

Algoritmos

Computación: Ciencia y Tecnología
del Mundo Digital – IIC1005

Sept. 2019

Yadran Eterovic

... o la ciencia (y el arte) de resolver problemas

... correcta y eficientemente

Algoritmo:

Método finito, determinista y eficaz para resolver problemas

método:

- secuencia de instrucciones que si las llevamos a cabo en orden realizan una tarea particular

finito:

- si llevamos a cabo las instrucciones, entonces, *siempre*, el algoritmo termina después de ejecutar un número finito de instrucciones
- además, el tiempo para terminar debería ser relativamente corto

determinista:

- cada instrucción es clara, no ambigua

eficaz:

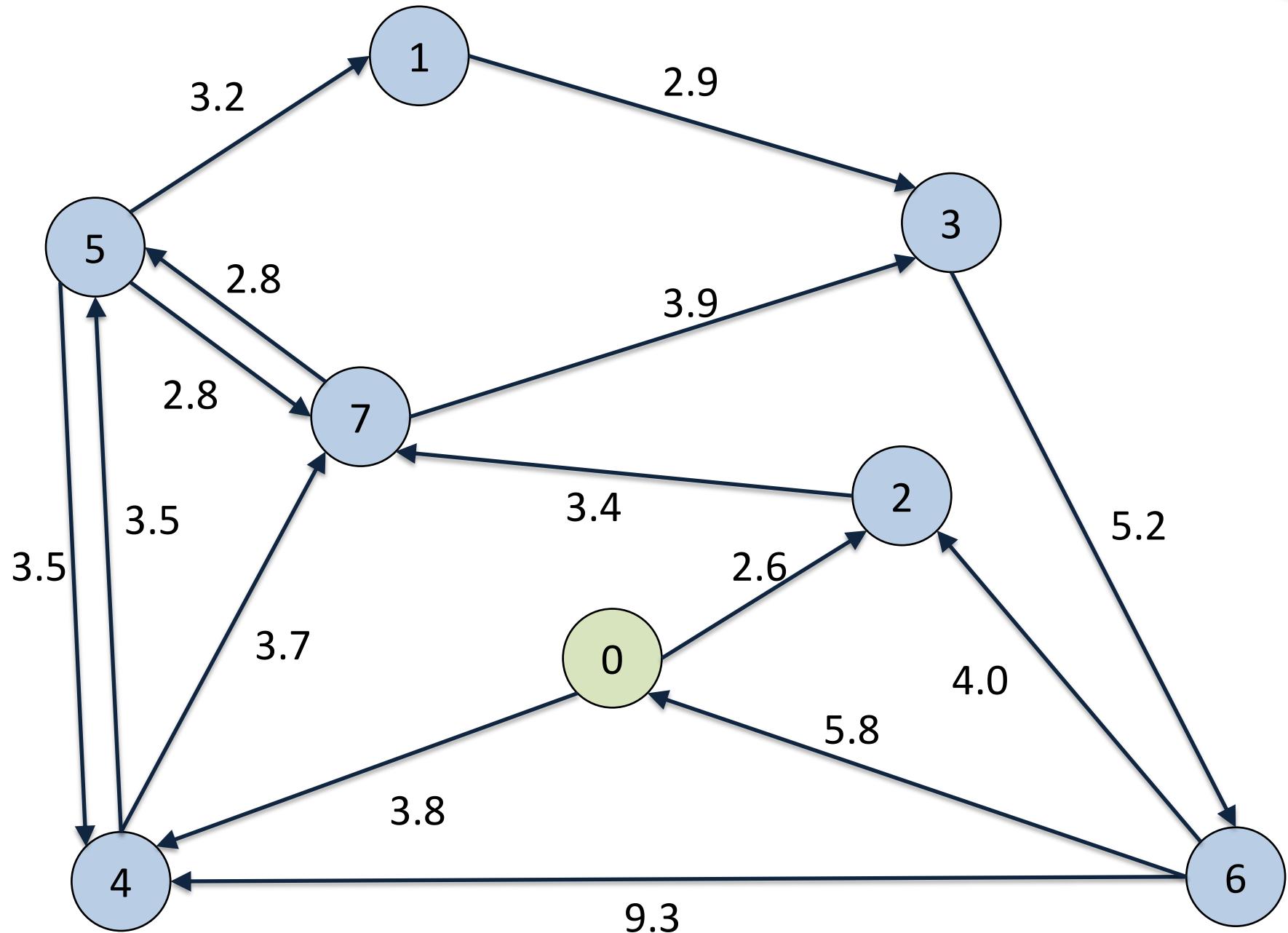
- cada instrucción es suficientemente básica —puede ser llevada a cabo, en principio, por una persona usando solo papel y lápiz

P.ej., la bomba de incendios de una comuna quiere saber cuáles son los caminos más rápidos para llegar a distintos puntos claves de la comuna —colegios, hospitales, centros comerciales, cines, etc.— en caso de incendio u otras emergencias

Para esto, numeró los puntos del 1 al n (la bomba es el punto 0)

... y midió los tiempos de viaje, en minutos, entre pares de puntos conectados por caminos en buen estado

... obteniendo los resultados que se muestran en la figura de la próxima diapositiva (para $n = 7$)



