$$(a^m)^n = a^{mn},$$

$$(a^m)^n = (a^n)^m$$

$$a^m a^n = a^{m+n}$$

Logaritmos

Notación:

$\log n = \log_2 n$

$\ln n = \log_e n$

Propiedades:

$a = b^{\log_b a}$,

$\log_c (ab) = \log_c a + \log_c b$

$\log_b a^n = n \log_b a$

Propiedades (cont.)

$$\log_b a = \log_c a / \log_c b,$$

$$\log_b (1/a) = - \log_b a,$$

$$\log_b a = 1 / \log_a b,$$

$$a^{\log_b n} = n^{\log_b a}$$

Ejemplo

- Ordenar las siguientes funciones con respecto de su orden de crecimiento.
- $\log n$
- $4^{\log n}$
- 2^{n}
- 2^{2}
- 1/n
- $(3/2)^{n}$

Encontrar la máxima componente en L

- Input : n, L
- * L : arreglo de tamaño n*
  Output : *max*
  * componente en L de mayor valor*
- *max* := L[1]
- **for** *ind* = 2 **to** n
  if *max* < L[*ind*]
  max := L[*ind*]
- **end**
- **return**(*max*)

Búsqueda secuencial

- Input: *n*. *L*. *x*
- * *L* arreglo de tamaño *n* ; *x* elemento buscado *
- Output: *ind*
- * posición de *x* en *L* (0 si no está) *
- 1. *ind* := *n*
- 2. **while** *ind* ≥ 1 **and** *L*[*ind*] ≠ *x*
- 3. *ind* := *ind* **-** 1
- 4. **end**
- 5. **return**(*ind*)

Tiempo Promedio

D_n el conjunto de todas las instancias de tamaño n para el problema.

A su vez el conjunto D_n puede dividirse en subconjuntos ajenos de instancias, para los cuales el algoritmo requiera la misma cantidad de tiempo, I_j.

Tiempo promedio (cont)

Sea $p(I_j)$ la probabilidad de que suceda una instancia de la clase I_j.

Sea $t(I_j)$ el tiempo requerido por el algoritmo para una instancia de la clase I_j.

Definimos el tiempo promedio

$$A(n) = \sum_j p(I_j)t(I_j)$$

Análisis de Búsqueda Secuencial

- Peor de los casos: $W(n) = n$

- Análisis Promedio:
- Suposiciones:
- Elementos de la lista son distintos
- x se encuentra en la lista
- x tiene la misma probabilidad de estar en cualquier posición

Búsqueda binaria

- Input: n, L, x
- Output: *ind*
- 1. *first* := 1, *last* := n
- 2. *found* := 0;
- 3. **while** *first* ≤ *last* **and**
- **not** *found* **do**
- 4. *ind* := $\lfloor$(*first* + *last*) / 2$\rfloor$
- 5. **if** $x = L($*ind*$)$ **then** *found*:=1
- 6. **elseif** $x < L[$*ind*$]$ *last*:=*ind*-1
- 7. **else** *first* := *ind* + 1
- 8. **if not** *found* **then** *index* := 0
- 9. **end**

Relación de recurrencia

- $W(n) = 1 + W(\lfloor n/2 \rfloor)$, $n > 1$
- $W(1) = 1$

Divide y Vencerás

Dividir el problema en varios subproblemas similares al original pero de menor tamaño.

Vencer Resolviendo los problemas recursivamente

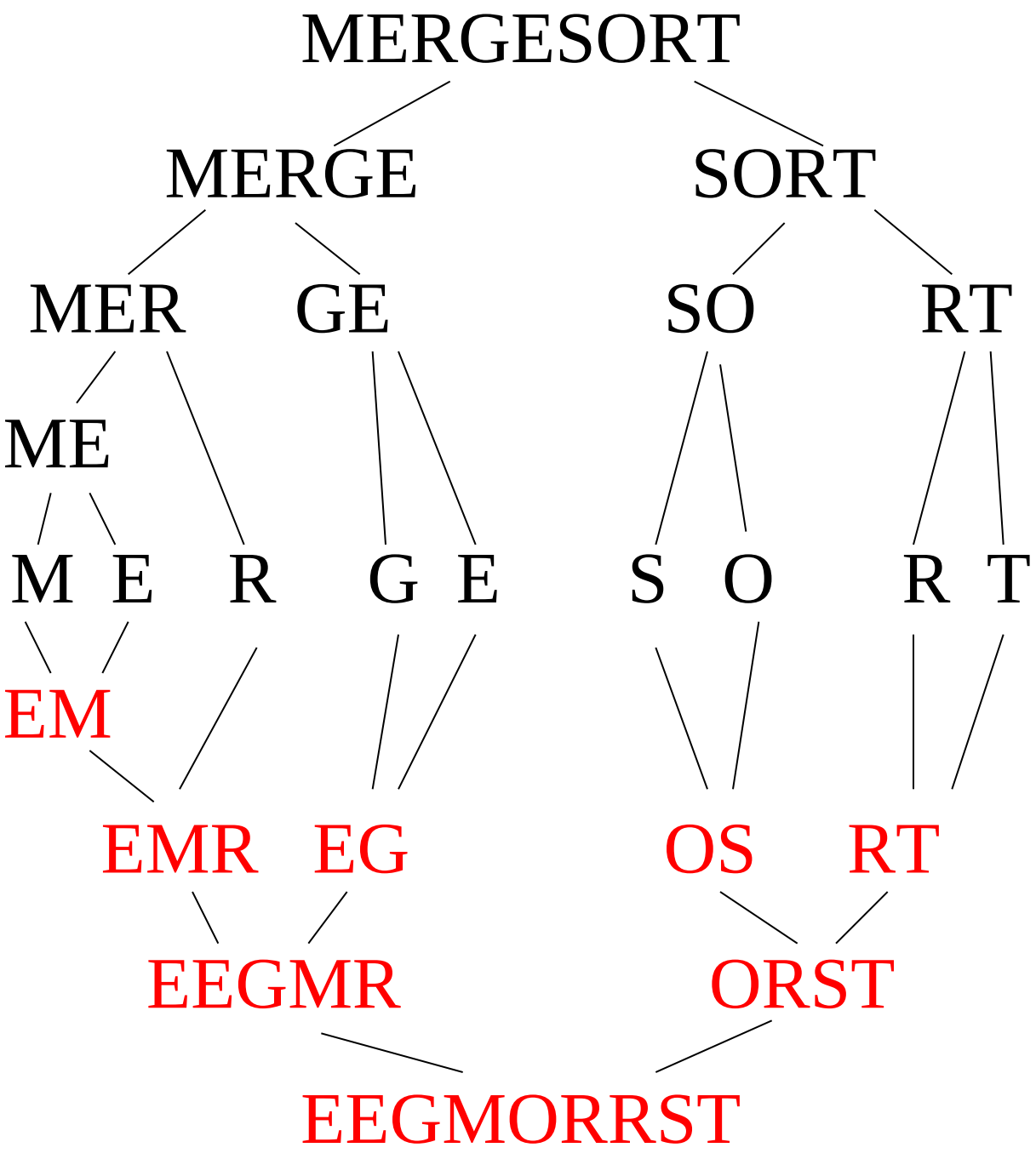
Combinar estas soluciones para crear una solución al problema original.

Ordenamiento por Mezcla

Dividir: Dividir el arreglo a ser ordenado en 2 subarreglos de tamaño $n/2$ cada uno.

Vencer: Ordenar los dos subarreglos recursivamente usando ordenamiento por mezcla.

Combinar: Mezclar los dos subarreglos para producir el arreglo ordenado.


MERGESORT
MERGE
SORT
MER
GE
SO
RT
ME
M
E
R
G
E
S
O
R
T
EM
EMR
EG
OS
RT
EEGMR
ORST
EEGMORRST

Análisis de divide y vencerás

Ecuación de recurrencia:

Si el problema se divide en a subproblemas, donde cada uno es de tamaño $1/b$ del original, y se requiere de un tiempo D para dividir y C para combinar, obtenemos

$$T(n) = D(n) + a\,T(n/b) + C(n)$$

$$T(1) = \mathbf{O}(1)$$

Análisis de Ordenamiento por Mezcla

Dividir: Tiempo constante $\mathbf{O}(1)$

Vencer: Resolver dos subproblemas de tamaño $n/2$. Esto requiere $2\,T(n/2)$

Combinar: Mezclar los elementos de dos subarreglos ordenados requiere $C(n)$.

Ecuación de recurrencia

$$T(n) = 2\,T(n/2) + n$$

$$T(1) = 0$$

Programación Dinámica

Divide y vencerás es una técnica para el diseño de algoritmos.

La programación dinámica es otra técnica, la cual también resuelve un problema combinando las soluciones de subproblemas.

La diferencia es que p.d. se aplica cuando los subproblemas no son independientes.

Programación Dinámica

Es aplicable cuando los subproblemas comparten subsubproblemas.

Cada subsubproblema se resuelve una sola vez, se guarda su resultado en una tabla, y cada vez que el subsubproblema reaparece, simplemente se recupera su solución.

Multiplicación de una sucesión de matrices

Dada una sucesión de matrices A_1, A_2,…, A_n, se pide multiplicarlas, esto es calcular el producto $A_1A_2 \ldots A_n$.

El producto de matrices es una operación binaria, es decir la única manera de multiplicar matrices es por pares.

ej. $ABC = A\,(BC) = (AB)\,C$

Número de multiplicaciones escalares

$C = AB$

Si A es una matriz p x q

y B es una matriz q x r

entonces C es una matriz p x r

El número de multiplicaciones necesarias para calcular C es pqr.

Costo de la asociación

Dadas 3 matrices A, B , C de dimensiones 10 x 100, 100 x 5 y 5 x 50, respectivamente.

¿Cuánto cuesta multiplicar $(AB)C$?

Y ¿cuánto cuesta $A(BC)$?

$(AB)C$

- A 10 x 100
- B 100 x 5
- C 5 x 50
- AB 10 x 5
- Para calcular el producto AB se requieren (10)(100)(5) = 5,000 multiplicaciones escalares.
- Para calcular (AB) C se requieren (10) (5) (50) = 2,500
- Total 7,500 multiplicaciones escalares

$A(BC)$

- A 10 x 100
- B 100 x 5
- C 5 x 50
- BC 100 x 50
- Para calcular el producto BC se requieren (100)(5)(50) = 25,000 multiplicaciones escalares.
- Para calcular $A(BC)$ se requieren (10) (100) (50) = 50,000
- Total 75,000 multiplicaciones escalares

El caso de *n* matrices

Si tenemos *n* matrices para multiplicar, el número de posibles asociaciones es

$$C(n) = \frac{1}{n+1}\binom{2n}{n} = \Omega(4^n / n^{3/2})$$

Los números de Catalan

Pasos para un algoritmo de programación dinámica

1. Caracterizar la estructura de una solución óptima.
2. Definir recursivamente el valor de una solución óptima
3. Calcular el valor de una solución óptima
4. Construir una solución óptima a partir de la información calculada.

- Dimensiones de las matrices:

- A_1 p_o x p_1
- A_2 p_1 x p_2

- A_i p_{i-1} x p_i

- A_n p_{n-1} x p_n

Primer paso

Notación: $A_{i\ldots j}$ la matriz que resulta al evaluar el producto $A_i A_{i+1}\ldots A_j$.

Una asociación óptima del producto $A_{1\ldots n}$ divide la secuencia en $A_{1\ldots k}$ y $A_{k+1\ldots n}$, para algún entero k.

Así, una solución óptima para el problema contiene soluciones óptimas a subproblemas.

Segundo paso

Solución recursiva.

$m[i,j]$ = mínimo número de multiplicaciones escalares necesarias para calcular la matriz $A_{i \ldots j}$.

$m[i,j] = m[i,k]+m[k+1,j]+ p_{i-1} p_k p_j$.

donde k minimiza la expresión sobre $i \leq k < j$.

$m[i,i] = 0$.

matriz	dimensión
A 1	30 x 35
A2	35 x 15
A3	15 x 5
A4	5 x 10
A5	10 x 20
A6	20 x 25

Mínimo número

1	2	3	4	5	6	
0	15,750	7,875	9,375	11,875	15,125	1
	0	2,625	4,375	7,125	10,500	2
		0	750	2,500	5,375	3
			0	1,000	3,500	4
				0	5,000	5
					0	6

Posición

- Esta tabla muestra las posiciones donde debe de factorizarse.

1	2	3	4	5	6	
*	1	1	3	3	3	1
	*	2	3	3	3	2
		*	3	3	3	3
			*	4	5	4
				*	5	5
					*	6

Complejidad del algoritmo

Si existen n matrices que multiplicar se deben de calcular $n(n - 1)$ términos.

En el peor de los casos el cálculo de cada término requiere n operaciones

El algoritmo es $\boldsymbol{O}(n^3)$

El problema de la mochila

Este problema se puede formular de la siguiente forma:

$$\max p_1 x_1 + p_2 x_2 + \ldots + p_n x_n$$

sujeto a

$$a_1 x_1 + a_2 x_2 + \ldots + a_n x_n \leq B$$

$$x_i = 0 \text{ ó } 1$$

Ejemplo

$$\max 16x_1 + 19x_2 + 23x_3 + 28x_4$$

sujeto a

$$2x_1 + 3x_2 + 4x_3 + 5x_4 \leq 7$$

$$x_i = 0 \text{ ó } 1$$

Caso continuo

- El caso continuo se puede resolver fácilmente, se evalúa el cociente, beneficio/peso y se ordena decrecientemente.

- p_i / a_i
- 8, 6.3, 5.75, 5.6
- x_1 =1, x_2 =1, x_3 =0.5
- con objetivo 46.5

Programación Dinámica

Definamos

$z_k(d)$ = la mejor solución al problema, si tenemos una mochila de capacidad d y consideramos únicamente los objetos 1,2,…,k.

Así que $z_n(B)$ es la solución óptima para el problema original.

Función Recursiva

Procedemos a calcular recursivamente $z_n(d)$ a partir de z_{n-1}, la cual a su vez se calcula de z_{n-2}, y así sucesivamente.

La recursión se inicializa con

$z_1(d) = p_1$ si $a_1 \leq d$

0 en otro caso

Función recursiva

$$z_k(d) = \begin{cases} z_{k-1}(d) & \text{si } a_k > d \\ \max\{z_{k-1}(d),\ p_k + z_{k-1}(d - a_k)\} & \text{si } a_k \leq d \end{cases}$$

Número de operaciones

Ejemplo

$$\max 16x_1 + 19x_2 + 23x_3 + 28x_4$$

sujeto a

$$2x_1 + 3x_2 + 4x_3 + 5x_4 \leq 7$$

$$x_i = 0 \text{ ó } 1$$

Tabla

Resultados de la programación dinámica

	1	2	3	4	5	6	7
1	0	16	16	16	16	16	16
2	0	16	19	19	35	35	35
3	0	16	19	23	35	39	42
4	0	16	19	23	35	39	44

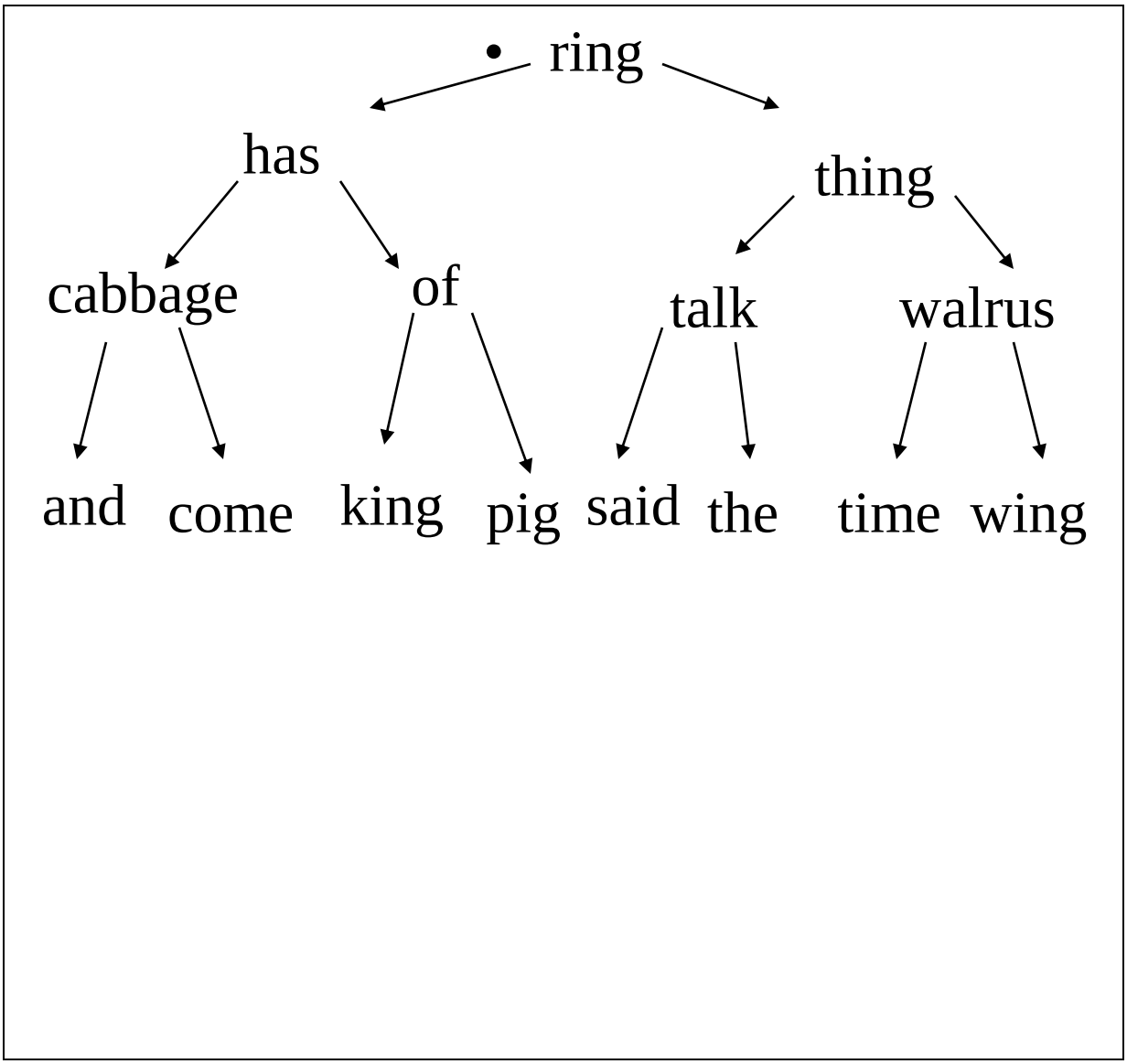
Arboles binarios óptimos

- Cómo construir un árbol binario de búsqueda para minimizar el tiempo de búsqueda promedio.

- Condición: Conocemos que algunos valores se buscan con más frecuencia que otros.

Arboles de búsqueda

- El valor almacenado en cada nodo es mayor que todos los valores en su subárbol izquierdo y menor que todos los valores en su subárbol derecho.
- Para buscar un valor particular, comenzamos en la raíz y seguimos la rama izquierda o la rama derecha dependiendo si el valor buscado es menor o mayor que el actual.


ring
has
thing
cabbage
of
talk
walrus
and
come
king
pig
said
the
time
wing

- Número de comparaciones requeridas para encontrar un valor es uno más el nivel del nodo en el árbol

- Si el árbol contiene n nodos, y está completamente balanceado, entonces en el peor de los casos la búsqueda es $O(\log n)$, si es arbitraria, puede ser $O(n)$.

- Valores buscados son $K_1, K_2 , \ldots K_n$
- La probabilidad de buscar cada uno es respectivamente $p_1, p_2 , \ldots p_n$
- c_i es el número de comparaciones necesarias para encontrar K_i. El promedio de nodos examinados es
- $A(T) = \sum_{i=1,\ldots,n} p_i c_i$

and	30	.15	4	.6
cabbage	5	.025	3	.075
come	10	.05	4	.2
has	5	.025	2	.05
king	10	.05	4	.2
of	25	.125	3	.375
pig	5	.025	4	.1
ring	15	.075	1	.075
said	15	.075	4	.3
talk	10	.05	3	.15
the	30	.15	4	.6
thing	15	.075	2	.15
time	10	.05	4	.1
walrus	5	.025	3	.075
wing	10	.05	4	.2
Total	200	1.0		3.25

Solución recursiva

- $A[i,j]$ = mínimo tiempo promedio para un árbol binario de búsqueda con valores $K_i < ... < K_j$

$A[i,j] =$

$$\min_{i \le k \le j} [A[i, k\text{-}1] + \Sigma_{q=i,..,k\text{-}1} p_q + A[k+1, j] + \Sigma_{q=k+1 \ldots j} p_q + p_k$$

$$= \min_{i \le k \le j} [A[i, k\text{-}1] + A[k+1,j]] + \Sigma_{q=i \ldots j} p_q$$

Cotas sobre algunos algoritmos de ordenamiento

Ordenamiento por inserción

Despúes de cada comparación, o bien no se mueve ningún elemento, o simplemente se intercambian dos elementos adyacentes.

- Todos los algoritmos que se basan en movimiento local después de cada comparación requieren el mismo tiempo.

Permutaciones

- Una permutación π de n objetos es una función 1-1 de $N = \{1,2,\ldots,n\}$ en sí mismo.

- Podemos considerar que el input para un algoritmo de ordenamiento es $\pi(1), \pi(2),\ldots, \pi(n)$.

Inversiones

- Un par ($\pi(i)$, $\pi(j)$) tal que $i < j$ y $\pi(i) > \pi(j)$ se llama *inversión*.
- Si existe una inversión ($\pi(i)$, $\pi(j)$), los elementos i-ésimo y j -ésimo de la lista se encuentran fuera de orden uno con respecto del otro.
- ej. 5,3,1,2,4 contiene 6 inversiones: (5,3), (5,1), (5,2), (5,4), (3,1), (3,2)

Inversiones

- Ordenar significa eliminar las inversiones presentes.

- Si un algoritmo elimina a lo más una inversión después de cada comparación (ej. inserción) entonces el número de comparaciones es al menos igual al número de inversiones.

Peor de los casos

- Existe una permutación que contiene exactamente $n(n\text{-}1)/2$ inversiones.

- El peor de los casos de cualquier algoritmo que elimine a lo más una inversión después de cada comparación debe de estar en $\mathbf{\Omega}\ (n^2)$.

Caso promedio

- Calculemos el número promedio de inversiones que hay en una permutación.

- Cada permutación puede aparearse con su *transpuesta* $\pi(n)$, $\pi(n\text{-}1),\ldots,$ $\pi(1)$. Sea π' la transpuesta de π.
- ej. La transpuesta de 5,3,1,2,4 es 4,2,1,3,5.

Inversiones en la transpuesta

- Si $j < i$ entonces (i,j) es una inversión exactamente en una de las dos permutaciones π ó π'

- Existen $n(n-1)/2$ pares de enteros. Por lo tanto cada par de permutaciones π, π' comparten $n(n-1)/2$ inversiones entre ellas, y por lo tanto un promedio de $n(n-1)/4$.

Teorema

- Cualquier algoritmo que ordene por comparación de valores y elimina a lo más una inversión después de cada comparación debe de efectuar cuando menos $n(n-1) / 2$ comparaciones en el peor de los casos y al menos $n(n-1) / 4$ comparaciones en promedio.

QUICK SORT

- **Dividir** El arreglo $A[p..r]$ se divide en dos subarreglos $A[p..q]$ y $A[q+1..r]$ de manera que los elementos del primer arreglo sean menores o iguales a los del segundo.
- **Conquistar** Los dos subarreglos se ordenan recursivamente.
- **Combinar** el arreglo ya queda ordenado.

Procedimiento

QUICKSORT(A, p, r)

if $p < r$

then q:= PARTITION(A, p, r)

QUICKSORT(A, p, q)

QUICKSORT(A, q+1, r)

Partición

PARTITION(A, p, r)

$x := A[p]$

$j := p-1$, $k := r+1$

while TRUE

do repeat $k := k - 1$

until $A[k] \leq x$

repeat $j := j + 1$

until $A[j] \geq x$

if $j < k$

then $A[j] \leftrightarrow A[k]$

else return k

5 3 2 6 4 1 3 7

j k

5 3 2 6 4 1 3 **7**

j k

3 3 2 6 4 1 **5 7**

j k

3 3 2 6 4 1 **5 7**

j k

3 3 2 1 4 **6 5 7**

j k

3 3 2 1 4 6 5 7

k j

Peor de los casos

- El peor de los casos ocurre precisamente cuando el arreglo ya está ordenado.
- En este caso se produce una particion de 1 y otra partición de n-1 elementos.

$$T(n) = T(n\text{-}1) + \Theta(n)$$

$$= \sum_{k=1,\ldots,n} \Theta(k) = \Theta(\sum_{k=1,\ldots,n} k)$$

$$= \Theta(n^2)$$

Mejor de los casos

- Partición balanceada
- $T(n) = 2\,T(n/2) + \Theta(n)$
- $T(n) = \Theta(n \log n)$

Versión Aleatorizada

- Un algoritmo es aleatorio cuando su comportamiento está determinado no sólo por los datos de entrada, sino también por los valores producidos por un generador de números aleatorios.

- *Ninguna instancia en particular presenta su caso peor*.

Quicksort Aleatorizado

RAND_PARTITION(A,p,r)

j := RANDOM(p,r)

intercambiar $A[j] \leftrightarrow A[p]$

return PARTITION(A,p,r)

RAND_QUICKSORT(A, p, r)

if $p < r$

then q:= RAND_ PARTITION(A, p, r)

RAND_ QUICKSORT(A, p, q)

RAND_ QUICKSORT(A, q+1, r)

Análisis de la partición

El *rango* de un elemento en un conjunto es el número de elementos menores o iguales a él.

El valor de q devuelto por el procedimiento depende únicamente del rango de $x = A[p]$.

La probabilidad de que rango(x) = j es $1/n$ para $j = 1,\ldots,n$.

Si rango(x) = 1

la parte “menor” de la partición termina con un sólo elemento.

Este evento tiene probabilidad $1/n$.

Si rango(x) ≥ 2

la parte “menor” termina con j elementos, para cada $j = 1,2, \ldots, n-1$.

Cada uno de estos eventos tiene probabilidad $1/n$.

Recurrencia

$$T(n) = 1/n \left(T(1) + T(n\text{-}1) + \sum_{q = 1,\ldots,n\text{-}1} (T(q) + T(n\text{-}q) \right) + \Theta(n)$$

$$1/n \left(T(1) + T(n\text{-}1) \right) = 1/n \left(\Theta(1) + O(n^2) \right) = O(n)$$

$$T(n) = 1/n \left(\sum_{q = 1,\ldots,n\text{-}1} (T(q) + T(n\text{-}q) \right) + \Theta(n)$$

$$T(n) = 2/n \sum_{q = 1,\ldots,n\text{-}1} T(q) + \Theta(n)$$

Usaremos inducción:

$T(1) \leq b$ es cierto para alguna b

Supondremos que

$T(k) \leq ak \log k + b$ para $k \leq n$ -1

Debemos establecer el resultado para $k = n$.

$T(n) \leq an \log n + b$

$$T(n) = 2/n \sum_{q=1,\ldots,n-1} T(q) + \Theta(n)$$

$$\leq 2/n \sum_{q=1,\ldots,n-1} (a\, q \log q + b) + \Theta(n)$$

$$\leq 2\, a/n \sum_{q=1,\ldots,n-1} q \log q + 2b(n-1)/n + \Theta(n)$$

Usando la identidad

$$\int x \ln x\, dx = \frac{1}{2} x^2 \ln x - \frac{1}{4} x^2$$

deducimos una cota para la sumatoria,

$$\sum_{k=1}^{n-1} k \log k \leq \frac{1}{2} n^2 \log n - \frac{1}{4} n^2$$

con esta cota obtenemos:

$$T(n) \leq 2\, a/n \left(1/2\, n^2 \log n - 1/4\, n^2\right) + 2b(n-1)/n + \Theta(n)$$

$$\leq a\, n \log n - a\, n/2 + 2b + \Theta(n)$$

$$= a\, n \log n + b + \Theta(n) + b - a\, n/2$$

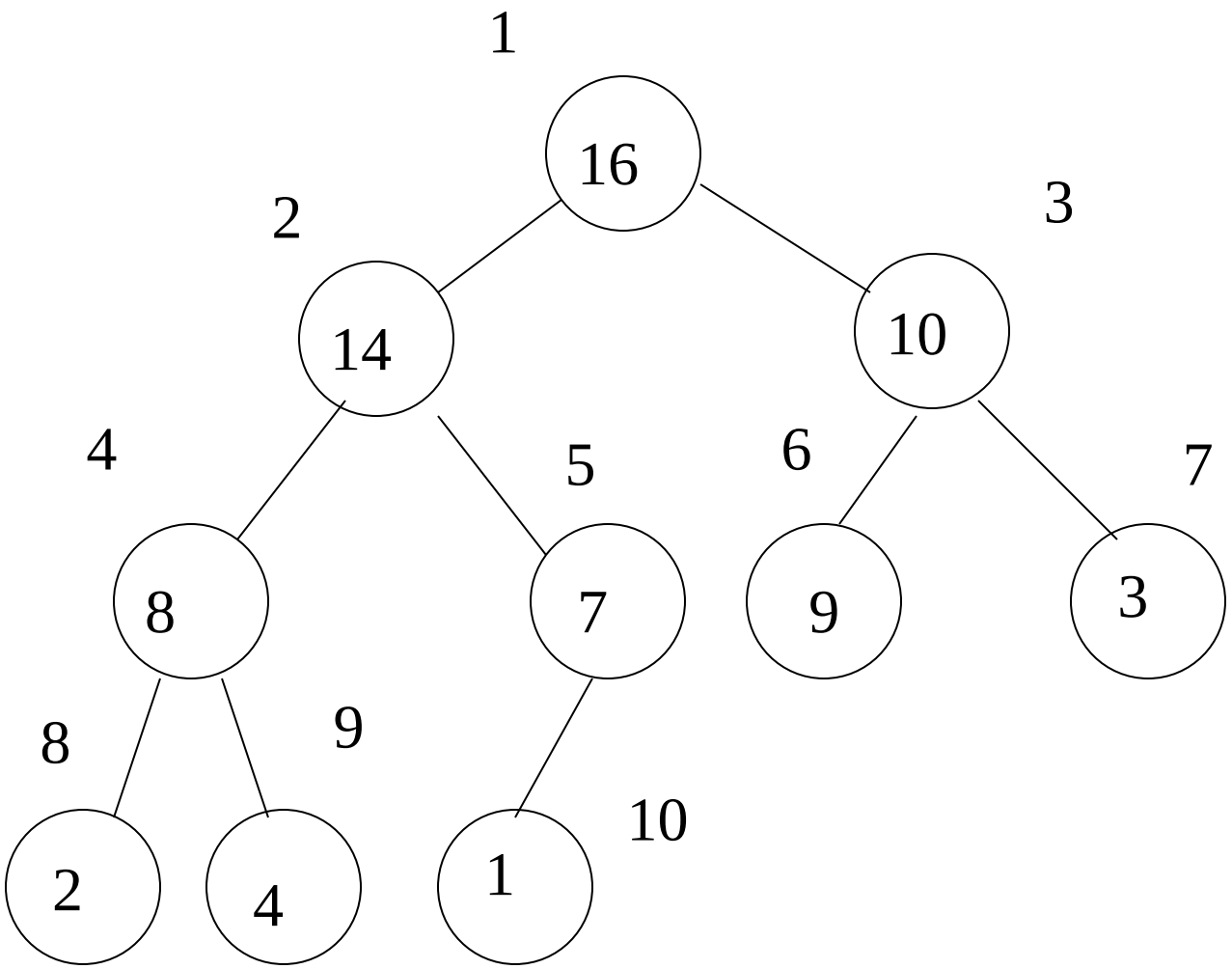
$$\leq a\, n \log n + b$$

Heap Sort

- Un heap es un arreglo que puede visualizarse como un árbol binario. Cada nodo del árbol corresponde a un elemento del arreglo.
- El árbol se encuentra completamente lleno en todos sus niveles excepto quizá en el último, el cual está lleno desde la izquierda hasta un cierto punto.

