

Tempo di esecuzione
e
Analisi asintotica

Corso di Algoritmi

Progettazione e analisi di algoritmi

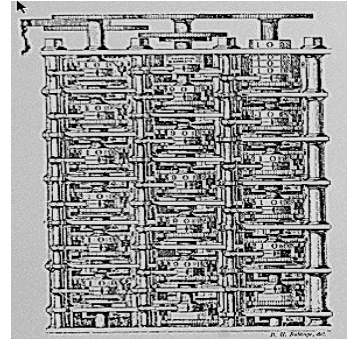
- **Progettazione:** tecnica Divide-et-impera, Greedy, Programmazione dinamica
- **Analisi «asintotica»** delle risorse utilizzate:
spazio e tempo.

Computational Tractability

As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise - By what course of calculation can these results be arrived at by the machine in the shortest time? - *Charles Babbage*



Charles Babbage (1864)



Analytic Engine (schematic)

Tempo di esecuzione

Cosa significa?

Esempio: ricerca del massimo fra n numeri a_1, \dots, a_n .

Un algoritmo:

```
max ← a1
for i = 2 to n {
    if (ai > max)
        max ← ai
}
```

Quale il tempo di esecuzione?

- Numero di secondi?
- Implementato con quale struttura dati, linguaggio, macchina, compilatore.....?
- Su quanti numeri? 100, 1.000, 1.000.000?

Analisi asintotica

Vogliamo analizzare l'efficienza dell'algoritmo

- Indipendentemente da implementazione, hardware etc.
- Al crescere della taglia dell'input

Studieremo il tempo in funzione della taglia dell'input : $T(n)$

Studieremo la crescita della funzione $T(n)$ al crescere di n

Analisi «asintotica»:

- per n arbitrariamente grande
- per n che tende a infinito
- da un certo punto in poi
- per ogni $n \geq n_0$

Vantaggi dell'analisi asintotica

- **Indipendente** da hardware
- Effettuabile con pseudocodice **prima** di implementare l'algoritmo
- Considera **infiniti** input

Alternativa? Analisi su dati campione.

Svantaggi: bisogna avere **già** implementato l'algoritmo;
analizza numero **finito** di dati

Casi di interesse

Esempio: problema dell'ordinamento

INPUT: n numeri a_1, \dots, a_n

OUTPUT: permutazione dei numeri in cui ogni numero sia minore del successivo

Esistono svariati algoritmi che lo risolvono

Qual è il tempo di esecuzione per ordinare **n** elementi con un fissato algoritmo (per esempio InsertionSort)?

Può dipendere dalla **distribuzione** dei numeri fra di loro

(es.: sono già ordinati, sono ordinati in senso inverso, sono tutti uguali, etc.)

Caso peggiore, migliore, medio

Analisi del **caso peggiore**: qualunque sia la distribuzione dell'input $T(n)$ è limitata superiormente da $f(n)$

Analisi del **caso migliore**: qualunque sia la distribuzione dell'input $T(n)$ è limitata inferiormente da $g(n)$ (poco significativa)

Analisi del **caso medio**: nel caso di una distribuzione media o random (difficile da determinare)

Worst-Case Analysis

- Worst case running time. Obtain bound on **largest possible** running time of algorithm on input of a given size N .
 - Generally captures efficiency in practice.
 - Draconian view, but hard to find effective alternative.
- Average case running time. Obtain bound on running time of algorithm on **random** input as a function of input size N .
 - Hard (or impossible) to accurately model real instances by random distributions.
 - Algorithm tuned for a certain distribution may perform poorly on other inputs.

Quando un algoritmo può essere considerato efficiente?

Polynomial-Time

- Brute force. For many non-trivial problems, there is a natural brute force search algorithm that checks every possible solution.
 - Typically takes 2^N time or worse for inputs of size N .
 - Unacceptable in practice.
- Desirable scaling property. When the input size doubles, the algorithm should only slow down by some constant factor C .

Interval scheduling

There exists constants $c > 0$ and $d > 0$ such that on every input of size N , its running time is bounded by $c N^d$ steps.

- Def. An algorithm is **poly-time** if the above scaling property holds.

choose $C = 2^d$

Worst-Case Polynomial-Time

- Def. An algorithm is **efficient** if its running time is polynomial.
- Justification: **It really works in practice!**
 - Although $6.02 \times 10^{23} \times N^{20}$ is technically poly-time, it would be useless in practice.
 - In practice, the poly-time algorithms that people develop almost always have low constants and low exponents.
 - Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.
- Exceptions.
 - Some poly-time algorithms do have high constants and/or exponents, and are useless in practice.
 - Some exponential-time (or worse) algorithms are widely used because the worst-case instances seem to be rare.



simplex method

Unix grep

Why It Matters

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1,000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10,000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100,000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1,000,000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Efficiency = polynomial

Polynomial-time solvability emerged as a formal notion of efficiency by a gradual process, motivated by the work of a number of researchers including Cobham, Rabin, Edmonds, Hartmanis, and Stearns.

Similarly, the use of asymptotic order or growth notation to bound the running time of algorithms—as opposed to working out exact formulas with leading coefficients and lower-order terms—is a modeling decision that was quite non-obvious at the time it was introduced;

Tempo di esecuzione

Esempio: algoritmo per la ricerca del massimo fra n numeri a_1, \dots, a_n

```
1.  max ← a1
2.  for i = 2 to n {
3.      if (ai > max)
4.          max ← ai
    }
```

Taglia dell'input = n

Tempo di un **assegnamento** = c_1 (costante, non dipende da n)

Tempo di un incremento = c_1

Tempo di un **confronto** = c_2

Linea 1: c_1

Linea 2: $n (c_1 + c_2)$

Linea 3 – 4 (una esecuzione): $\leq c_2 + c_1$

$$T(n) \leq c_1 + n (c_1 + c_2) + (n-1) (c_2 + c_1) = 2(c_1 + c_2) n - c_2 = A n + B$$

dove A e B sono costanti non quantificabili a priori (dipendono dall'implementazione)

Operazioni elementari

Assegnamento, incremento, confronto sono considerate **operazioni elementari** all'interno dell'algoritmo della ricerca del massimo.

Richiedono tempo **costante** (= non dipendente dalla taglia n dell'input) ma a priori non quantificabile

Tempo di esecuzione $T(n)$ sarà misurato in termini del **numero di operazioni elementari** per eseguire l'algoritmo su un input di taglia n

Notazioni asintotiche

Nell'analisi asintotica analizziamo $T(n)$

1. A meno di costanti moltiplicative (perché non quantificabili)
2. Asintoticamente (per considerare input di taglia arbitrariamente grande)

Le notazioni asintotiche:

O, Ω, Θ

ci permetteranno il confronto tra funzioni, mantenendo queste caratteristiche

Diremo per esempio che l'algoritmo per la ricerca del massimo ha un tempo di esecuzione lineare, $T(n) = O(n)$, essendo $T(n) \leq An + B = \Theta(n)$

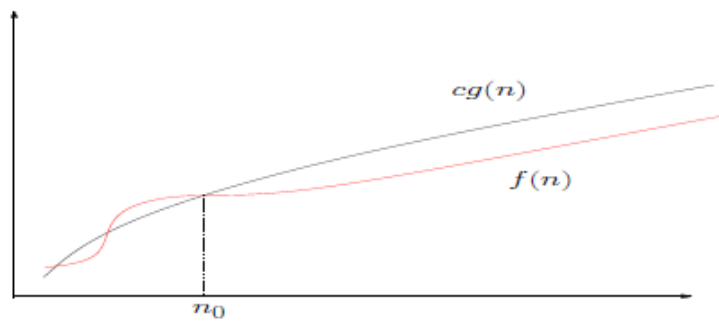
Notazioni Asintotiche: notazione O

Date $f : n \in N \rightarrow f(n) \in R_+$, $g : n \in N \rightarrow g(n) \in R_+$,
scriveremo

$$f(n) = O(g(n))$$

$\Leftrightarrow \exists c > 0, \exists n_0$ tale che $f(n) \leq cg(n), \forall n \geq n_0$

Informalmente, $f(n) = O(g(n))$ se $f(n)$ **non** cresce più
velocemente di $g(n)$. Graficamente



Esempi

● $10n^3 + 2n^2 + 7 = O(n^3)$

Occorre provare che

$$\exists c, n_0 : 10n^3 + 2n^2 + 7 \leq cn^3, \forall n \geq n_0$$

Si ha: $10n^3 + 2n^2 + 7 \leq 10n^3 + 2n^3 + 7$
 $\leq 10n^3 + 2n^3 + n^3 = 13n^3, \forall n \geq 2.$

Quindi la disuguaglianza è soddisfatta per $c = 13$ e $n_0 = 2$.

Esempi: un polinomio è O del suo primo termine

Più in generale, possiamo provare che:

$$a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0 = O(n^k)$$

Infatti

$$\begin{aligned} a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0 & \\ & \leq |a_k| n^k + |a_{k-1}| n^{k-1} + \dots + |a_1| n + |a_0| \\ & \leq |a_k| n^k + |a_{k-1}| n^k + \dots + |a_1| n^k + |a_0| n^k \\ & = (|a_k| + |a_{k-1}| + \dots + |a_1| + |a_0|) n^k \\ & = c n^k \end{aligned}$$

$$\implies a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0 = O(n^k)$$

quindi

$$n^3 + 100n + 200 = O(n^3)$$

$$20n^3 + n^5 + 100n = O(n^5)$$

$$10n^2 + n^{5/2} + 7n = O(n^{5/2})$$

$$10n + 3n^7 + 5n^6 + 9n^3 + 34n^2 + 22n^5 + n^{8/3} + 4n^{7/2} + 23n^{11/2} = O(n^7)$$

⋮

Notazione asintotica Ω

Notazione duale di O:

$$f(n) = \Omega(g(n)) \text{ se e solo se } g(n) = O(f(n))$$

Notazioni Asintotiche: notazione Θ

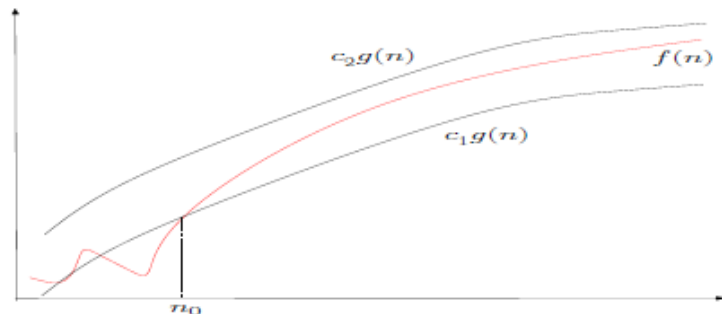
Date $f : n \in N \rightarrow f(n) \in R_+$, $g : n \in N \rightarrow g(n) \in R_+$,
scriveremo

$$f(n) = \Theta(g(n))$$

$$\Leftrightarrow \exists n_0, c_1, c_2 > 0 : c_1g(n) \leq f(n) \leq c_2g(n), \forall n \geq n_0$$

Equivalentemente

$$f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n)) \text{ e } f(n) = \Omega(g(n))$$



Notazione Θ

- Date due funzioni $f(n)$ scriveremo

$$f(n) = O(g(n))$$

se $f(n)$ non cresce più velocemente di $g(n)$

- Scriveremo invece

$$f(n) = \Omega(g(n))$$

se $f(n)$ cresce almeno tanto velocemente di $g(n)$

- Scriveremo infine

$$f(n) = \Theta(g(n))$$

se $f(n)$ e $g(n)$ **crescono allo stesso modo**

⊖ limitazione esatta

- È perfettamente legittimo dire che $n^3 + n\sqrt{n} \log n + 10 = O(n^3)$,
ma è più preciso dire che $n^3 + n\sqrt{n} \log n + 10 = \Theta(n^3)$
- È corretto dire che $n^{\frac{1}{\log n}} = O(n)$,
ma è più preciso dire che

$$n^{\frac{1}{\log n}} = \left(2^{\log n}\right)^{\frac{1}{\log n}} = 2 = \Theta(1)$$

- È quindi una questione di precisione nel linguaggio...

⊖ è una limitazione esatta (*tight bound*)

Tornando a $T(n)$

$T(n) = O(g(n))$ significa che il tempo di esecuzione, anche nel caso peggiore, è limitato superiormente da $g(n)$

$T(n) = \Omega(g(n))$ significa che il tempo di esecuzione, anche nel caso migliore, è limitato inferiormente da $g(n)$

$T(n) = \Theta(g(n))$ significa che nel caso peggiore è $O(g(n))$ e nel caso migliore è $\Omega(g(n))$

(in pratica non vi è distinzione fra tempo di esecuzione nel caso peggiore e migliore)

Esempio. $T_I(n)$, tempo di esecuzione di InsertionSort è $T_I(n) = \Omega(n)$ e $T_I(n) = O(n^2)$

$T_M(n)$, tempo di esecuzione di MergeSort è $T_M(n) = \Theta(n \log n)$

In termini di analisi ... matematica

• se $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c \neq 0$ allora

$$f(n) = O(g(n)) \quad \text{e} \quad g(n) = O(f(n)) \quad (\text{ovvero } f(n) = \Theta(g(n)))$$

• se $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$ allora

$$f(n) = O(g(n)) \quad \text{ma} \quad g(n) \neq O(f(n))$$

• se $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$ allora

$$f(n) \neq O(g(n)) \quad \text{ma} \quad g(n) = O(f(n))$$

Esempio

Sia $f(n) = \log n$ e $g(n) = \sqrt{n}$. Usando la regola de l'Hôpital abbiamo

$$\begin{aligned}\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} &= \lim_{n \rightarrow \infty} \frac{\log n}{\sqrt{n}} \\ &= \lim_{n \rightarrow \infty} \frac{1/n}{1/(2\sqrt{n})} \\ &= \lim_{n \rightarrow \infty} \frac{2}{\sqrt{n}} = 0\end{aligned}$$

da cui otteniamo immediatamente

$$\log n = O(\sqrt{n})$$

FINE

Notazioni asintotiche

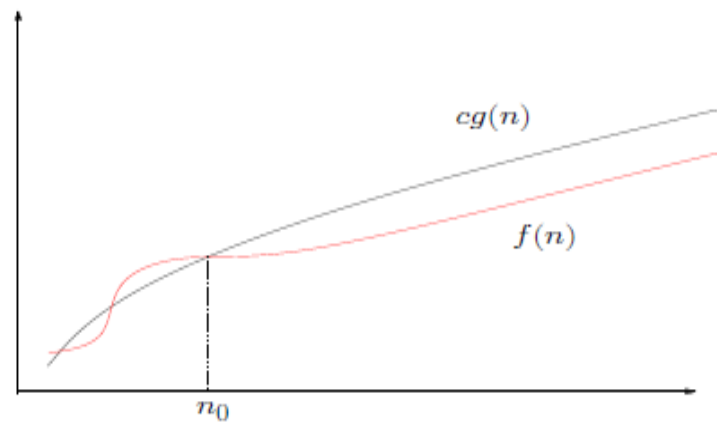
Notazioni Asintotiche: notazione O

Date $f : n \in N \rightarrow f(n) \in R_+$, $g : n \in N \rightarrow g(n) \in R_+$,
scriveremo

$$f(n) = O(g(n))$$

$\Leftrightarrow \exists c > 0, \exists n_0$ tale che $f(n) \leq cg(n), \forall n \geq n_0$

Informalmente, $f(n) = O(g(n))$ se $f(n)$ **non** cresce più
velocemente di $g(n)$. Graficamente



Asymptotic Order of Growth

- **Upper bounds.** $T(n)$ is $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$.
- **Lower bounds.** $T(n)$ is $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.
- **Tight bounds.** $T(n)$ is $\Theta(f(n))$ if $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$.
- *Ex:* $T(n) = 32n^2 + 17n + 32$.
 - $T(n)$ is $O(n^2)$, $O(n^3)$, $\Omega(n^2)$, $\Omega(n)$, and $\Theta(n^2)$.
 - $T(n)$ is not $O(n)$, $\Omega(n^3)$, $\Theta(n)$, or $\Theta(n^3)$.

⊕ limitazione esatta

- È perfettamente legittimo dire che

$$n^3 + n\sqrt{n}\log n + 10 = O(n^3),$$

ma è più preciso dire che $n^3 + n\sqrt{n}\log n + 10 = \Theta(n^3)$

- È corretto dire che $n^{\frac{1}{\log n}} = O(n)$,

ma è più preciso dire che

$$n^{\frac{1}{\log n}} = \left(2^{\log n}\right)^{\frac{1}{\log n}} = 2 = \Theta(1)$$

- È quindi una questione di precisione nel linguaggio...

⊕ è una limitazione esatta (*tight bound*)

Tornando a $T(n)$

$T(n) = O(g(n))$ significa che il tempo di esecuzione, anche nel caso peggiore, è limitato superiormente da $g(n)$

$T(n) = \Omega(g(n))$ significa che il tempo di esecuzione, anche nel caso migliore, è limitato inferiormente da $g(n)$

$T(n) = \Theta(g(n))$ significa che nel caso peggiore è $O(g(n))$ e nel caso migliore è $\Omega(g(n))$

(in pratica non vi è distinzione fra tempo di esecuzione nel caso peggiore e migliore)

Esempio.

$T_I(n)$, tempo di esecuzione di InsertionSort è $T_I(n) = \Omega(n)$ e $T_I(n) = O(n^2)$

$T_M(n)$, tempo di esecuzione di MergeSort è $T(n) = \Theta(n \log n)$

Notation

- Slight abuse of notation. $T(n) = O(f(n))$.
 - Asymmetric:
 - $f(n) = 5n^3$; $g(n) = 3n^2$
 - $f(n) = O(n^3) = g(n)$
 - but $f(n) \neq g(n)$.
 - Better notation: $T(n) \in O(f(n))$.
- Meaningless statement: “Any comparison-based sorting algorithm requires at least $O(n \log n)$ comparisons”.
 - Use Ω for lower bounds.