6

Los bomberos quieren saber, p.ej., si para ir al punto 7 les conviene hacerlo pasando por el punto 2 o pasando por el 4

... o si para ir al punto 3 les conviene primero llegar al 7 o primero llegar al 1

Los algoritmos deben ser inventados, validados y analizados

Inventado:

- posiblemente usando alguna de las técnicas de diseño de algoritmos que han demostrado ser útiles —*backtracking*, dividir para conquistar, algoritmos codiciosos, programación dinámica

Validado:

- hay que *demostrar* que calcula la respuesta correcta para todos los *inputs* legales posibles —por inducción, por contradicción

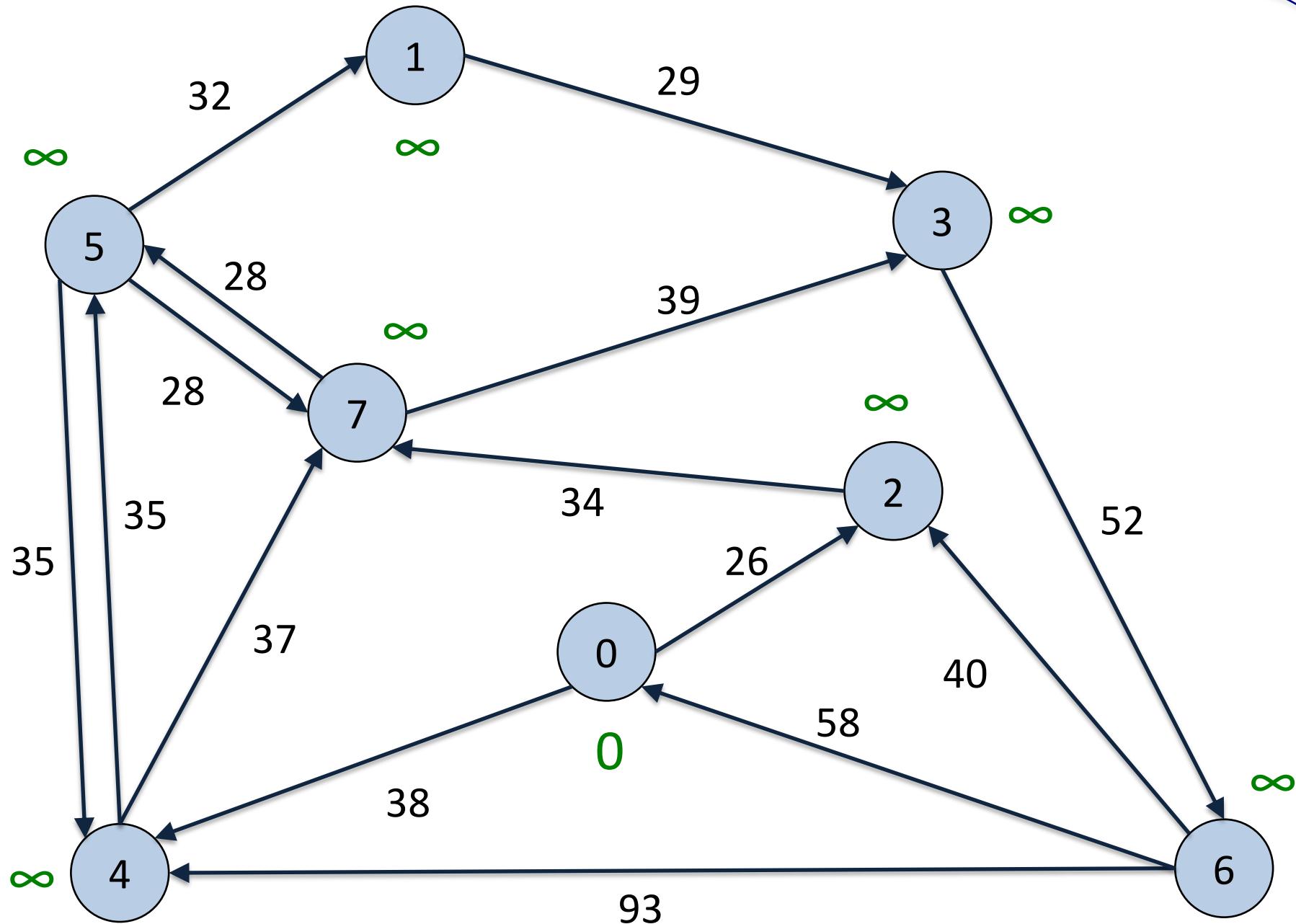
Analizado:

- determinar cuánto tiempo de computación y cuánta memoria necesita

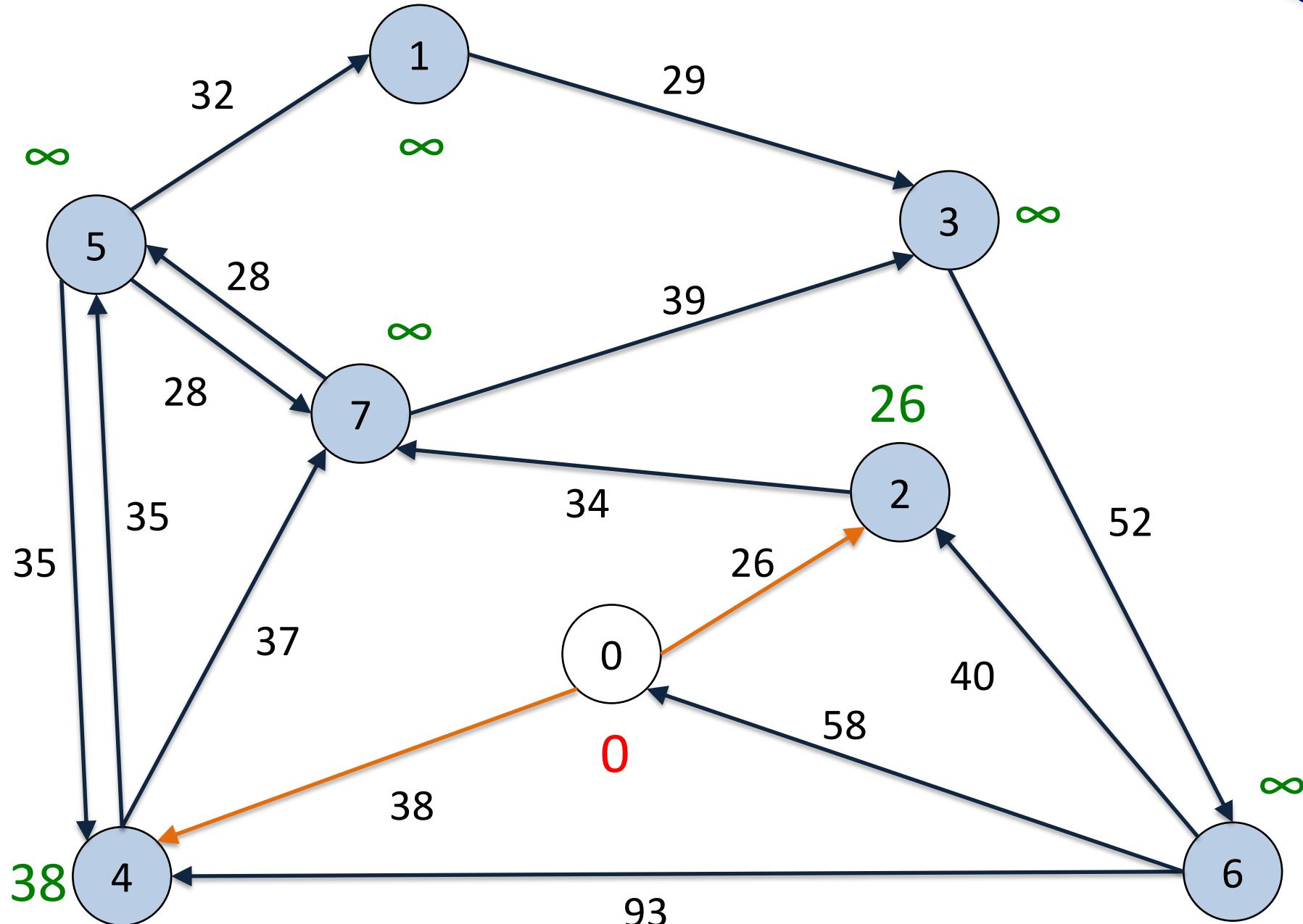
El algoritmo de Dijkstra para encontrar los caminos más cortos (o más rápidos) desde un punto, o *nodo*, a todos los otros nodos