Propiedad heap

- Para cualquier nodo i distinto de la raíz
- $A[\text{padre}(i)] \geq A[i]$
- El valor almacendado en un nodo es a lo más el valor almacendo en el padre del nodo.



(16,14,10,8,7,9,3,2,4,1)

Heaps

- Atributos:
- *long*[*A*], número de elementos en el arreglo.
- *heap_size*[*A*], número de elementos en el arreglo que se encuentran dentro del heap.
- La raíz del árbol es *A*[1].
- nodo j
- padre $\lfloor j / 2 \rfloor$
- hijo izquierdo $2j$
- hijo derecho $2j + 1$

Propiedades

- $A[\, j \,] \geq A[\,$ hijo$[\, j \,]\,]$

- altura de un nodo: número de aristas en el camino más largo hacia abajo, desde el nodo hasta una hoja.
- altura del árbol: altura de su raíz.
- Para un heap de n elementos su altura es $O(\log n)$.

Procedimientos básicos

- HEAPIFY Mantener la propiedad de heap $O(\log n)$
- BUILD_HEAP $O(n)$ produce un heap a apartir de un arreglo arbitrario
- HEAPSORT $O(n \log n)$ ordena un arreglo
- EXTRACT_MAX $O(\log n)$
- INSERT $O(\log n)$
- colas con prioridad

Heapify

- Input:
- Un arreglo *A*, y un índice *j* dentro del arreglo.
- Precondiciones:
- los árboles binarios con raíz en left(*j*) y en right(*j*) son heaps, pero el valor *A*[*j*] puede ser menor que sus hijos.
- Output:
- heap con raíz en *A*[*j*]

HEAPIFY(A,j)

l := left(j)

r := right(j)

if $l \leq heap\_size[A]$ **and** $A[l] > A[j]$

then $mayor := l$

else $mayor := j$

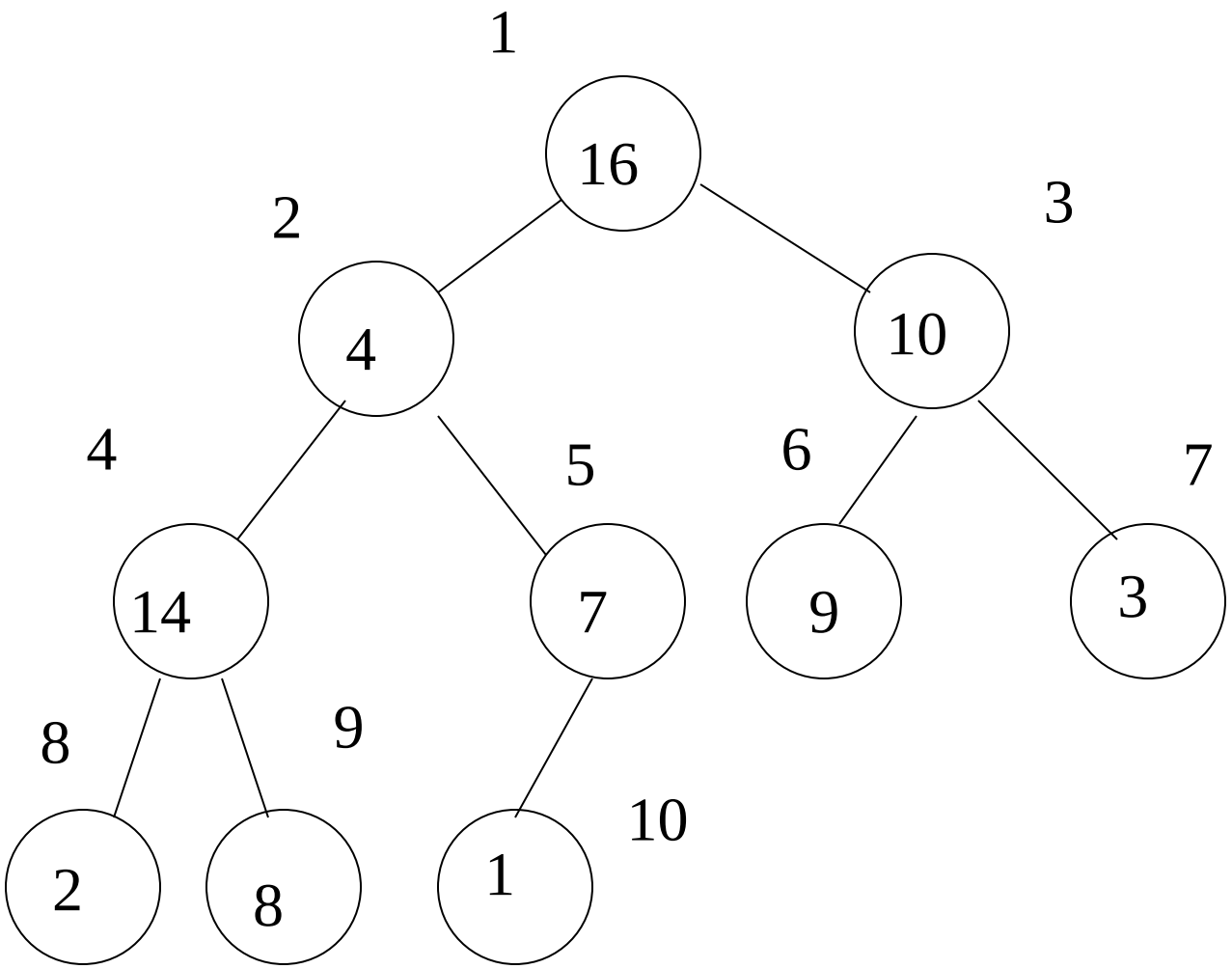
if $r \leq heap\_size[A]$ **and** $A[r] > A[mayor]$

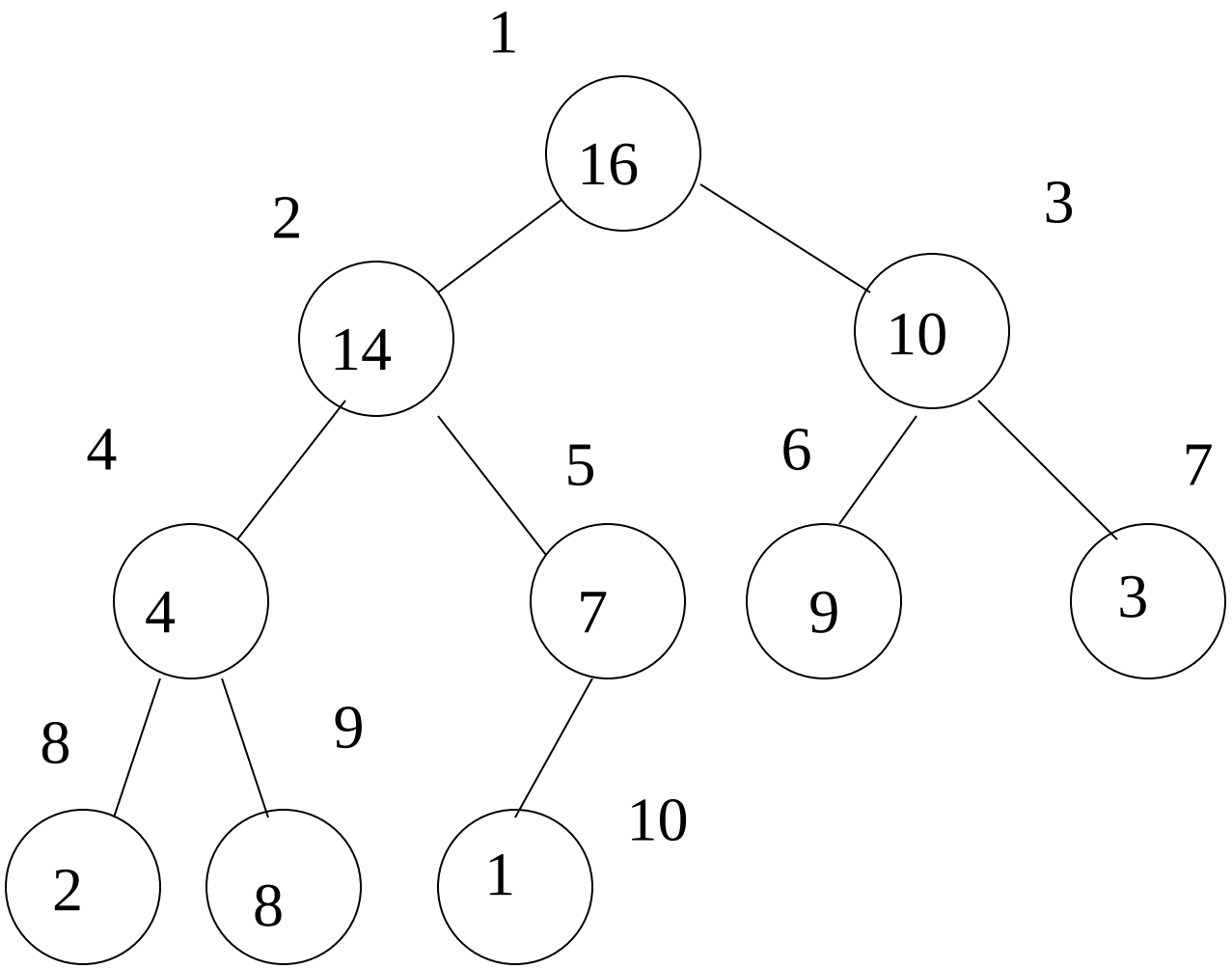
then $mayor := r$

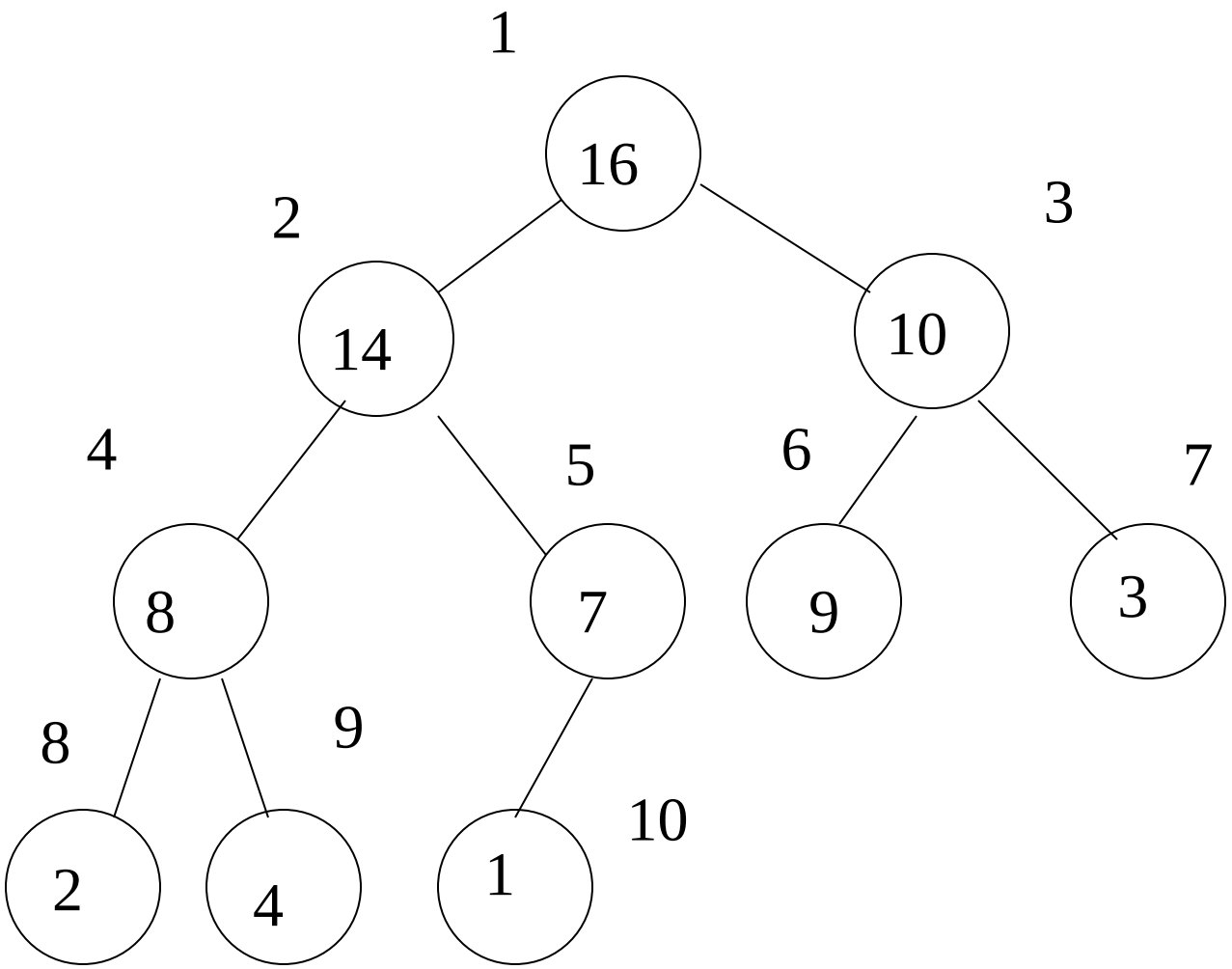
if $mayor \neq j$

then intercambiar $A[mayor] \leftrightarrow A[j]$

HEAPIFY (A,$mayor$)


1
16
2
3
4
10
4
5
6
7
14
7
9
3
8
9
10
2
8
1


1
16
2
3
14
10
4
5
6
7
4
7
9
3
8
9
10
2
8
1


1
16
2
3
14
10
4
5
6
7
8
7
9
3
8
9
10
2
4
1

BUILD_HEAP(A)

input: arreglo $A[1,2,\ldots,n]$

$n = long[A]$

output : heap $A[1,2,\ldots,n]$

$heap\_size[A] := long[A]$

for $j := \lfloor long[A]/2 \rfloor$ **downto** 1

do HEAPIFY(A,j)

A = (4,1,3,2,16,9,10,14,8,7)

- En un heap binario completo de n nodos, existen $n/2$ nodos que son hojas (altura $h = 0$)
- $n/4$ nodos con altura 1
- $n/8$ nodos con altura 2 , …
- En general, existen a lo más
- $\lceil n / 2^{h+1} \rceil$ nodos de altura h.

$$\sum_{h=0}^{\lg n} \left\lceil \frac{n}{2^{h+1}} \right\rceil h \leq \sum_{h=0}^{\lg n} \frac{n}{2^h} h = n \sum_{h=0}^{\lg n} \frac{h}{2^h}$$

HEAPSORT(A)

input: arreglo $A[1,2,\ldots,n]$

output: arreglo A ordenado

BUILD_HEAP(A)

for j = *long*[A] **downto** 2

do intercambiar $A[1] \leftrightarrow A[j]$

heap_size[A] = *heap_size*[A]-1

HEAPIFY(A,1)

Colas con prioridad

- Estructura de datos para mantener un conjunto de S elementos cada uno con un valor asociado.
- INSERT(S,x) inserta el elemento x en el conjunto S
- MAX(S) regresa el elemento de S con el máximo valor
- EXTRACT_MAX(S) elimina y regresa el elemento de S con el máximo valor

EXTRACT_MAX(*A*)

input: heap *A*
if *heap_size*[*A*] < 1
  then error
max := *A*[1]
A[1] := *A*[*heap_size*[*A*]]
heap_size[*A*] := *heap_size*[*A*] **-** 1
HEAPIFY(*A*,1)
return *max*

HEAP_INSERT(*A*,*valor*)

input: heap *A*

heap_size[*A*] := *heap_size*[*A*] +1
j := *heap_size*[*A*]
while *j* > 1 **and** *A*[padre(*j*)]< *valor*
    do *A*[*j*] := *A*[padre(*j*)]
        j := padre(*j*)
A[*j*] := *valor*

Lectura

- Thanks, heaps.
- *Jon Bentley*

- **Communications of the ACM**

- March 1985 vol.28 Number 3

Medianas y estadísticas de orden

- La *i*-ésima estadística de orden de un conjunto de *n* elementos es el *i*-ésimo menor elemento.
- **Ejemplo** El mínimo del conjunto es la primer estadística de orden, y el máximo es la *n*-ésima estadística de orden. La mediana se encuentra en la posición $i = \lfloor (n+1)/2 \rfloor$ y $\lceil (n+1)/2 \rceil$

Problema de selección

Seleccionar la *i*-ésima estadística de orden en un conjunto de *n* elementos.

Input: Un conjunto *A* de *n* elementos distintos y un número *i*, entre 1 y *n*.

Output: El elemento *x* que es mayor que exactamente *i*-1 elementos de *A*.

Observemos que el problema se puede resolver en tiempo $O(n \log n)$.

Existen algoritmos más eficientes.

Mínimo y máximo

- ¿Cuántas comparaciones ese requieren para determinar el mínimo de un conjunto de *n* elementos?
- ¿Cuántas comparaciones ese requieren para determinar el máximo de un conjunto de *n* elementos?

Máximo y mínimo simultáneos

- $2n - 2$ comparaciones.

Selección en tiempo lineal promedio

- Nuevamente introduciendo aleatoreidad, se puede encontrar un algoritmo que funciona eficientemente en promedio.
- El procedimiento RAND_SELECT usa RAND_QUICK_SORT la diferencia se encuentra en que éste procesa ambos lados de la partición, aquel procesa sólo uno.

Selección Aleatorizada

RAND_SELECT *(A,p,r,i)*

if $p = r$

  then return $A[p]$

q:= RAND_ PARTITION(A, p, r)

k:= $q - p + 1$

if $i \leq k$

  then return

  RAND_ SELECT(A, p, q, i)

  else return

  RAND_ SELECT(A, q+1, r, i-k)

Peor Caso

- El peor caso es $\Theta(n^2)$, aún para encontrar el mínimo, porque podríamos tener tan mala suerte que siempre particionemos con respecto al mayor elemento remanente.
- El algoritmo trabaja bien en el caso promedio, y ya que está aleatoreizado ningún input en particular presenta el peor caso.

Relación de recurrencia

$$T(n) \le \frac{1}{n}\left(T(\max(1, n-1)) + \sum_{k=1}^{n-1} T(\max(k, n-k))\right) + O(n)$$

$$\le \frac{1}{n}\left(T(n-1) + 2\sum_{k=\lceil n/2 \rceil}^{n-1} T(k)\right) + O(n)$$

$$\le \frac{2}{n}\sum_{k=\lceil n/2 \rceil}^{n-1} T(k) + O(n)$$

- Supongamos que $T(n) \le cn$ para alguna c.

$$T(n) \le \frac{2}{n} \sum_{k=\lceil n/2 \rceil}^{n-1} ck + O(n)$$

$$\le \frac{2c}{n} \left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k \right) + O(n)$$

$$= \frac{2c}{n} \left(\frac{1}{2}(n-1)n - \frac{1}{2}\left(\left\lceil \frac{n}{2} \right\rceil - 1\right)\left\lceil \frac{n}{2} \right\rceil \right) + O(n)$$

$$\le c(n-1) - \frac{c}{n}\left(\frac{n}{2} - 1\right)\left(\frac{n}{2}\right) + O(n)$$

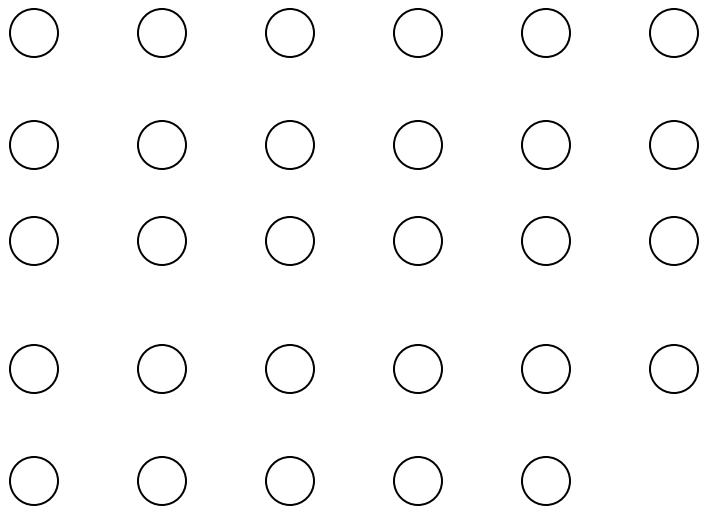
$$= c\left(\frac{3}{4}n - \frac{1}{2}\right) + O(n) \le cn$$

Selección en tiempo lineal para el peor caso

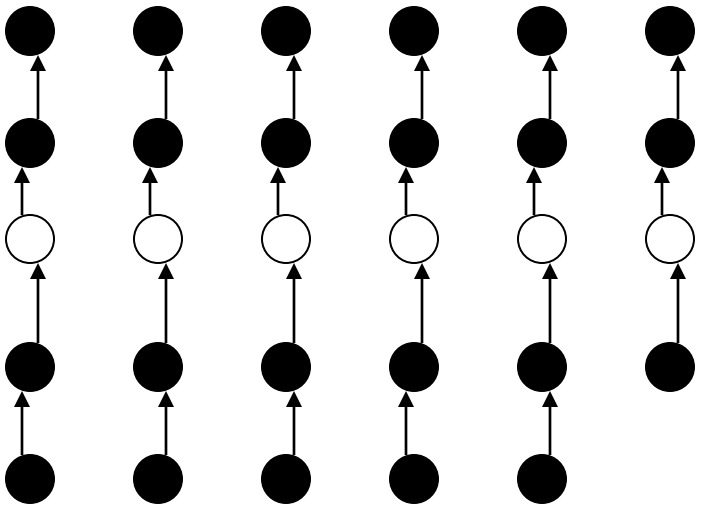
- La estrategia es similar: encontrar el elemento deseado mediante particiones recursivas.
- En cambio, la idea es *garantizar* una buena partición.
- Se modifica la versión determinística de PARTITION

Pasos

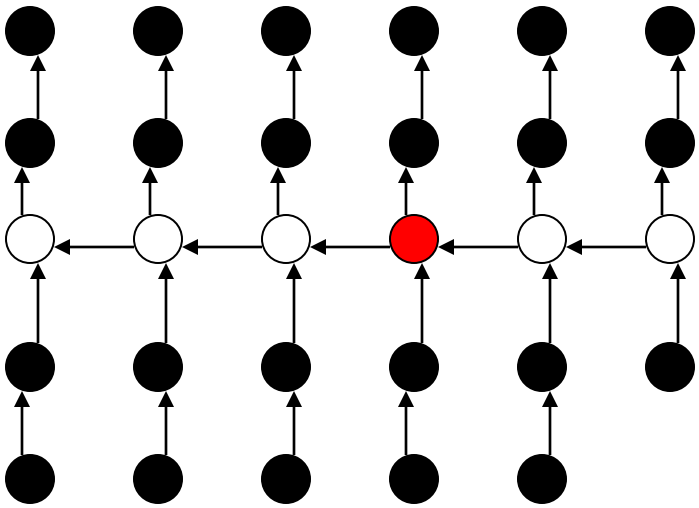
- 1. Dividir los n elementos en $\lfloor n/5 \rfloor$ grupos de 5 elementos cada uno, y a lo más un grupo adicional con los elementos restantes. $n = 29$



- 2. Encontrar la mediana de cada uno de los $\lceil n/5 \rceil$ grupos cada uno ordenado por inserción y tomando el elemento medio. (Si el grupo tiene un número par de elementos tomar la mayor de las dos medianas).



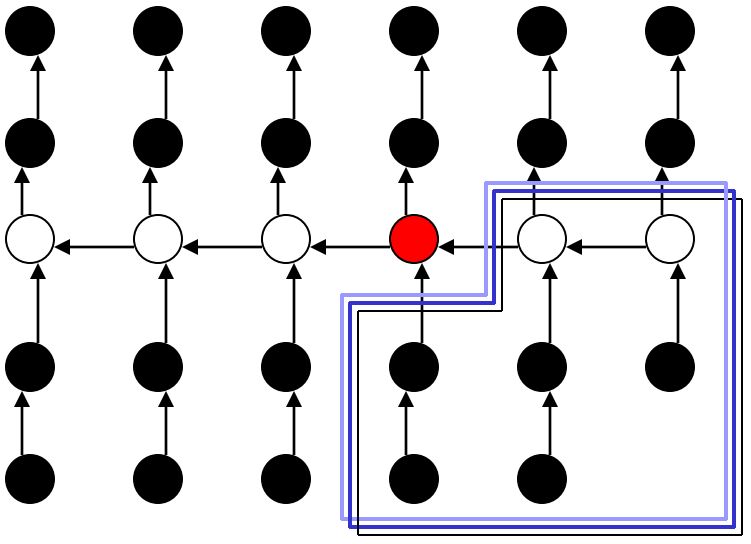
- 3. Usar SELECT recursivamente para encontrar la mediana x del conjunto de $\lceil n/5 \rceil$ medianas encontradas en el paso 2.



- 4. Dividir el arreglo de entrada con elemento pivote x, la mediana de las medianas usando PARTITION. Sea k el número de elementos en la parte inferior de la partición, de manera que $n - k$ es el número de elementos en la parte superior.
- 5. Usar SELECT recursivamente para encontrar el i-ésimo menor elemento en el lado inferior si $i \leq k$, o el $(i\text{-}k)$-ésimo menor elemento en el lado superior si $i > k$.

Análisis

- Cota inferior sobre el número de elementos mayores que el pivote x.



- Cota inferior sobre el número de elementos mayores que el pivote x.

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5}\right\rceil\right\rceil - 2\right) \geq \frac{3n}{10} - 6$$

- Similarmente el número de elementos que son menores que x es al menos $3n/10 - 6$.
- En el peor de los casos SELECT se aplica recursivamente en un conjunto de a lo más $7n/10 + 6$ elementos en el paso 5.

Relación de recurrencia

- Pasos 1, 2 y 4 $O(n)$
- Paso 3 $T(\lceil n/5 \rceil)$
- Paso 5 a lo más $T(7n/10+6)$

- $T(n) \leq \Theta(1)$
- $T(n) \leq T(\lceil n/5 \rceil) + T(7n/10+6) + O(n)$

El tiempo es lineal

- Por inducción, supongamos que $T(n) \leq cn$ para alguna c y $n \leq 80$.
- $T(n) \leq c\lceil n/5 \rceil + c(7n/10+6) + O(n)$
- $\leq cn/5 + c + 7cn/10 + 6c + O(n)$
- $\leq 9cn/10 + 7c + O(n)$
- $\leq cn$

GRAFOS

- **Definición Formal**
- Un grafo G, es un par (V, E) donde V es un conjunto finito no vacío cuyos elementos son llamados vértices, y E es un conjunto de subconjuntos de V de cardinalidad dos, sus elementos son llamados aristas

Digrafos

- Un digrafo G, es un par (V,E) donde V es un conjunto finito no vacío cuyos elementos son llamados vértices, y $E \subseteq V \times V$ sus elementos son llamados arcos.
- **Convención**
- $| V | = n$
- $| E | = m$

Representaciones

- **Matricial**
- **Adyacencia**
- supondremos que los vértices están numerados 1,2,...,n.
- $A = (a_{ij})$, donde
- $a_{ij} = 1$ si $(i,j) \in E$
- $a_{ij} = 0$ en otro caso.