Funzioni più utilizzate

Scaletta:

Man mano che si scende
troviamo funzioni che crescono
più velocemente (in senso stretto)

Espressione O	nome
$O(1)$	costante
$O(\log \log n)$	log log
$O(\log n)$	logaritmico
$O(\sqrt[c]{n}), c > 1$	sublineare
$O(n)$	lineare
$O(n \log n)$	$n \log n$
$O(n^2)$	quadratico
$O(n^3)$	cubico
$O(n^k) (k \geq 1)$	polinomiale
$O(a^n) (a > 1)$	esponenziale
$O(n!)$	fattoriale

Asymptotic Bounds for Some Common Functions

- **Polynomials.** $a_0 + a_1n + \dots + a_d n^d$ is $\Theta(n^d)$ if $a_d > 0$.
- **Polynomial time.** Running time is $O(n^d)$ for some constant d independent of the input size n .
- **Logarithms.** $O(\log_a n) = O(\log_b n)$ for any constants $a, b > 0$.
↑
can avoid specifying the base
- **Logarithms.** For every $x > 0$, $\log n = O(n^x)$.
↑
log grows slower than every polynomial
- **Exponentials.** For every $r > 1$ and every $d > 0$, $n^d = O(r^n)$.
↑
every exponential grows faster than every polynomial

Più in dettaglio

Informalmente....

- ❑ Un esponenziale cresce più velocemente di qualsiasi polinomio
- ❑ Un polinomio cresce più velocemente di qualsiasi potenza di logaritmo

Più precisamente:

$$n^d = O(r^n) \quad \text{per ogni } d > 0 \text{ e } r > 1$$

$$\log_b n^k = O(n^d) \quad \text{per ogni } k, d > 0 \text{ e } b > 1$$

E ancora

Informalmente....

- ❑ Nel confronto fra esponenziali conta la base
- ❑ Nel confronto fra polinomi conta il grado
- ❑ Nel confronto fra logaritmi ... la base non conta

Per esempio:

$$2^n = O(3^n)$$

$$n^2 = O(n^3)$$

$$\log_{10} n = \log_2 n (\log_{10} 2) = \Theta(\log_2 n)$$

Polinomi vs logaritmi

Un polinomio cresce più velocemente di qualsiasi potenza di logaritmo.

Per esempio

• Proviamo che

$$\log_2 n = O(n).$$

Occorre provare che $\exists c, n_0 : \log_2 n \leq cn \quad \forall n \geq n_0$

Per induzione su n : Per $n = 1$ abbiamo $\log_2 1 = 0 \leq 1$.

In generale, per $n \geq 1$

$$\begin{aligned} \log_2(n+1) &\leq \log_2(n+n) = \log_2(2n) \\ &= \log_2 2 + \log_2 n = 1 + \log n \\ &\leq 1 + n \text{ (per ipotesi induttiva)} \end{aligned}$$

Abbiamo quindi provato che

$$\log n \leq n \quad \forall n \geq 1 \implies \boxed{\log n = O(n)}$$

Lo proveremo con $c=1$ e $n_0=1$, cioè $\log_2 n \leq n, \quad \forall n \geq 1$

In pratica

Per stabilire l'ordine di crescita di una funzione dovremo tenere ben presente la «**scaletta**» e alcune **proprietà** delle notazioni asintotiche

Properties

- **Transitivity.**
 - If $f = O(g)$ and $g = O(h)$ then $f = O(h)$.
 - If $f = \Omega(g)$ and $g = \Omega(h)$ then $f = \Omega(h)$.
 - If $f = \Theta(g)$ and $g = \Theta(h)$ then $f = \Theta(h)$.

- **Additivity.**
 - If $f = O(h)$ and $g = O(h)$ then $f + g = O(h)$.
 - If $f = \Omega(h)$ and $g = \Omega(h)$ then $f + g = \Omega(h)$.
 - If $f = \Theta(h)$ and $g = \Theta(h)$ then $f + g = \Theta(h)$.

Transitività

Se $f = O(g)$ e $g = O(h)$ allora $f = O(h)$

Ipotesi:

esistono costanti $c, n_0 > 0$ tali che per ogni $n \geq n_0$ si ha $f(n) \leq c \cdot g(n)$

esistono costanti $c', n'_0 > 0$ tali che per ogni $n \geq n'_0$ si ha $g(n) \leq c' \cdot h(n)$

Tesi (Dobbiamo mostrare che):

esistono costanti $c'', n''_0 > 0$ tali che per ogni $n \geq n''_0$ si ha $f(n) \leq c'' \cdot h(n)$

Quanto valgono c'', n''_0 ?

$f(n) \leq c \cdot g(n) \leq c \cdot c' \cdot h(n)$ per ogni $n \geq n_0$ e $n \geq n'_0$

$$c'' = c \cdot c'$$

$$n''_0 = \max \{n_0, n'_0\}$$

Additività

Se $f = O(h)$ e $g = O(h)$ allora $f + g = O(h)$

Ipotesi:

esistono costanti $c, n_0 > 0$ tali che per ogni $n \geq n_0$ si ha $f(n) \leq c \cdot h(n)$

esistono costanti $c', n'_0 > 0$ tali che per ogni $n \geq n'_0$ si ha $g(n) \leq c' \cdot h(n)$

Tesi (Dobbiamo mostrare che):

esistono costanti $c'', n''_0 > 0$ tali che per ogni $n \geq n''_0$ si ha $f(n) + g(n) \leq c'' \cdot h(n)$

Quanto valgono c'', n''_0 ?

$f(n) + g(n) \leq c \cdot h(n) + c' \cdot h(n) = (c + c') h(n)$ per ogni $n \geq n_0$ e $n \geq n'_0$

$$c'' = c + c'$$

$$n''_0 = \max \{n_0, n'_0\}$$

Due regole fondamentali

Nel determinare l'ordine di crescita asintotica di una funzione

1. Possiamo trascurare i termini additivi di ordine inferiore
2. Possiamo trascurare le costanti moltiplicative

ATTENZIONE!

Le regole NON servono però per determinare esplicitamente le costanti c ed n_0 .

Prima regola

«Possiamo trascurare i termini additivi di ordine inferiore»

Cosa significa formalmente?

Se $g = O(f)$ allora $f + g = \Theta(f)$

Ipotesi:

g è di ordine inferiore a f : $g = O(f)$:

esistono costanti $c, n_0 > 0$ tali che per ogni $n \geq n_0$ si ha $g(n) \leq c \cdot f(n)$

Tesi (Dobbiamo mostrare che):

$f + g = \Theta(f)$

$f + g = \Omega(f)$: esistono $c'', n''_0 > 0$ tali che per ogni $n \geq n''_0$ si ha $f(n) + g(n) \geq c'' \cdot f(n)$

Dato che $f = O(f)$ e $g = O(f)$ per l'additività: $f + g = O(f)$.

$f(n) + g(n) \geq f(n)$ essendo $g(n) \geq 0$; $c'' = 1$ ed $n''_0 = 0$

Seconda regola

«Possiamo trascurare le costanti moltiplicative»

Cosa significa formalmente?

Per ogni costante $a > 0$ allora $a \cdot f = \Theta(f)$

Ipotesi: $a > 0$

Tesi

esistono costanti $c, n_0 > 0$ tali che per ogni $n \geq n_0$ si ha $a \cdot f(n) \leq c \cdot f(n)$

esistono costanti $c', n'_0 > 0$ tali che per ogni $n \geq n'_0$ si ha $a \cdot f(n) \geq c' \cdot f(n)$

$c = a$

$c' = 1$

Per confrontare crescita di due funzioni

Basterà usare:

- La «**scaletta**»
- Le proprietà di **additività** e **transitività**
- Le due **regole fondamentali**

Esercizio 1

Vero o Falso?

• $3n^5 - 16n + 2 = O(n^5)$?

• $3n^5 - 16n + 2 = O(n)$?

• $3n^5 - 16n + 2 = O(n^{17})$?

• $3n^5 - 16n + 2 = \Omega(n^5)$?

• $3n^5 - 16n + 2 = \Omega(n)$?

• $3n^5 - 16n + 2 = \Omega(n^{17})$?

• $3n^5 - 16n + 2 = \Theta(n^5)$?

• $3n^5 - 16n + 2 = \Theta(n)$?

• $3n^5 - 16n + 2 = \Theta(n^{17})$?

Esercizio 2

Per ciascuna delle seguenti coppie di funzioni $f(n)$ e $g(n)$, dire se $f(n) = O(g(n))$, oppure se $g(n) = O(f(n))$.

● $f(n) = (n^2 - n)/2, \quad g(n) = 6n$

● $f(n) = n + 2\sqrt{n}, \quad g(n) = n^2$

● $f(n) = n + \log n, \quad g(n) = n\sqrt{n}$

● $f(n) = n^2 + 3n, \quad g(n) = n^3$

● $f(n) = n \log n, \quad g(n) = n\sqrt{n}/2$

● $f(n) = n + \log n, \quad g(n) = \sqrt{n}$

● $f(n) = 2(\log n)^2, \quad g(n) = \log n + 1$

● $f(n) = 4n \log n + n, \quad g(n) = (n^2 - n)/2$

● $f(n) = (n^2 + 2)/(1 + 2^{-n}), \quad g(n) = n + 3$

● $f(n) = n + n\sqrt{n}, \quad g(n) = 4n \log(n^3 + 1)$

NOTA: Esistono anche funzioni (particolari) non confrontabili tramite O

Esercizio 3

Date le seguenti funzioni

$\log n^5, n^{\log n}, \log^2 n, 10\sqrt{n}, (\log n)^n, n^n, n \log \sqrt{n},$
 $n \log^3 n, n^2 \log n, \sqrt{n \log n}, 10 \log \log n, 3 \log n,$

ordinarle scrivendole da sinistra a destra in modo tale che la funzione $f(n)$ venga posta a sinistra della funzione $g(n)$ se $f(n) = O(g(n))$.

Proviamo...

$10 \log \log n, 3 \log n, \log n^5, \log^2 n, 10\sqrt{n}, \sqrt{n \log n}, n \log \sqrt{n}$
 $n \log^3 n, n^2 \log n, n^{\log n}, (\log n)^n, n^n$

Solved Exercise 1, pag. 65-66

Ordinare le seguenti funzioni

$$f_1(n) = 10^n$$

$$f_2(n) = n^{1/3}$$

$$f_3(n) = n^n$$

$$f_4(n) = \log_2 n$$

$$f_5(n) = 2^{\sqrt{\log_2 n}}$$

in senso crescente.

Cioè, se $g(n)$ segue la funzione $f(n)$ allora $f(n) = O(g(n))$.

Soluzione.

Parte più facile: $f_4(n) = \log_2 n$, $f_2(n) = n^{1/3}$, $f_1(n) = 10^n$, $f_3(n) = n^n$

E $f_5(n) = 2^{\sqrt{\log_2 n}}$?

Continuare lo svolgimento ...

Esercizi «per casa»

- Esercizi delle slides precedenti
- Es. 3, 4, 5 e 6 di pagg. 67-68 del libro [KT]
- Esercizi di esami (es. 1-9 dall'elenco online)

Programmazione dinamica

Primi esempi

Dynamic Programming Applications

Areas.

Bioinformatics.

Control theory.

Information theory.

Operations research.

Computer science: theory, graphics, AI, systems,

Some famous dynamic programming algorithms.

Viterbi for hidden Markov models.

Unix diff for comparing two files.

Smith-Waterman for sequence alignment.

Bellman-Ford for shortest path routing in networks.

Cocke-Kasami-Younger for parsing context free grammars.

Programmazione dinamica e Divide et Impera

Entrambe le tecniche dividono il problema in sottoproblemi: dalle soluzioni dei sottoproblemi è possibile risalire alla soluzione del problema di partenza.

Divide et Impera dà luogo in modo naturale ad algoritmi ricorsivi.

La programmazione dinamica può dar luogo ad algoritmi:

- ricorsivi con annotazione delle soluzioni su una tabella
- iterativi

Idea chiave:

calcolare la soluzione di ogni sottoproblema 1 sola volta ed annotarla su una **tabella**

Un primo esempio: calcolo dei numeri di Fibonacci

Sequenza dei numeri di Fibonacci:

1, 1, 2, 3, 5, 8, 13, ?

1, 1, 2, 3, 5, 8, 13, **21**, ...

In generale il prossimo numero è la somma degli ultimi due.
Come si formalizza?

Indicando con F_n (oppure $F(n)$) l' n-esimo numero,
abbiamo la seguente relazione di ricorrenza:

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{se } n \geq 3 \\ 1 & \text{se } n = 1, 2 \end{cases}$$

I numeri di Fibonacci

Motivazione: problema di dinamica delle popolazioni studiato da *Leonardo da Pisa* (anche noto come *Fibonacci*)

I numeri di Fibonacci godono di una gamma stupefacente di proprietà, si incontrano nei modelli matematici di svariati fenomeni e sono utilizzabili per molti procedimenti computazionali; essi inoltre posseggono varie generalizzazioni interessanti. A questi argomenti viene espressamente dedicato un periodico scientifico, *The Fibonacci Quarterly*.

I numeri di Fibonacci sono legati alla “sezione aurea” Φ .

$$\lim_{n \rightarrow \infty} \frac{F_n}{F_{n-1}} = \phi$$
$$\phi = \frac{1 + \sqrt{5}}{2} = 1,6180339887\dots$$

Come calcolare l' n-esimo numero di Fibonacci?

Un primo approccio numerico.

Possiamo usare una funzione matematica che calcoli direttamente i numeri di Fibonacci.

Infatti si può dimostrare che:

$$F_n = \frac{1}{\sqrt{5}} \left(\phi^n - \hat{\phi}^n \right)$$

$$\begin{aligned} \phi &= \frac{1+\sqrt{5}}{2} \approx +1.618 \\ \hat{\phi} &= \frac{1-\sqrt{5}}{2} \approx -0.618 \end{aligned}$$

e quindi...

Un primo algoritmo numerico

```
Fibonacci1 (n)
```

```
return  $1/\sqrt{5} (\Phi^n - \Phi^{-n})$ 
```

Però:

Problemi di accuratezza sul numero di cifre decimali;
il risultato non sarebbe preciso.

Un secondo algoritmo ricorsivo

Secondo approccio: utilizzare la definizione ricorsiva, che lavora solo su interi

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{se } n \geq 3 \\ 1 & \text{se } n = 1, 2 \end{cases}$$

Fibonacci2 (n)

If $n \leq 2$ **then Return** 1

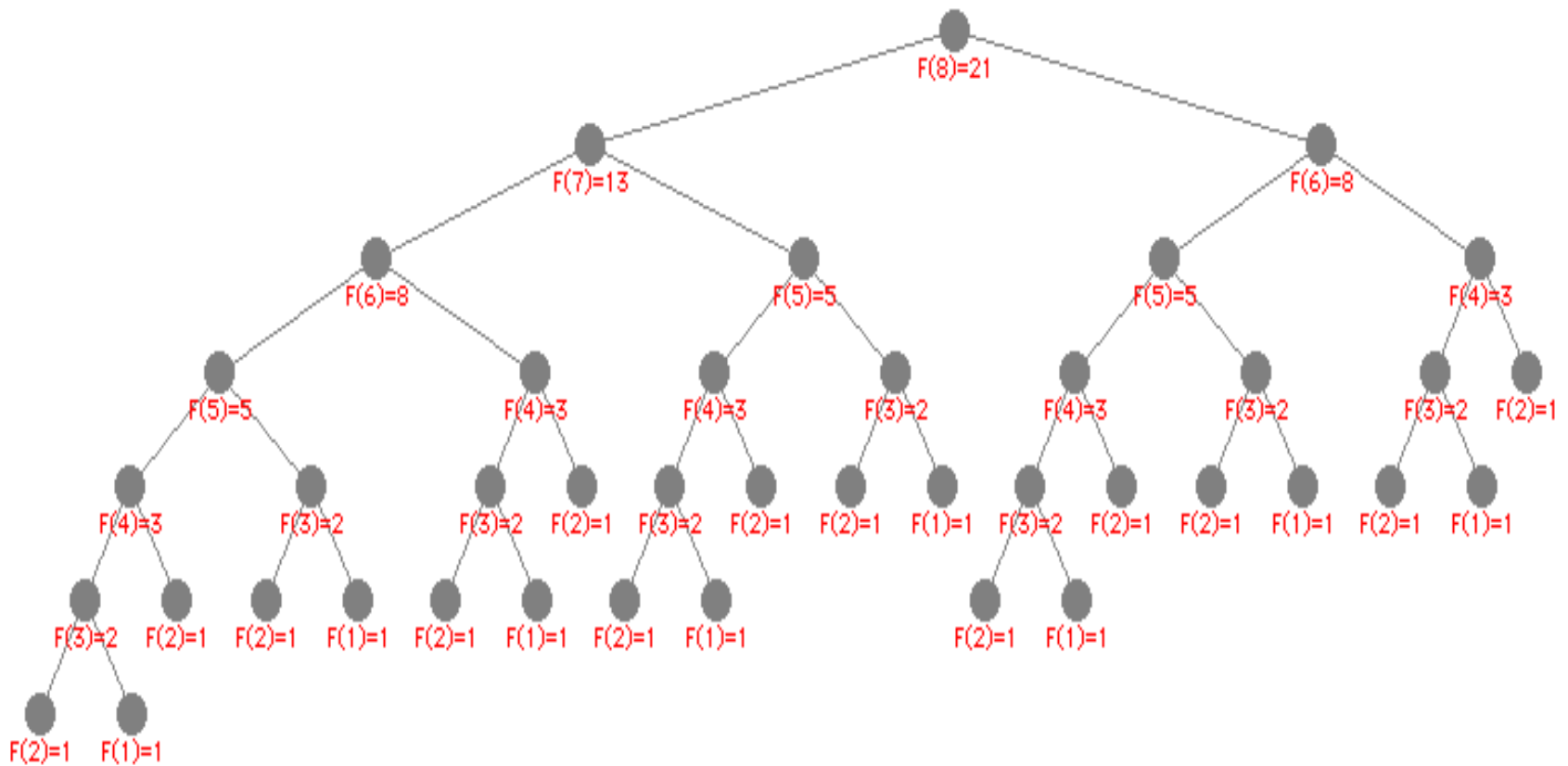
Else Return Fibonacci2 (n-1) + Fibonacci2 (n-2)

Un esempio

Fibonacci2(n)

If $n \leq 2$ then Return 1

Else Return Fibonacci2(n-1) + Fibonacci2(n-2)



Analisi del tempo di esecuzione

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + \Theta(1) & n \geq 3 \\ T(n) &= \Theta(1) & n = 1, 2 \end{aligned}$$

$$T(n) = O(2^n)$$

Dimostrazione per induzione.