... es de tipo codicioso

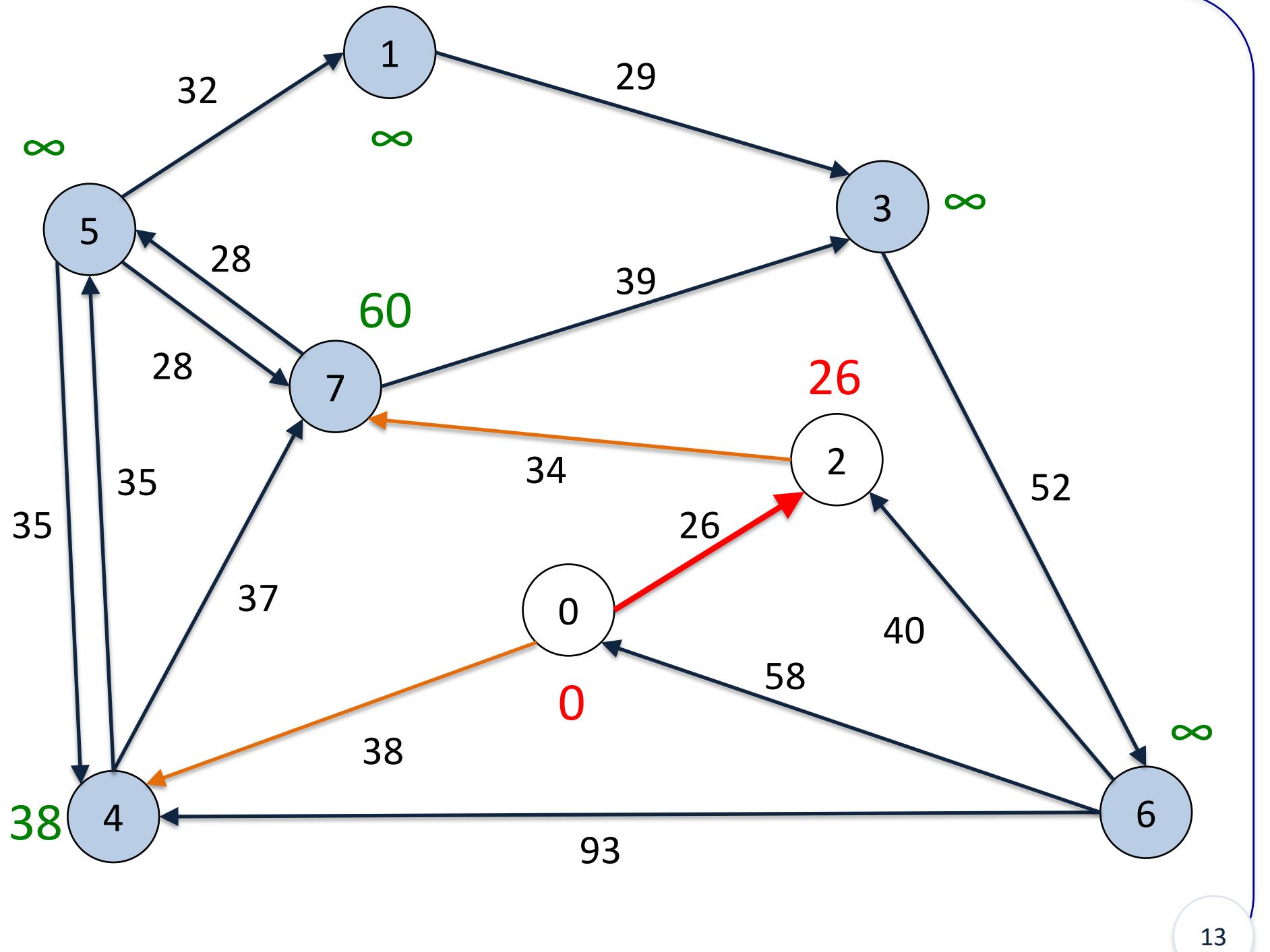
... y (se puede demostrar que) efectivamente resuelve el problema si todas las distancias (o todos los tiempos de viaje) son no negativas