- **Matricial**
- **Incidencia**
- supondremos que los vértices están numerados 1,2,...,n y también las aristas 1,2,...,m.
- $B = (b_{ij})$, n x m donde
- $b_{ij} = 1$ si la arista j es incidente al vértice i.
- $b_{ij} = 0$ en otro caso.

- **Listas de adyacencia**

- $1 \rightarrow 2 \rightarrow 5$
- $2 \rightarrow 1 \rightarrow 5 \rightarrow 3 \rightarrow 4$
- $3 \rightarrow 2 \rightarrow 4$
- $4 \rightarrow 2 \rightarrow 5 \rightarrow 3$
- $5 \rightarrow 4 \rightarrow 1 \rightarrow 2$

Definiciones

- Si (v_1,v_2) es una arista en E, decimos que los vértices v_1 y v_2 son adyacentes y que la arista (v_1,v_2) es incidente sobre los vértices v_1 y v_2.

- El grafo $G' = (V', E')$ es un subrafo de $G = (V, E)$ si, $V' \subseteq V$ y $E' \subseteq E$.

Definiciones

- Una trayectoria del vértice v_p al vértice v_q en G, es una sucesión de vértices v_p, v_i, v_j,… v_k, v_q, tal que (v_p, v_i), (v_i, v_j),…, (v_k, v_q) son aristas en E.
- Una trayectoria simple es una trayectoria donde todos los vértices, excepto quizá el primero y el último son distintos.

Definiciones

- Un *ciclo*, es una trayectoria simple donde el primero y el último vértice son los mismos.

- Diremos que un grafo G es *conexo*, si para cada par de vértices distintos v_i, v_j en V, existe una trayectoria de v_i a v_j
- Una *componente conexa* de G es un subgrafo maximal conexo.

Definiciones

- Un *árbol* es un grafo conexo y *acíclico* (sin ciclos).
- Las siguientes afirmaciones son equivalentes.

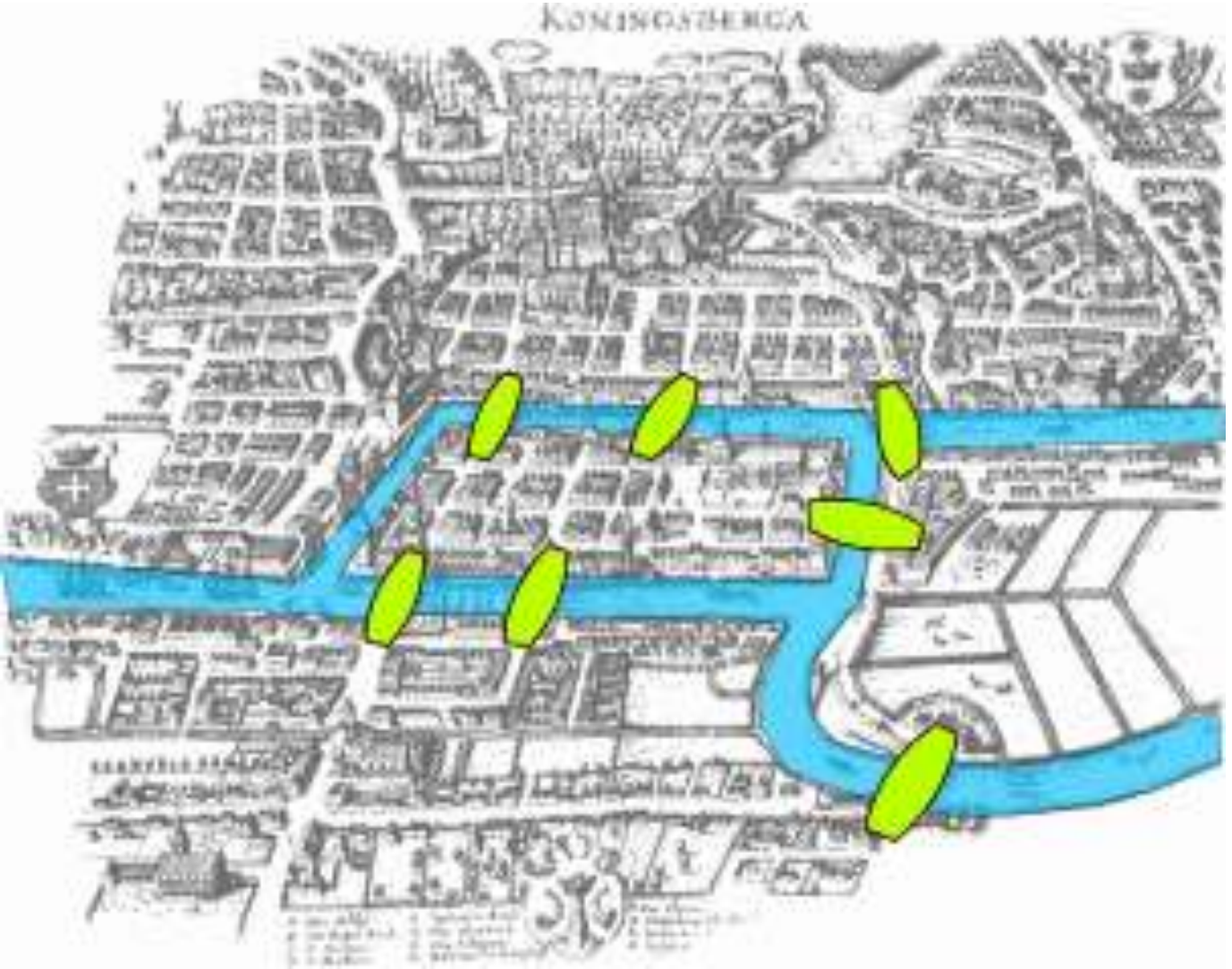
1) *G* es un árbol.
2) Cada par de vértices de *G* están unidos por una única trayectoria
3) *G* es conexo y tiene *n* -1 aristas
4) *G* es acíclico y tiene *n* -1 aristas

PROBLEMAS EN GRAFOS

- Ciclo Euleriano

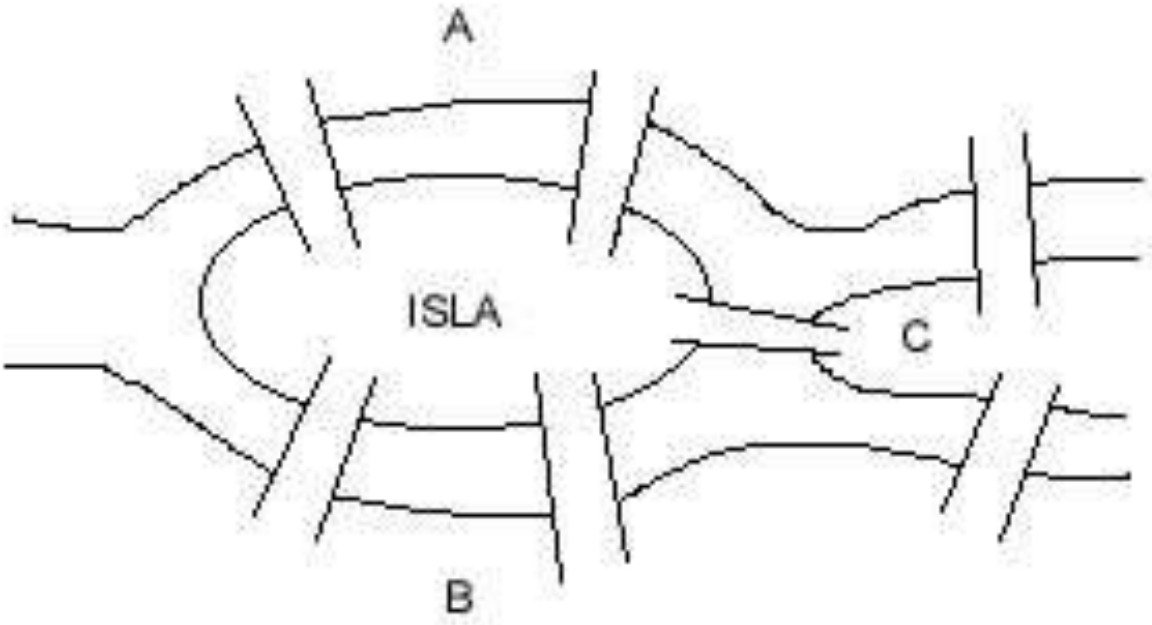
- Instancia: Grafo $G = (V,E)$

- Pregunta: ¿Es posible recorrer todas aristas del grafo exactamente una vez, iniciando el recorrdio en un vértice y terminando en el mismo?


KONINGSBERGA

Puentes de Königsberg

- Río Pregel



ALFAGUARA

Los puentes de Königsberg

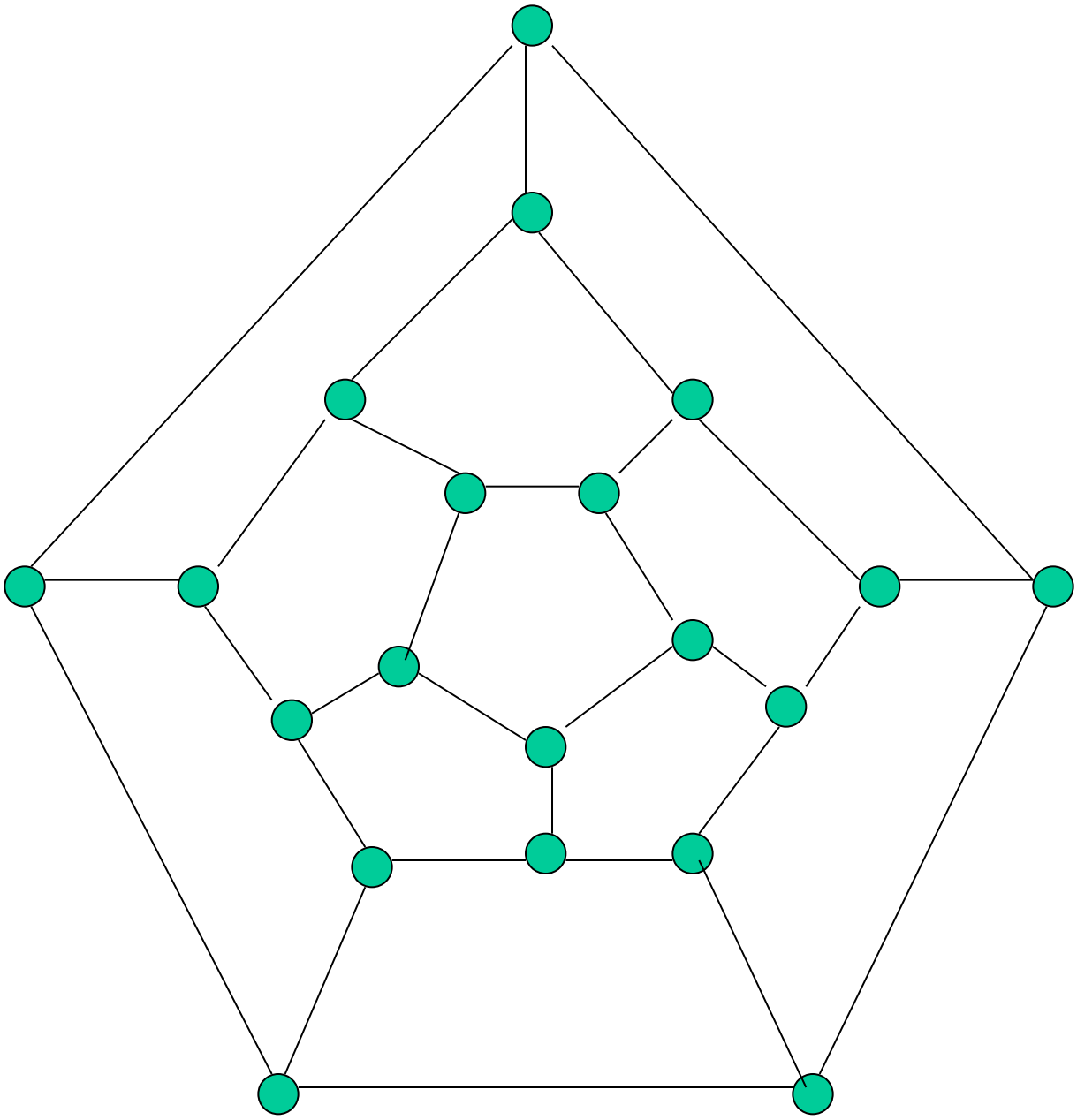
David Toscana

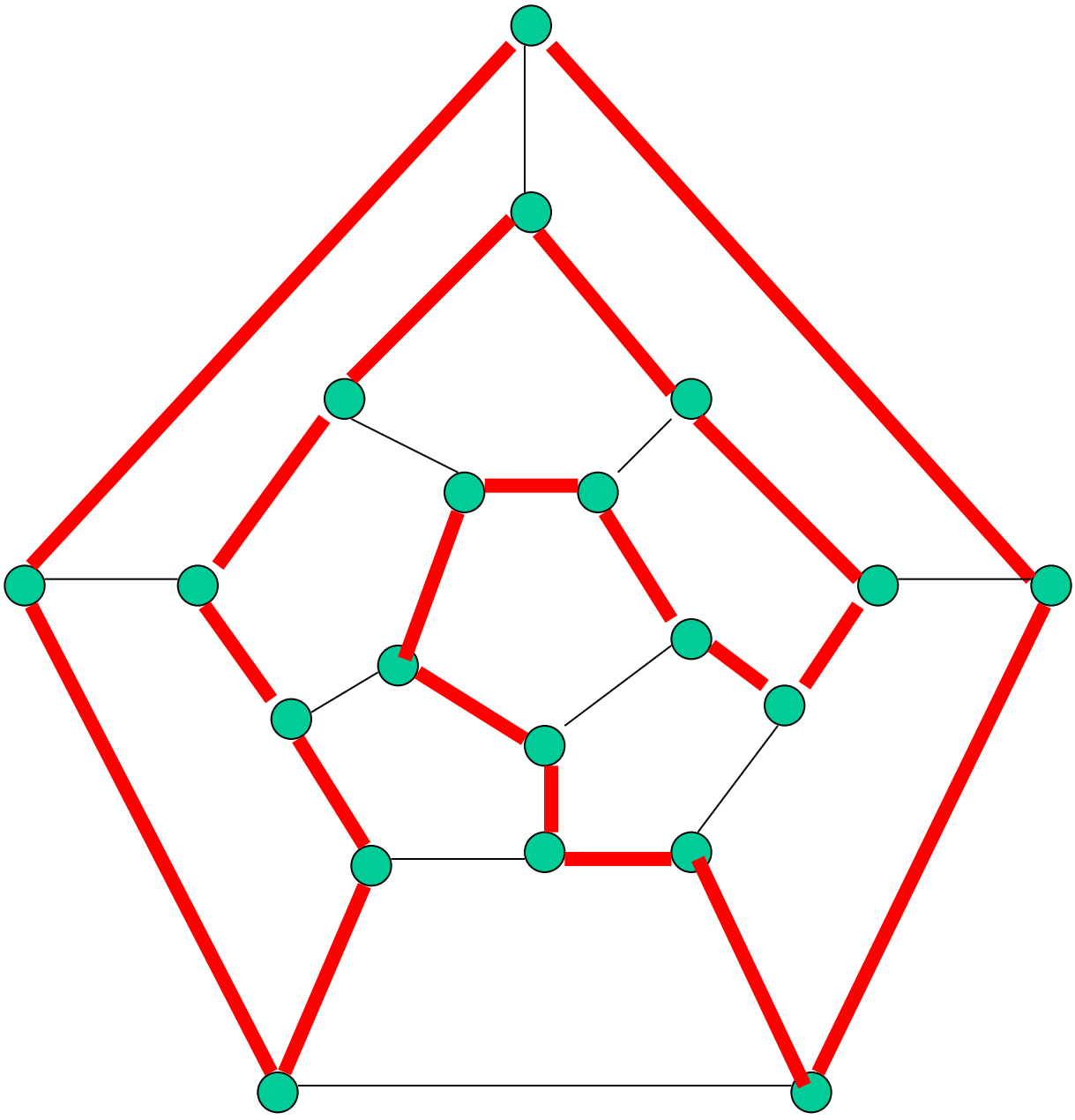
PROBLEMAS EN GRAFOS

- Ciclo Hamiltoniano

- Instancia: Grafo $G = (V,E)$

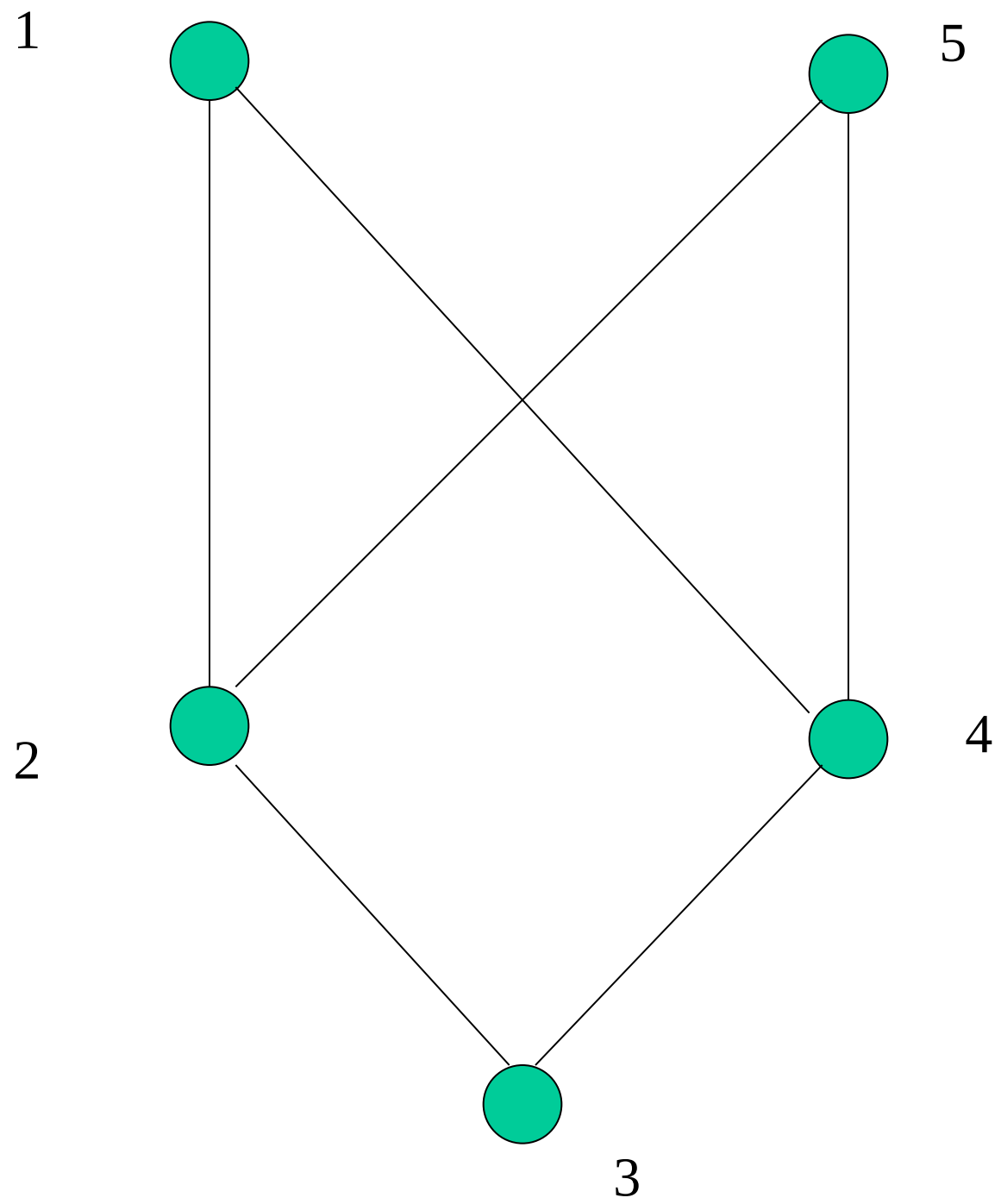
- Pregunta: ¿Es posible visitar todos los vértices del grafo exactamente una vez, partiendo de un vértice y regresando al mismo?

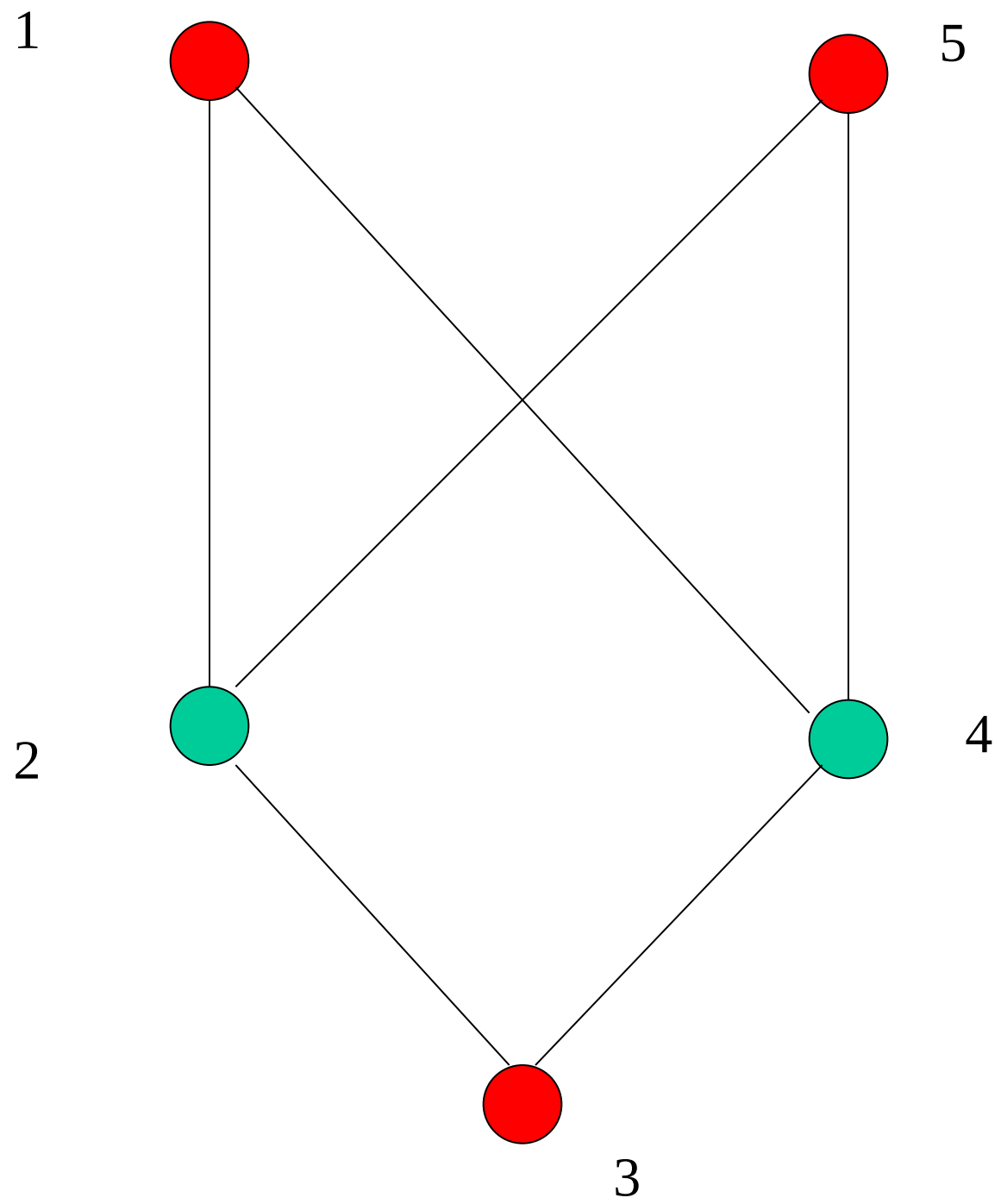


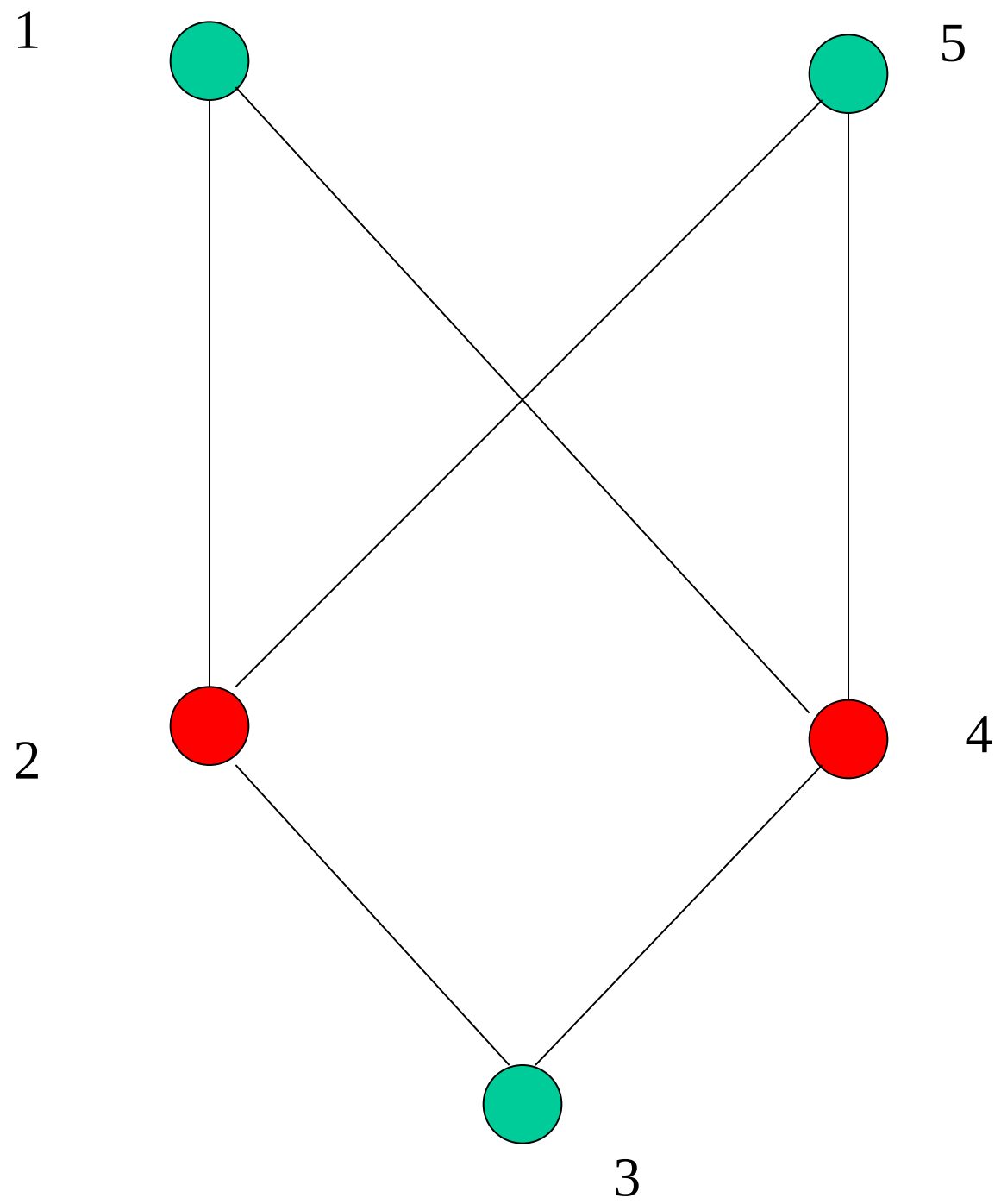


PROBLEMAS EN GRAFOS

- Cubierta por vértices mínima
- Instancia: $G = (V, E)$
- Una cubierta por vértices es un conjunto $V' \subseteq V$ tal que para cada arista (u,v) al menos uno de los dos u ó v pertenece a V'.
- Pregunta: ¿Cuál es la cubierta por vértices de menor tamaño?