Sia $c > 0$ tale che $T(n) = T(n-1) + T(n-2) + c$ e $T(1), T(2) \leq c$.

Tesi: $T(n) \leq c 2^n$.

Base: $T(1) \leq c \leq c 2^1 = 2c$; $T(2) \leq c \leq c 2^2 = 4c$

Ipotesi induttiva: $T(n-1) \leq c 2^{n-1}$ e $T(n-2) \leq c 2^{n-2}$

Passo induttivo: $T(n) \leq c 2^{n-1} + c 2^{n-2} + c \leq c 2^{n-1} + c 2^{n-2} + c 2^{n-2} =$
 $= c 2^{n-1} + c (2^{n-2} + 2^{n-2}) = c 2^{n-1} + c (2 \cdot 2^{n-2}) = c 2^{n-1} + c 2^{n-1} = c 2^n$

Nota: $T(n) \approx F(n) \approx \Phi^n$

Già calcolare $F(100)$ con le attuali tecnologie richiederebbe un tempo inaccettabile

Si può fare di meglio?

Migliorare l'algoritmo ricorsivo (versione con annotazione)

Perché non memorizzare le soluzioni dei sottoproblemi via via calcolate?

Basta un array $F[1..n]$ per memorizzare tutti valori già calcolati. L'array F inizialmente è vuoto.

```
Fibonacci3-memo (j)
```

```
If  $j \leq 2$  then  $F[j]=1$ 
```

```
    Return  $F[j]$ 
```

```
Else if  $F[j]$  non è vuoto then Return  $F[j]$ 
```

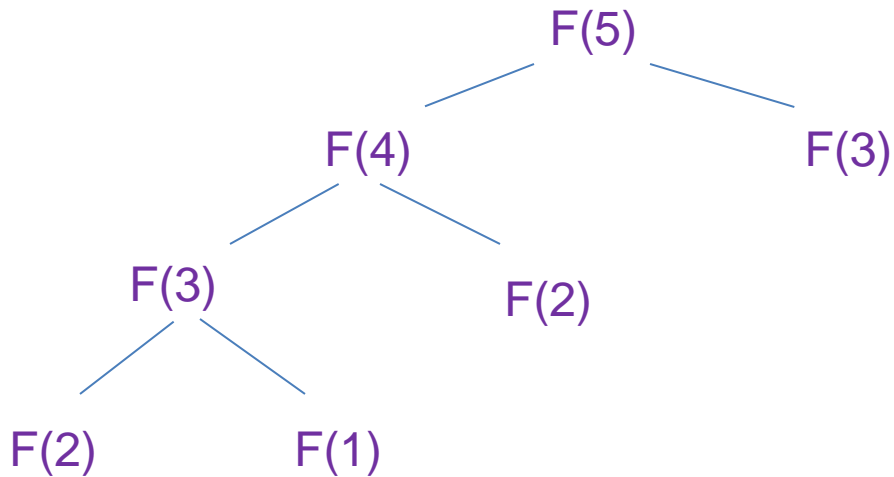
```
Else Define  $F[j]=$  Fibonacci3-memo ( $j-1$ ) +  
                Fibonacci3-memo ( $j-2$ )
```

```
    Return  $F[j]$ 
```

```
Endif
```

Esempio j=5

```
Fibonacci3-memo(j)
If j ≤ 2 then F[j]=1
    Return F[j]
Else if F[j] non è vuoto then Return F[j]
Else Define F[j]= Fibonacci3-memo(j-1)+
    Fibonacci3-memo(j-2)
    Return F[j]
Endif
```



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

1	1	2	3	
---	---	---	---	--

1	1	2		
---	---	---	--	--

1	1			
---	---	--	--	--

	1			
--	---	--	--	--

--	--	--	--	--

Migliorare l'algoritmo ricorsivo (versione iterativa)

```
Fibonacci3-iter(n)
```

```
F[1]=1
```

```
F[2]=1
```

```
For i=3,...,n
```

```
    F[i]= F[i-1]+F[i-2]
```

```
Endfor
```

```
Return F[n]
```

Esempio: n=5

1				
---	--	--	--	--

1	1			
---	---	--	--	--

1	1	2		
---	---	---	--	--

1	1	2	3	
---	---	---	---	--

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

Tempo di esecuzione

Il tempo di esecuzione di `Fibonacci3-iter` è $\Theta(n)$

Anche il tempo di esecuzione di `Fibonacci3-memo` è $\Theta(n)$: ogni differente chiamata ricorsiva è eseguita solo una volta, richiede tempo costante, e ci sono $O(n)$ diverse chiamate a `Fibonacci3-memo`.

Confronto prestazioni

L'algoritmo `Fibonacci3-iter` e `Fibonacci3-memo` impiegano tempo **proporzionale** a n invece di **esponenziale** in n come `Fibonacci2`.

Tempo effettivo richiesto da implementazioni in C dei due seguenti algoritmi su piattaforme diverse:

	<code>fibonacci2(58)</code>	<code>fibonacci3(58)</code>
Pentium IV 1700MHz	15820 sec. (\simeq 4 ore)	0.7 milionesimi di secondo
Pentium III 450MHz	43518 sec. (\simeq 12 ore)	2.4 milionesimi di secondo
PowerPC G4 500MHz	58321 sec. (\simeq 16 ore)	2.8 milionesimi di secondo

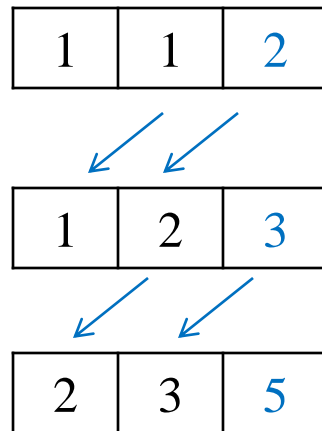
Analisi dello spazio necessario

Lo spazio di memoria necessario per eseguire `Fibonacci3-iter` e `Fibonacci3-memo` è

$$S(n) = \Theta(n)$$

perché entrambi gli algoritmi usano un array con n celle.

Ma in realtà possono bastare 3 celle (per qualsiasi n):
per il calcolo di $F[j]$ servono solo i valori nelle 2 celle precedenti



Programmazione dinamica: caratteristiche

`Fibonacci3-memo` e `Fibonacci3-iter` sono algoritmi di **programmazione dinamica**: perché?

1. La soluzione al problema originale si può ottenere da soluzioni a sottoproblemi
2. Esiste una relazione di ricorrenza per la funzione che dà il valore ottimale ad un sottoproblema
3. Le soluzioni ai sottoproblemi sono via via memorizzate in una **tabella**

Due implementazioni possibili:

- Con annotazione (*memoized*) o *top-down*
- Iterativa o *bottom-up*

Programmazione dinamica vs Divide et Impera

Entrambe le tecniche dividono il problema in sottoproblemi: dalle soluzioni dei sottoproblemi è possibile risalire alla soluzione del problema di partenza

Dobbiamo allora considerare la tecnica Divide et Impera superata?

NO: La programmazione dinamica risulta più efficiente quando:

- Ci sono dei sottoproblemi ripetuti
- Ci sono solo un numero **polinomiale** di sottoproblemi (da potere memorizzare in una **tabella**)

Per esempio: nel MergeSort non ci sono sottoproblemi ripetuti.

6.1 Weighted Interval Scheduling

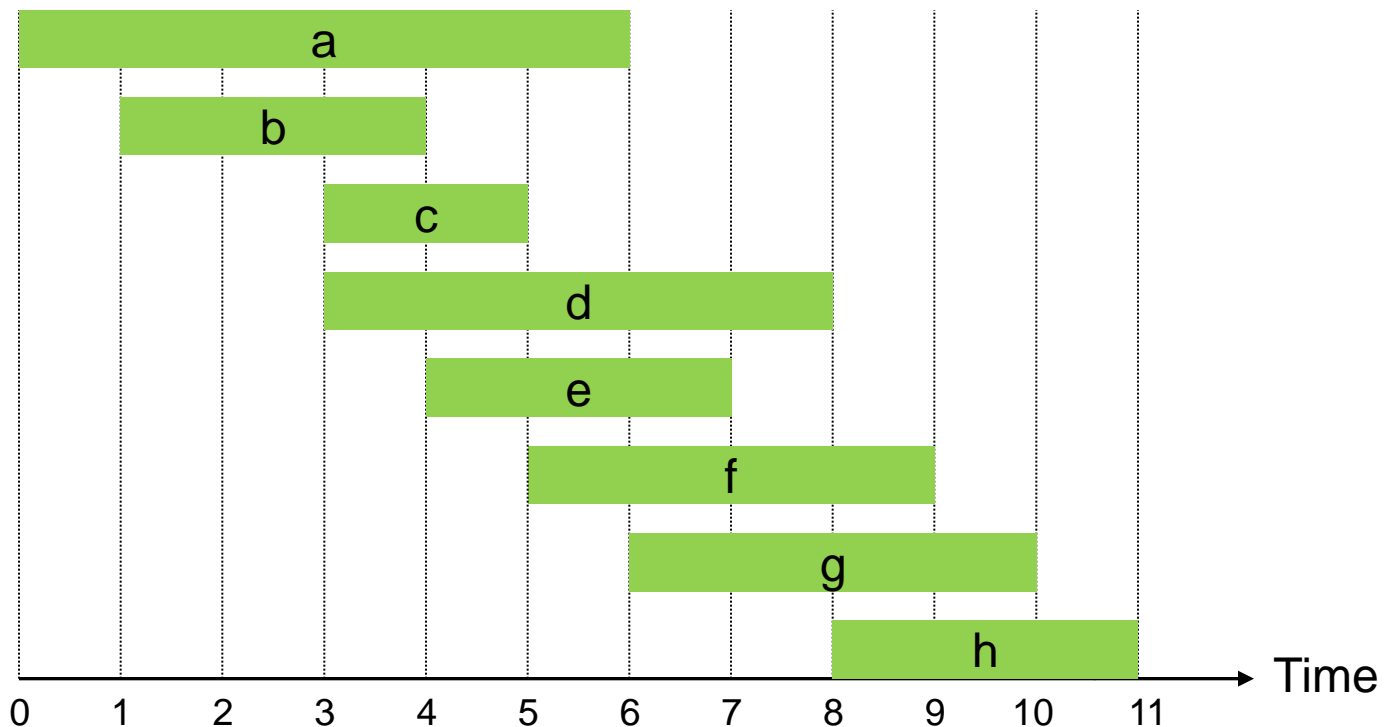
Interval Scheduling

Interval scheduling problem.

Job j starts at s_j , finishes at f_j .

Two jobs **compatible** if they don't overlap.

Goal: find **biggest** subset of mutually compatible jobs.



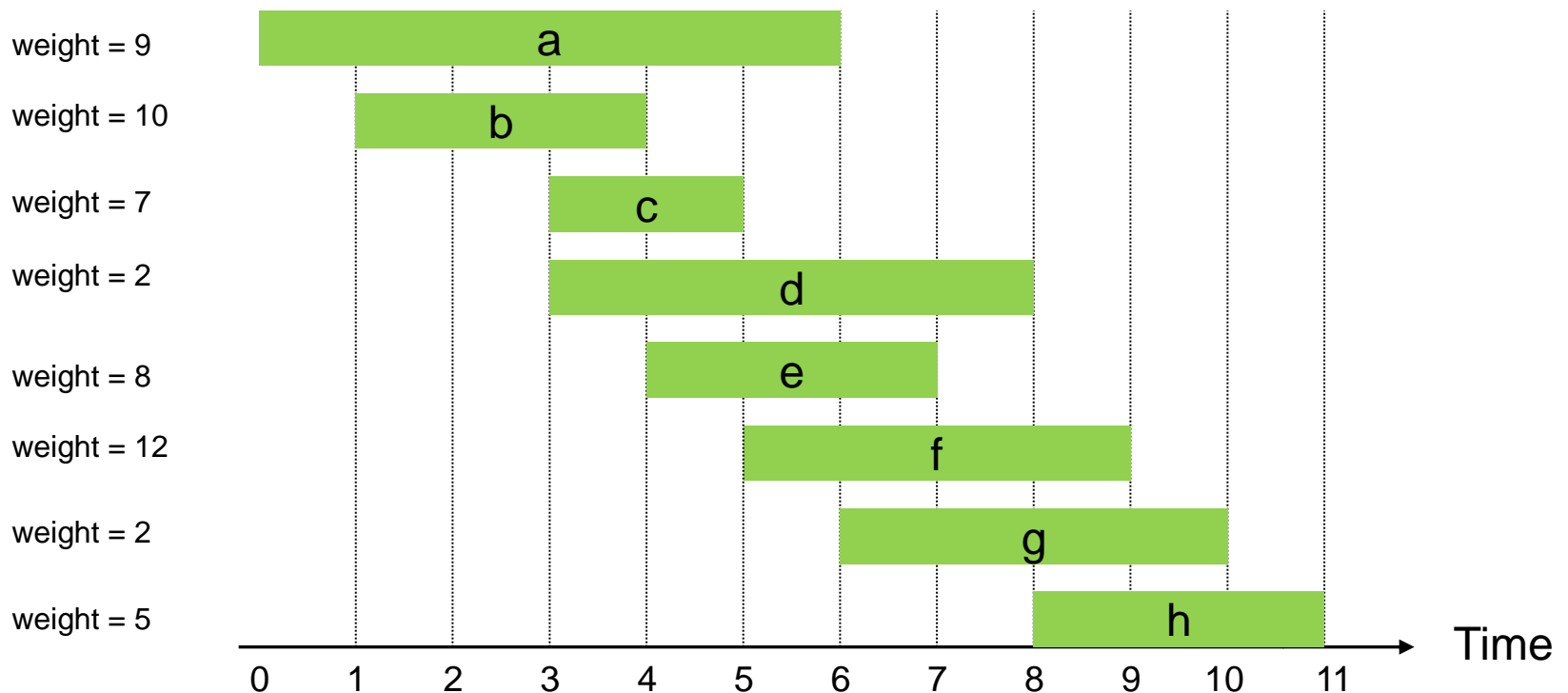
Weighted Interval Scheduling

Weighted interval scheduling problem.

Job j starts at s_j , finishes at f_j , and has **weight** or value v_j .

Two jobs **compatible** if they don't overlap.

Goal: find maximum **weight** subset of mutually compatible jobs.



Weighted Interval Scheduling

Weighted interval scheduling problem.

Job j starts at s_j , finishes at f_j , and has **weight** or value v_j .

Two jobs **compatible** if they don't overlap.

Goal: find **maximum weight** subset of mutually compatible jobs.

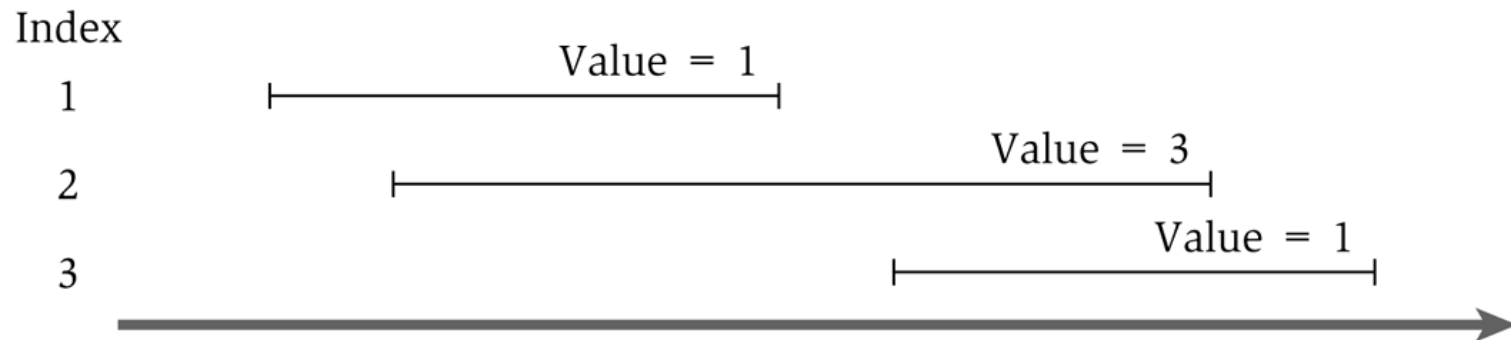


Figure 6.1 A simple instance of weighted interval scheduling.