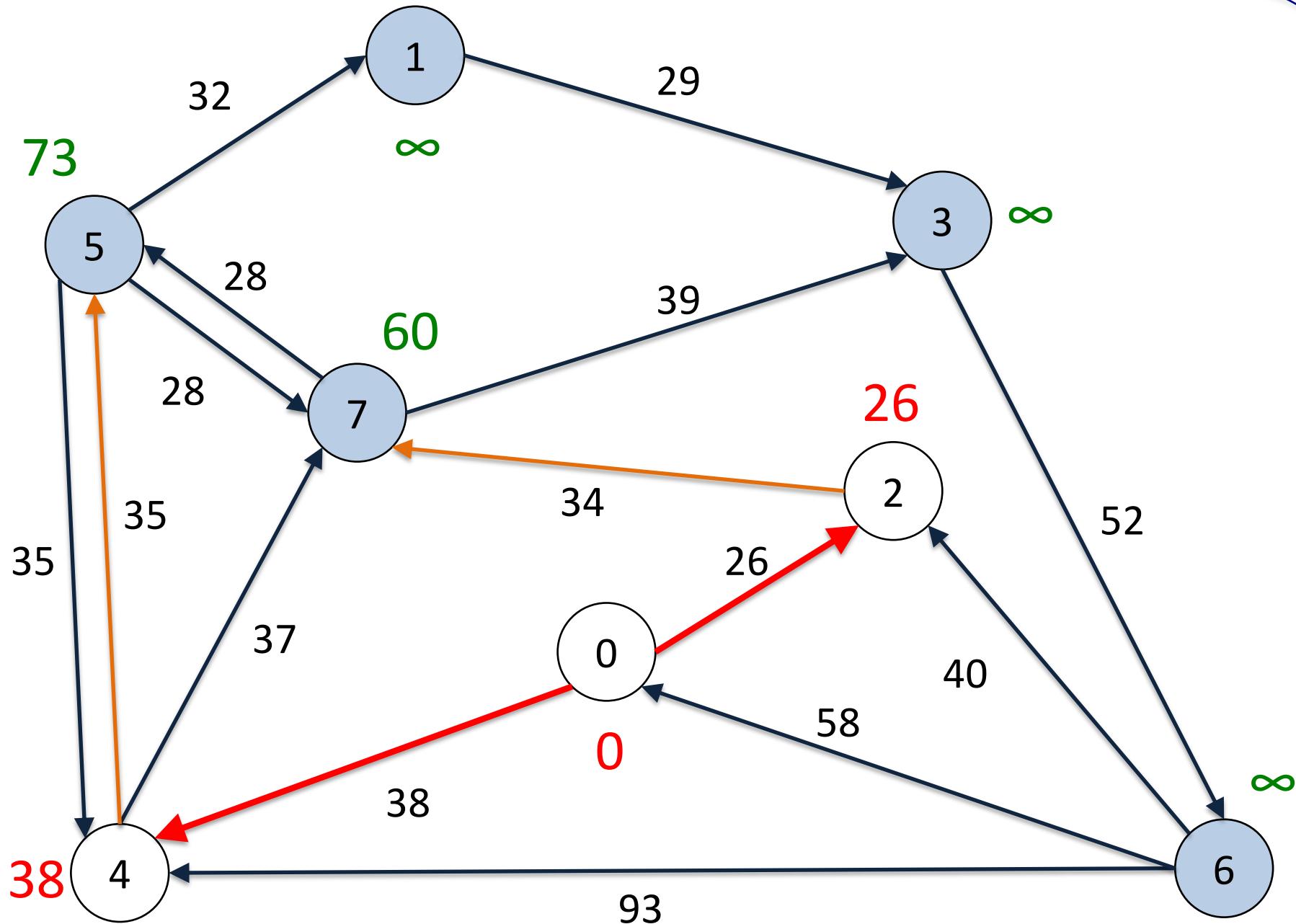
11



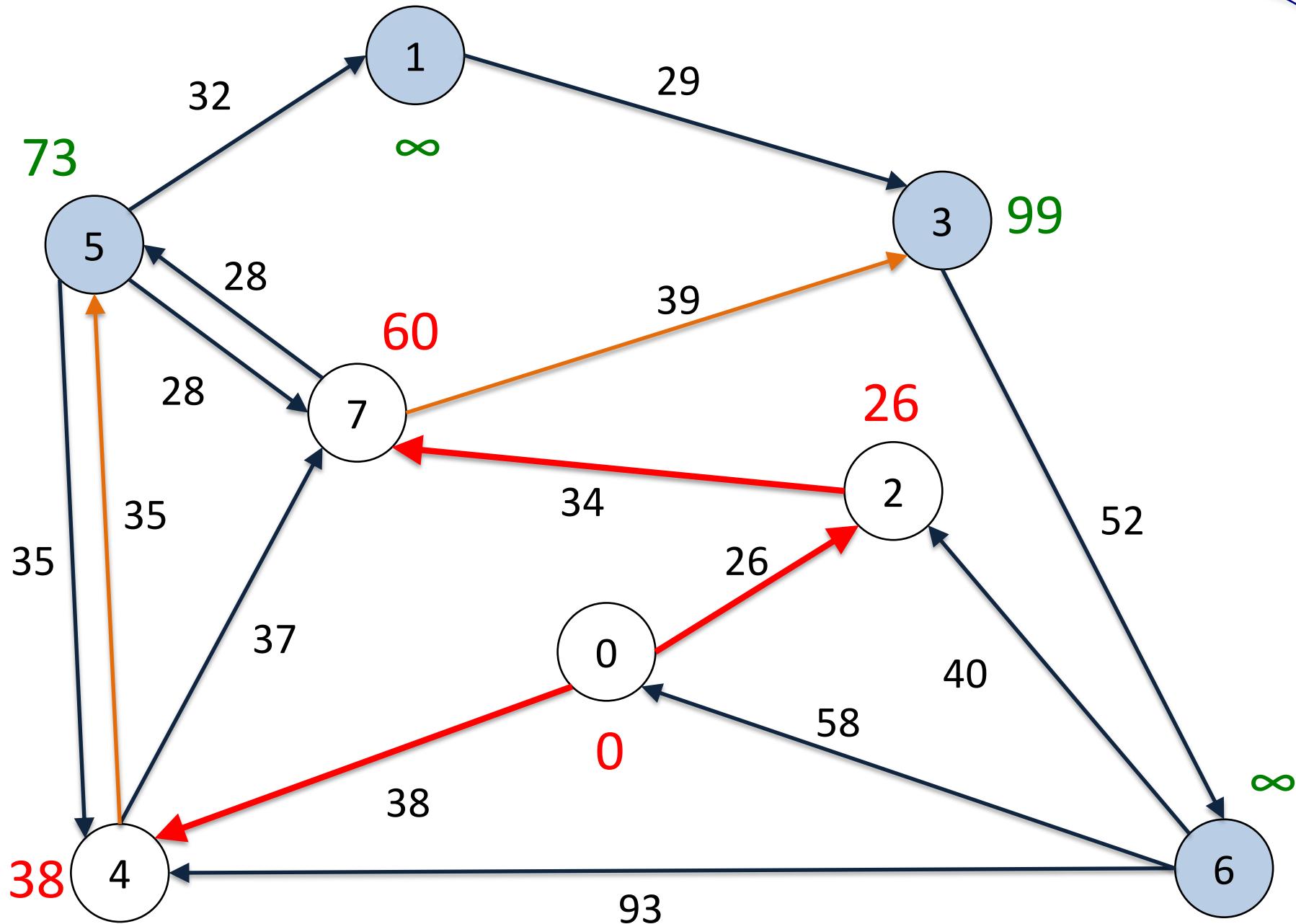
12



13

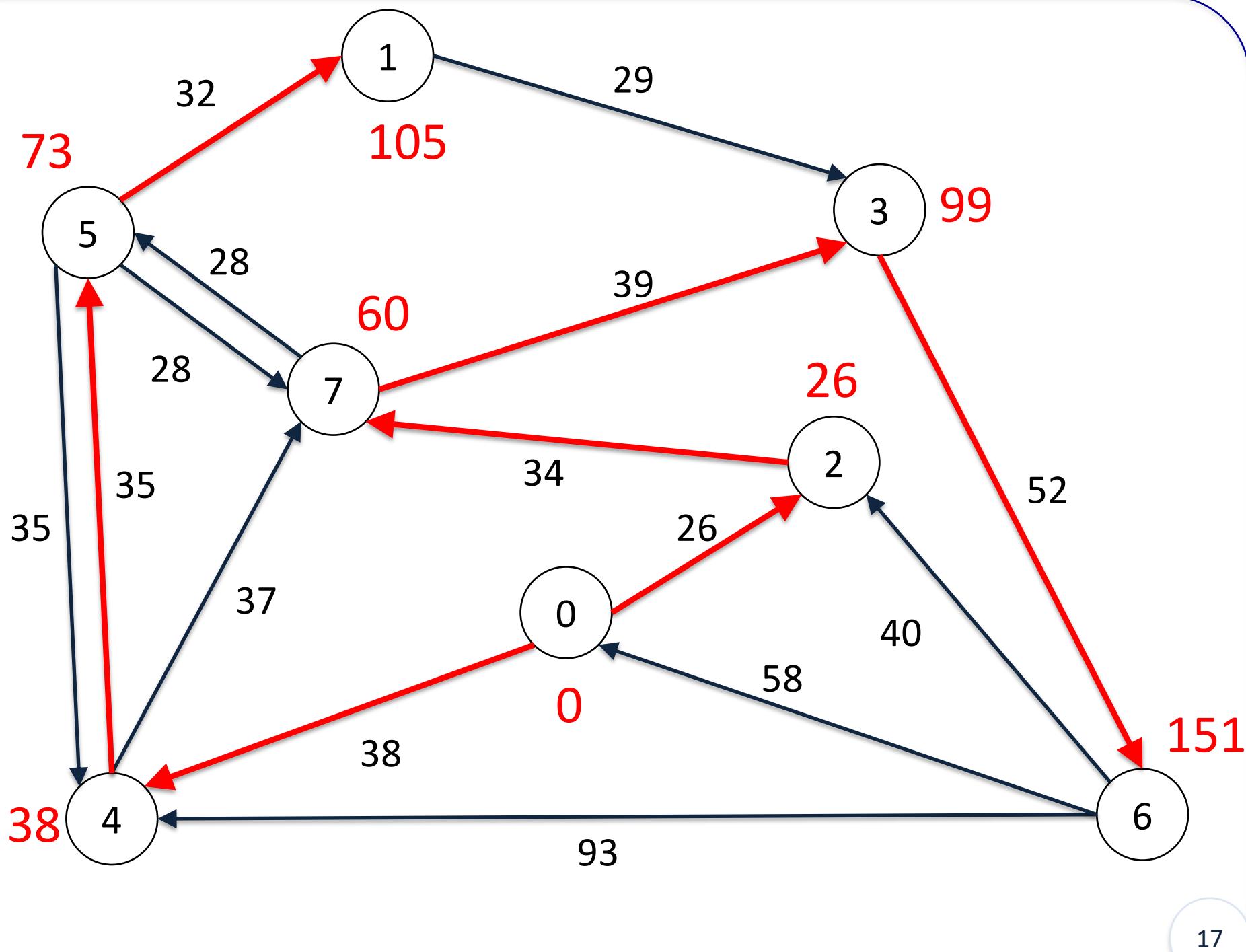


14



15

• • •



**Analizar un algoritmo es
predecir razonadamente los recursos que el algoritmo necesita**

Principalmente, tiempo de computación:

- a veces, memoria, ancho de banda de comunicación, accesos al disco

... en un computador (relativamente) convencional:

- las instrucciones son ejecutadas una después de la otra
 - ... operaciones aritméticas comunes, movimientos simples de datos, e instrucciones de control
 - ... cada una toma una cantidad de tiempo constante
- los tipos de datos son *entero* y *punto flotante*
- no consideramos *caches* ni memoria virtual

Tiempo de computación:

Número de pasos, u operaciones primitivas, ejecutados

Definimos “paso” de modo que sea lo más independiente posible del computador:

- un segmento sintáctica o semánticamente significativo de un programa, cuyo tiempo de ejecución es independiente de las características del problema particular
- se necesita una cantidad de tiempo constante para ejecutar cada línea de nuestros programas (los de esta presentación)

Orden de crecimiento y la notación O()

Al determinar el tiempo de computación de un algoritmo, ya sea para el peor caso o el caso promedio, no nos interesa tanto la fórmula “exacta”, sino poder acotar superiormente esa fórmula