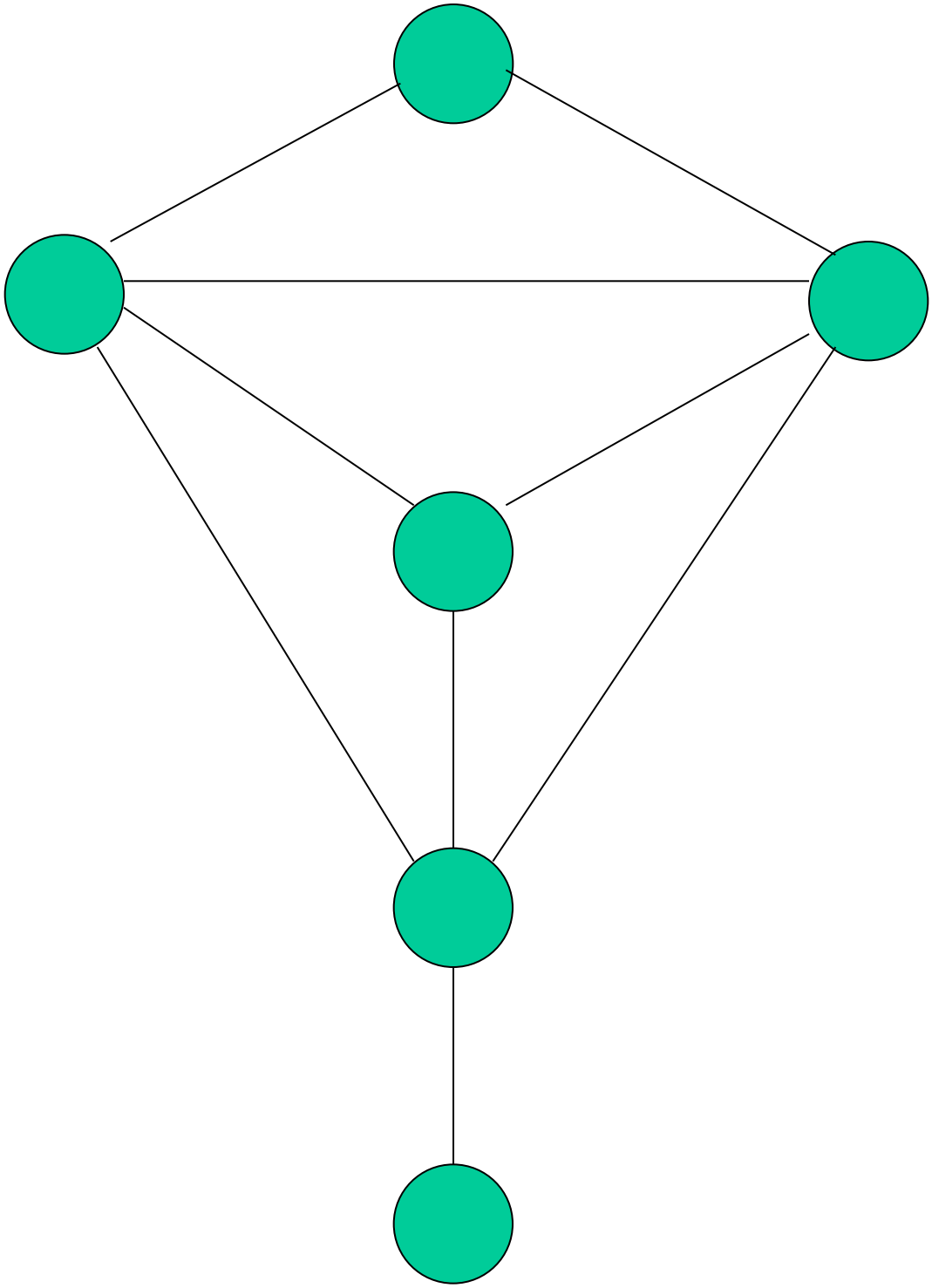

1
5
2
4
3


1
5
2
4
3


1
5
2
4
3

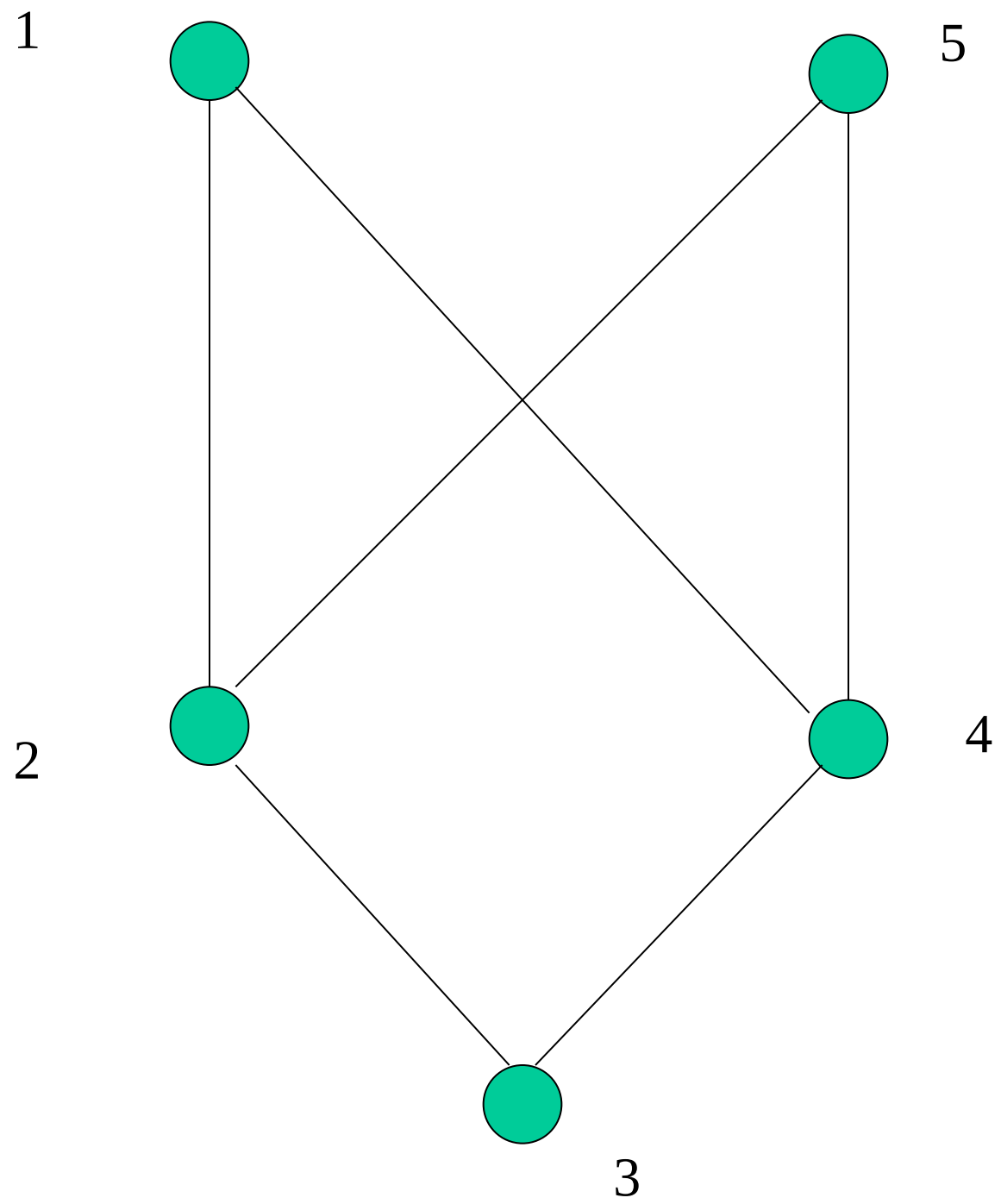
PROBLEMAS EN GRAFOS

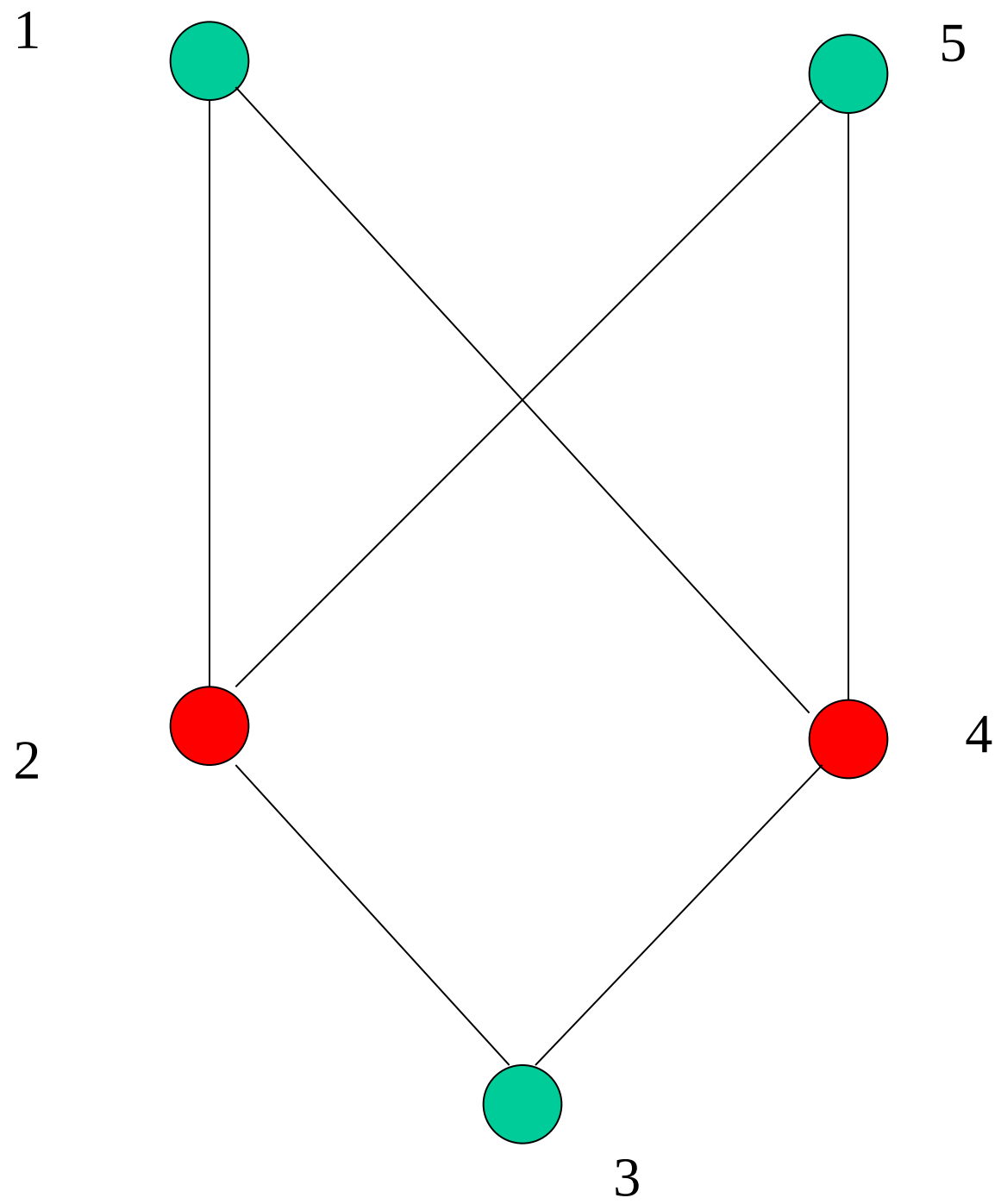
- Máximo clique
- Instancia: $G = (V,E)$
- Un *clique* es un subconjunto V' de V tal que cualesquier dos vértices de V' están unidos por una arista en E.
- Pregunta: ¿Cuál es el clique de mayor tamaño en G?

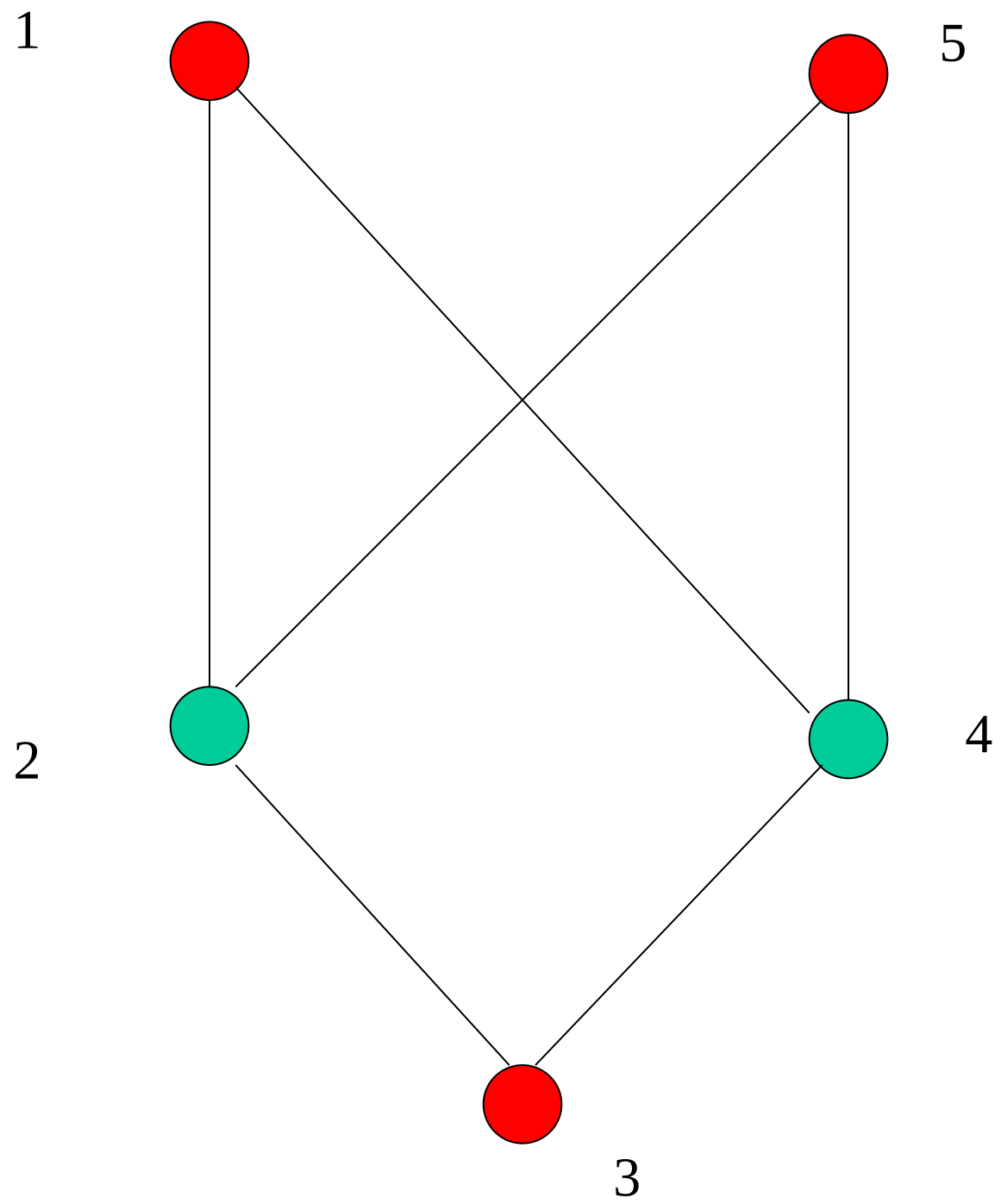


PROBLEMAS EN GRAFOS

- Conjunto independiente
- Instancia $G = (V, E)$
- Un conjunto independiente es un subconjunto $V' \subseteq V$ tal que, si u y v están en V' , la arista (u,v) no está en E.
- Pregunta: ¿Cuál es el máximo conjunto independiente en G?


1
5
2
4
3


1
5
2
4
3

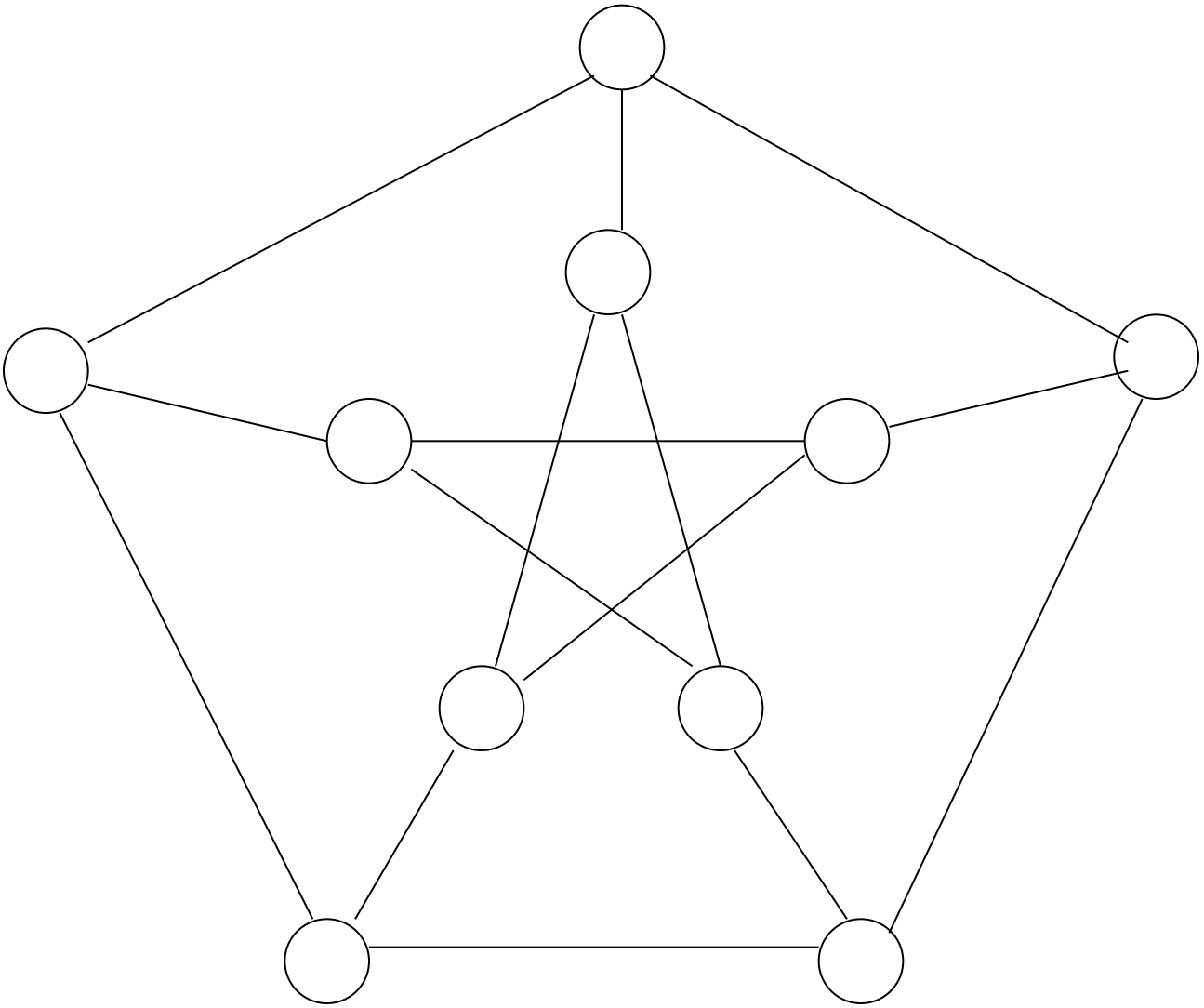

1
5
2
4
3

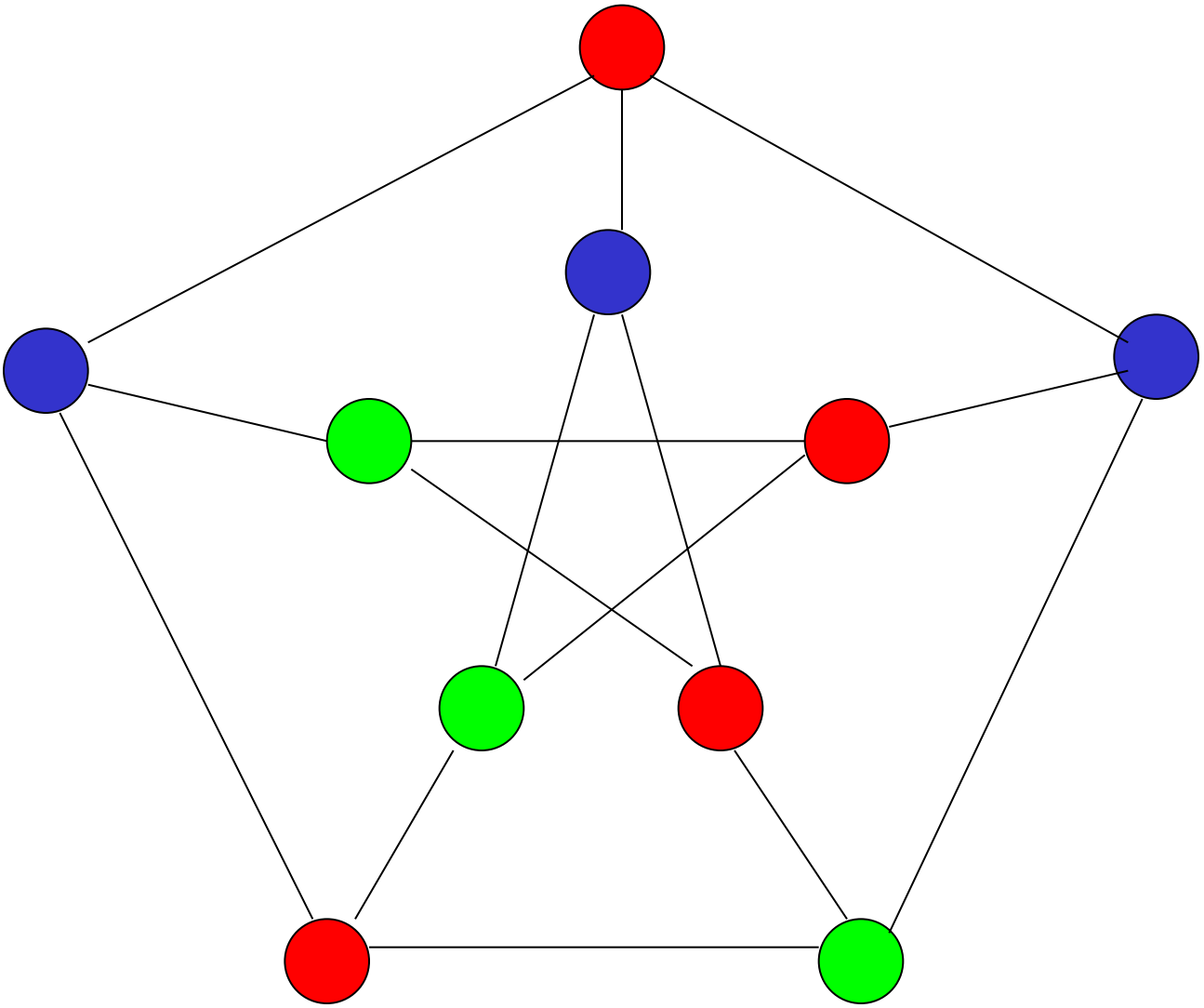
PROBLEMAS EN GRAFOS

- Coloreo de grafos
- Instancia $G = (V, E)$
- Un coloreo de G es un mapeo

  $$c: V \rightarrow \mathbf{N}$$

  tal que si u es adyacente a v , entonces $c(u) \neq c(v)$
- Pregunta: ¿Cuál es el coloreo en G tal que $|c(V)|$ es mínimo?





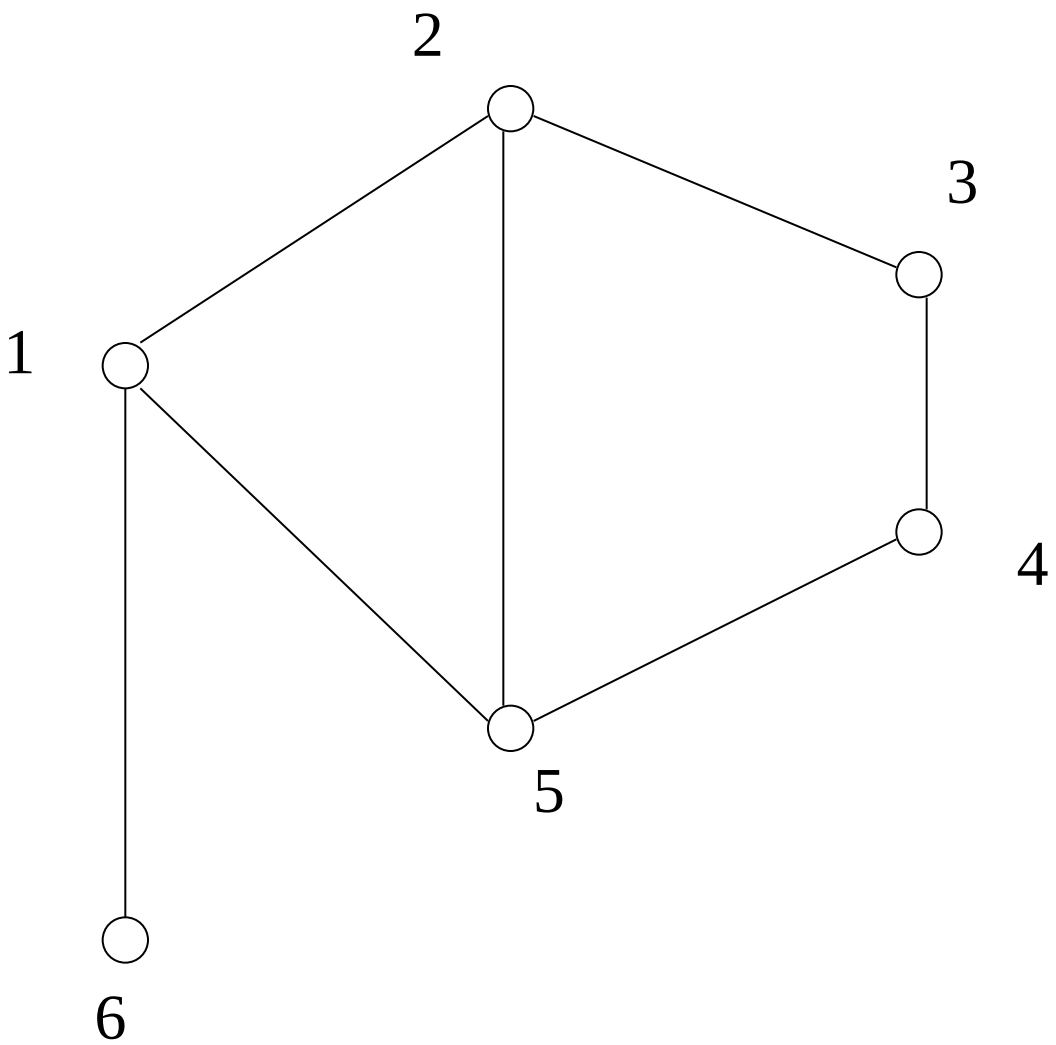
Michael Trick

http://mat.gsia.cmu.edu/COLOR/instances.html

Recorrido de Grafos

- Búsqueda hacia lo ancho
- Breadth-First Search (BFS)
- Búsqueda hacia lo profundo
- Depth-First Search (DFS)

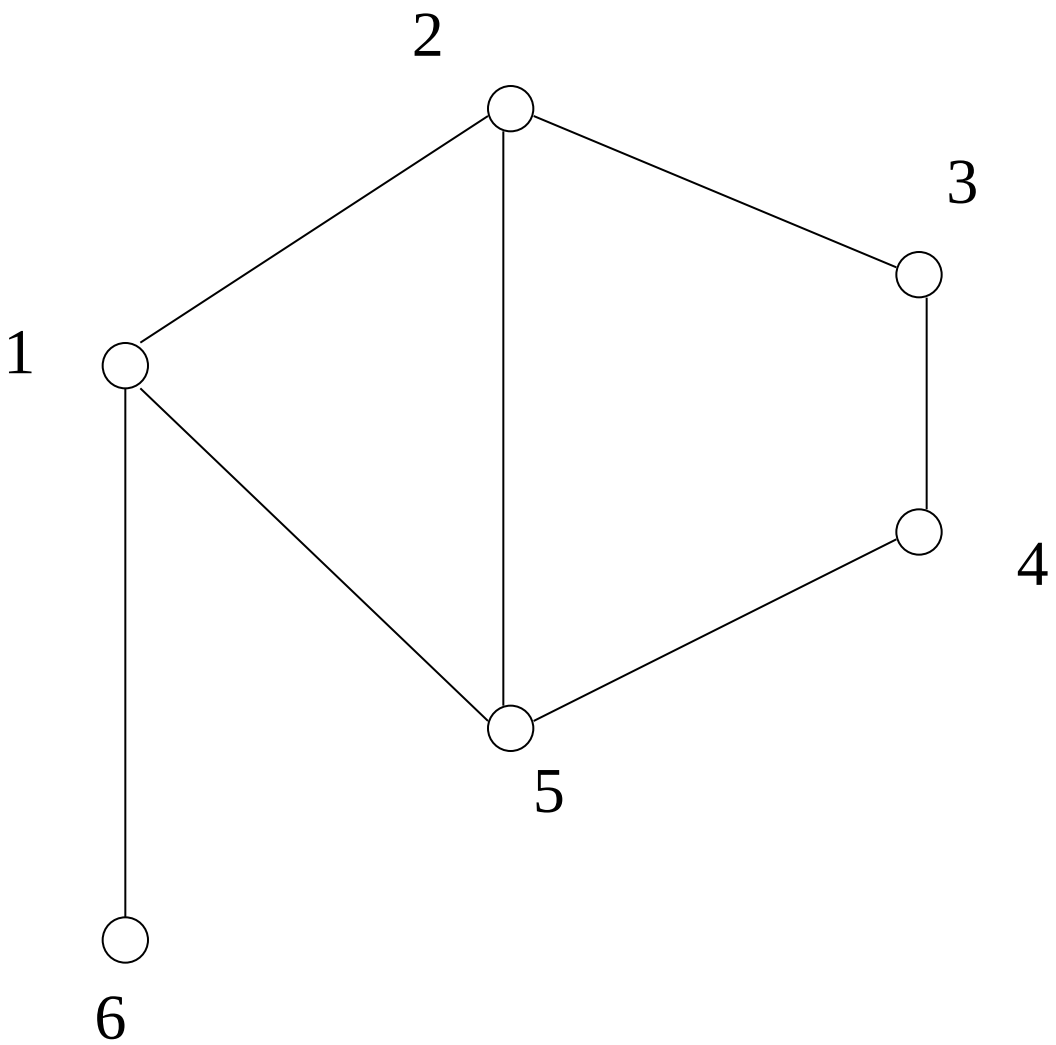
BFS



BFS

```
procedure BFS(adylist, v)
var Q, w
initialize Q = ∅;
visit and mark v; insert v in Q;
while Q ≠ ∅ {
    x := dequeue (Q);
    for (w in adylist(x) & notmarked) {
        visit & mark w;
        enqueue (Q, w);
    }
}
```

DFS



DFS

procedure DFS(*AdyList*, *v*)
var *w*;
visit & mark *v*;
while (unmarked *w* in adylist(*v*)){
    DFS(*w*);
}

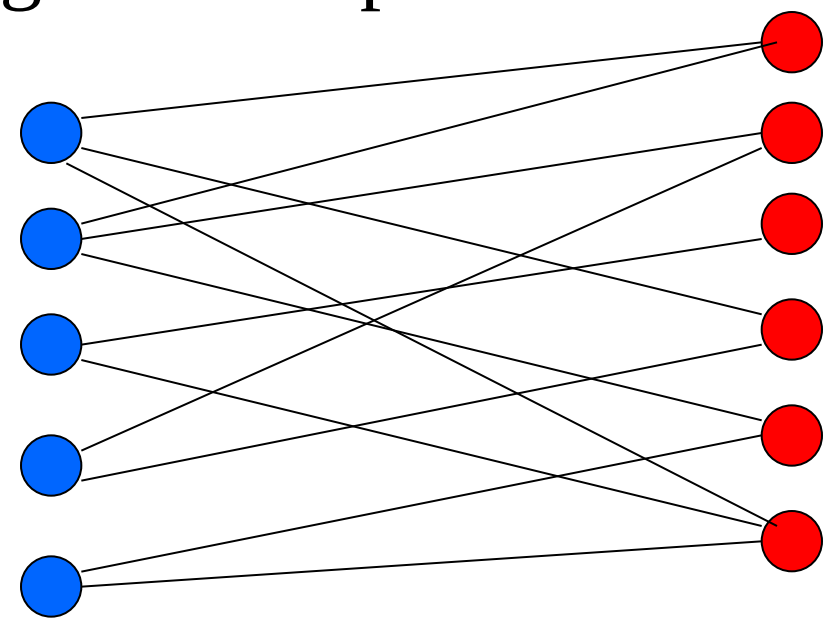
Aplicaciones BFS
Componentes conexas

```
init_search(g)
c = 0;
for (i=1; i<= n; i++) {
   if ( not discovered[i]) {
         c = c + 1;
         print(component: c);
         bfs(g,i);
   }
}
```

Aplicaciones BFS
2-Coloreo de grafos

Un grafo es bipartita si puede colorearse usando sólo dos colores.