Insiemi compatibili: $\{1,3\}$ di peso 2

$\{2\}$ di peso **3** è la soluzione ottimale

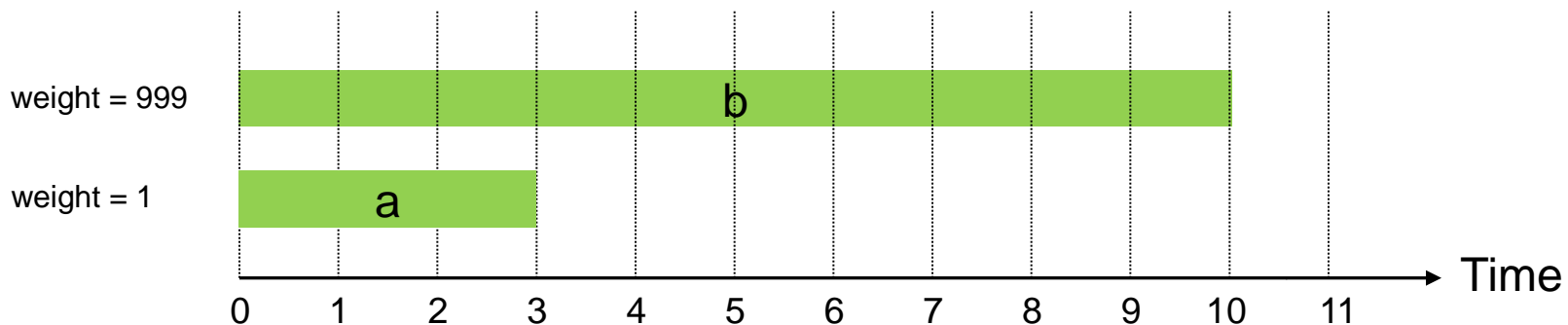
Unweighted Interval Scheduling Review

Note: Greedy algorithm works if all **weights are 1**.

Consider jobs in ascending order of finish time.

Add job to subset if it is compatible with previously chosen jobs.

Observation. Greedy algorithm can fail spectacularly if arbitrary weights are allowed.

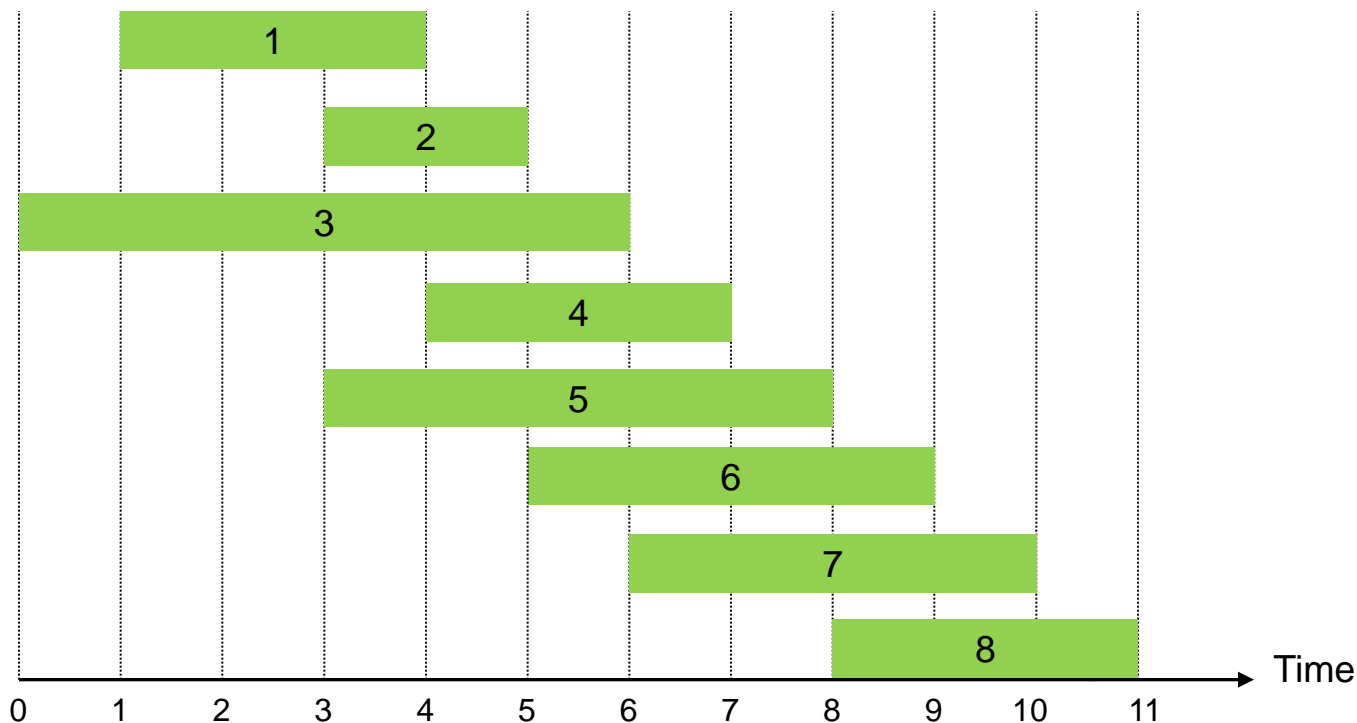


Weighted Interval Scheduling

Notation. Label jobs by finishing time: $f_1 \leq f_2 \leq \dots \leq f_n$.

Def. $p(j)$ = largest index $i < j$ such that job i is compatible with j .

Ex: (independently from weights) $p(8) = 5$, $p(7) = 3$, $p(2) = 0$.



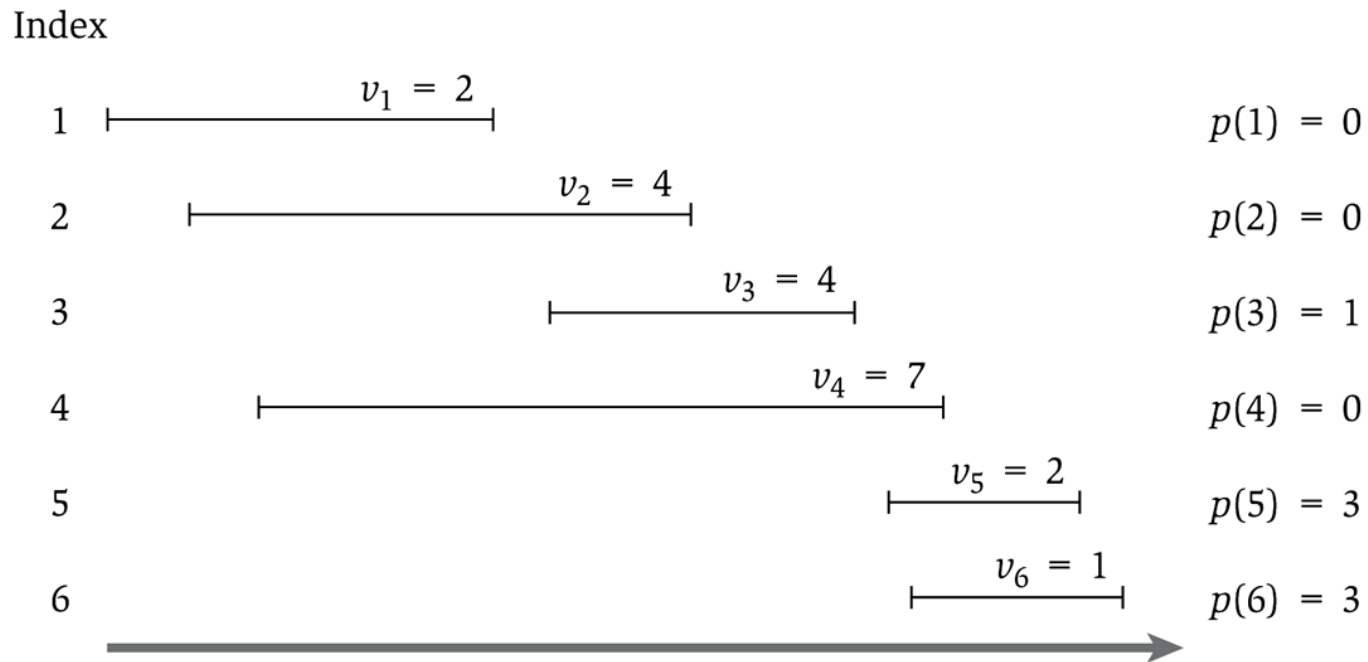


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j .

Soluzione = {5, 3, 1}
 Valore = 2+4+2 = 8

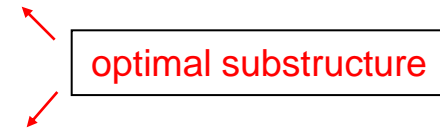
Dynamic Programming: Binary Choice

Notation. $\mathbf{OPT}(j)$ = value of optimal solution to the problem consisting of job requests 1, 2, ..., j.

Case 1: OPT selects job j.

can't use incompatible jobs { $p(j) + 1, p(j) + 2, \dots, j - 1$ }

must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., $p(j)$



Case 2: OPT does not select job j.

must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max \{v_j + OPT(p(j)), OPT(j-1)\} & \text{otherwise} \end{cases}$$

Weighted Interval Scheduling: Brute Force

Brute force **recursive** algorithm.

```
Input:  $n, s_1, \dots, s_n, f_1, \dots, f_n, v_1, \dots, v_n$ 
```

```
Sort jobs by finish times so that  $f_1 \leq f_2 \leq \dots \leq f_n$ .
```

```
Compute  $p(1), p(2), \dots, p(n)$ 
```

```
Compute-Opt( $j$ ) {  
    if ( $j = 0$ )  
        return 0  
    else  
        return  $\max(v_j + \text{Compute-Opt}(p(j)), \text{Compute-Opt}(j-1))$   
}
```

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{v_j + OPT(p(j)), OPT(j-1)\} & \text{otherwise} \end{cases}$$

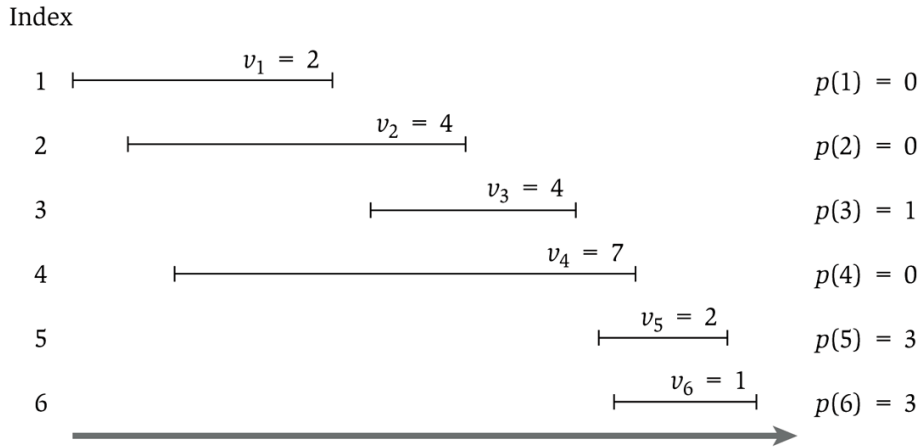


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j .

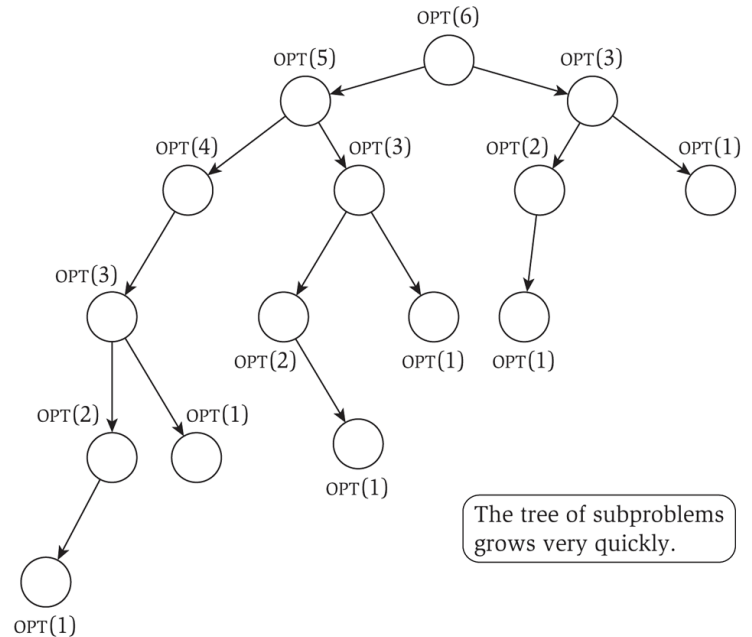
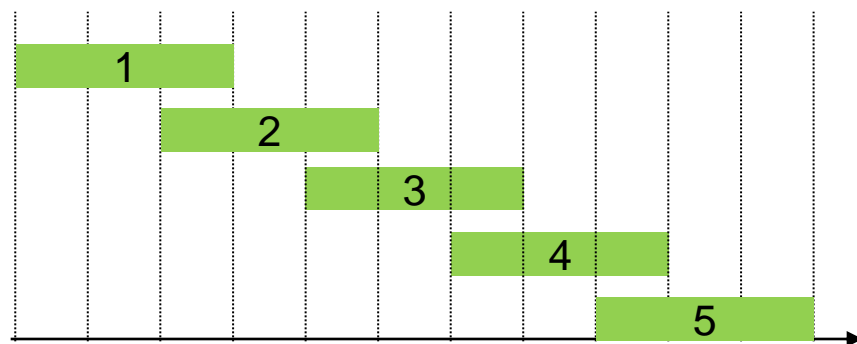


Figure 6.3 The tree of subproblems called by `Compute-Opt` on the problem instance of Figure 6.2.

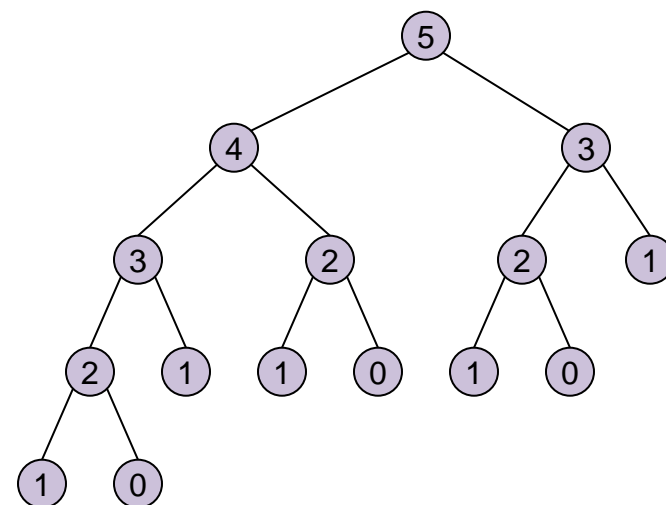
Weighted Interval Scheduling: Brute Force

Observation. Recursive algorithm fails spectacularly because of **redundant sub-problems** \Rightarrow exponential algorithms.

Ex. Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.



$$p(1) = 0, p(j) = j-2$$



Weighted Interval Scheduling: Memoization

Memoization. Store results of each sub-problem in a cache; lookup as needed.

```
Input:  $n, s_1, \dots, s_n, f_1, \dots, f_n, v_1, \dots, v_n$ 
```

```
Sort jobs by finish times so that  $f_1 \leq f_2 \leq \dots \leq f_n$ .
```

```
Compute  $p(1), p(2), \dots, p(n)$ 
```

```
for  $j = 1$  to  $n$ 
```

```
     $M[j] = \text{empty}$ 
```

```
 $M[j] = 0$ 
```

```
M-Compute-Opt( $j$ ) {
```

```
    if ( $M[j]$  is empty)
```

```
         $M[j] = \max(w_j + \text{M-Compute-Opt}(p(j)), \text{M-Compute-Opt}(j-1))$ 
```

```
    return  $M[j]$ 
```

```
}
```

Weighted Interval Scheduling: Running Time

Claim. Memoized version of algorithm takes $O(n \log n)$ time.

Sort by finish time: $O(n \log n)$.

Computing $p(\cdot)$: $O(n)$ after sorting by start time (exercise).

$M\text{-Compute-Opt}(j)$: each invocation takes $O(1)$ time and either

(i) returns an existing value $M[j]$

(ii) fills in one new entry $M[j]$ and makes two recursive calls

Progress measure $\Phi = \#$ nonempty entries of $M[\]$.

initially $\Phi = 0$, throughout $\Phi \leq n$.

(ii) increases Φ by 1 \Rightarrow at most $2n$ recursive calls.

Overall running time of $M\text{-Compute-Opt}(n)$ is $O(n)$.

Remark. $O(n)$ if jobs are pre-sorted by start and finish times.