Si $T(n)$ representa la fórmula exacta

... y $f(n)$ representa una fórmula más simple

... decimos que $T(n) = O(f(n))$ —se lee “ $T(n)$ es O de $f(n)$ ”

... si existen constantes positivas c y n_0 tales que

... $T(n) \leq cf(n)$ cuando $n \geq n_0$

Órdenes de crecimiento comunes en función del tamaño n del problema

1	instrucción o paso	sumar dos números
$\log n$	dividir por la mitad	búsqueda binaria
n	<i>loop</i>	encontrar el máximo
$n \log n$	dividir para conquistar	<i>mergeSort</i>
n^2	<i>loop double</i>	revisar todos los pares
n^3	<i>loop triple</i>	revisar todos los tríos
2^n	búsqueda exhaustiva	revisar todos los subconjuntos
$n!$	búsqueda exhaustiva	revisar todas las permutaciones

La ventaja de pasar de un algoritmo $O(n^2)$ a uno $O(n\log n)$ para el mismo problema

... es que ahora puedes resolver el problema para inputs de tamaño mucho más grande

... aproximadamente $n/\log n$ veces más grande

Tiempos de ejecución (en segundos) de varios algoritmos —con distintos órdenes de crecimiento— para un mismo problema

n	$O(n^3)$	$O(n^2)$	$O(n \log n)$	$O(n)$
100	0.00016	0.000006	0.000005	0.000002
1,000	0.096	0.00037	0.00006	0.00002
10,000	86.67	0.0333	0.00062	0.00022
100,000	NA	3.33	0.0067	0.0022
1,000,000	NA	NA	0.075	0.023

Todo lo anterior está muy bien ... sin embargo ...