BFS puede usarse para detectar si un grafo es bipartita.



PROBLEMAS EN GRAFOS

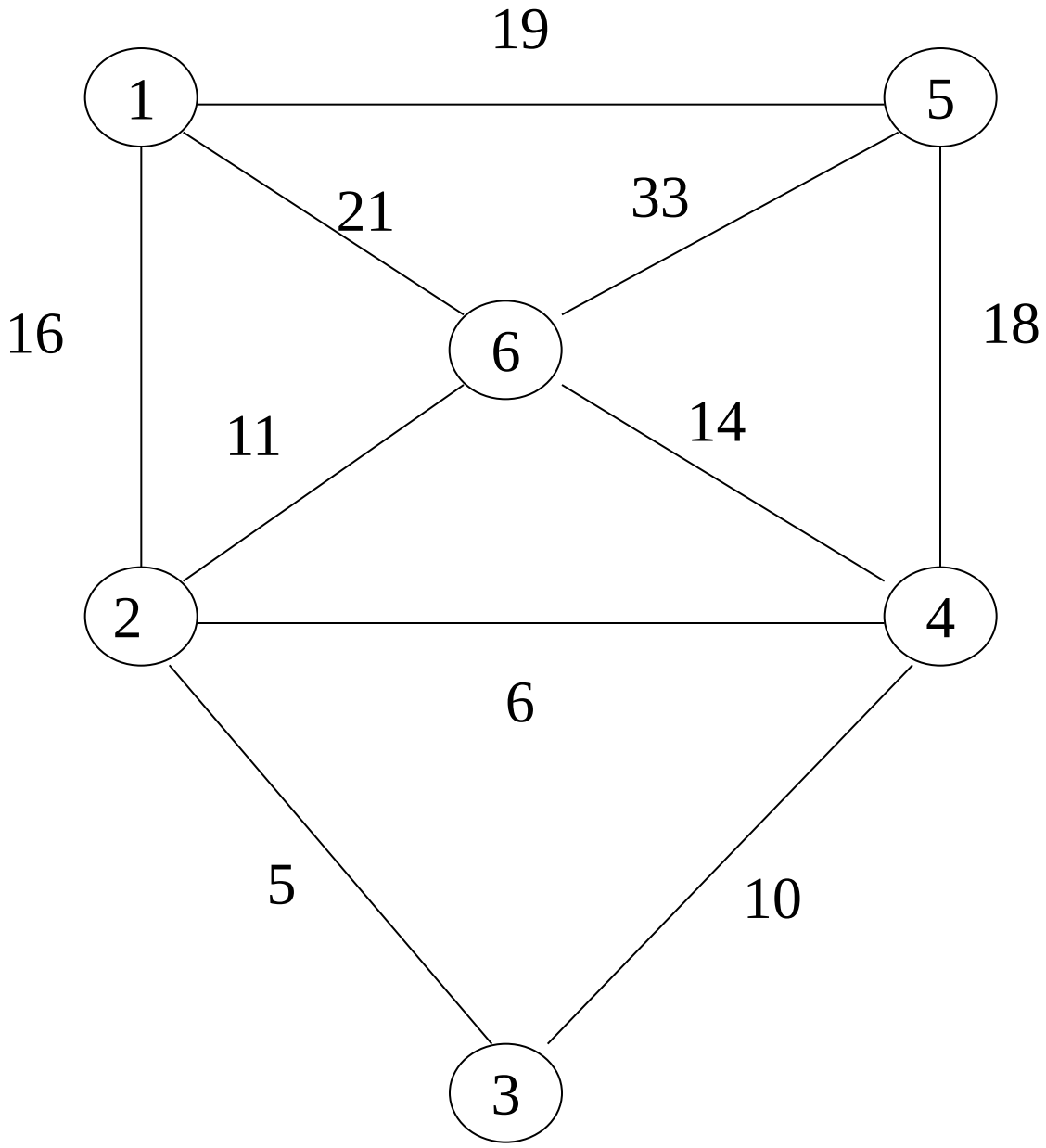
- Mínimo árbol abarcador (MST)
- Instancia $G = (V, E)$ y una función de peso $w(e)$, un entero para cada e en E
- Un árbol abarcador es un árbol que contiene a todos los vértices de V.
- Pregunta: ¿Cuál es árbol abarcador de mínimo peso?

PROBLEMAS EN GRAFOS

- Camino más corto
- Instancia $G = (V, E)$ y una función de distancia $w(e)$, un entero para cada e en E.
- Pregunta: Dados dos vértices u y v en E, ¿cuál es la trayectoria en G que une a u con v de distancia mínima?

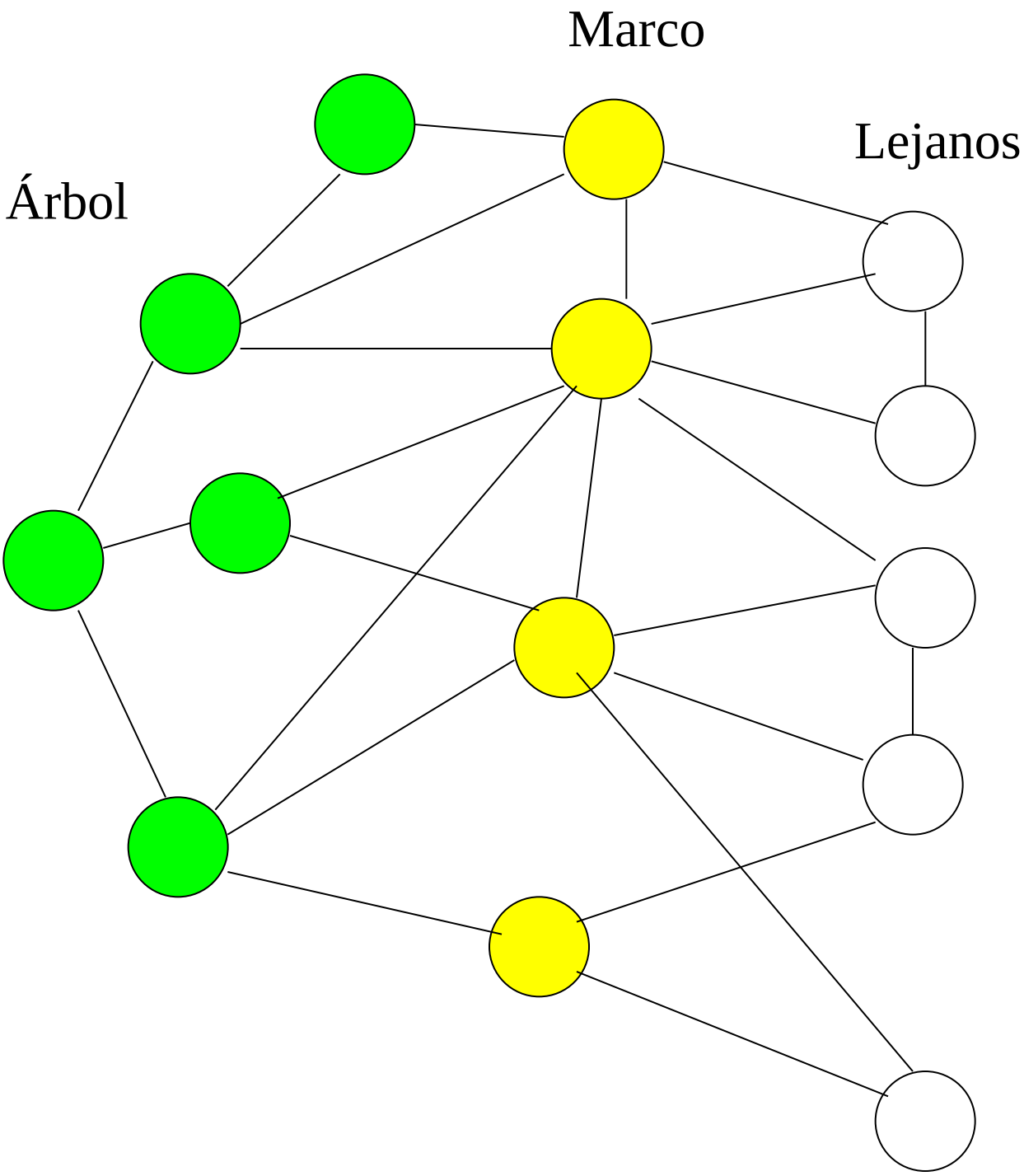
ALGORITMO DE DIJKSTRA-PRIM

- Estrategia:
- Elegir un vértice inicial arbitrario, y de allí el árbol empieza a ramificarse desde la parte del árbol construida hasta el momento agregando un nuevo vértice y una nueva arista en cada iteración.


19
1
5
21
33
16
6
18
11
14
2
4
6
5
10
3

- Durante la ejecución del algoritmo, los vértices se dividen en tres subconjuntos ajenos:

- Vértices árbol: en el árbol construido hasta el momento.
- Vértices marco: no se encuentran en el árbol, pero son adyacentes a algún vértice en el árbol.
- Vértices lejanos: el resto


Marco
Lejanos
Árbol

- Estructura general del algoritmo:

Seleccionar un vértice arbitrario
    para iniciar el árbol;

while existen vértices marco **do**
    seleccionar una arista de mínimo
    peso entre un vértice árbol y un
    vértice marco;
    añadir la arista seleccionada y el
    vértice marco al árbol ;
end

1. Sea x un vértice arbitrario

   $V_T := \{u\}$; $E_T := \varnothing$;

2. **while** $V \neq V_T$ **do**

3. **for** cada vértice marco v adyacente a u **do**

   if $w(u,v) <$ w(candidato e adyacente con v) **then**

   (u,v)reemplaza a e.

4. **for** cada lejano v adyacente a u,

   v es ahora vértice marco

   (u,v) es ahora candidato

5. encontrar un candidato e con peso mínimo;

   u:= vértice marco incidente con e;

   añadir u y e al árbol;

```
x:=1; status[1] := árbol;
Contador:= 0;
ListaMarco:=0;
for y:=2 to n  status[y]:=lejano;

while Contador < n - 1
   pointer:=adjList[x];
   while ptr≠nil
        y:= ptr↑.vertex;
        if status[y]=marco and
        ptr↑.peso < PesoMarco[y]
            padre[y]:=x;
            PesoMarco[y]:= ptr↑.peso;
        end
```

if *status*[*y*]=*lejano*

status[*y*]:=*marco*;

y se liga al principio de la lista marco;

padre[*y*]:=*x*;

PesoMarco[*y*]:= *ptr*↑.*peso*;

end;

ptr := *ptr* ↑.*link*;

end;

recorrer la lista marco para encontrar un candidato con peso mínimo.

x:=vértice marco incidente;

eliminar *x* de la lista marco;

status[*x*]:=*árbol*;inc(*Contador)*;

end;

Teorema

Sea $G = (V,E,w)$ un grafo conexo con peso w y sea $E' \subseteq E$ un subconjunto de las aristas en algún árbol abarcador mínimo $T = (V, E_T)$ para G. Sea V' el conjunto de vértices incidentes con las aristas en E'. Si (u,v) es una arista de mínimo peso tal que $u \in V'$ y $v \notin V'$, entonces $E' \cup \{(u,v)\}$ es un subconjunto de un árbol abarcador mínimo.

Camino más corto

- Una gran cantidad de problemas de optimización combinatoria pueden formularse y resolverse como problemas de camino más corto.
- Problemas más complejos pueden resolverse con procedimientos que llaman como subrutinas a problemas de camino más corto.

Observaciones

- Calcular el camino más corto desde un nodo *origen* hasta un nodo *terminal* es igual de difícil (o fácil) que calcular los caminos más cortos desde un nodo *origen* hasta todos los nodos del grafo.
- Utilizaremos ideas de *programación dinámica*.

Algunas formulaciones

- Trayectorias más confiables
- En una red de comunicaciones, la probabilidad de que la conexión de j a k se encuentre operativa es p_{jk}. Así que la probabilidad de que todas las conexiones en una trayectoria dada estén operativas es el producto. ¿Cuál es la trayectoria más confiable de un nodo a otro?

Ecuaciones de Bellman

- a_{ij} = la longitud (finita) del arco (i,j) si tal arco existe.
- u_j = la longitud de un camino más corto desde el origen hasta el nodo j.

- $u_1 = 0$
- $u_j = \min_{k \neq j} \{u_k + a_{kj}\}$ $(j = 2,3,\ldots,n)$

Suposiciones

- Existe una trayectoria de longitud finita desde el origen a cada uno de los nodos restantes.
- Todos los circuitos dirigidos tienen longitud estrictamente positiva.
- Bajo estas condiciones las ecuaciones de Bellman tienen solución única finita.

Construcción de las trayectorias

- u_1, u_2, …,$u_{n\text{-}1}$, u_n, soluciones.
- Para encontrar una trayectoria de longitud u_j hasta el nodo j, encontrar un arco (k,j) que satisfaga $u_j = u_k + a_{kj}$, luego hallar un arco (l,k) tal que $u_k = u_l + a_{lk}$, continuar de esta forma hasta alcanzar el origen.
- Eventualmente, se debe llegar al origen, en otro caso habría un ciclo nulo.

Estructura de la solución

- Se seleccionan exactamente *n*-1 arcos.
- No se forma ningún ciclo.
- Se ha obtenido un árbol con raíz en el origen.
- **Teorema**: Existe un árbol con raíz en el origen, tal que la trayectoria del origen a cada nodo es un camino más corto.

Dificultades

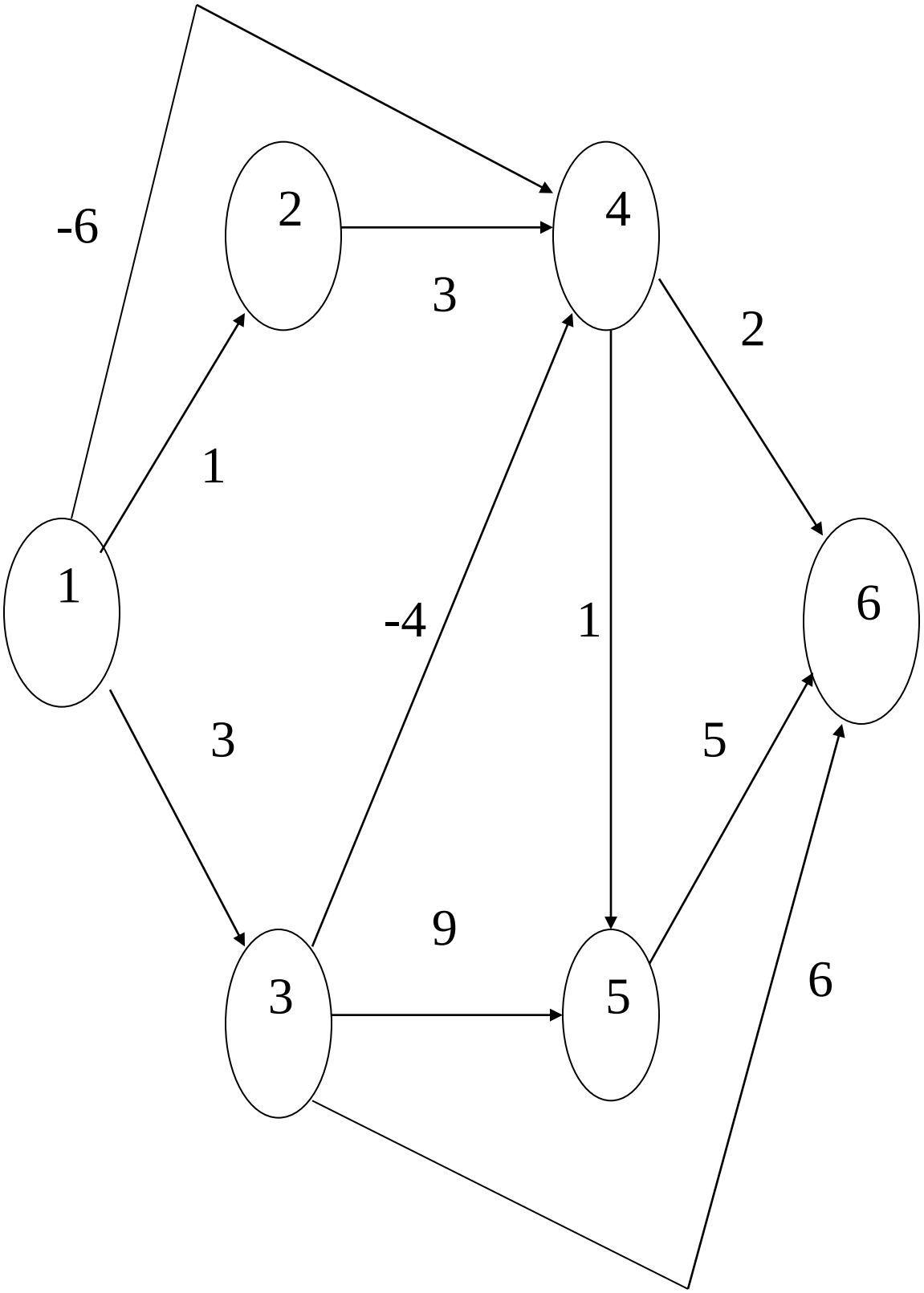
- Ecuaciones no lineales
- Relaciones funcionales implícitas
- Cada u_j se encuentra expresada como una función no lineal de las restantes.

- Encontrar métodos adecuados para superar estas dificultades en situaciones particulares

Redes sin circuitos

- Existe una numeración de los nodos tal que hay un arco dirigido de i a j sii $i < j$
- $u_1 = 0$
- $u_j = \min_{k<j} \{u_k + a_{kj}\}$ $(j = 2,3,\ldots,n)$

- Estas ecuaciones pueden resolverse fácilmente mediante sustitución hacia adelante.


-6
2
4
3
2
1
1
6
-4
1
3
5
9
3
5
6

Complejidad

- Sumas:
- $0 + 1 + 2 + \ldots + n - 1 = n(n-1)/2$

- Comparaciones:
- $1 + 2 + n-2 = (n-1)(n-2) / 2$

- $\boldsymbol{O}(n^2)$

Observaciones

- Es obvio que una red sin circuitos no puede tener circuitos negativos, independiente de la longitud de sus arcos.
- Se puede reemplazar cada arco por su negativo y resolver el problema.
- Equivalente a resolver el problema del camino más largo. En general este es un problema intratable.

Redes con arcos positivos

- Estrategia: Etiquetar los nodos. En cada iteración algunas etiquetas serán “tentativas” y otras “permanentes”.
- Permanente: la longitud del camino más corto hasta el nodo.
- Tentativa: cota superior sobre esa longitud.

Procedimiento

- Inicialmente, el único nodo permanente es el origen, con etiqueta $u_1 = 0$.
- El resto de los nodos se etiqueta tentativamente con $u_j = a_{1j}$.
- Paso general: hallar el nodo k cuya etiqueta tentativa sea la mínima. Declarar esta etiqueta como permanente.

- Actualizar las restantes etiquetas tentativas:

$$u_j = \min \{u_j, u_k + a_{kj}\}.$$

- El procedimiento termina cuando ya no hay etiquetas tentativas.

- Este procedimeinto se conoce como el Algoritmo de Dijkstra.

Algoritmo (Arcos positivos)

- **Inicialización**

  $u_1 = 0$; $u_j = a_{1j}$ ($j=2,\ldots,n$);

  $P = \{1\}$; $T = N\text{-}P$;

  while ($T \neq \varnothing$) {

  $k = \arg(\min_{j \in T} \{u_j\})$;

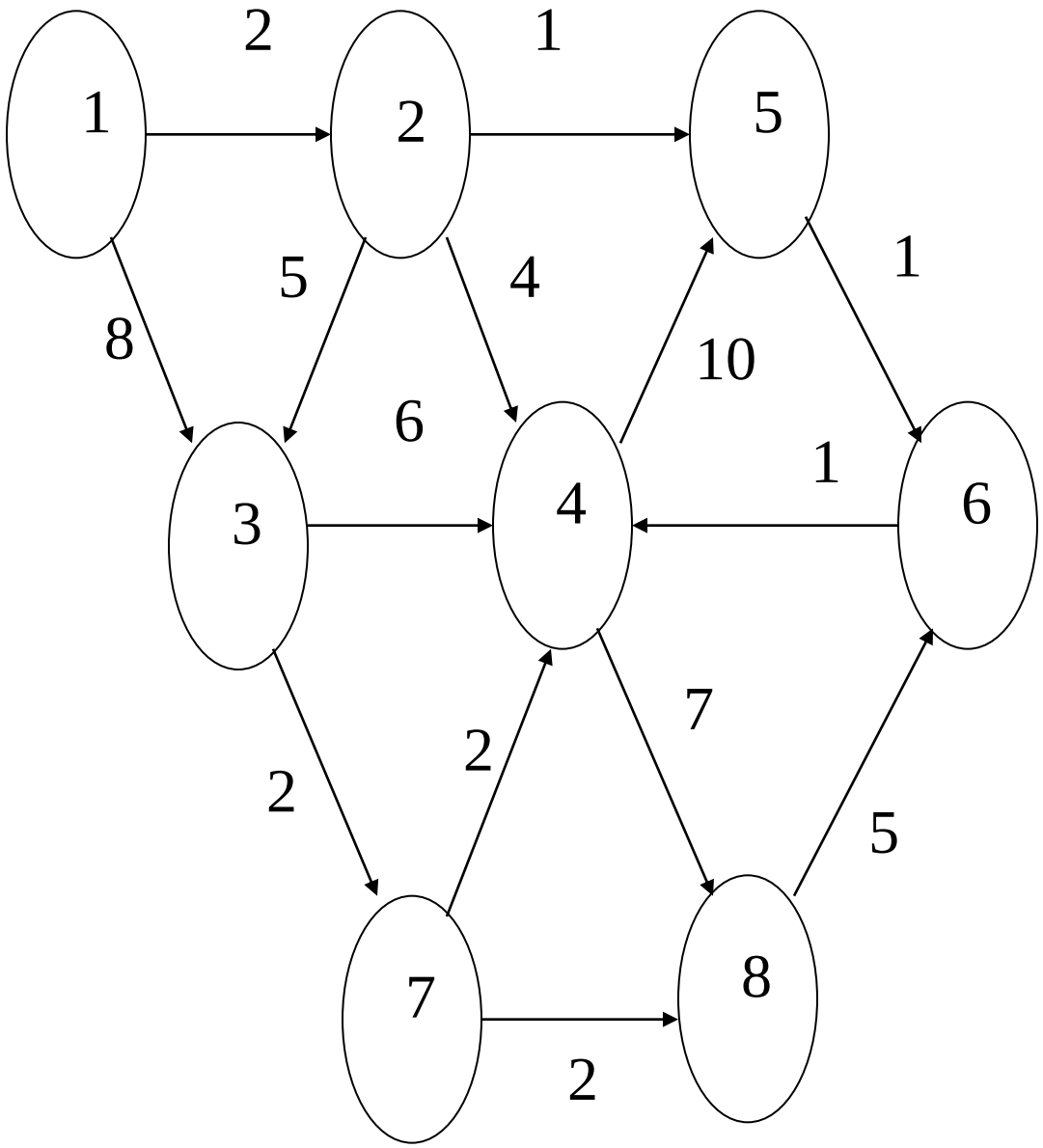
  $T = T - k$; $P = P \cup k$;

  $u_j = \min \{u_j, u_k + a_{kj}\}$;

  end (while)}

Complejidad

- Inicialización n asignaciones
- Primer renglón del while:
- $(n\text{-}2) + (n\text{ -}3) + \ldots = (n\text{-}1)(n\text{-}2)/2$ comparaciones.
- Segundo renglón:
- $(n\text{-}2) + (n\text{ -}3) + \ldots = (n\text{-}1)(n\text{-}2)/2$ comp.
- $(n\text{-}2) + (n\text{ -}3) + \ldots = (n\text{-}1)(n\text{-}2)/2$ sumas


1
2
5
3
4
6
7
8
2
1
8
5
4
10
1
6
2
7
2
5
2

El Algoritmo de Dijkstra no funciona si existen arcos con peso negativo.

Aproximaciones sucesivas

- Método general: no suponemos que la red es acíclica, tampoco que todos los arcos son positivos.
- Seguimos suponiendo que no hay ciclos negativos.
- Se atribuye a Bellman y Ford.

Algoritmo de Bellman-Ford

Instancia: $G = (V,E,d)$

d función de distancia: entero, puede ser negativo, pero no se permiten ciclos negativos

Pregunta: ¿Cuál es la menor distancia del origen al vértice j?

$u_j^{(m)}$ = longitud de un camino más corto desde el origen hasta el nodo *j*, sujeto a la restricción de que la trayectoria contiene a lo más *m* arcos.

$u_1^{(1)} = 0$

$u_j^{(1)} = d_{1j}$

Obs. Si no existe un arco de *i* a *j* la distancia es infinito.

$u_j^{(m+1)} = \min\{u_j^{(m)}, \min\{u_k^{(m)} + d_{kj}\}\}$

Complejidad: $O(n^3)$

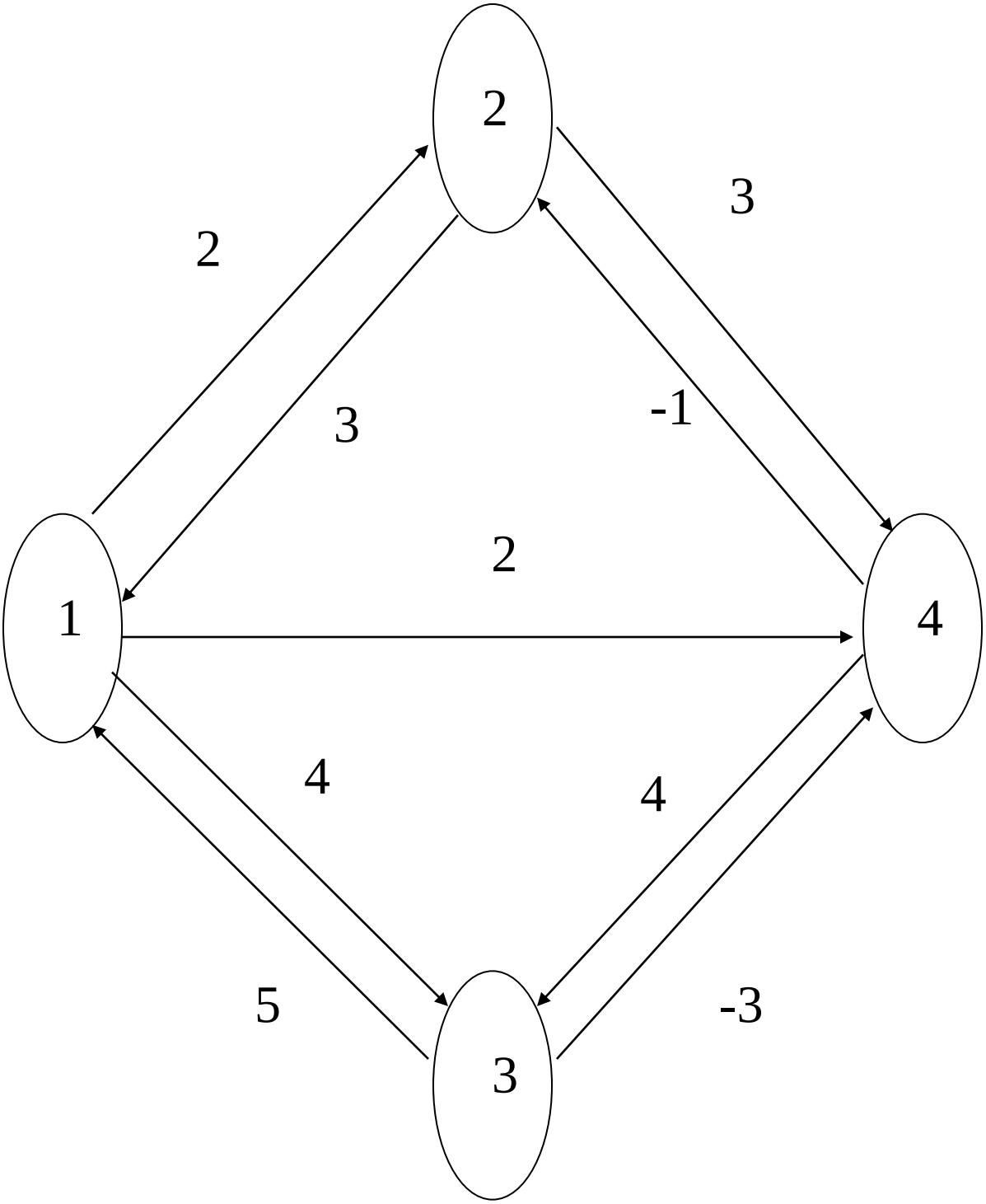
- Inicialmente, hacemos

$$u^{(1)}_1 = 0,$$

$$u^{(1)}_j = a^{(1)}_{1j} \quad j \neq 1$$

- calculamos aproximaciones de orden m+1 a partir de las aproximaciones de orden m.
- Deseamos que para cada nodo j:

$$u^{(1)}_j \geq u^{(2)}_j \geq u^{(3)}_j \geq \ldots u^{(m)}_j \geq u^{(m+1)}_j \ldots$$


2
3
2
3
-1
2
1
4
4
4
5
-3
3

Complejidad

- Las ecuaciones deben de resolverse para $m = 1,2,\ldots,n-2$.
- Para cada valor de m hay n ecuaciones a ser resueltas.
- La solución de cada ecuación requiere n -1 sumas y minimización sobre n alternativas.
- El cálculo es $O(n^3)$

Camino más corto entre todos los pares de vértices

Para encontrar el camino más corto entre todos los pares de vértices se podría utilizar alguno de los algoritmos anteriores, cada uno de los vértices se considera como el vértice inicial, y se calcula el camino más corto a cada uno de los vértices restantes.