Weighted Interval Scheduling: Bottom-Up

Bottom-up dynamic programming. Unwind recursion.

```
Input:  $n, s_1, \dots, s_n, f_1, \dots, f_n, v_1, \dots, v_n$ 
```

```
Sort jobs by finish times so that  $f_1 \leq f_2 \leq \dots \leq f_n$ .
```

```
Compute  $p(1), p(2), \dots, p(n)$ 
```

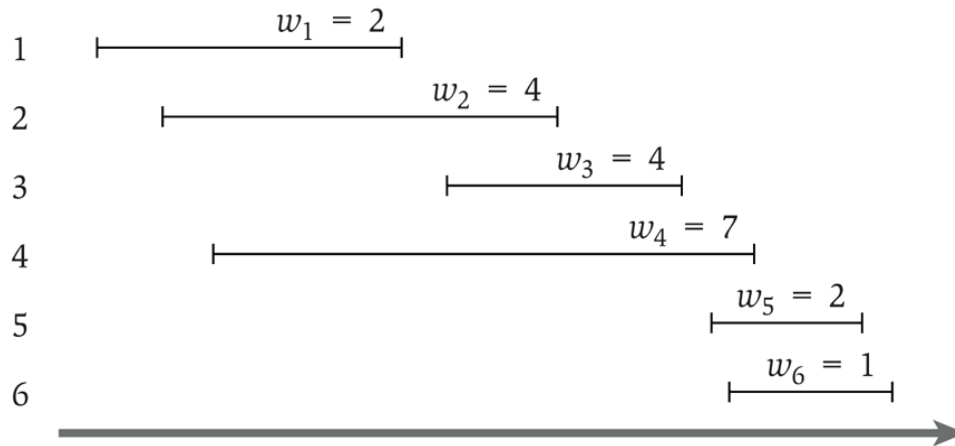
```
Iterative-Compute-Opt {  
     $M[0] = 0$   
    for  $j = 1$  to  $n$   
         $M[j] = \max(v_j + M[p(j)], M[j-1])$   
}
```

```

Iterative-Compute-Opt {
    M[0] = 0
    for j = 1 to n
        M[j] = max(wj + M[p(j)], M[j-1])
    }

```

Index



$$p(1) = 0$$

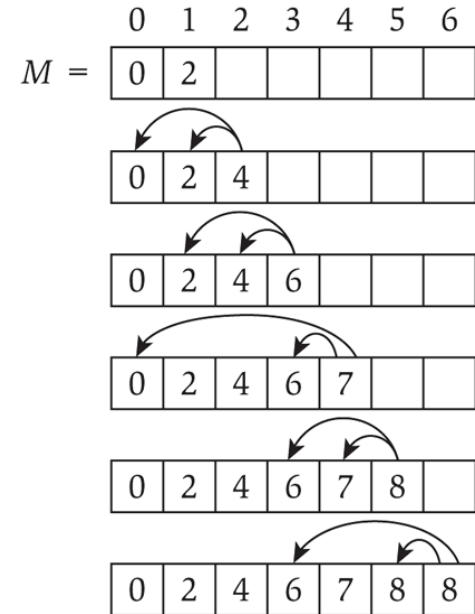
$$p(2) = 0$$

$$p(3) = 1$$

$$p(4) = 0$$

$$p(5) = 3$$

$$p(6) = 3$$



(b)

$$M[1] = \max (2+M[0], M[0]) = \max (2+0, 0) = 2$$

$$M[2] = \max (4+M[0], M[1]) = \max (4+0, 2) = 4$$

$$M[3] = \max (4+M[1], M[2]) = \max (4+2, 4) = 6$$

$$M[4] = \max (7+M[0], M[3]) = \max (7+0, 6) = 7$$

$$M[5] = \max (2+M[3], M[4]) = \max (2+6, 7) = 8$$

$$M[6] = \max (1+M[3], M[5]) = \max (1+6, 8) = 8$$

Copyright © 2005 Pearson
Addison-Wesley. All rights
reserved.

Esempio del calcolo di una soluzione

$$OPT(j) = \begin{cases} 0 & \text{if } j=0 \\ \max\{v_j + OPT(p(j)), OPT(j-1)\} & \text{otherwise} \end{cases}$$

$$M[1] = \max (2+M[0], M[0]) = \max (2+0, 0) = 2$$

$$M[2] = \max (4+M[0], M[1]) = \max (4+0, 2) = 4$$

$$M[3] = \max (4+M[1], M[2]) = \max (4+2, 4) = 6$$

$$M[4] = \max (7+M[0], M[3]) = \max (7+0, 6) = 7$$

$$M[5] = \max (2+M[3], M[4]) = \max (2+6, 7) = 8$$

$$M[6] = \max (1+M[3], M[5]) = \max (1+6, 8) = 8$$

$$M = \begin{array}{|c|c|c|c|c|c|c|} \hline & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 0 & 0 & 2 & 4 & 6 & 7 & 8 & 8 \\ \hline \end{array}$$

$M[6]=M[5]$: 6 non appartiene a OPT

$M[5]=v_5+M[3]$: OPT contiene 5 e una soluzione ottimale al problema per {1,2,3}

$M[3]=v_3+M[1]$: OPT contiene 5, 3 e una soluzione ottimale al problema per {1}

$M[1]=v_1+M[0]$: OPT contiene 5, 3 e 1 (e una soluzione ottimale al problema vuoto)

Soluzione = {5, 3, 1}

Valore = 2+4+2 = 8

Weighted Interval Scheduling: Finding a Solution

Q. Dynamic programming algorithms computes optimal **value**.

What if we want the **solution itself** (the set of intervals)?

A. Do some post-processing.

```
Run M-Compute-Opt(n)
Run Find-Solution(n)

Find-Solution(j) {
    if (j = 0)
        output nothing
    else if ( $v_j + M[p(j)] > M[j-1]$ )
        print j
        Find-Solution(p(j))
    else
        Find-Solution(j-1)
}
```

of recursive calls $\leq n \Rightarrow O(n)$.

Tecnica Divide et Impera

Algoritmi basati sulla tecnica Divide et Impera:

In questo corso:

- Ricerca binaria
- Mergesort (ordinamento)
- Quicksort (ordinamento)
- Moltiplicazione di interi
- Moltiplicazione di matrici (non in programma)

NOTA: nonostante la tecnica Divide-et-Impera sembri così «semplice» ben due «top ten algorithms of the 20° century» sono basati su di essa:
Fast Fourier Transform (FFT)

Quicksort

"Great algorithms are the poetry of computation,"

says Francis Sullivan of the Institute for Defense Analyses' Center for Computing Sciences in Bowie, Maryland.

He and Jack Dongarra of the University of Tennessee and Oak Ridge National Laboratory

have put together a sampling that might have made Robert Frost beam with pride--had the poet been a computer jock.

Their list of 10 algorithms having "the greatest influence on the development and practice of science and engineering in the 20th century" appears in the January/February issue of Computing in Science & Engineering.

If you use a computer, some of these algorithms are no doubt crunching your data as you read this.

The drum roll, please:

1. **1946: The Metropolis Algorithm for Monte Carlo.** Through the use of random processes, this algorithm offers an efficient way to stumble toward answers to problems that are too complicated to solve exactly.
2. **1947: Simplex Method for Linear Programming.** An elegant solution to a common problem in planning and decision-making.
3. **1950: Krylov Subspace Iteration Method.** A technique for rapidly solving the linear equations that abound in scientific computation.
4. **1951: The Decompositional Approach to Matrix Computations.** A suite of techniques for numerical linear algebra.
5. **1957: The Fortran Optimizing Compiler.** Turns high-level code into efficient computer-readable code.
6. **1959: QR Algorithm for Computing Eigenvalues.** Another crucial matrix operation made swift and practical.
7. **1962: Quicksort Algorithms for Sorting.** For the efficient handling of large databases.
8. **1965: Fast Fourier Transform.** Perhaps the most ubiquitous algorithm in use today, it breaks down waveforms (like sound) into periodic components.
9. **1977: Integer Relation Detection.** A fast method for spotting simple equations satisfied by collections of seemingly unrelated numbers.
10. **1987: Fast Multipole Method.** A breakthrough in dealing with the complexity of n-body calculations, applied in problems ranging from celestial mechanics to protein folding.

Ordinamento

INPUT: un insieme di n oggetti a_1, a_2, \dots, a_n
presi da un dominio totalmente ordinato secondo \leq

OUTPUT: una permutazione degli oggetti a'_1, a'_2, \dots, a'_n tale che
 $a'_1 \leq a'_2 \leq \dots \leq a'_n$

Applicazioni:

- Ordinare alfabeticamente lista di nomi, o insieme di numeri, o insieme di compiti d'esame in base a cognome studente
- Velocizzare altre operazioni (per es. è possibile effettuare ricerche in array ordinati in tempo $O(\log n)$)
- Subroutine di molti algoritmi (per es. *greedy*)
-

Algoritmi per l'ordinamento

Data l'importanza, esistono svariati algoritmi di ordinamento, basati su tecniche diverse:

Insertionsort

Selectionsort

Heapsort

Mergesort

Quicksort

Bubblesort

Countingsort

.....

Ognuno con i suoi aspetti positivi e negativi.

Il Mergesort e il Quicksort sono entrambi basati sulla tecnica Divide et Impera, ma risultano avere differenti prestazioni

Mergesort

Dato un array di n elementi

I) **Divide**: trova l'indice della posizione centrale e divide l'array in due parti ciascuna con $n/2$ elementi (più precisamente $\lfloor n/2 \rfloor$ e $\lceil n/2 \rceil$)

II) Risolve i due sottoproblemi **ricorsivamente**

III) **Impera**: fonde i due sotto-array ordinati usando la procedura Merge

$$T(n) = \Theta(1) + 2T(n/2) + \Theta(n)$$

La soluzione è $T(n) = \Theta(n \log n)$

Nota:

Il tempo di esecuzione di Merge è $\Theta(n)$ (e non solo $O(n)$).

Infatti:

nel caso peggiore faremo $O(n)$ confronti:

1	3	5
---	---	---

2	4	6
---	---	---

nel caso migliore faremo $\Omega(n)$ confronti

1	2	3
---	---	---

4	5	6
---	---	---

Ricorda: Il tempo di esecuzione di un algoritmo è $\Theta(f(n))$ se nel caso peggiore è $O(f(n))$ e nel caso migliore è $\Omega(f(n))$

Il tempo di esecuzione di Mergesort è $\Theta(n \log n)$

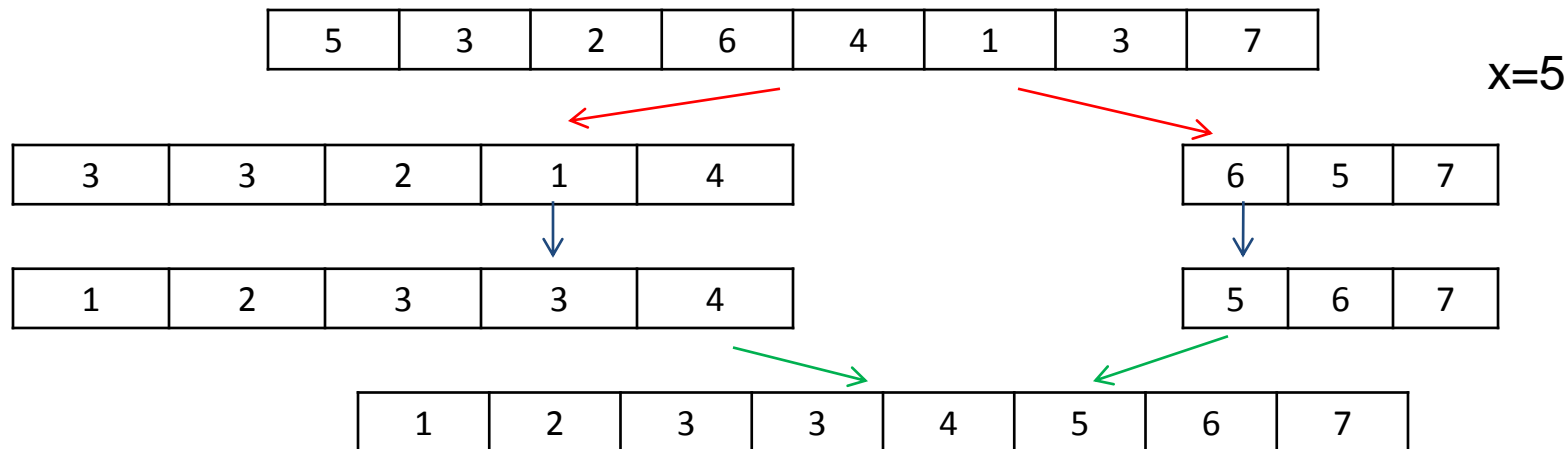
Quicksort

Dato un array di n elementi

I) **Divide**: scegli un elemento x dell'array (detto "pivot" o perno) e partiziona la sequenza in elementi $\leq x$ ed elementi $\geq x$

II) Risolvi i due sottoproblemi **ricorsivamente**

III) **Impera**: restituisci la concatenazione dei due sotto-array ordinati



Scelta del pivot

L'algoritmo funziona per qualsiasi scelta (primo / ultimo / ...),
ma se vogliamo algoritmo “deterministico” devo fissare la scelta;
nel seguito sceglieremo il **primo**.

Altrimenti: scelgo “random” e avrò “algoritmi randomizzati”
(vedi Kleinberg & Tardos, dopo)

Partizionamento

Partiziona l'array in elementi $\leq x$ ed elementi $\geq x$

Banalmente:

scorro l'array da 1 ad n e inserisco gli elementi \leq pivot in un nuovo array e quelli \geq del pivot in un altro nuovo array

Però:

1) avrei bisogno di array ausiliari

2) di che dimensione? I due sotto-array hanno un numero variabile di elementi

Partizione “in loco”

Partition:

- pivot = $A[1]$
- Scorri l'array da destra verso sinistra (con un indice j) e da sinistra verso destra (con un indice i) :
 - da destra verso sinistra, ci si ferma su un elemento $<$ del pivot
 - da sinistra verso destra, ci si ferma su un elemento $>$ del pivot;
- Scambia gli elementi
- Riprendi la scansione finché i e j si incrociano

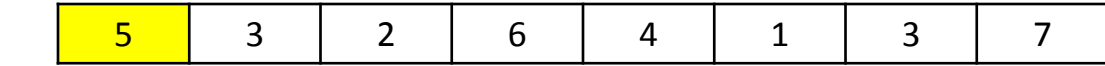
Partition (Hoare 1962)

```
Partition (A, p, r)

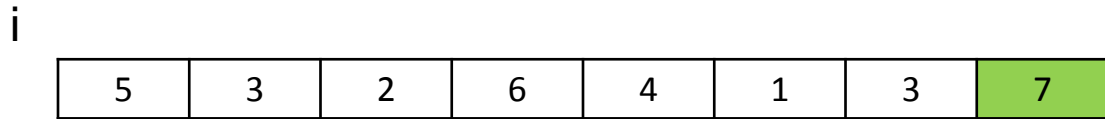
x = A[p]
i = p-1
j = r+1
while True
    do repeat j=j-1 until A[j] ≤ x
    repeat i=i+1 until A[i] ≥ x
    if i < j
        then scambia A[i] ↔ A[j]
    else return j
```

Esiste un diverso algoritmo per il partizionamento, e quindi per il Quicksort, dovuto a N. Lomuto ed esistono piccole varianti di questo (che potreste incontrare cambiando libro di testo)

Partizione in loco: un esempio



pivot = 5



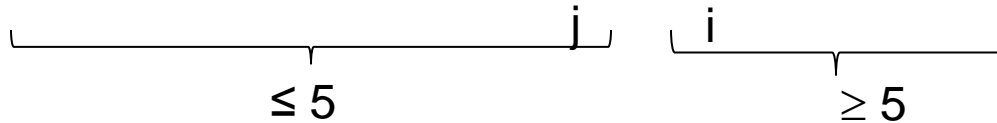
Scambia 3 con 5



Scambia 1 con 6



Restituisce $q = j$



Correttezza di Partition

Perché funziona?

Ad ogni iterazione (quando raggiungo il while):

la “parte verde” di sinistra (da p ad i) contiene elementi ≤ 5 ;

la “parte verde” di destra (da j a r) contiene elementi ≥ 5 .