Queremos poder resolver problemas muy grandes más rápidamente de lo que la CPU más rápida permite

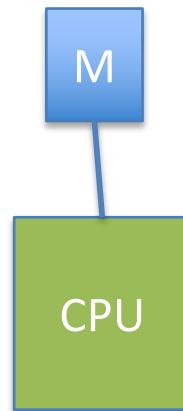
Nuestra única opción es tratar de dividir el problema en varios subproblemas

... resolver cada uno de los subproblemas en una CPU diferente en paralelo —posiblemente comunicándose una CPU con otra de vez en cuando

... y finalmente encontrar la solución al problema original combinando las soluciones a los subproblemas

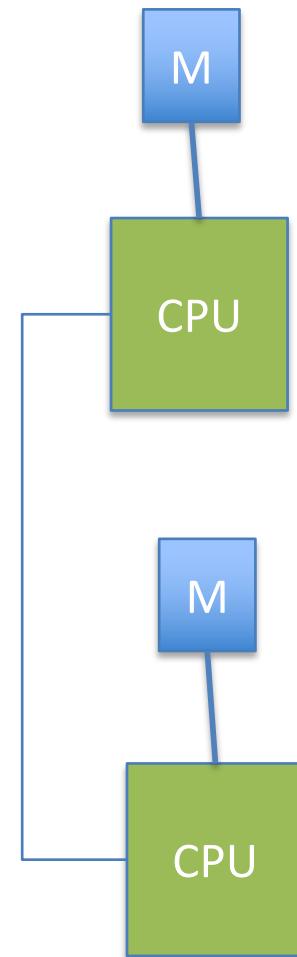
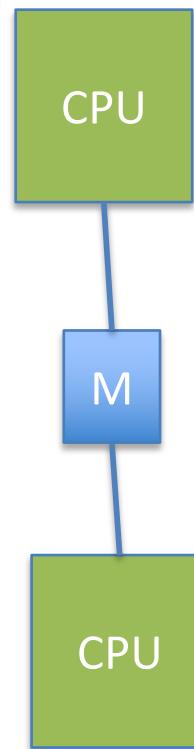
Siglo XX (1960s a 1990s):

- para un procesador, o CPU
- cada vez con más memoria disponible —desde kilobytes hasta megabytes (y, hoy, gigabytes)



Siglo XXI:

- para múltiples CPUs
 - ... ya sea, con memoria compartida (por las múltiples CPUs)
 - ... o con memoria distribuida, en que cada CPU tiene su propia memoria
- se habla de *algoritmos concurrentes, distribuidos, paralelos*



Cuando la ejecución de un algoritmo es dividida entre múltiples CPUs
—llamamos *proceso* al código ejecutado en cada CPU—

... aparecen dos temas nuevos de los cuales preocuparse*:

- comunicación —los procesos necesitan compartir información
- sincronización —los procesos necesitan ponerse de acuerdo

*recordemos que todos los procesos están colaborando para resolver el mismo problema