Complejidad: $O(n^3)$ y $O(n^4)$ resp.

Algoritmo de Floyd-Warshall

Dada una trayectoria $\{v_1, v_2, ..., v_{l-1}, v_l\}$ un vértice **intermedio** es cualquier vértice en la trayectoria distinto de v_1 o de v_l

$d_{ij}^{(k)}$ = la longitud de un camino más corto de i a j, sujeto a la condición de que todos los vértices intermedios estén en el conjunto $\{1,2,...,k\}$.

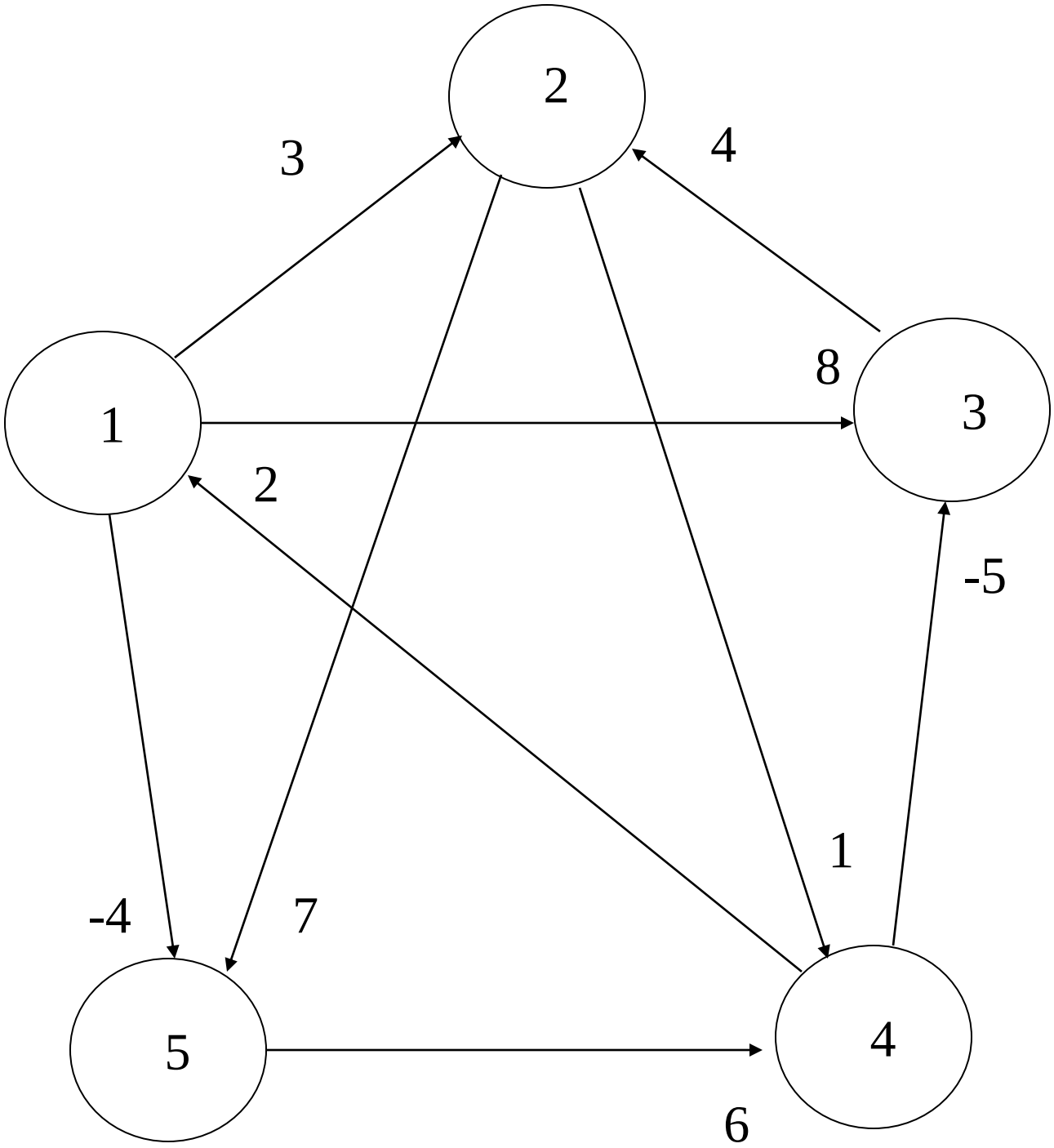
Recurrencia

$d_{ij}^{(0)} = a_{ij}$

$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$

para $k \geq 1$

Iniciamos construyendo la matriz $D^{(0)}$, la cual contiene $\{d_{ij}^{(0)}\}$, la solución se encuentra en la matriz $D^{(n)}$, conteniendo $\{d_{ij}^{(n)}\}$


2
3
4
1
8
3
2
-5
1
-4
7
5
4
6

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$\Pi^{(0)} = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 2 & 2 \\ 0 & 3 & 0 & 0 & 0 \\ 4 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 5 & 0 \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$\Pi^{(1)} = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 2 & 2 \\ 0 & 3 & 0 & 0 & 0 \\ 4 & 1 & 4 & 0 & 1 \\ 0 & 0 & 0 & 5 & 0 \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$\Pi^{(2)} = \begin{pmatrix} 0 & 1 & 1 & 2 & 1 \\ 0 & 0 & 0 & 2 & 2 \\ 0 & 3 & 0 & 2 & 2 \\ 4 & 1 & 4 & 0 & 1 \\ 0 & 0 & 0 & 5 & 0 \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$\Pi^{(3)} = \begin{pmatrix} 0 & 1 & 1 & 2 & 1 \\ 0 & 0 & 0 & 2 & 2 \\ 0 & 3 & 0 & 2 & 2 \\ 4 & 3 & 4 & 0 & 1 \\ 0 & 0 & 0 & 5 & 0 \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

$$\Pi^{(4)} = \begin{pmatrix} 0 & 1 & 4 & 2 & 1 \\ 4 & 0 & 4 & 2 & 1 \\ 4 & 3 & 0 & 2 & 1 \\ 4 & 3 & 4 & 0 & 1 \\ 4 & 3 & 4 & 5 & 0 \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

$$\Pi^{(5)} = \begin{pmatrix} 0 & 3 & 4 & 5 & 1 \\ 4 & 0 & 4 & 2 & 1 \\ 4 & 3 & 0 & 2 & 1 \\ 4 & 3 & 4 & 0 & 1 \\ 4 & 3 & 4 & 5 & 0 \end{pmatrix}$$

void Floyd-Warshall (A, $U^{(n)}$)

N = filas[A];

$U^{(0)} = A$;

for ($k = 1$; $k \leq n$; ++k) {

    for ($i = 1$; $i \leq n$; ++i) {

        for ($j = 1$; $j \leq n$; ++j) {

$$u_{ij}^{(k)} = \min (u_{ij}^{(k-1)}, u_{ik}^{(k-1)} + u_{kj}^{(k-1)});$$

}}};

Construcción del camino

Se puede calcular la matriz Π de predecesores simultáneamente.

$\pi_{ij}^{(k)}$ = predecesor del vértice j en un camino más corto desde i con todos los vértices intermedios en el conjunto $\{1,2,...,k\}$.

Recurrencia

$\pi_{ij}^{(0)} = \text{NIL si } i = j \text{ o } a_{ij} = \infty$

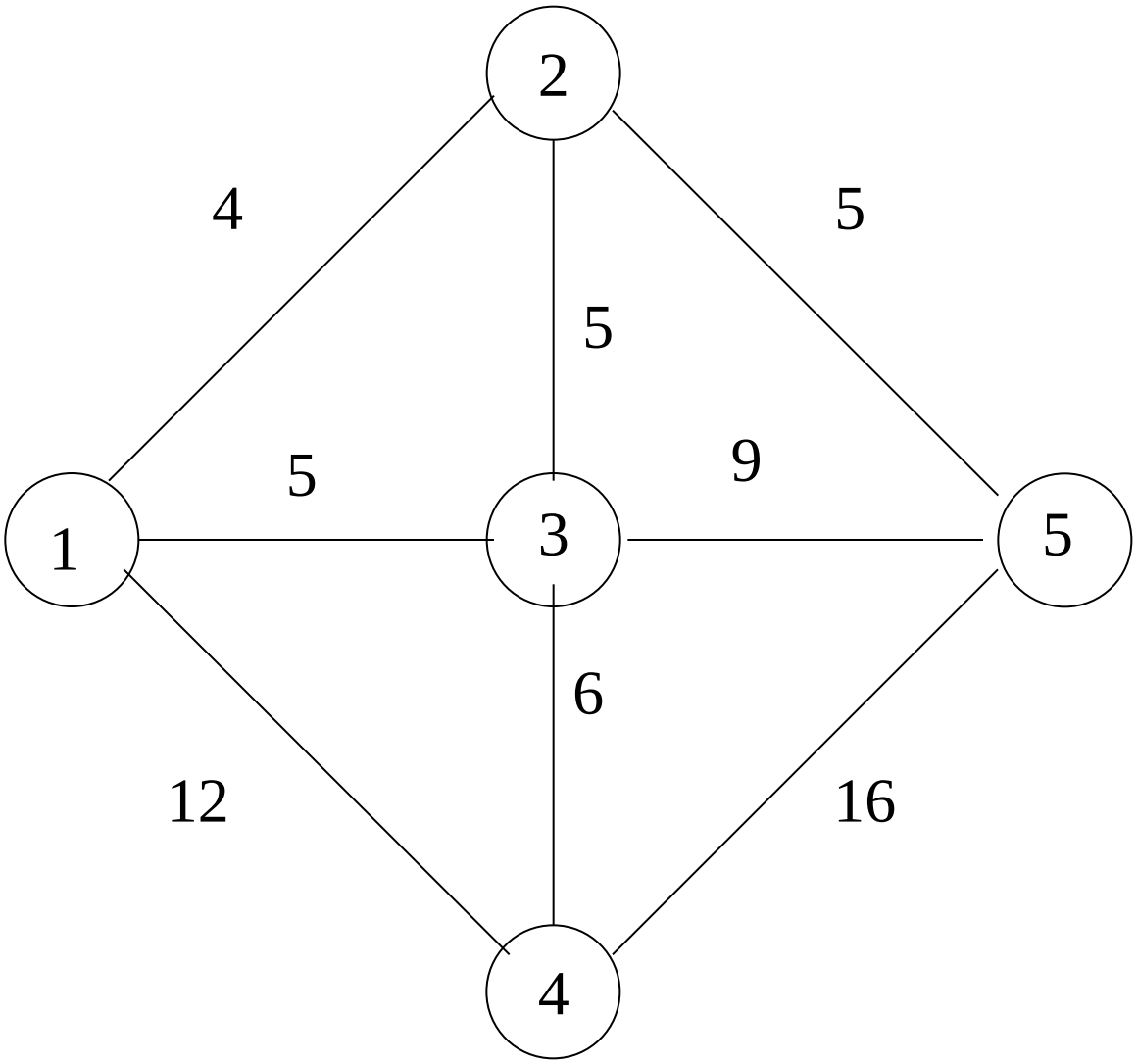
$\pi_{ij}^{(k)} = \pi_{ij}^{(k\text{-}1)} \text{ si } d_{ij}^{(k\text{-}1)} \leq d_{ik}^{(k\text{-}1)} + d_{kj}^{(k\text{-}1)}$

$\pi_{ij}^{(k)} = \pi_{kj}^{(k\text{-}1)} \text{ si } d_{ij}^{(k\text{-}1)} > d_{ik}^{(k\text{-}1)} + d_{kj}^{(k\text{-}1)}$

El camino más corto se construye usando la matriz $\Pi^{(n)}$

Caminos por arcos

- Problema del Cartero Chino
- Mei Ko Kuan
- Instancia: G = (V, E, w)
- Encontrar un circuito cerrado de mínima longitud que recorra cada arista de un grafo conexo no dirigido.
- Se permite recorrer aristas más de una vez.


2
4
5
5
5
9
1
3
5
6
12
16
4

Caminos por arcos

- Problema del Cartero Chino
- Mei-Ko Kuan
- Instancia: G = (V, E, w)
- Encontrar un circuito cerrado de mínima longitud que recorra cada arista de un grafo conexo no dirigido.
- Se permite recorrer aristas más de una vez.

Problemas $\mathcal{NP}$-Completos

- Definiremos algunas nociones cuyo propósito es distinguir entre los problemas *tratables*, (esto es “no tan difíciles”) y los problemas *intratables* (“difíciles”).
- Enunciaremos algunos problemas que usaremos como ejemplos.

Programación de tareas

- n trabajos $J_1, J_2,\ldots, J_n$, deben de ejecutarse uno a uno. Tenemos los tiempos de ejecución $t_1, t_2,\ldots, t_n$, tiempos límites $d_1, d_2,\ldots, d_n$, y penalizaciones $p_1, p_2,\ldots, p_n$ por no terminar a tiempo.
- Una programación es una permutación π de $\{1,2,\ldots,n\}$, donde $J_{\pi(1)}$, es el primer trabajo a ser ejecutado, $J_{\pi(2)}$, el segundo, etc.

- El castigo total para una programación particular es:

$$P_\pi = \sum_{j=1,\ldots,n} p_{\pi(j)}$$

para j tal que $t_{\pi(1)} + \ldots + t_{\pi(j)} > d_{\pi(j)}$

Optimización: Determinar el mínimo castigo posible.

Decisión: Dado además, un entero no negativo k, existe una programación con $P_\pi \leq k$?

Empacamiento

Instancia: un número ilimitado de cajitas cada una de capacidad 1, y n objetos con tamaños s_1, s_2,... s_n, donde $0 < s_j < 1$.

Optimización: Determinar el mínimo número de cajitas en donde se pueden empacar los n objetos.

Decisión: ¿Dado k, caben los n objetos en k cajitas?

Suma de Subconjuntos

Instancia n objetos con pesos , s_1, s_2,... s_n, enteros positivos.

Optimización: Entre los subconjuntos de objetos, cuya suma no excede C, ¿cuál es aquél de mayor suma?

Decisión: ¿Existe un subconjunto de objetos cuya suma sea exactamente C?

Satisfactibilidad

La madre de todos los problemas intratables.

Satisfactibilidad

Instancia: Un conjunto de variables lógicas. Una cláusula, esto es, una sucesión de varibales booleanas separadas por el operador **o** ($\vee$). Una forma normal conjuntiva (CNF), una sucesión de cláusulas separadas por el operador **y** ($\wedge$)

Decisión: ¿Existe una asignación de valores lógicos T , F, para las variables tal que la CNF tenga valor T ?

Agente Viajero

Instancia: $G = (V,E,d)$ grafo completo

Optimización: Encontrar un circuito Hamiltoniano de longitud mínima.

Decisión: Dado un entero k, ¿existe un ciclo Hamiltoniano cuya longitud sea a lo más k?

La clase $\mathcal{P}$

- $\mathcal{P}$ es la clase de problemas de decisión que están acotados polinomialmente.

  MST

  Camino más corto

  Trayectoria Euleriana

La clase $\mathcal{NP}$

Para una instancia dada, una solución es un objeto que satisface los criterios del problema y justifica una respuesta afirmativa.

Una solución propuesta es simplemente un objeto de la clase adecuada- puede o no satisfacer los criterios-

$\mathcal{NP}$ es la clase de problemas de decisión para los cuales una solución propuesta dada, puede checarse rápidamente (en tiempo polinomial) para comprobar si realmente es una solución, est es, si satisface los requerimientos del problema.

Una solución propuesta puede ser descrita mediante una cadena de símbolos de algún conjunto finito.

El tamaño de una cadena es el número de símbolos que contiene.

Checar una solución consiste en checar que tiene sentido, y que satisface los criterios del problema.

Algoritmo no determinístico

1. Fase no determinística. Una cadena de caracteres, *s*, completamente arbitraria se escribe en algún lugar de la memoria.
2. Fase determinística. Leer el input, y la cadena *s*.

   Eventualmente, la ejecución termina con el output *sí*, o *no*, o bien puede que se cicle y no termine nunca.

- Decimos que un algoritmo no determinístico está acotado polinomialmente, si existe un polinomio *p* tal que, para cada input de tamaño *n* **para el cual la respuesta es *sí***, la ejecución del algoritmo produce una respuesta *sí* en un número máximo de $p(n)$ pasos.

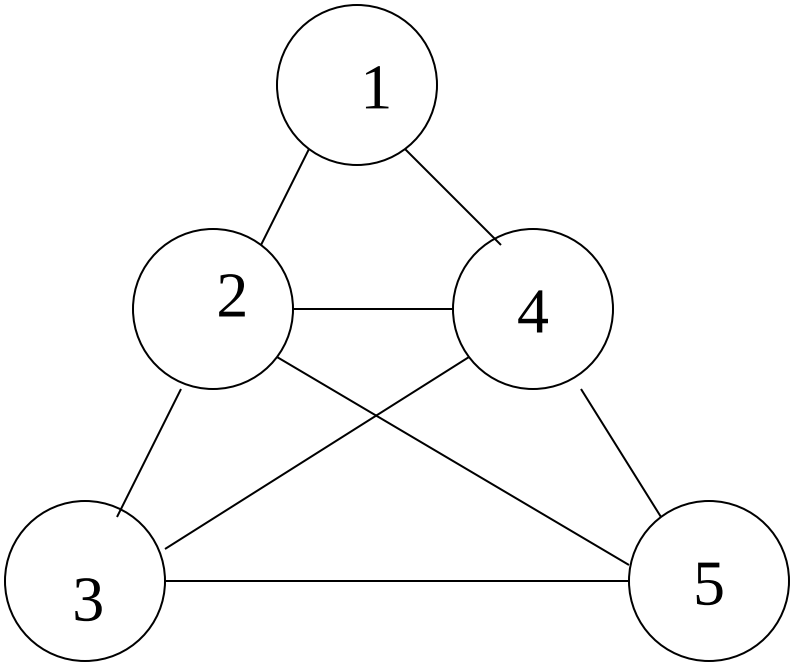
La clase $\mathcal{NP}$

- $\mathcal{NP}$ es la clase de problemas de decisión para los cuales existe un algoritmo no determinístico acotado polinomialmente.

- (**N**ondeterministic **P**olynomially bounded)

Ejemplo

- El problema es detectar si un grafo G es k-coloreable.

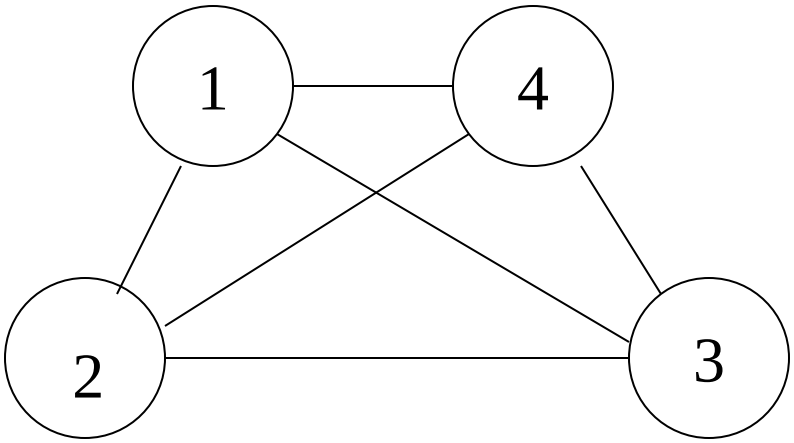


Input: 4, 5,
k, *n*
(1,2),(1,4),(2,4),(2,3),(3,5),(2,5),(3,4),(4,5)

Posibles outputs

- *string* *Output*

- RGRBG *no*
- RGRB *no*
- RBYGO *no*
- RGRBY *sí*
- R#*&/$ *no*



Input: 3, 4,

k, *n*

(1,2), (1,3), (1,4), (2,3), (2,4), (3,4)

Posibles outputs

string	*Output*
RGRBG	*no*
RGYB	*no*
RGRB	*no*
R#*&/$	*no*

Ejemplos

- Coloreo de Grafos
- Circuito Hamiltoniano
- Programación de tareas con castigo
- Empacamiento
- Problema de la mochila
- Satisfactibilidad
- Agente Viajero
- Número Compuesto

El problema del número primo

- Complemento del Problema del Número Compuesto
- FC el conjunto de instancias factibles para el problema del número compuesto
- FP el conjunto de instancias factibles para el problema del número primo

- FC = D \ FP

Teorema (Pratt, 1975)

- Un entero n es primo si y sólo si existe a entero, tal que

i) $a^{n-1} \equiv 1 \pmod{n}$, y

ii) $a^{(n-1)/p} \not\equiv 1 \pmod{n}$, para todos los divisores primos p de $n-1$.

Teorema

$$\mathcal{P} \subseteq \mathcal{NP}$$

Pregunta

$$¿\mathcal{P} = \mathcal{NP}?$$

Reducciones Polinomiales

- Un problema de decisión queda especificado por:
- el conjunto de todas las instancias de ese problema *D*
- el conjunto de las instancias factibles *F*, es decir el conjunto de instancias para las cuales la respuesta es afirmativa.
- Ej. Número compuesto
- $D = \{2,3,4,5,..\}$ $F = \{4,6,8,9,...\}$

- Supongamos que queremos resolver un problema $\Pi_1 = (D_1, F_1)$
- y tenemos un algoritmo para resolver otro problema $\Pi_2 = (D_2, F_2)$
- Supongamos también que tenemos una función T, la cual mapea D_1 en D_2 de forma que $d_1 \in F_1$, si y sólo si $T(d_1) \in F_2$

- Entonces, tenemos un algoritmo para resolver Π_1
- Si T puede calcularse en tiempo polinomial, se llama *transformación polinomial* de Π_1 en Π_2
- Y decimos que Π_1 es polinomialmente reducible a Π_2

$$\Pi_1 \propto \Pi_2$$

Ejemplo

Π_1 es el problema de Ciclo Hamiltoniano (*HC*)

Π_2 es el problema del Agente Viajero (*TSP*)

$$HC \propto TSP$$

Teorema de Cook
1971

- Cualquier problema que pertenece a la clase $\mathcal{NP}$ puede transformarse polinomialmente en SAT.
- Esto es si $\Pi \in \mathcal{NP}$, entonces

$$\Pi \propto \text{SAT}$$

3COL $\propto$ SAT

- Para cada vértice definimos 3 variables Booleanas.
- R_j = T si el vértice j se colorea de rojo, F en otro caso.
- A_j azul , V_j verde
- i) Cada vértice debe de tener exactamente un color asignado:
- al menos un color $(R_j \vee A_j \vee V_j)$
- a lo más uno $\neg(R_j \wedge A_j)$

- Usando las leyes de DeMorgan

$$(R_j \vee A_j \vee V_j) \wedge (\overline{R}_j \vee \overline{A}_j)$$

$$\wedge (\overline{R}_j \vee \overline{V}_j) \wedge (\overline{A}_j \vee \overline{V}_j)$$

- ii) Para cada par de vértices adyacentes (j,k) no se les puede asignar el mismo color:

$$\overline{(R_j \wedge R_k)}$$

- Usando nuevamente las leyes de DeMorgan

$$(\overline{R_j} \vee \overline{R_k}) \wedge (\overline{A_j} \vee \overline{A_k}) \wedge (\overline{V_j} \vee \overline{V_k})$$

Teorema

- Si $\Pi_1 \propto \Pi_2$ y Π_2 está en $\mathcal{P}$, entonces Π_1 está en $\mathcal{P}$.
- Si $\Pi_1 \propto \Pi_2$ y $\Pi_2 \propto \Pi_3$, entonces $\Pi_1 \propto \Pi_3$

- La relación $\Pi_1 \propto \Pi_2$ puede interpretarse como una relación de orden ya que $\Pi_1 \propto \Pi_1$
- Π_2 es “más difícil” que Π_1

Los problemas más difíciles

- Decimos que un problema Π es $\mathcal{NP}$-*Completo* ($\mathcal{NPC}$) si
- i) está en $\mathcal{NP}$
- ii) para cualquier otro problema Π' en $\mathcal{NP}$, $\Pi' \propto \Pi$
- Este conjunto de problemas constituye la clase $\mathcal{NPC}$.

Algunas consecuencias de la definición

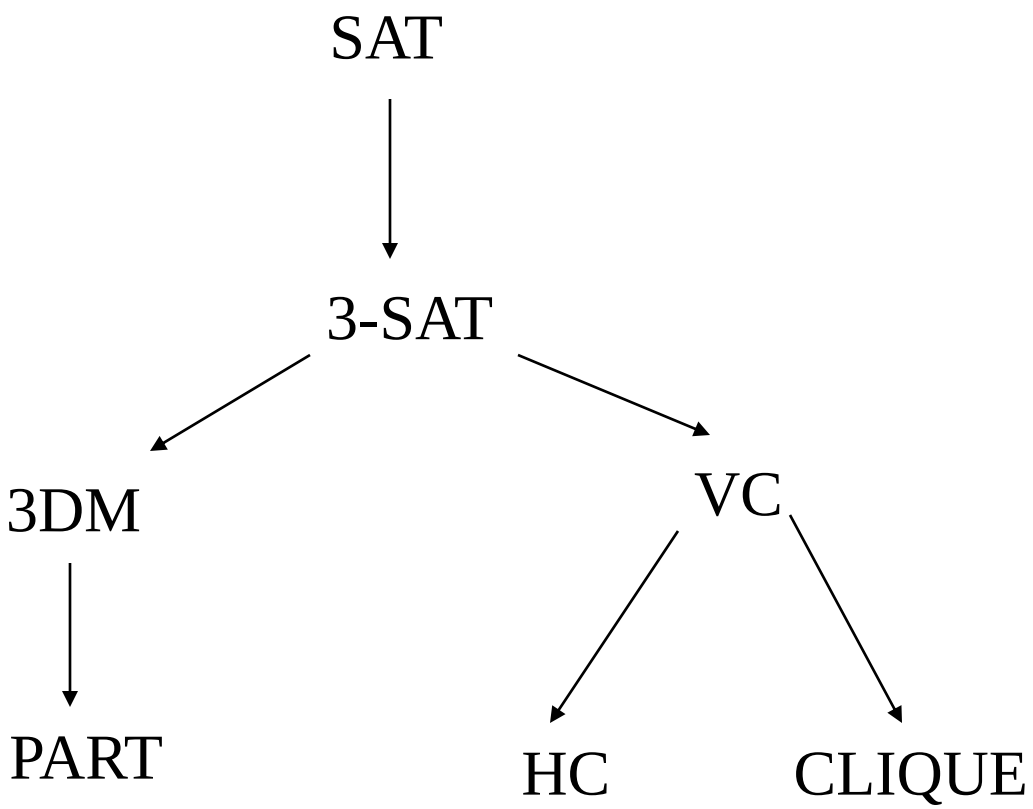
- Si Π_1 está en $\mathcal{NPC}$ y se puede encontrar un algoritmo polinomial para Π_1 entonces cualquier problema en la clase $\mathcal{NP}$ puede resolverse polinomialmente.
- Si Π_1 está en la clase $\mathcal{NPC}$, Π_2 en la clase $\mathcal{NP}$ y $\Pi_1 \propto \Pi_2 \Rightarrow$ Π_2 está en la clase $\mathcal{NPC}$

Teorema de Cook
1971

- SAT se encuentra en la clase $\mathcal{NPC}$

- Esto significa que cualquier problema en $\mathcal{NP}$ se puede transformar polinomialmente en SAT

La lista de Karp (1972)



3-SAT es $\mathcal{NPC}$

- Instancia: Igual que en SAT, pero con la restricción de que cada cláusula contiene exactamente 3 literales.
- ej. $(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3 \vee \neg x_4)$
- Sabemos que SAT está en $\mathcal{NPC}$, basta probar que 3-SAT es más difícil que SAT, esto es
- SAT $\propto$ 3-SAT

Dada una instancia de SAT se debe de transformar polinomialmente en 3-SAT

$U = \{u_1, u_2, \ldots, u_n\}$ conjunto de variables booleanas en SAT

$C = \{c_1, c_2, \ldots, c_m\}$ conjunto de cláusulas en SAT

Construiremos una colección *C*' de cláusulas con tres literales en un conjunto *U*'de variables, tal que *C*' es satisfactible si y sólo si *C* lo es.

c_j dada por $\{z_1, z_2, \ldots, z_k\}$ donde las z's son todas las literales derivadas de las variables en U

Caso 1. k=1 $U_j' = \{y_j^1, y_j^2\}$

$C_j' = \{\{z_1 \vee y_j^1 \vee y_j^2\} \wedge \{z_1 \vee y_j^1 \vee \neg y_j^2\} \wedge \{z_1 \vee \neg y_j^1 \vee y_j^2\} \wedge \{z_1 \vee \neg y_j^1 \vee \neg y_j^2\}\}$

Caso 2. k=2 $U_j' = \{y_j^1\}$

$C_j' = \{\{z_1 \vee z_2 \vee y_j^1\} \wedge \{z_1 \vee z_2 \vee \neg y_j^1\}$

Caso 3. k=3

$U_j' = \varnothing$

$C_j' = \{\{c_j'\}\}$

Caso 4. $k > 3$. $U_j' = \{y_j^i \mid 1 \leq i \leq k\text{-}3\}$

$$C_j' = \{\{z_1 \vee z_2 \vee y_j^1\} \wedge \{\neg y_j^i \vee z_{i+2} \vee y_j^{i+1} \mid 1 \leq i \leq k\text{-}4\} \wedge \{\neg y_j^{k-3} \vee z_{k-1} \vee z_k\}$$

Problemas en $\mathcal{NPC}$

- Coloreo de Grafos
- Circuito Hamiltoniano
- Programación de tareas con castigo
- Empacamiento
- Problema de la mochila
- Satisfactibilidad
- Agente Viajero

Compendio de Problemas NP-completos

- http://www.nada.kth.se/~viggo/problemlist/compendium.html

El problema del número compuesto (diapositiva vieja)

El problema se encuentra en $\mathcal{NP}$

Hemos visto un algoritmo exponencial para resolverlo.