Tale affermazione è vera all’inizio e si mantiene vera ad ogni iterazione (per induzione)

Analisi Partition

Il tempo di esecuzione è $\Theta(n)$


```
Quicksort (A, p, r)
```

```
  if p < r then  
    q = Partition (A, p, r)  
    Quicksort(A, p, q)  
    Quicksort(A, q+1, r)
```

Correttezza: la concatenazione di due array ordinati in cui l'array di sinistra contiene elementi minori o uguali degli elementi dell'array di destra è un array ordinato

Analisi: $T(n) = \Theta(n) + T(k) + T(n-k)$

Se k sono gli elementi da p a q (e $n-k$ i rimanenti da $q+1$ a r)
con $1 \leq k \leq n-1$

Analisi Quicksort (caso peggiore)

Un primo caso: ad ogni passo il pivot scelto è il minimo o il massimo degli elementi nell'array (la partizione è $1 \mid n-1$):

$$T(n) = T(n-1) + T(1) + \Theta(n)$$

essendo $T(1) = \Theta(1)$

$$T(n) = T(n-1) + \Theta(n)$$

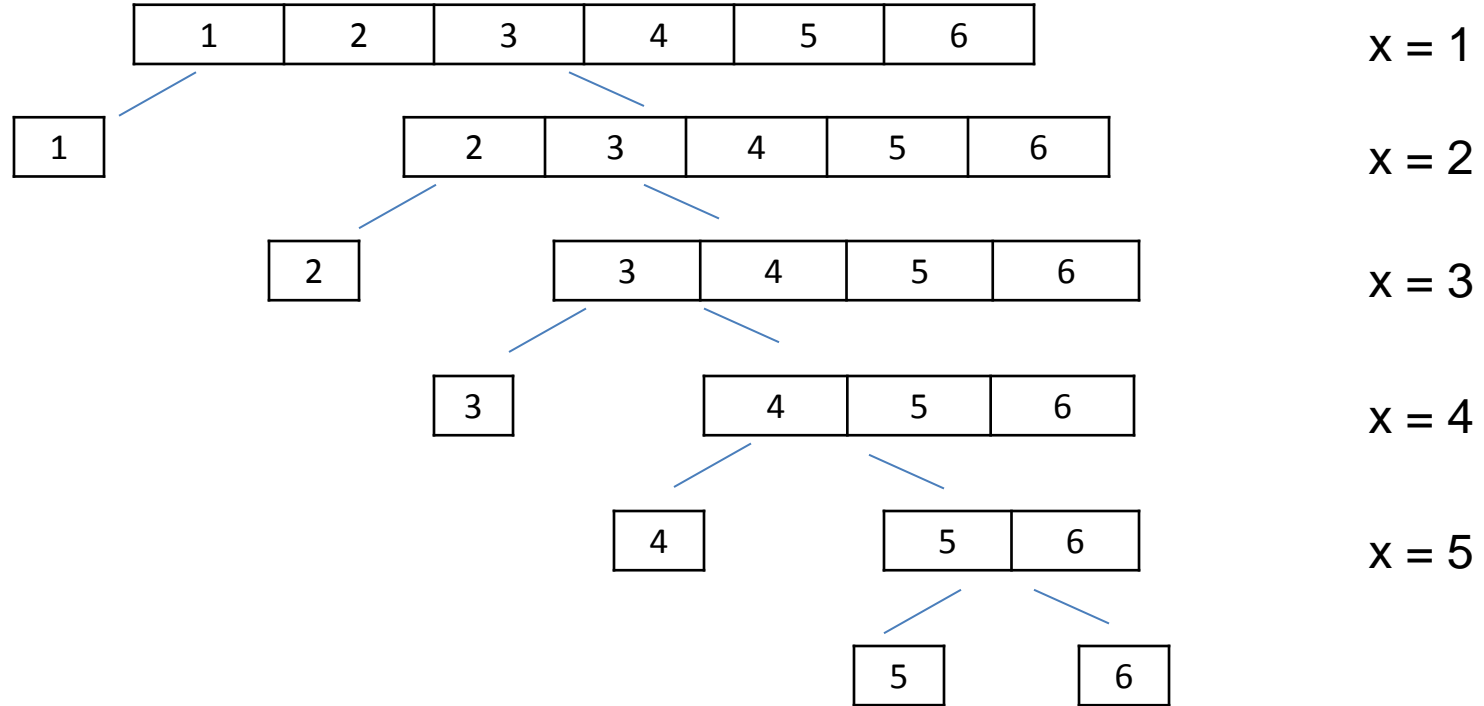
La cui soluzione è $T(n) = \Theta(n^2)$

Si può dimostrare che questo è il **caso peggiore**; quindi per il Quicksort:

$$T(n) = O(n^2)$$

Un esempio del caso peggiore del Quicksort

Un array ordinato



Analisi Quicksort (caso migliore)

Un altro caso: ad ogni passo il pivot scelto è la “mediana” degli elementi nell’array (la partizione è $n/2 \mid n/2$):

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

La cui soluzione è $T(n) = \Theta(n \log n)$

(è la stessa relazione di ricorrenza del Mergesort)

Si può dimostrare che questo è il **caso migliore**; quindi: $T(n) = \Omega(n \log n)$

Riassumendo, per il Quicksort: $T(n) = O(n^2)$ e $T(n) = \Omega(n \log n)$

Il caso migliore è diverso dal caso peggiore quindi

$T(n)$ **non** è Θ di nessuna funzione

Is Quicksort ... quick?

Il **Quicksort** non ha un «buon» caso peggiore, ma ha un **buon caso medio** (si può dimostrare che anche nel caso medio si comporta come nel caso migliore), per cui si può considerare una sua versione «randomizzata»

Algoritmo randomizzato:

- Introduce una chiamata a **random(a, b)** (che restituisce un numero a caso fra a e b ($a < b$))
- Forza l'algoritmo a comportarsi come nel caso medio
- Non esiste una distribuzione d'input «peggiore» a priori

Nota: sul libro di testo trovate solo una versione randomizzata.

Per il resto potete fare riferimento al libro di Cormen, Leiserson, Rivest, (Stein)

Introduzione agli algoritmi, o ad altri testi consigliati.

QuickSort randomizzato

```
Random-Partition (A, p, r)
```

```
  i ← random(p,r)
```

```
  scambia A[i] <-> A[p]
```

```
  return Partition(A, p, r)
```

```
  if p < r then
```

```
    q ← Random-Partition (A,p,r)
```

```
    Random-Quicksort(A, p, q)
```

```
    Random-Quicksort(A, q+1, r)
```

Quicksort vs Mergesort

Fase		MergeSort	Tempi	QuickSort	Tempi
I	Divide	$q = \lfloor (p+r)/2 \rfloor$	$\Theta(1)$	PARTITION	$\Theta(n)$
II	Ricorsione	$\lfloor n/2 \rfloor \mid \lceil n/2 \rceil$	$2T(n/2)$	$k \mid n - k$	$T(k) + T(n-k)$
III	Combina	MERGE	$\Theta(n)$	niente	$\Theta(1)$
			$T(n) = 2T(n/2) + \Theta(n)$		$T(n) = T(k) + T(n-k) + \Theta(n)$
			$T(n) = \Theta(n \log n)$		$T(n) = O(n^2), T(n) = \Omega(n \log n)$

Da ricordare sulla complessità dell'ordinamento

Esistono algoritmi di ordinamento con tempo nel caso peggiore $\Theta(n^2)$ e $\Theta(n \log n)$

Esistono anche algoritmi di ordinamento con tempo nel caso peggiore $\Theta(n)$, ma ... **non** sono basati sui confronti e funzionano **solo** sotto certe ipotesi.

Inoltre si può dimostrare che **tutti** gli algoritmi di ordinamento basati sui confronti richiedono $\Omega(n \log n)$ confronti nel caso peggiore!

Si dice che $\Omega(n \log n)$ è una delimitazione inferiore (*lower bound*) al problema dell'ordinamento, cioè al numero di confronti richiesti per ordinare n oggetti.

Delimitazione inferiore (*lower bound*) =

quantità di risorsa **necessaria** per risolvere un determinato problema

Indica la difficoltà intrinseca del problema.

5.5 Integer Multiplication

Divide –et – Impera per la moltiplicazione

Esprimere il prodotto di due interi a n cifre tramite prodotti di due interi con un **numero inferiore** di cifre.

Esempio: (base 10) $123.456 = 123 \cdot 1000 + 456 = 123 \cdot 10^3 + 456$

(base 2) $1101 = 11 \cdot 2^2 + 01$

$$\begin{aligned} 1101 &= 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = \\ &= (1 \cdot 2 + 1) \cdot 2^2 + (0 \cdot 2^1 + 1 \cdot 2^0) \end{aligned}$$

Dato un intero x a n bit

$$x = x_1 \cdot 2^{n/2} + x_2$$

dove x_1 e x_2 hanno $n/2$ bit

Divide-and-Conquer Multiplication: Warmup

To multiply two n -digit integers:

Multiply four pairs of $\frac{1}{2}n$ -digit integers.

Add two pairs of $\frac{1}{2}n$ -digit integers, and shift to obtain result.

$$\begin{aligned}x &= 2^{n/2} \cdot x_1 + x_0 \\y &= 2^{n/2} \cdot y_1 + y_0 \\xy &= (2^{n/2} \cdot x_1 + x_0)(2^{n/2} \cdot y_1 + y_0) = 2^n \cdot x_1 y_1 + 2^{n/2} \cdot (x_1 y_0 + x_0 y_1) + x_0 y_0\end{aligned}$$

$$T(n) = \underbrace{4T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n)}_{\text{add, shift}} \Rightarrow T(n) = \Theta(n^2)$$

↑
assumes n is a power of 2

Karatsuba Multiplication

To multiply two n-digit integers:

Add two pairs of $\frac{1}{2}n$ digit integers.

Multiply three different pairs of $\frac{1}{2}n$ -digit integers (A, B, C).

Add, subtract, and shift $\frac{1}{2}n$ -digit integers to obtain result.

$$\begin{aligned}x &= 2^{n/2} \cdot x_1 + x_0 \\y &= 2^{n/2} \cdot y_1 + y_0 \\xy &= 2^n \cdot x_1 y_1 + 2^{n/2} \cdot (x_1 y_0 + x_0 y_1) + x_0 y_0 \\&= 2^n \cdot x_1 y_1 + 2^{n/2} \cdot ((x_1 + x_0)(y_1 + y_0) - x_1 y_1 - x_0 y_0) + x_0 y_0\end{aligned}$$

A

B

A

C

C

Theorem. [Karatsuba-Ofman, 1962]

Can multiply two n-digit integers in $O(n^{1.585})$ bit operations.

Karatsuba Algorithm

Recursive-Multiply(x,y):

Write $x = x_1 \cdot 2^{n/2} + x_0$

$y = y_1 \cdot 2^{n/2} + y_0$

Compute $x_1 + x_0$ and $y_1 + y_0$

$p = \text{Recursive-Multiply}(x_1 + x_0, y_1 + y_0)$

$x_1y_1 = \text{Recursive-Multiply}(x_1, y_1)$

$x_0y_0 = \text{Recursive-Multiply}(x_0, y_0)$

Return $x_1y_1 \cdot 2^n + (p - x_1y_1 - x_0y_0) \cdot 2^{n/2} + x_0y_0$

$$T(n) \leq \underbrace{T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + T(1 + \lceil n/2 \rceil)}_{\text{recursive calls}} + \underbrace{\Theta(n)}_{\text{add, subtract, shift}}$$

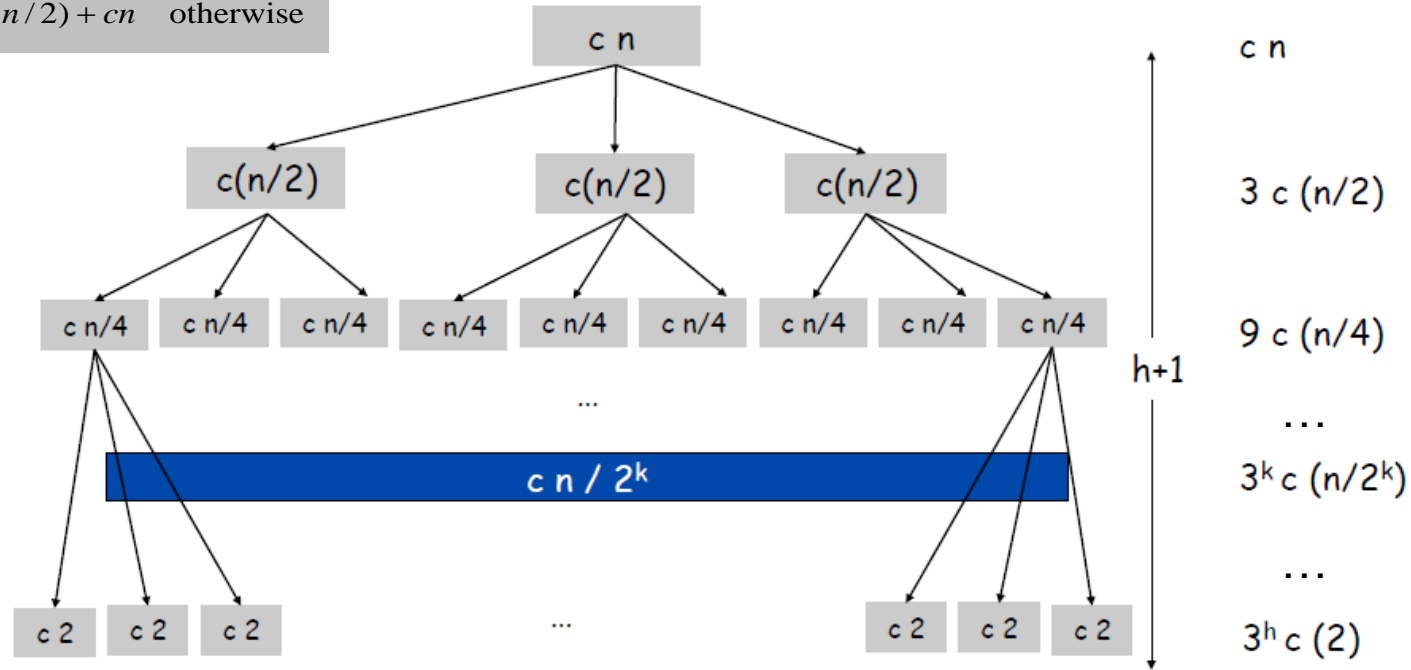
$$\Rightarrow T(n) = O(n^{\log_2 3}) = O(n^{1.585})$$

Per semplificare risolveremo:

$$T(n) = \begin{cases} 2 & \text{if } n = 2 \\ 3T(n/2) + n & \text{otherwise} \end{cases}$$

Soluzione con albero di ricorsione

$$T(n) = \begin{cases} 2 & \text{if } n = 2 \\ 3T(n/2) + cn & \text{otherwise} \end{cases}$$



$$n / 2^h = 2 \quad \text{cioè} \quad h = \log_2 n - 1$$

$$T(n) = \sum_{k=0}^{\log_2 n - 1} cn \left(\frac{3}{2}\right)^k$$

Un po' di calcoli...

$$\begin{aligned} T(n) &= \sum_{k=0}^{\log_2 n - 1} cn \left(\frac{3}{2}\right)^k = cn \sum_{k=0}^{\log_2 n - 1} \left(\frac{3}{2}\right)^k = cn \frac{\left(\frac{3}{2}\right)^{\log_2 n} - 1}{\frac{3}{2} - 1} = cn \frac{\left(\frac{3}{2}\right)^{\log_2 n} - 1}{\frac{1}{2}} \\ &\leq cn \frac{\left(\frac{3}{2}\right)^{\log_2 n}}{\frac{1}{2}} = 2cn \frac{3^{\log_2 n}}{2^{\log_2 n}} = 2c \times 3^{\log_2 n} = 2c \times n^{\log_2 3} \end{aligned}$$

$$T(n) = O(n^{\log_2 3})$$

MEMENTO
(da ricordare)

$$\sum_{k=0}^N a^k = \frac{a^{1+N} - 1}{a - 1}$$

$$\text{Inoltre: } \sum_{k=0}^N a^k \leq \sum_{k=0}^{\infty} a^k = \lim_{n \rightarrow \infty} \frac{a^{1+N} - 1}{a - 1} = \begin{cases} \frac{1}{1-a} & \text{se } a < 1 \\ \text{diverge} & \text{altrimenti} \end{cases}$$

Matrix Multiplication and decimal wars!

(non in programma, lettura facoltativa)

Matrix Multiplication

Matrix multiplication. Given two n-by-n matrices A and B, compute $C = AB$.

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$$

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

Matrix Multiplication: Warmup

Divide-and-conquer.

Divide: partition A and B into $\frac{1}{2}n$ -by- $\frac{1}{2}n$ blocks.

Conquer: multiply 8 $\frac{1}{2}n$ -by- $\frac{1}{2}n$ recursively.

Combine: add appropriate products using 4 matrix additions.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\begin{aligned} C_{11} &= (A_{11} \times B_{11}) + (A_{12} \times B_{21}) \\ C_{12} &= (A_{11} \times B_{12}) + (A_{12} \times B_{22}) \\ C_{21} &= (A_{21} \times B_{11}) + (A_{22} \times B_{21}) \\ C_{22} &= (A_{21} \times B_{12}) + (A_{22} \times B_{22}) \end{aligned}$$

$$T(n) = \underbrace{8T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n^2)}_{\text{add, form submatrices}} \Rightarrow T(n) = \Theta(n^3)$$

Matrix Multiplication: Key Idea

Key idea. multiply 2-by-2 block matrices with only 7 multiplications.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$C_{11} = P_5 + P_4 - P_2 + P_6$$

$$C_{12} = P_1 + P_2$$

$$C_{21} = P_3 + P_4$$

$$C_{22} = P_5 + P_1 - P_3 - P_7$$

$$P_1 = A_{11} \times (B_{12} - B_{22})$$

$$P_2 = (A_{11} + A_{12}) \times B_{22}$$

$$P_3 = (A_{21} + A_{22}) \times B_{11}$$

$$P_4 = A_{22} \times (B_{21} - B_{11})$$

$$P_5 = (A_{11} + A_{22}) \times (B_{11} + B_{22})$$

$$P_6 = (A_{12} - A_{22}) \times (B_{21} + B_{22})$$

$$P_7 = (A_{11} - A_{21}) \times (B_{11} + B_{12})$$

Fast Matrix Multiplication

Fast matrix multiplication. (Strassen, 1969)

Divide: partition A and B into $\frac{1}{2}n$ -by- $\frac{1}{2}n$ blocks.

Compute: 14 $\frac{1}{2}n$ -by- $\frac{1}{2}n$ matrices via 10 matrix additions.

Conquer: multiply 7 $\frac{1}{2}n$ -by- $\frac{1}{2}n$ matrices recursively.

Combine: 7 products into 4 terms using 8 matrix additions.

Analysis.

Assume n is a power of 2.

$T(n)$ = # arithmetic operations.

$$T(n) = \underbrace{7T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n^2)}_{\text{add, subtract}} \Rightarrow T(n) = \Theta(n^{\log_2 7}) = O(n^{2.81})$$

Fast Matrix Multiplication in Theory

Q. Multiply two 2-by-2 matrices with only 7 scalar multiplications?

A. Yes! [Strassen, 1969]

$$\Theta(n^{\log_2 7}) = O(n^{2.81})$$

Q. Multiply two 2-by-2 matrices with only 6 scalar multiplications?

A. Impossible. [Hopcroft and Kerr, 1971]

$$\Theta(n^{\log_2 6}) = O(n^{2.59})$$

Q. Two 3-by-3 matrices with only 21 scalar multiplications?

A. Also impossible.

$$\Theta(n^{\log_3 21}) = O(n^{2.77})$$

Q. Two 70-by-70 matrices with only 143,640 scalar multiplications?

A. Yes! [Pan, 1980]

$$\Theta(n^{\log_{70} 143640}) = O(n^{2.80})$$

Decimal wars!

December, 1979: $O(n^{2.521813})$

January, 1980: $O(n^{2.521801})$

Fast Matrix Multiplication in Theory

Best known. $O(n^{2.376})$ [Coppersmith-Winograd, 1987.]