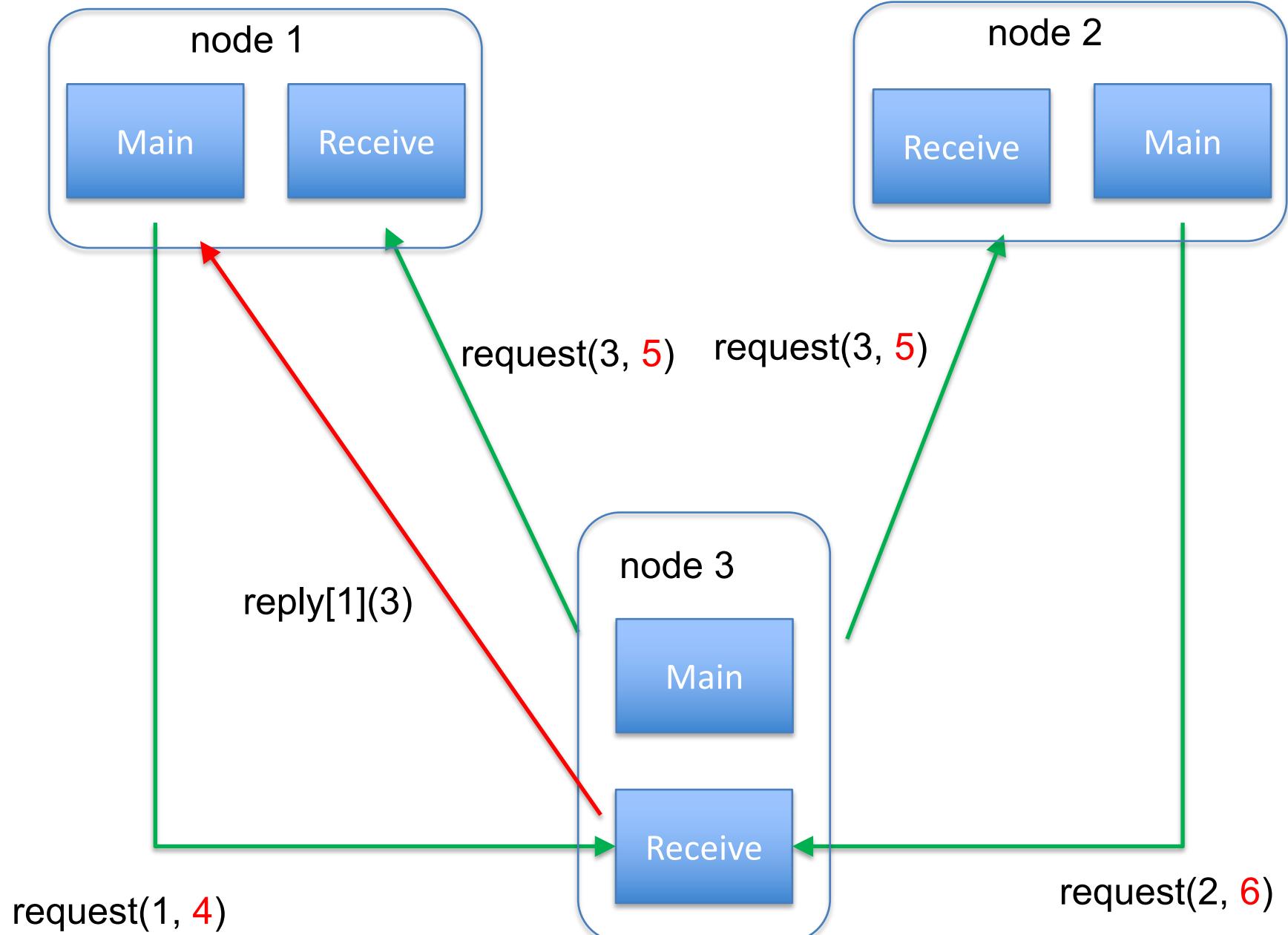
P.ej., en un sistema con memoria distribuida —independientemente del problema particular que estén tratando de resolver los procesos— cada proceso de vez en cuando necesita escribir información en una base de datos común

... lo cual debe hacerse bajo *exclusión mutua* —solo un proceso al mismo tiempo puede estar escribiendo en la base de datos

→ si dos o más procesos quieren escribir en la base de datos en un momento determinado

... entonces estos procesos necesitan ponerse de acuerdo para decidir cuál va a escribir primero, cuál va a escribir a continuación, etc.

→ versión distribuida del algoritmo de la panadería, de Ricart y Agrawala



Cuando el nodo **myID** quiere escribir en la base de datos:

- primero, define su propio turno, **myNumber**, como uno más que el más grande de todos los turnos que ha “visto”
- luego, envía una solicitud de permiso a cada uno de los otros nodos y se queda esperando a que todos respondan dando sus permisos
- solo cuando ha recibido todas las respuestas, escribe en la base de datos

Por otra parte, cuando el nodo **myID** recibe una solicitud de permiso de otro nodo, **nodeID**, compara el turno de **nodeID** con el propio:

- si el turno de **nodeID** es menor que el propio —**requestNumber << myNumber**— entonces le responde inmediatamente dándole el permiso
- en caso contrario, pospone la respuesta y agrega el número del otro nodo a una cola local, **deferred**

Finalmente, cuando el nodo **myIDF** termina de escribir, le responde a todos los nodos que están en su cola **deferred**

Código ejecutado
por los procesos
Main y **Receive** en
el nodo **myID**

```
int myNumber = 0, highest = 0
set deferred = Ø
bool requestCS = false
process Main:
    while (true):
        NCS
        requestCS = true
        myNumber = highest + 1
        for (all other nodes i)
            send request[i](myID, myNumber)
        await replies from all other nodes
        escribe en la base de datos
        requestCS = false
        for (all nodes in deferred)
            nodeID = deferred.remove()
            send reply[nodeID](myID)

process Receive:
    int nodeID, requestNumber
    while (true):
        receive request[myID](nodeID, requestNumber)
        highest = max(highest, requestNumber)
        if (!requestCS || requestNumber << myNumber)
            send reply[nodeID](myID)
        else
            deferred.insert(nodeID)
```

Bibliografía

M. Ben Ari, *Principles of Concurrent and Distributed Programming* (2nd ed.), Addison Wesley, 2006

T. Cormen, C. Leiserson, R. Rivest, C. Stein, *Introduction to Algorithms* (3rd ed.), MIT Press, 2009

E. Horowitz, S. Sahni, S. Rajasekeran, *Computer Algorithms* (2nd ed.), Silicon Press 2008

R. Sedgewick, K. Wayne, *Algorithms* (4th ed.), Addison-Wesley 2011

M. Weiss, *Data Structures and Algorithm Analysis in C++* (4th ed.), Pearson 2013