No se conoce ningún algoritmo polinomial.

No se ha podido probar que el problema está en $\mathcal{NPC}$.

El problema del número compuesto (diapositiva nueva)

El problema se encuentra en $\mathcal{NP}$

Hemos visto un algoritmo exponencial para resolverlo.

http://mathworld.wolfram.com/news/2002-08-07_primetest/

El problema de decisión de los primos se encuentra en $\mathcal{P}$.

¿ $\mathcal{P} = \mathcal{NP}$?

- No se ha podido probar que sean iguales. Basta probar que un problema en $\mathcal{NPC}$ admite una solución polinomial.

- Tampoco se ha podido probar que sean diferentes.

- La conjetura más importante en la teoría de la computación.

Lectura

- Turing Award Lecture

- **Combinatorics, Complexity and Randomness**

- Richard M. Karp

- Communications of the ACM.
- February 1986, Vol. 29 No. 2

Algoritmos de aproximación

- Problema $\mathcal{NP}$-*hard* ($\mathcal{NP}$-*duro*)

- La definición de problemas $\mathcal{NPC}$ aplica a problemas de decisión.
- En la práctica es más común encontrar la versión de optimización.
- Un problema es $\mathcal{NP}$- *duro* si es al menos tan difícil como un problema $\mathcal{NPC}$

- ¿Qué podemos hacer si nos topamos con un problema *NP-duro*?
- Enfoque: usar algoritmos rápidos (polinomiales) para los cuales no tenemos garantía de que nos produzcan la mejor solución, pero esperamos que nos den una solución cercana al óptimo.

- Tales algoritmos son llamados algoritmos de aproximación o bien heurísticas

- Para hacer afirmaciones precisas acerca del comportamiento de un algoritmo de aproximación (qué tan buenos son los resultados) requerimos de algunas definiciones.

Formulación

$$\min \sum_{j=1}^{n} \sum_{i=1}^{n} c_{ij} x_{ij}$$

$$s.a \qquad \sum_{j=1}^{n} x_{ij} = 1 \qquad \forall i = 1,...,n$$

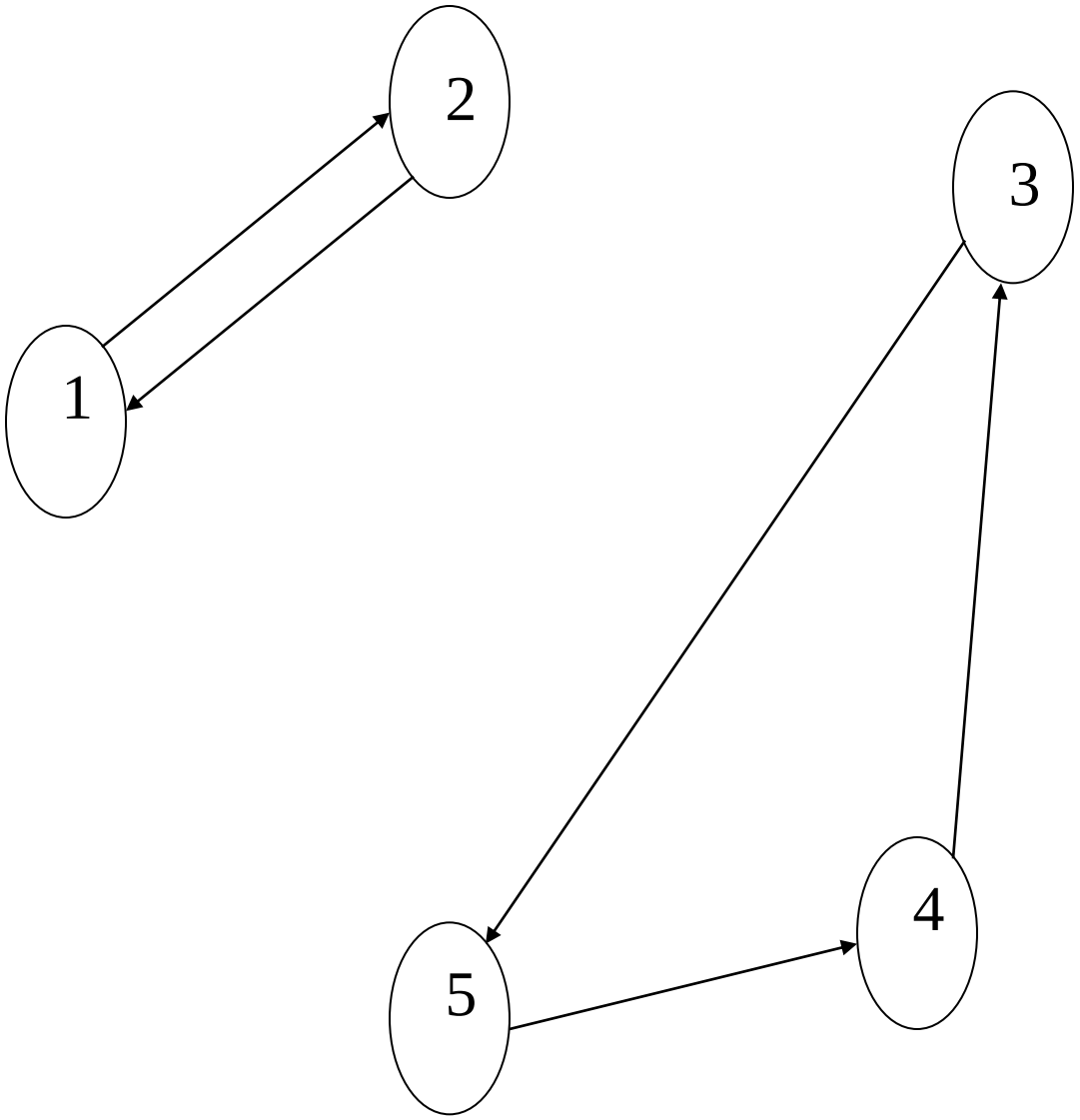
$$\sum_{i=1}^{n} x_{ij} = 1 \qquad \forall j = 1,...,n$$

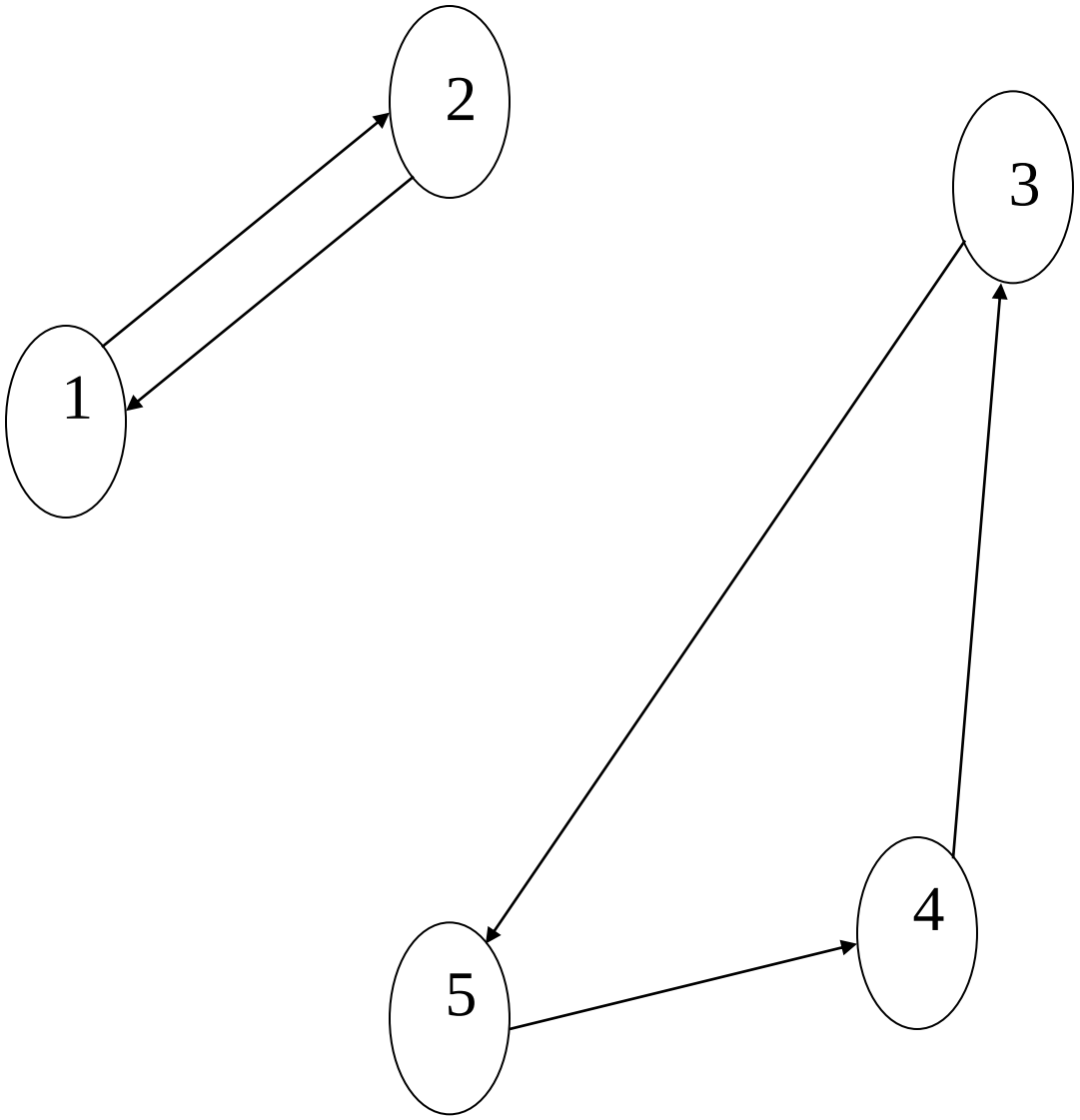
$$x_{ij} = 0,1$$

	1	2	3	4	5
1	-	1.1	2.8	2.4	2.6
2	0.7	-	2.9	2.5	2.7
3	2.6	3.1	-	0.4	0.6
4	2.6	3.1	0.8	-	0.6
5	2.7	3.2	2.9	0.5	-

- Asignación Óptima

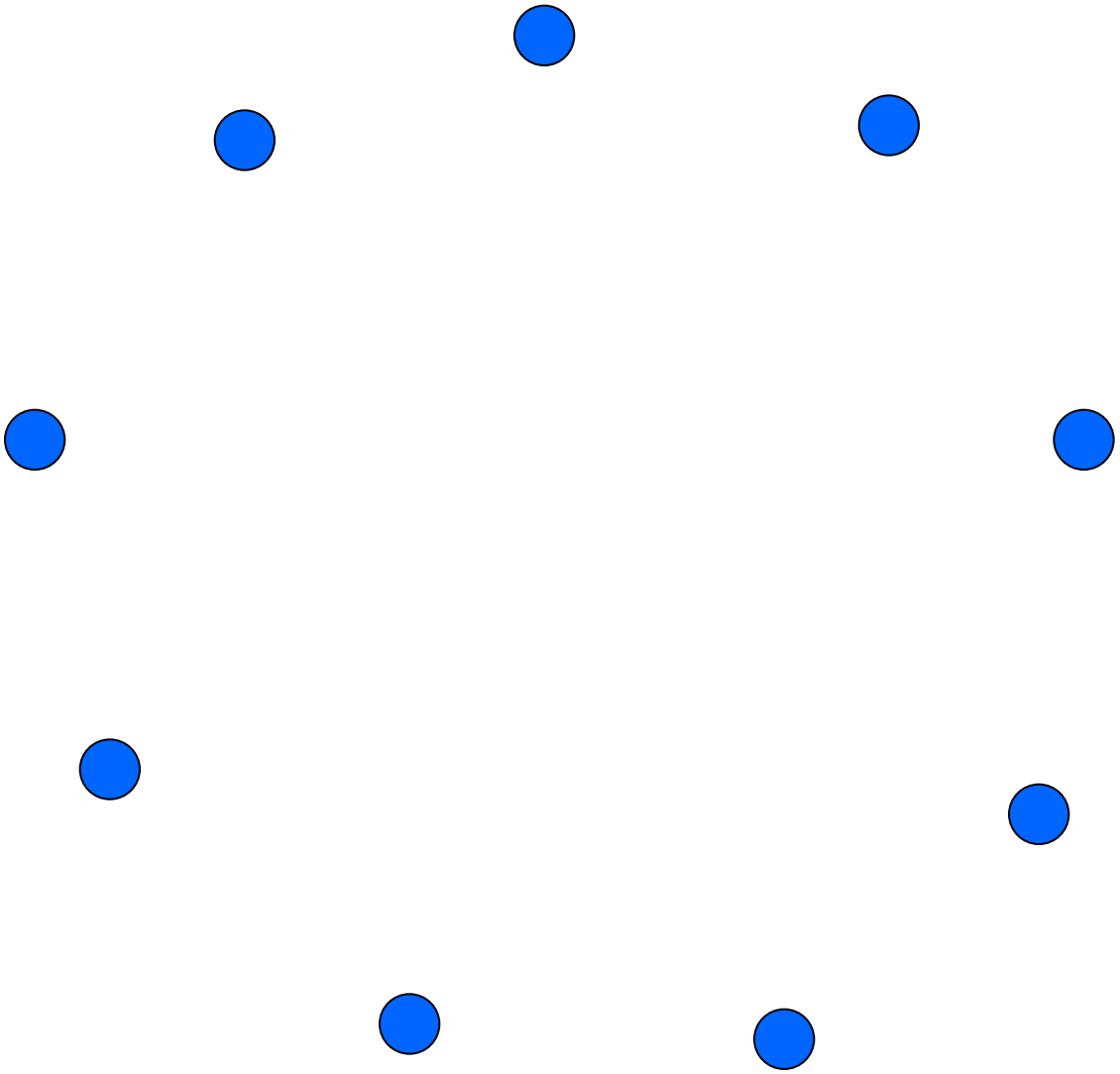
- $1 \rightarrow 2$
- $2 \rightarrow 1$
- $3 \rightarrow 5$
- $4 \rightarrow 3$
- $5 \rightarrow 4$

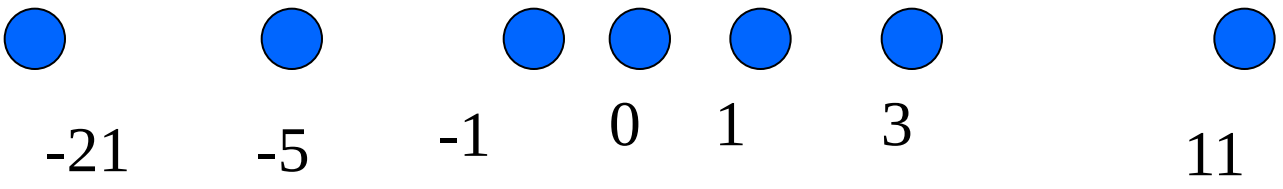

2
3
1
4
5


2
3
1
4
5

Heurísticas Miopes (Greedy)

- TSP
- 1. **Vecino más cercano**
  Empezar con un vértice arbitrario j_1, y construir una trayectoria $j_1, j_2,..., j_k, j_{k+1},\ldots, j_n$, donde j_{k+1}, se elige como el vértice que se encuentra más cerca de j_k y no está en la trayectoria.




-21
-5
-1
0
1
3
11

2. **Inserción más cercana**. Dado un subtour T y un vértice j en $V \setminus T$. Sea $d(j,T) = \min_{k \in T} c_{jk}$, y sea j^* la que minimiza $d(j,T)$. Y sea k^* la que minimiza c_{j^*k}. Entonces j^* es el vértice "más cercano" a T, y k^* es el nodo en T más cercano a j^*.

Construir un subtour en $T \cup \{j^*\}$ insertando k^* entre j^* y uno de sus dos vecinos en T.

Proceder iterativamente hasta construir un tour completo.

***k*-Intercambio**. Las heurísticas de búsqueda local son también útiles para el TSP.

Aquí suponemos que se tiene construido un tour. Eliminar *k* aristas del tour y reemplazarlas con otras *k* aristas que no se encuentren dentro del tour.

Garantía de Optimalidad

Si Π es un problema de minimización, e I es cualquier instancia en D_{Π}, definimos el cociente $R_A(I)$ mediante,

$$R_A(I) = \frac{A(I)}{OPT(I)}$$

Para un problema de maximización, el cociente se define como

$$R_A(I) = \frac{OPT(I)}{A(I)}$$

Notemos que $1 \leq R_A \leq \infty$

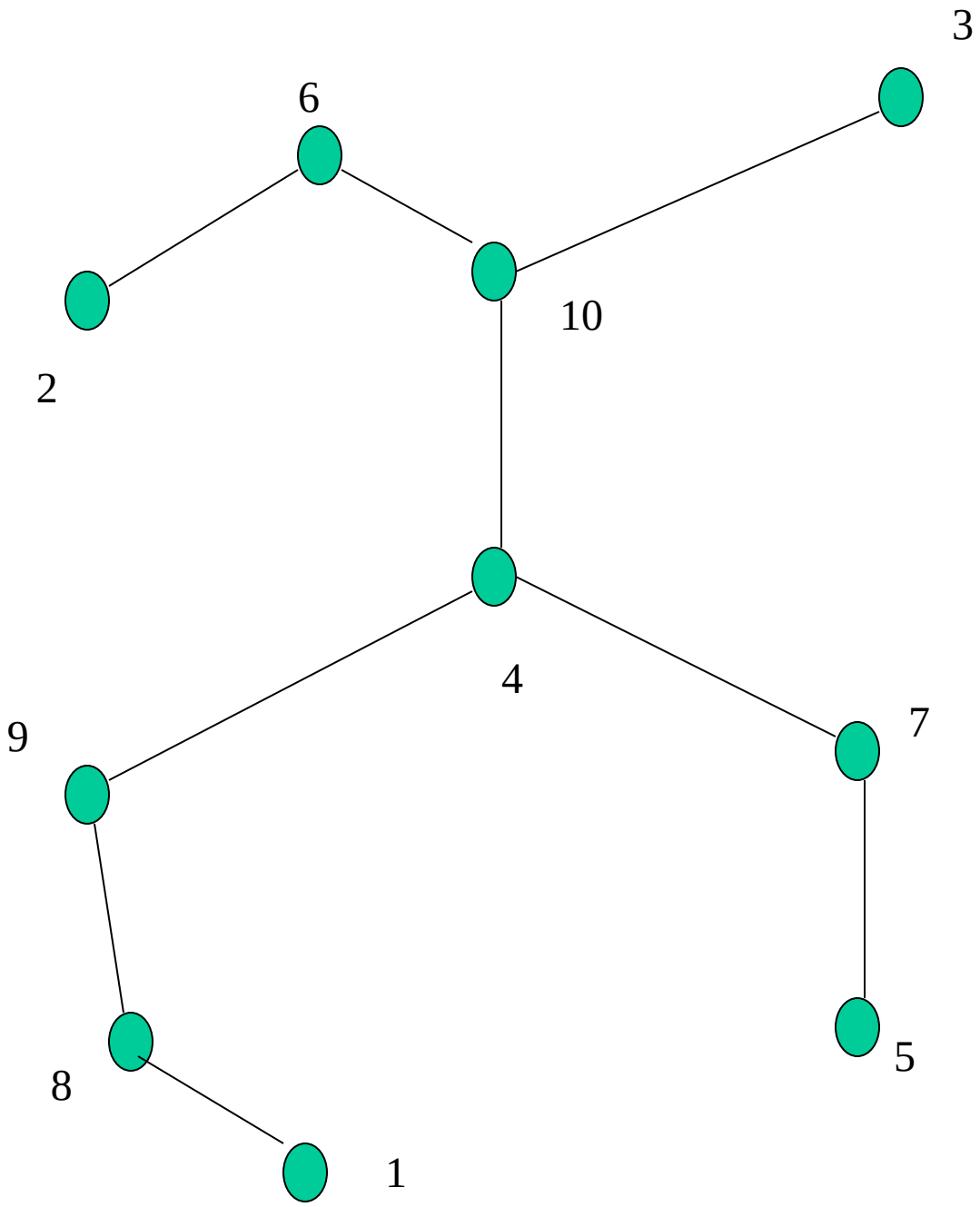
TSP Métrico

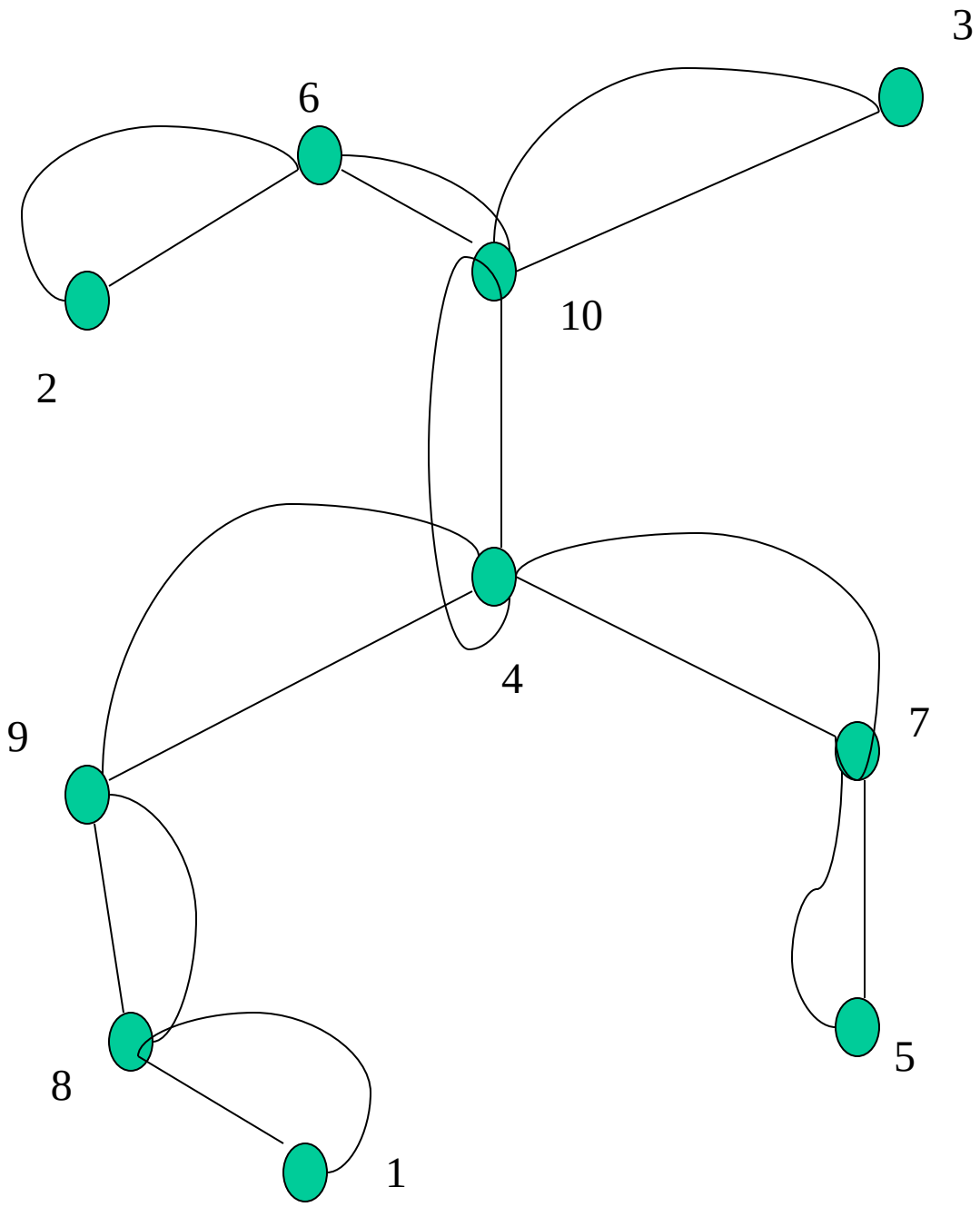
- $d(a,c) \leq d(a,b) + d(b,c)$
- Teorema
- Para cualquier instancia I con m ciudades del TSP métrico
- $NN(I) \leq 1/2 (\log m + 1)OPT(I)$
- más aún, existen instancias para las cuales
- $NN(I) > 1/3(\log(m+1))OPT(I)$

	2	3	4	5	6	7	8	9	10
1	96	105	50	41	86	46	29	56	70
2		78	49	94	21	64	63	41	37
3			60	84	61	54	86	76	51
4				45	35	20	26	17	18
5					80	36	55	59	64
6						46	50	28	8
7							45	37	30
8								21	45
9									25

Heurística MST doble

- Encontrar un MST de G.
- Duplicar cada arista $e \in E$ y construir un ciclo Euleriano Q en el grafo resultante.
- Extraer un tour T en G a partir de Q, suprimiendo las repeticiones de vértices.