Conjecture. $O(n^{2+\varepsilon})$ for any $\varepsilon > 0$.

Caveat. Theoretical improvements to Strassen are progressively less practical.

6.6 *Sequence Alignment*

E' capitato anche a voi?

Di digitare sul computer una parola in maniera sbagliata (per esempio usando un dizionario sul Web):

AGORITNI

E sentirsi chiedere:

«Forse cercavi ALGORITMI?»

Come fanno a capirlo? Sanno veramente cosa abbiamo in mente????

Non trovando AGORITNI sul dizionario ha cercato una parola «simile», «vicina»

String Similarity

How similar are two strings?

ocurrance

occurrence

o c u r r a n c e -

o c c u r r e n c e

6 mismatches, 1 gap

o c - u r r a n c e

o c c u r r e n c e

1 mismatch, 1 gap

o c - u r r - a n c e

o c c u r r e - n c e

0 mismatches, 3 gaps

Edit Distance

Applications.

Basis for Unix diff

Speech recognition

Computational biology (sequenze di simboli nel DNA rappresentano proprietà degli organismi)

Edit distance. [Levenshtein 1966, Needleman-Wunsch 1970]

Gap penalty δ ; mismatch penalty α_{pq} (you may assume $\alpha_{pp}=0$).

Cost = sum of gap and mismatch penalties.

C T G A C C T A C C T

- C T G A C C T A C C T

C C T G A C T A C A T

C C T G A C - T A C A T

$$\alpha_{TC} + \alpha_{GT} + \alpha_{AG} + 2\alpha_{CA}$$

$$2\delta + \alpha_{CA}$$

Sequence Alignment

Goal: Given two strings $X = x_1 x_2 \dots x_m$ and $Y = y_1 y_2 \dots y_n$ find alignment of minimum cost.

Def. An **alignment** M is a set of ordered pairs $x_i - y_j$ such that each item occurs in at most one pair and no crossings.

Def. The pair $x_i - y_j$ and $x_{i'} - y_{j'}$ **cross** if $i < i'$, but $j > j'$.

$$\text{cost}(M) = \underbrace{\sum_{(x_i, y_j) \in M} \alpha_{x_i y_j}}_{\text{mismatch}} + \underbrace{\sum_{i: x_i \text{ unmatched}} \delta + \sum_{j: y_j \text{ unmatched}} \delta}_{\text{gap}}$$

Ex: CTACCG **vs** TACATG.

Sol: $M = x_2 - y_1, x_3 - y_2, x_4 - y_3, x_5 - y_4, x_6 - y_6$.

x_1	x_2	x_3	x_4	x_5		x_6
C	T	A	C	C	-	G
-	T	A	C	A	T	G
	y_1	y_2	y_3	y_4	y_5	y_6

Sequence Alignment: Problem Structure

Def. $OPT(i, j)$ = min cost of aligning strings $x_1 x_2 \dots x_i$ and $y_1 y_2 \dots y_j$.

Case 1: OPT matches x_i - y_j .

pay mismatch for x_i - y_j + min cost of aligning two strings
 $x_1 x_2 \dots x_{i-1}$ and $y_1 y_2 \dots y_{j-1}$

Case 2a: OPT leaves x_i unmatched.

pay gap for x_i and min cost of aligning $x_1 x_2 \dots x_{i-1}$ and $y_1 y_2 \dots y_j$

Case 2b: OPT leaves y_j unmatched.

pay gap for y_j and min cost of aligning $x_1 x_2 \dots x_i$ and $y_1 y_2 \dots y_{j-1}$

$$OPT(i, j) = \begin{cases} j\delta & \text{if } i = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & \text{otherwise} \\ i\delta & \text{if } j = 0 \end{cases}$$

Sequence Alignment: Algorithm

```
Sequence-Alignment( $m, n, x_1x_2\dots x_m, y_1y_2\dots y_n, \delta, \alpha$ ) {  
  for  $i = 0$  to  $m$   
     $M[0, i] = i\delta$   
  for  $j = 0$  to  $n$   
     $M[j, 0] = j\delta$   
  
  for  $i = 1$  to  $m$   
    for  $j = 1$  to  $n$   
       $M[i, j] = \min(\alpha[x_i, y_j] + M[i-1, j-1],$   
                     $\delta + M[i-1, j],$   
                     $\delta + M[i, j-1])$   
  
  return  $M[m, n]$   
}
```

Analysis. $\Theta(mn)$ time and space.

English words or sentences: $m, n \leq 10$.

Computational biology: $m = n = 100,000$. 10 billions ops OK, but 10GB array?

Sequence Alignment: Linear **Space**

Q. Can we avoid using quadratic **space**?

Easy. Optimal **value** in $O(m + n)$ space and $O(mn)$ time.

Compute $\text{OPT}(i, \cdot)$ from $\text{OPT}(i-1, \cdot)$.

No longer a simple way to recover alignment itself.

Theorem. [Hirschberg 1975]

Optimal **alignment** in $O(m + n)$ **space** and $O(mn)$ time.

Clever combination of divide-and-conquer and dynamic programming.

Inspired by idea of Savitch from complexity theory.

Ricostruzione dell'allineamento

$$OPT(i,j) = \begin{cases} j\delta & \text{if } i = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & \text{otherwise} \\ i\delta & \text{if } j = 0 \end{cases}$$

Per ricostruire l'allineamento seguiamo il percorso all'indietro nella matrice

Esempio: X = mean, Y = name

$\delta = 2$

- costo mismatch fra vocali differenti=1
 - costo mismatch fra consonanti differenti=1
 - costo mismatch fra vocale e consonante=3
- La freccia nella casella (i,j) proviene dalla casella usata per ottenere il minimo

$$M[4,4] = \min\{\alpha_{ne} + M[3,3], \delta + M[3,4], \delta + M[4,3]\}$$

$$= \min\{3 + 5, 2 + 5, 2 + 4\} = 6$$

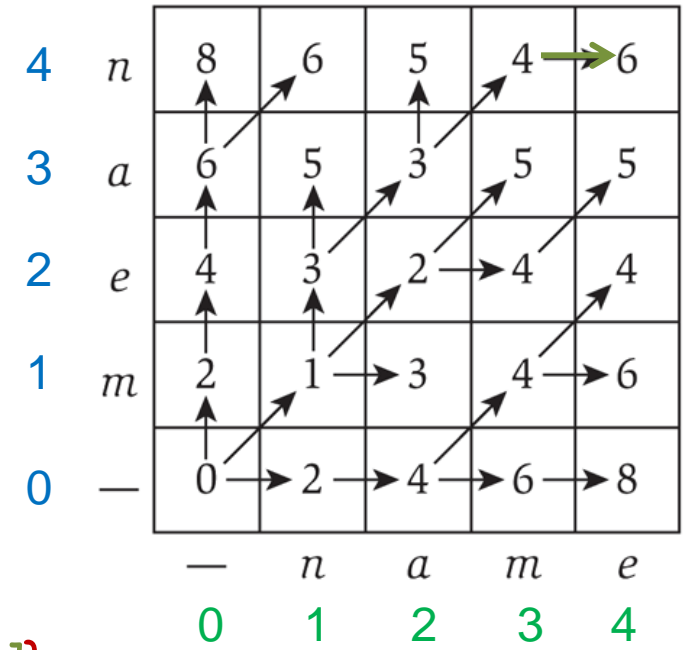
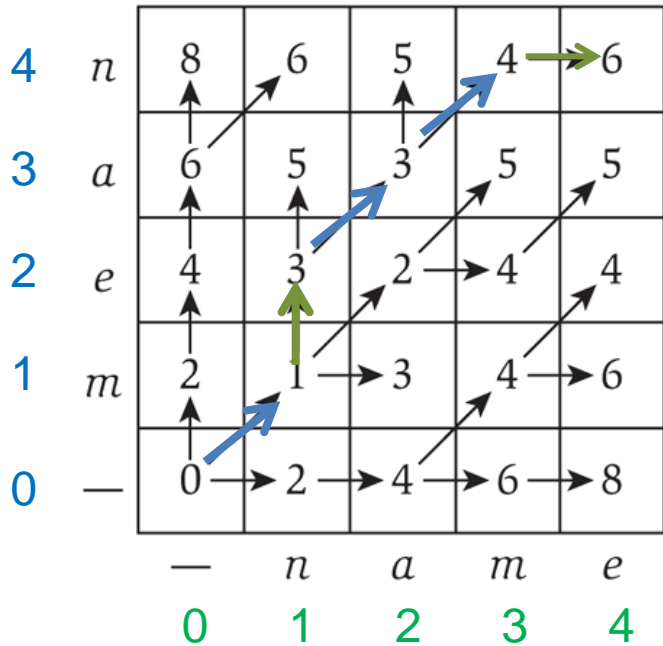


Figure 6.18 The OPT values for the problem of aligning the words *mean* to *name*.

Ricostruzione soluzione ottima



m e a n -

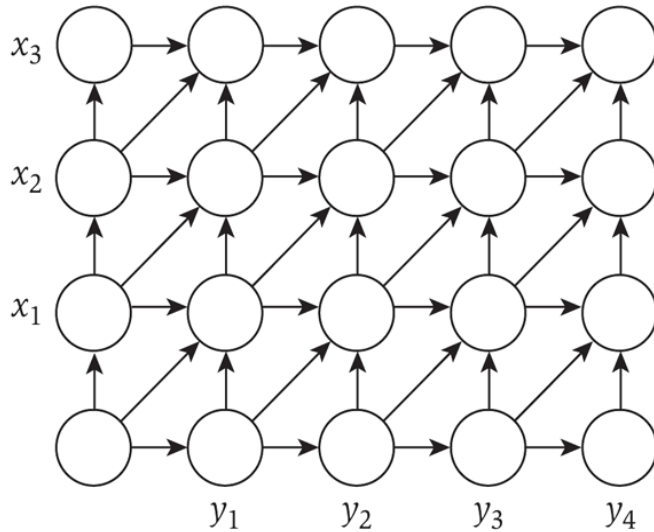
n - a m e

1 2 1 2 = 6

Figure 6.18 The OPT values for the problem of aligning the words *mean* to *name*.

Una visione grafica del problema del sequence alignment

Ad $X = x_1 x_2 x_3$ e $Y = y_1 y_2 y_3 y_4$ associamo il grafo G_{xy} seguente:



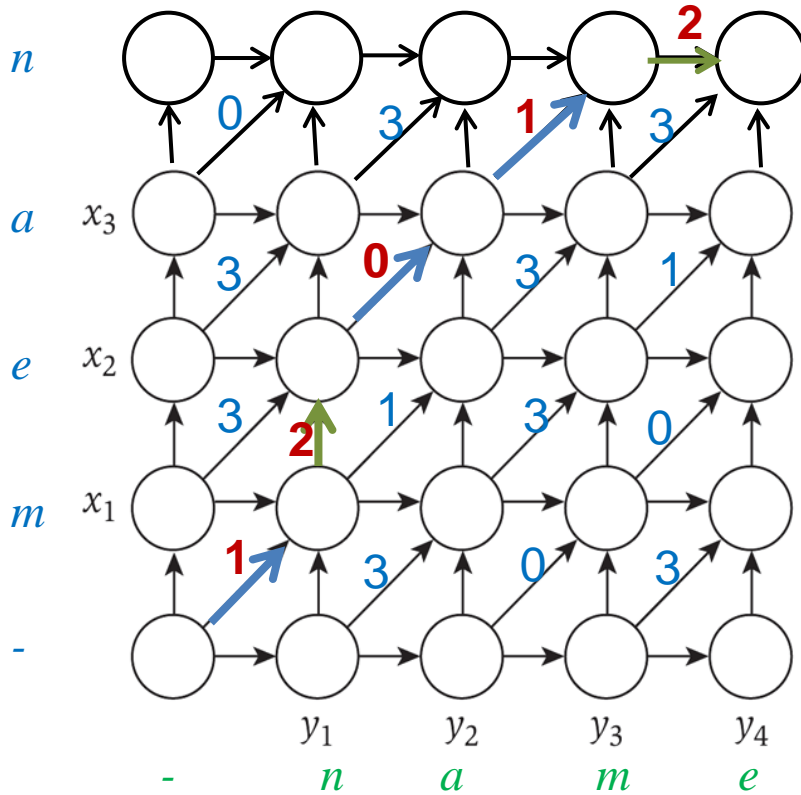
nodo (i,j) in corrispondenza di x_i e y_j

costi archi orizzontali e verticali = δ

costo arco diagonale verso $(i,j) = \alpha_{x_i, y_j}$

Figure 6.17 A graph-based picture of sequence alignment.

Un esempio



costi archi orizzontali e verticali = δ
 costo arco diagonale verso $(i,j) = \alpha_{x_i, y_j}$

$\delta = 2$

costo mismatch fra vocali differenti=1
 costo mismatch fra consonanti differenti=1
 costo mismatch fra vocale e consonante=3

Figure 6.17 A graph-based picture of sequence alignment.

Trovare allineamento di costo minimo fra X e Y equivale a cercare **il cammino di costo minimo** in G_{XY} dal nodo in basso a sinistra a quello in alto a destra.

Prova per induzione (fare)

Studieremo algoritmi per la ricerca di cammini di costo minimo in un grafo, ma non apporteranno miglioramenti al tempo di esecuzione.

Esercizi: varianti al problema dello zaino

Problema dello zaino: Esercizio 1

Si descriva ed analizzi un algoritmo per la seguente variazione del problema dello zaino: Dati n oggetti di peso w_1, w_2, \dots, w_n e valore v_1, v_2, \dots, v_n ed uno zaino di capacità W (tutti gli input sono >0), trovare il massimo valore di un sottoinsieme degli oggetti il cui peso totale è al massimo W , con la condizione che ogni oggetto può essere preso anche più di una volta.

(La variazione rispetto al problema del testo, consiste nel superamento del vincolo che ogni oggetto poteva essere preso al massimo una sola volta.)

Problema dello zaino: Esercizio 2

Si descriva ed analizzi un algoritmo per la seguente variazione del problema dello zaino: Dati n oggetti di peso w_1, w_2, \dots, w_n e valore v_1, v_2, \dots, v_n ed uno zaino di capacità W (tutti gli input sono >0), trovare il massimo valore di un sottoinsieme degli oggetti il cui peso totale è al massimo W , con la condizione che ogni oggetto può essere preso al massimo 2 volte.

(La variazione rispetto al problema del testo, consiste nel superamento del vincolo che ogni oggetto poteva essere preso al massimo una sola volta.)

Problema dello zaino: Esercizio 3

Si descriva ed analizzi un algoritmo per la seguente variazione del problema dello zaino: Dati n oggetti di peso w_1, w_2, \dots, w_n e valore v_1, v_2, \dots, v_n ed uno zaino di capacità W (tutti gli input sono >0), trovare il massimo valore di un sottoinsieme degli oggetti il cui peso totale è al massimo W , con la condizione che non possono essere presi due oggetti con indici consecutivi (ovvero gli oggetti i -esimo ed $(i+1)$ -esimo, per $i=1, 2, \dots, n-1$).

Dall'elenco

Esercizio 1

Lungo un fiume ci sono n approdi. A ciascuno di questi approdi é possibile fittare una canoa che puó essere restituita ad un altro approdo. E' praticamente impossibile andare controcorrente. Il costo del fitto di una canoa da un punto di partenza i ad un punto di arrivo j , con $i < j$, é denotato con $C(i, j)$. E' possibile che per andare da i a j sia piú economico effettuare alcune soste e cambiare la canoa piuttosto che fittare una unica canoa. Se si fitta una nuova canoa in $k_1 < k_2 < \dots < k_l$ allora il costo totale per il fitto é $C(i, k_1) + C(k_1, k_2) + \dots + C(k_{l-1}, k_l) + C(k_l, j)$.

Descrivere un algoritmo che dato in input i costi $C(i, j)$, determini il costo minimo per recarsi da 1 ad n . Analizzare la complessitá dell'algoritmo proposto.

Esempio. Sia $n = 4$, e $C(1, 2) = 1$, $C(1, 3) = 2$, $C(1, 4) = 4$, $C(2, 3) = 1$, $C(2, 4) = 1$, $C(3, 4) = 1$. Allora i possibili modi per andare da 1 a 4 sono: $1 \rightarrow 4$, $1 \rightarrow 2 \rightarrow 4$, $1 \rightarrow 3 \rightarrow 4$, $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$. I rispettivi costi sono: 4, 2, 3, 3. Il costo minimo é quindi 2.

Suggerimento: si usi la tecnica della scelta binaria.

Dall'elenco

Esercizio 2

Sia dato un grafo orientato con n vertici v_1, v_2, \dots, v_n in cui sono presenti solo gli archi $v_i \rightarrow v_j$ con $i < j$ e ad ogni arco $v_i \rightarrow v_j$ é associato un costo $c_{i,j}$, dove $c_{i,j}$ é un intero positivo.

Descrivere ed analizzare un algoritmo che, dato k , con $0 \leq k \leq n-2$, computi il costo minimo di un cammino da v_1 a v_n che attraversi esattamente altri k vertici.

Esempio. Sia G un grafo con 5 vertici v_1, v_2, v_3, v_4, v_5 e sia $c_{i,j} = (2j - i)$, con $i < j$, il costo associato all'arco $v_i \rightarrow v_j$. Sia $k = 2$. Allora i cammini da v_1 a v_5 che attraversano esattamente altri 2 vertici sono: $v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_5$, $v_1 \rightarrow v_2 \rightarrow v_4 \rightarrow v_5$, $v_1 \rightarrow v_3 \rightarrow v_4 \rightarrow v_5$, i cui costi sono, rispettivamente, 14, 15, e 16.

Suggerimento: si usi la tecnica dell'aggiunta di una variabile.