3
6
2
10
4
9
7
8
5
1


3
6
10
2
4
9
7
8
5
1

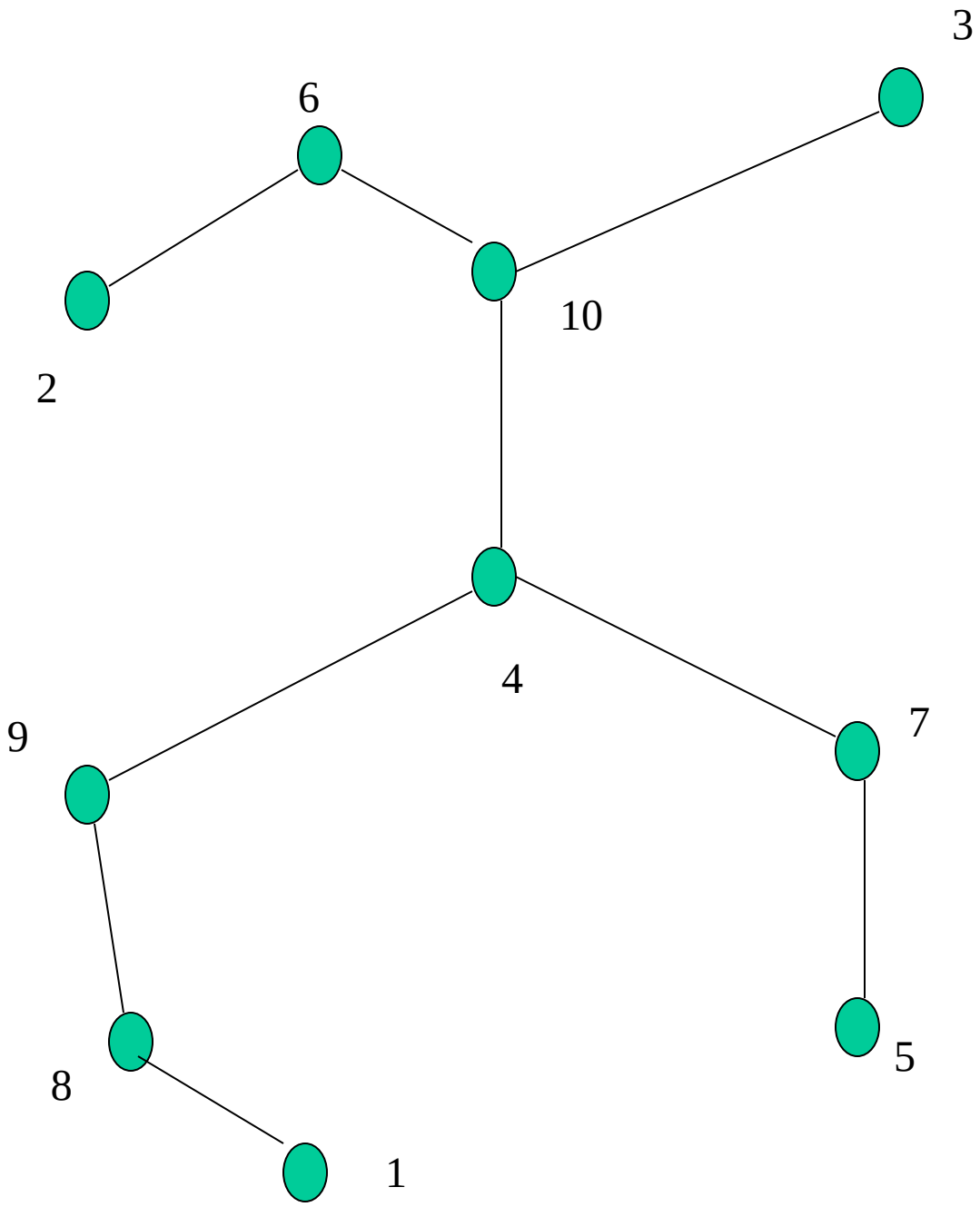
Garantía de Optimalidad

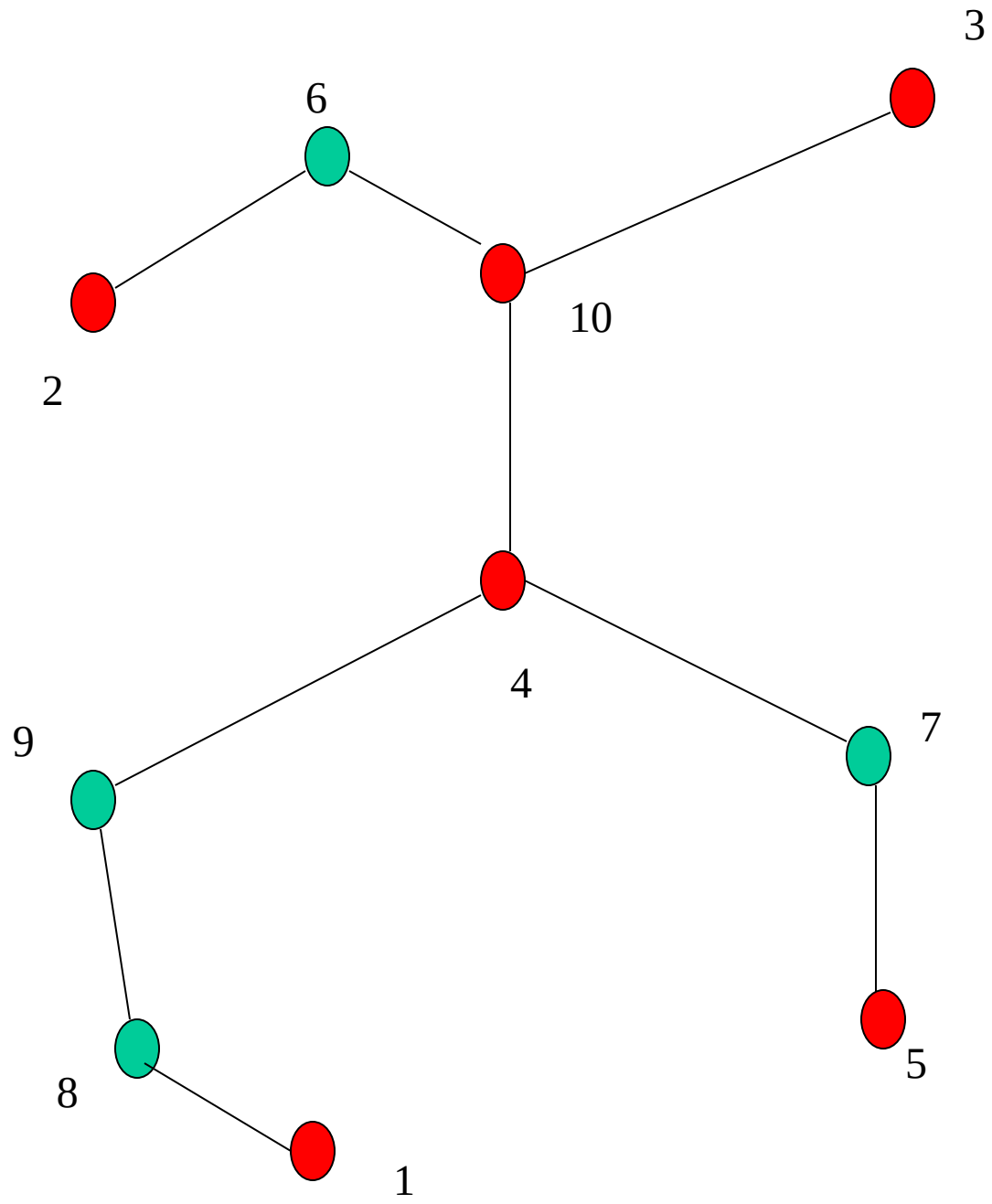
- Si las distancias son no negativas y se satisface la desigualdad del triángulo, entonces, cualquier tour producido por la heurística MST doble tiene una longitud no mayor al doble de la longitud de un tour óptimo.

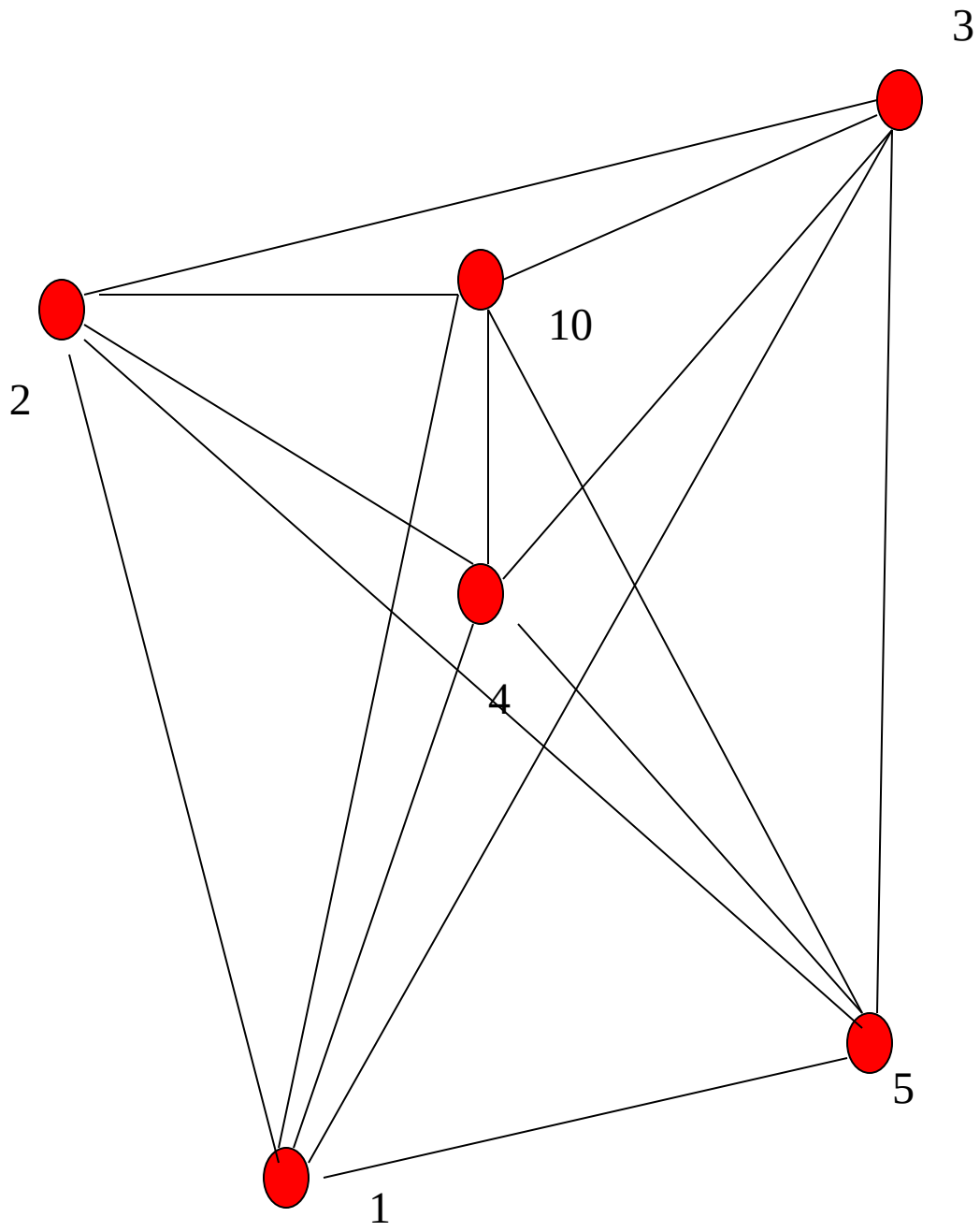
- $\text{MST}(I) \leq 2\ E_T \leq 2\ \text{OPT}(I)$

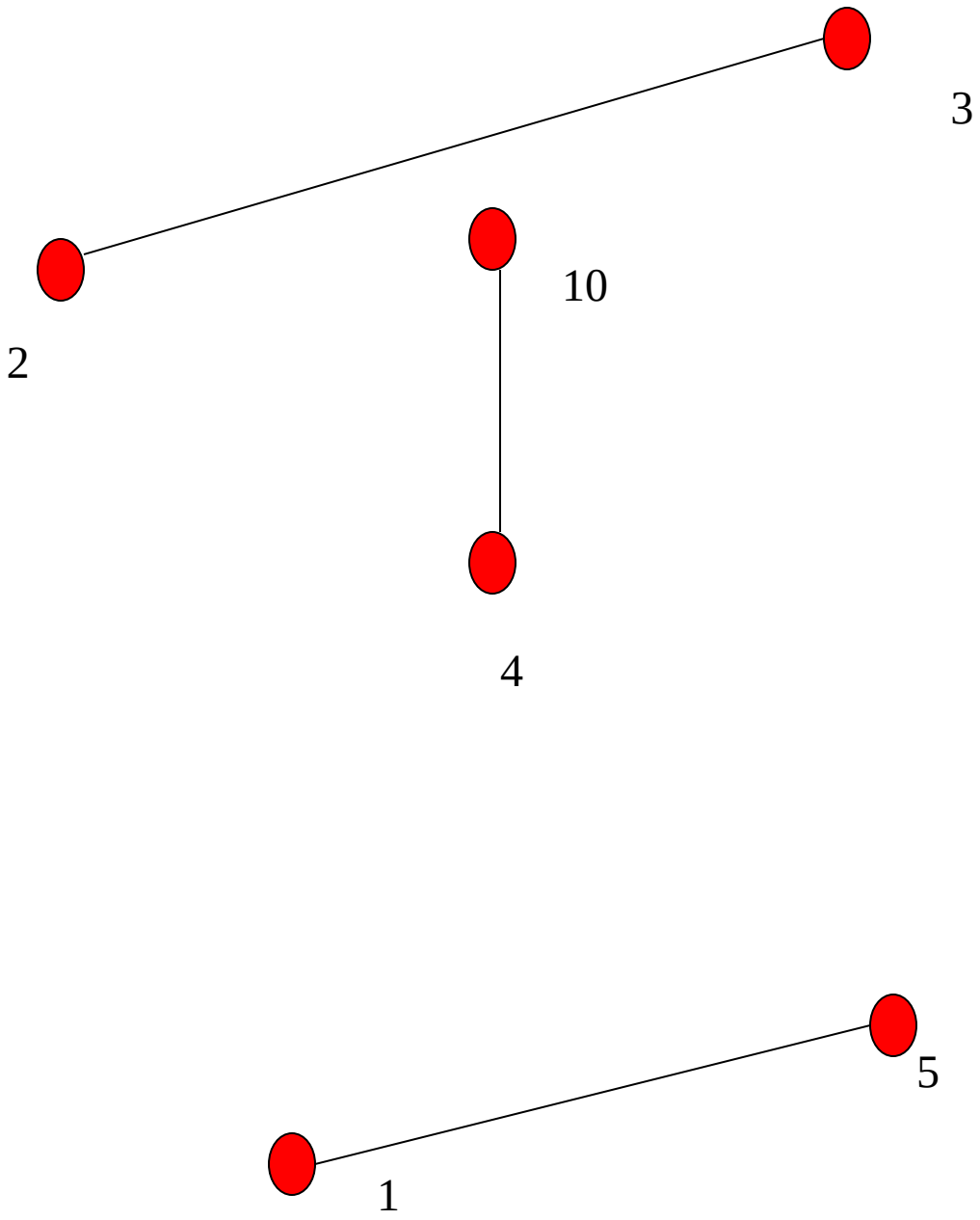
MST-Apareamiento Mínimo

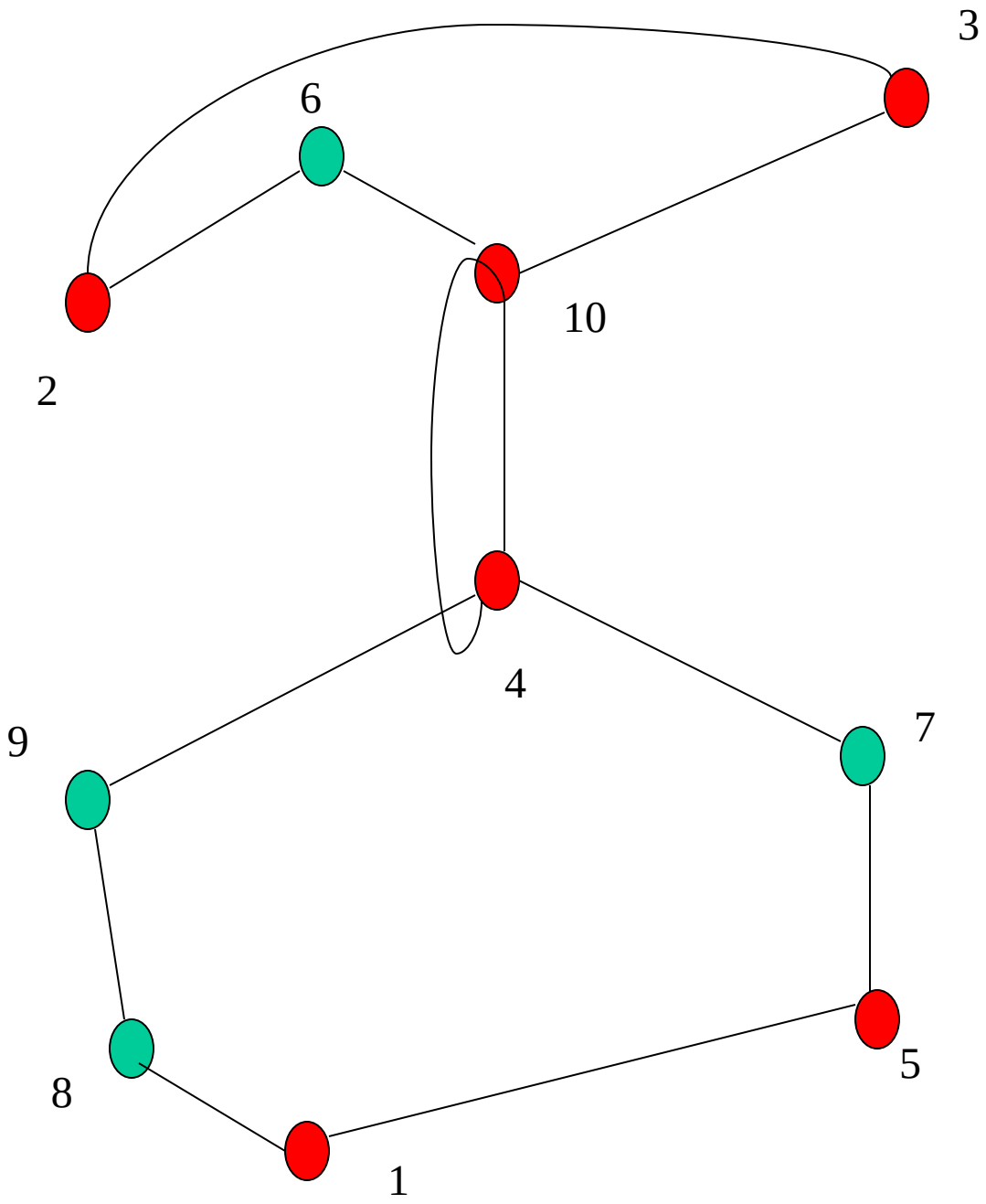
- Construir un MST
- Considerar únicamente aquellos vértices de grado impar en el MST resultante (un número par)
- Construir un apareamiento de peso mínimo
- Construir un grafo Euleriano
- Extraer un tour T en G, suprimiendo las repeticiones de vértices.


3
6
2
10
4
9
7
8
5
1


3
6
10
2
4
9
7
8
5
1


3
10
2
4
5
1


3
10
2
4
5
1


3
6
2
10
4
9
7
8
5
1

Garantía de Optimalidad

- $\mathrm{MM}(I) \leq 1/2\ \mathrm{OPT}(I)$
- $E_T \leq \mathrm{OPT}(I)$
- $\mathrm{MM}(I) + E_T \leq 3/2\ \mathrm{OPT}(I)$
-

Aproximar TSP es difícil

- Si eliminamos la hipótesis de la desigualdad del triángulo ya no se puede dar una garantía de optimalidad

Si existe una K , y un algoritmo polinomial tal que

$$A(I) \leq K\,\mathrm{OPT}(I)$$

entonces $\mathcal{P} = \mathcal{NP}$

- Supongamos que existe un algoritmo polinomial con esta propiedad.
- Entonces este algoritmo nos permite resolver HC en tiempo polinomial.
- Dado un grafo, $G = (V,E)$, construimos un grafo completo G' dando peso de 1 a las aristas originales y peso $K\,n$ a las que no están

- Entonces, si el grafo original contiene un ciclo hamiltoniano, el grafo completo contiene un tour de longitud mínima n.
- Aplicando el algoritmo a este grafo, se obtiene una solución con longitud $\leq K\,n$.
- En otro caso, cualquier tour contiene un arco de longitud $K\,n$, y por lo tanto la solución óptima tiene longitud $> K\,n$.
- Así que el grafo contiene un ciclo hamiltoniano, si y sólo si la solución es menor o igual que $K\,n$.

Paralelismo

- Hasta este punto, nuestro modelo de computación ha sido el de una computadora de propósito general, determinística, acceso random, que realiza una sola operación a la vez.
- Usaremos el término *algoritmo secuencial* para los algoritmos de un paso a la vez que hemos estudiado hasta ahora.

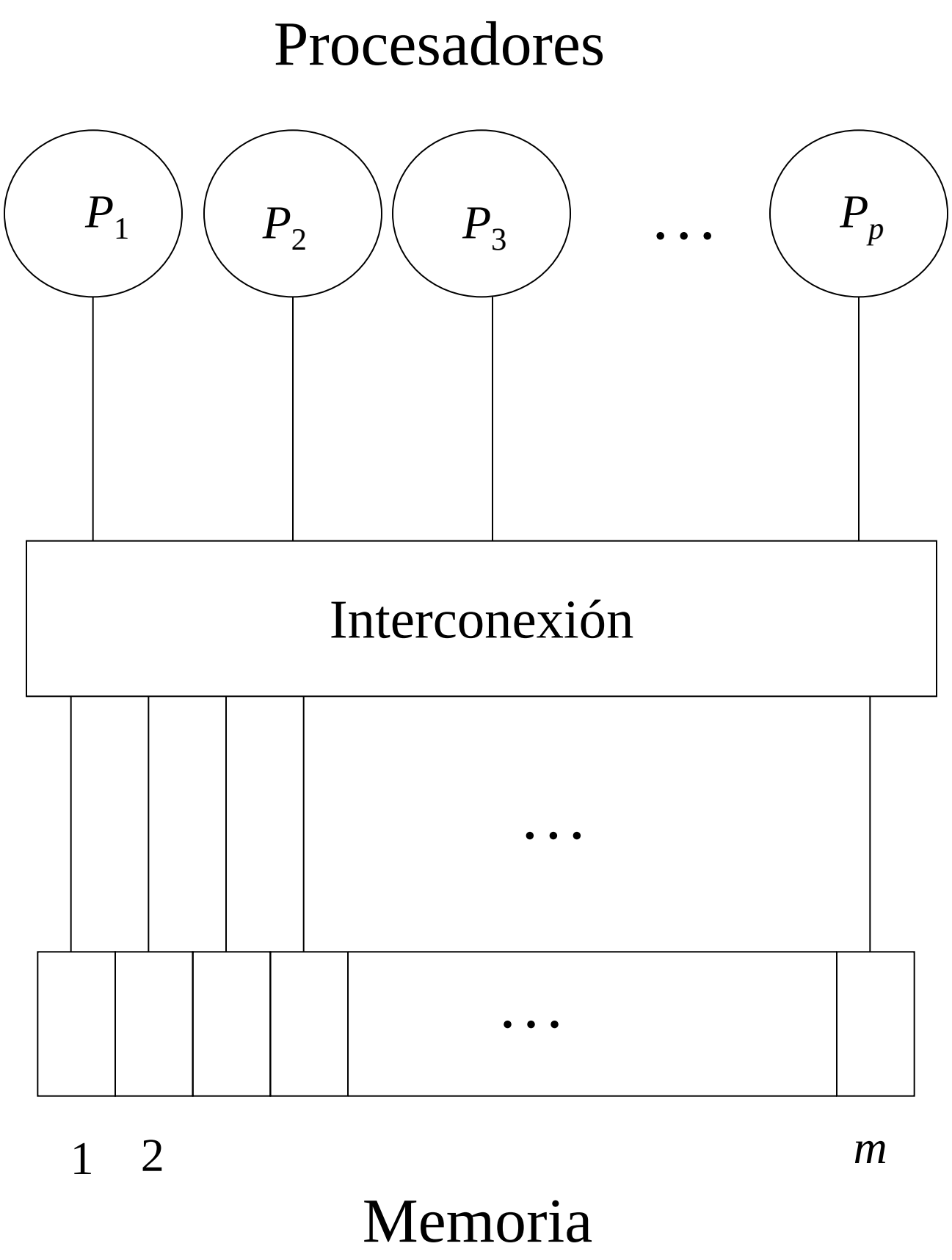
Algoritmos en paralelo

- Varias operaciones pueden ejecutarse simultáneamente en paralelo, esto es, algoritmos para máquinas que tienen más de un procesador trabajando en un problema al mismo tiempo.
- El propósito de este capítulo es introducir algunos de los conceptos, modelos formales, y técnicas para el análisis de algoritmos en paralelo.

Modelo **PRAM**

- **P** arallel
- **R** andom
- **A** ccess
- **M** achine

- p procesadores
- $P_1, P_2, \ldots, P_p$, conectados a una memoria compartida M.
- Cada procesador tiene una memoria local
- Toda la comunicación entre los procesadores se lleva a cabo vía la memoria compartida
- El input se encuentra en las primeras n celdas de la memoria
- El output se escribe en la primera celda
- Todas las celdas de memoria sin input tienen un 0


Procesadores
P_1
P_2
P_3
...
P_p
Interconexión
...
...
1
2
m
Memoria

- Cada paso tiene tres fases:
- a) Lectura
- b) Cálculo
- c) Escritura

- Procesadores PRAM están sincronizados:

  Empiezan cada paso al mismo tiempo; y todos los que tienen que escribir lo hacen al mismo tiempo.

  Cualquier número de procesadores puede leer la misma posición de memoria concurrentemente.

Conflictos en escritura

- Para resolver los conflictos en escritura se han propuesto algunas variaciones:

- CREW (Concurrent Read, Exclusive Write)
- Sólo un procesador puede escribir en una celda particular en un paso dado.

- CRCW (Concurrent Read, Common Write)
- Varios procesadores pueden escribir en una celda particular en un paso dado, siempre y cuando escriban el mismo valor.

- CRPW (Concurrent Read, Priority Write)
- Si varios procesadores tratan de escribir en una celda al mismo tiempo, gana el de menor índice.

Complejidad

- $\mathcal{NC}$

- Clase de problemas que pueden ser resueltos por un algoritmo en paralelo donde el número de procesadores p está acotado por un polinomio en el tamaño del input y el número de pasos acotado por un polinomio en el logaritmo del tamaño del input.
- $p(n) = O(n^k)$
- $T(n) = O((\log n)^m)$

$\mathcal{NC}$

- Nick’s Class
- Propuesto por Stephen Cook en honor de Nick Pippenger, quien fue el primero en proponer esta noción de computación paralela factible.

$\mathcal{NC}$

- Así como $\mathcal{P}$ puede considerarse como la clase de problemas tratables, $\mathcal{NC}$ puede verse como la clase de problemas que pueden ser resueltos eficientemente en una computadora en paralelo.
  - $\mathcal{NC}$ es un subconjunto de $\mathcal{P}$ porque las computadoras en paralelo pueden simularse mediante computadoras secuenciales.

Algunas medidas

- S_p *speed-up ratio* con respecto al mejor algoritmo secuencial

$$S_p = \frac{T(N)}{T_p(N)}$$

- $T(n)$ = tiempo requerido por el mejor algoritmo secuencial conocido
- $T_p(n)$ = tiempo que toma un algoritmo paralelo usando p procesadores

Eficiencia

$$E_p = \frac{S_p}{p}$$

Encontrar el máximo número en un arreglo

- Torneo en paralelo.
- Competir por pares: eliminatoria hasta llegar a las finales donde se elige al ganador.
- $\log_2 n$ competencias en paralelo

- read $M[i]$ into *big*;
- *incr* = 1;
- write -∞ into $M[n + i]$;
- **for** *step* = 1 **to** $\lceil \log n \rceil$

  read $M[i + incr]$ into *temp*;

  big = max (*big*, *temp*);

  incr = 2 * *incr* ;

  write *big* into $M[i]$;
- **end**

	P1	P2	P3	P4	P5	P6	P7	P8
M	16	12	1	17	23	19	4	8
leer	16	12	1	17	23	19	4	8
temp	12	1	17	23	19	4	8	-1000
comp	**16**	12	**17**	23	**23**	19	**8**	8
temp	17	23	23	19	8	8	-1000	-1000
comp	**17**	23	23	23	**23**	19	8	8
temp	23	19	8	8	-1000	-1000	-1000	-1000
comp	**23**	23	23	23	23	19	8	8

- Este algoritmo emplea CREW de manera que no hay conflictos de escritura.

- Modificándolo se puede usar para:
  - encontrar el mínimo de *n* números
  - calcular el **or** y **and** de *n* bits
  - calcular la suma de *n* números

Algoritmos fácilmente paralelizables

- Consideremos el problema de multiplicación de dos matrices A y B.

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \qquad \text{para } 1 \leq i, j \leq n$$

- asignar un procesador a cada elemento del producto, usando n^2 procesadores. Cada procesador P_{ij} calcula su c_{ij} en $2n$ pasos.

Resolución de conflictos de escritura

- Cálculo de la función Booleana **or** de n bits.
- *Input*: Bits $x_1, x_2, \ldots x_n$, en $M[1], M[2],\ldots, M[n]$
- *Output* $x_1 \vee x_2 \vee \ldots \vee x_n$ en $M[1]$
- P_i lee x_i en $M[i]$;
- si $x_i = 1$, entonces P_i escribe 1 en $M[1]$.

Un algoritmo rápido para encontrar Max

- Utilizando CRCW o CRPW podemos acelerar el tiempo de cálculo de Max, aumentando el número de procesadores.
- Se usan $n(n\text{-}1)$ procesadores.
- Estrategia: comparar todos los pares de valores en paralelo, y comunicar los resultados vía la memoria compartida.

- Se usa un arreglo *loser* que ocupa las celdas de memoria $M[n+1], \ldots, M[2n]$. Inicialmente todas las celdas de este arreglo tienen el valor 0. Si x_i pierde una comparación, entonces $loser[i]$ recibe un 1.
- Paso 1

  $P_{i,j}$ lee x_i (de $M[i]$)
- Paso 2

  $P_{i,j}$ lee x_j (de $M[j]$)

  $P_{i,j}$ compara x_i con x_j

  Sea k el índice del menor valor.

  $P_{i,j}$ escribe 1 en $looser[k]$.

- Paso 3

  $P_{i,i+1}$ lee *loser*[*i*] (y $P_{i,n}$ lee *loser*[*n*]);

  Cualquier procesador que lee un cero escribe x_i en *M*[1].

Mezcla de listas en paralelo

- Dos listas
- X = (x_1, x_2,... $x_{n/2}$)
- Y = (y_1, y_2,... $y_{n/2}$)

- *n* procesadores
- Cada procesador se asigna a un elemento de la lista, su misión es localizar la posición de ese elemento en la lista mezclada.

- Un procesador asociado con el elemento x_i en X, realiza una búsqueda binaria en la lista Y y localiza la menor j tal que $x_i < y_j$
- Con esto se determina que:
- x_i es mayor que i-1 elementos en X
- y mayor que j-1 elementos en Y
- su posición en la lista mezclada es $i + j$ -1

Ordenamiento

- *Input* : lista de n elementos en $M[1], ..M[n]$
- *Output*: los n elementos ordenados en orden no decreciente en $M[1], \ldots, M[n]$

for $t = 1$ **to** $\lceil \log n \rceil$ **do**

$k = 2^{t-1}$;

$P_i, \ldots, P_{i+2k-1}$ mezclan las dos listas ordenadas de tamaño k empezando en $M[i]$;

end