Esercizio 1 pag. 312 KT punto a) [continua...]

Exercises

1. Let $G = (V, E)$ be an undirected graph with n nodes. Recall that a subset of the nodes is called an *independent set* if no two of them are joined by an edge. Finding large independent sets is difficult in general; but here we'll see that it can be done efficiently if the graph is "simple" enough.

Call a graph $G = (V, E)$ a *path* if its nodes can be written as v_1, v_2, \dots, v_n , with an edge between v_i and v_j if and only if the numbers i and j differ by exactly 1. With each node v_i , we associate a positive integer *weight* w_i .

Consider, for example, the five-node path drawn in Figure 6.28. The *weights* are the numbers drawn inside the nodes.

The goal in this question is to solve the following problem:

Find an independent set in a path G whose total weight is as large as possible.

- (a) Give an example to show that the following algorithm *does not* always find an independent set of maximum total weight.

```
The "heaviest-first" greedy algorithm
Start with  $S$  equal to the empty set
While some node remains in  $G$ 
  Pick a node  $v_i$  of maximum weight
  Add  $v_i$  to  $S$ 
  Delete  $v_i$  and its neighbors from  $G$ 
Endwhile
Return  $S$ 
```

Esercizio 1 pag. 312 KT punti b) e c)

- (b) Give an example to show that the following algorithm also *does not* always find an independent set of maximum total weight.

Let S_1 be the set of all v_i where i is an odd number
Let S_2 be the set of all v_i where i is an even number
(Note that S_1 and S_2 are both independent sets)
Determine which of S_1 or S_2 has greater total weight,
and return this one

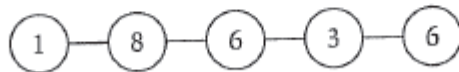


Figure 6.28 A path with weights on the nodes. The maximum weight of an independent set is 14.

- (c) Give an algorithm that takes an n -node path G with weights and returns an independent set of maximum total weight. The running time should be polynomial in n , independent of the values of the weights.

Tipici tempi di esecuzione e Divide-et-Impera

Linear Time: $O(n)$

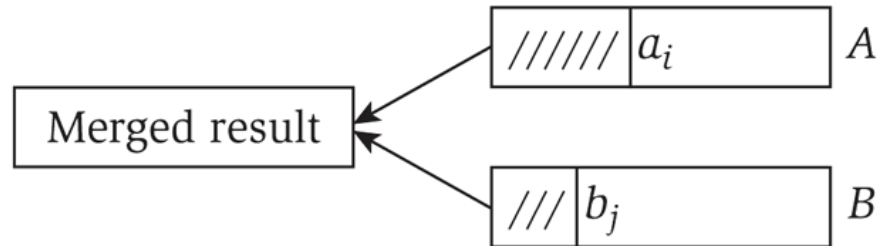
Linear time. Running time is at most a constant factor times the size of the input.

```
max ← a1
for i = 2 to n {
    if (ai > max)
        max ← ai
}
```

Computing the **maximum**. Compute maximum of n numbers a_1, \dots, a_n .

Linear Time: $O(n)$

Merge. Combine two sorted lists $A = a_1, a_2, \dots, a_n$ with $B = b_1, b_2, \dots, b_n$ into sorted whole.



```
i = 1, j = 1
while (both lists are nonempty) {
    if ( $a_i \leq b_j$ ) append  $a_i$  to output list and increment i
    else ( $a_i \leq b_j$ ) append  $b_j$  to output list and increment j
}
append remainder of nonempty list to output list
```

Claim. Merging two lists of size n takes $O(n)$ time.

Pf. After each comparison, the length of output list increases by 1.

Quadratic Time: $O(n^2)$

Quadratic time. Enumerate all pairs of elements.

Closest pair of points. Given a list of n points in the plane $(x_1, y_1), \dots, (x_n, y_n)$, find the pair that is closest.

$O(n^2)$ solution. Try all pairs of points.

```
min ← (x1 - x2)2 + (y1 - y2)2
for i = 1 to n {
  for j = i+1 to n {
    d ← (xi - xj)2 + (yi - yj)2
    if (d < min)
      min ← d
  }
}
```

← don't need to
take square roots

Cubic Time: $O(n^3)$

Cubic time. Enumerate all triples of elements.

Set disjointness. Given n sets S_1, \dots, S_n each of which is a subset of $1, 2, \dots, n$, is there some pair of these which are disjoint?

```
foreach set  $S_i$  {
  foreach other set  $S_j$  {
    foreach element  $p$  of  $S_i$  {
      determine whether  $p$  also belongs to  $S_j$ 
    }
    if (no element of  $S_i$  belongs to  $S_j$ )
      report that  $S_i$  and  $S_j$  are disjoint
  }
}
```

$O(n^3)$ solution. For each pairs of sets, determine if they are disjoint.

Polynomial Time: $O(n^k)$ Time

Independent set of size k. Given a graph, are there k nodes such that no two are joined by an edge?

k is a constant

$O(n^k)$ solution. Enumerate all subsets of k nodes.

```
foreach subset S of k nodes {  
    check whether S is an  
independent set  
    if (S is an independent set)  
        report S is an independent  
set  
    }  
}
```

Check whether S is an independent set = $O(k^2)$.

Number of k element subsets = $\binom{n}{k} = \frac{n(n-1)(n-2)\cdots(n-k+1)}{k(k-1)(k-2)\cdots(2)(1)} \leq \frac{n^k}{k!}$

$O(k^2 n^k / k!) = O(n^k)$.

Exponential Time

Independent set. Given a graph, what is maximum size of an independent set?

$O(n^2 2^n)$ solution. Enumerate all subsets.

```
S* ← φ
foreach subset S of nodes {
  check whether S is an independent set
  if (S is largest independent set seen so far)
    update S* ← S
}
```

Note the differences with **Independent set of size k.**

Sub-linear Time: $O(\log n)$

Tempo lineare: esamina tutto l'input eseguendo operazioni di tempo costante ad ogni passo

Tempo **sub**-lineare: Non è necessario esaminare **tutto** l'input!

Esempio. **Ricerca binaria:** ricerca di un elemento in un array **ordinato** (per esempio un vocabolario)

Tempo Logaritmico: $O(\log n)$

Esempio: Ricerca Binaria. Data una lista ordinata $A = a_1, \dots, a_n$ ed un valore key , determina l'indice i per cui $a_i = key$, se esso esiste.

```
first ← 1, last ← n
while (first ≤ last)
    mid ← (first + last)/2; (calcola punto mediano)
    if (key > amid)
        first = mid + 1; (ripete la ricerca nella metà di destra)
    else if (key < amid)
        last = mid - 1; (ripete la ricerca nella metà di sinistra)
    else
        return(mid)
return(non c'è)
```

Analisi

```
first ← 1, last ← n
while (first ≤ last)
  mid ← (first + last)/2; (calcola punto mediano)
  if (key > amid)
    first = mid + 1; (ripete la ricerca nella metà di destra)
  else if (key < amid)
    last = mid - 1; (ripete la ricerca nella metà di sinistra)
  else
    return(mid)
return(non c'è)
```

Dopo la prima iterazione, al più l'algoritmo riefettua la ricerca su $n/2$ elementi dopo la seconda iterazione, al più l'algoritmo riefettua la ricerca su $n/4 = n/2^2$ elementi, ... dopo la k -esima iterazione, al più l'algoritmo riefettua la ricerca su $n/2^k$ elementi. L'algoritmo si fermerà sicuramente al **primo** k per cui $n/2^k \leq 1$ (se non prima)
 $\Rightarrow k = O(\log n) \Rightarrow$ poichè il numero di operazioni in ciascuna delle $O(\log n)$ iterazioni è costante, il tempo di esecuzione totale è $O(\log n)$

$O(n \log n)$ Time

Molto comune perché

- E' il running time di **algoritmi divide-and-conquer** che dividono l'input in due parti, le risolvono ricorsivamente e poi combinano le soluzioni in tempo lineare.
- Running time di **algoritmi di ordinamento**.
Mergesort and Heapsort hanno usato $O(n \log n)$ confronti.
- Molti algoritmi usano l'ordinamento come passo più costoso. Per esempio molti algoritmi basati sulla **tecnica greedy**

Divide-and-Conquer

“Divide et impera”
Giulio Cesare

Divide-and-conquer.

- Break up problem into several parts.
- Solve each part recursively.
- Combine solutions to sub-problems into overall solution.

Examples: Binary Search, Mergesort,

Ricerca binaria (versione D-et-I)

Divide - et - Impera

- Dividi il problema in sottoproblemi
- Risolvi ogni sottoproblema ricorsivamente
- Combina le soluzioni ai sottoproblemi per ottenere la soluzione al problema

Ricerca binaria:

- Dividi l'array a metà (determinando l'elemento di mezzo)
- Risolvi ricorsivamente sulla metà di destra o di sinistra o su nessuna (a secondo del confronto con l'elemento di mezzo)
- Niente

Sorting

Sorting. Given n elements, rearrange in ascending order.

Obvious sorting applications.

- List files in a directory.
- Organize an MP3 library.
- List names in a phone book.
- Display Google PageRank results.

Problems become easier once sorted.

- Find the median.
- Find the closest pair.
- Binary search in a database.**
- Identify statistical outliers.
- Find duplicates in a mailing list

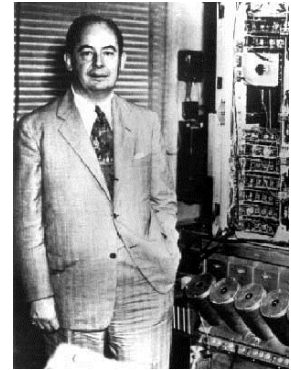
Non-obvious sorting applications.

- Data compression.
- Computer graphics.
- Interval scheduling.**
- Computational biology.
- Minimum spanning tree.**
- Supply chain management.
- Simulate a system of particles.
- Book recommendations on Amazon.
- Load balancing on a parallel computer.
- ...

Mergesort

Mergesort.

- Divide array into two halves.
- Recursively sort each half.
- Merge two halves to make sorted whole.



Jon von Neumann (1945)

A L G O R I T H M S

A L G O R I T H M S

A G L O R H I M S T

A G H I L M O R S T

divide $O(1)$

sort $2T(n/2)$

merge $O(n)$

Mergesort

Mergesort su una sequenza S con n elementi consiste di tre passi:

1. Divide: separa S in due sequenze $S1$ e $S2$, ognuna di **circa** $n/2$ elementi;
2. Ricorsione: ricorsivamente ordina $S1$ e $S2$
3. Conquer (impera): unisci $S1$ e $S2$ in un'unica sequenza *ordinata*

Mergesort

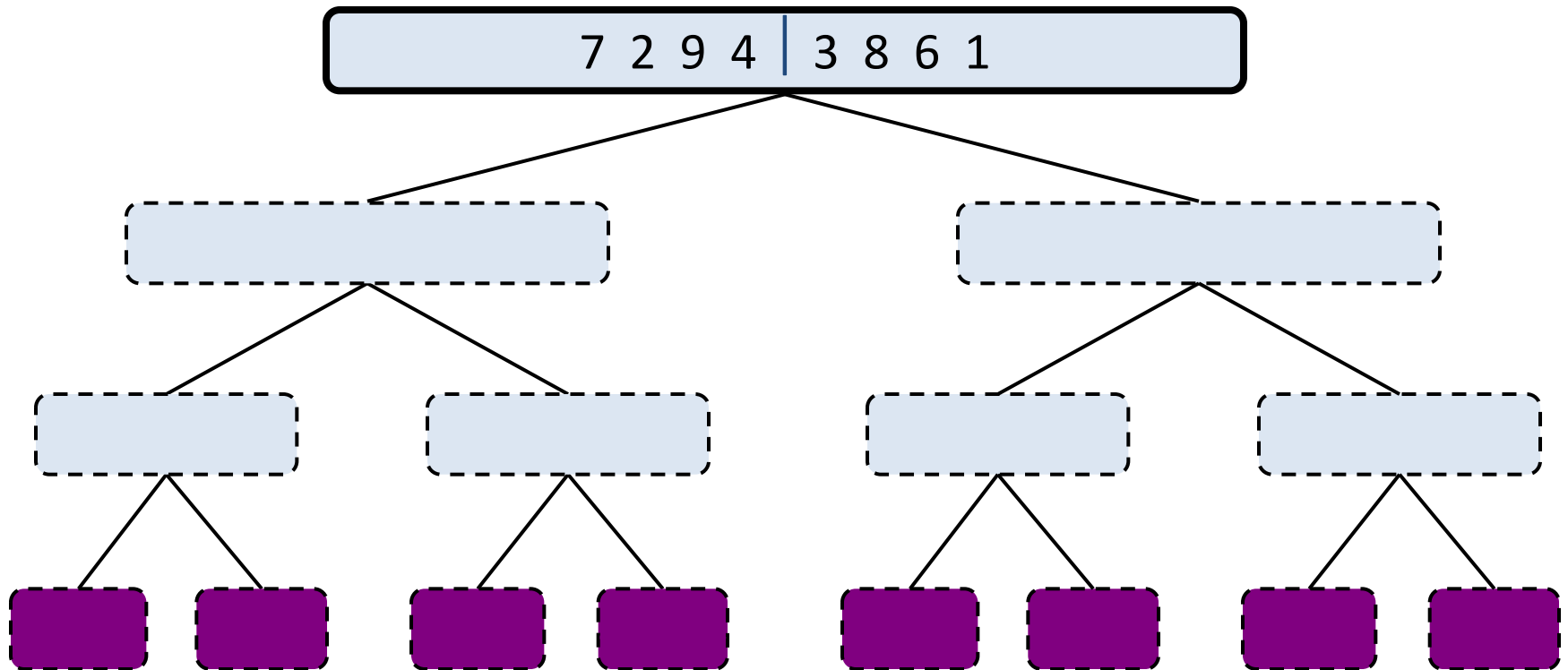
```
MERGE-SORT (A, p, r)
1  if p < r
2      then q ← ⌊ (p + r) / 2 ⌋
3          MERGE-SORT (A, p, q)
4          MERGE-SORT (A, q + 1, r)
5          MERGE (A, p, q, r)
```

Supponendo che:

la sequenza sia data come un array $A[p, \dots, r]$ con $n = r - p + 1$ elementi,
 $\text{MERGE}(A, p, q, r)$ «fonda» le sequenze $A[p, \dots, q]$ e $A[q+1, \dots, r]$

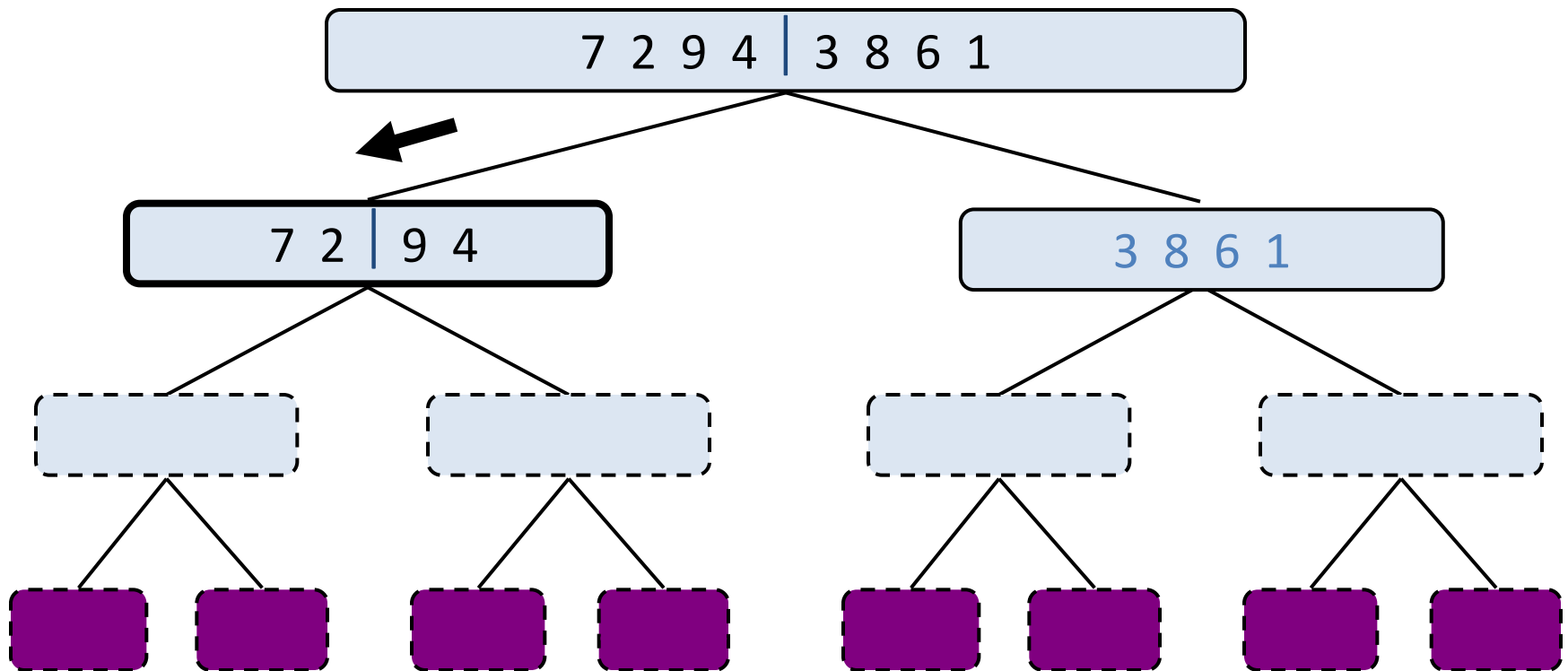
Esempio di esecuzione di MergeSort

- Divide



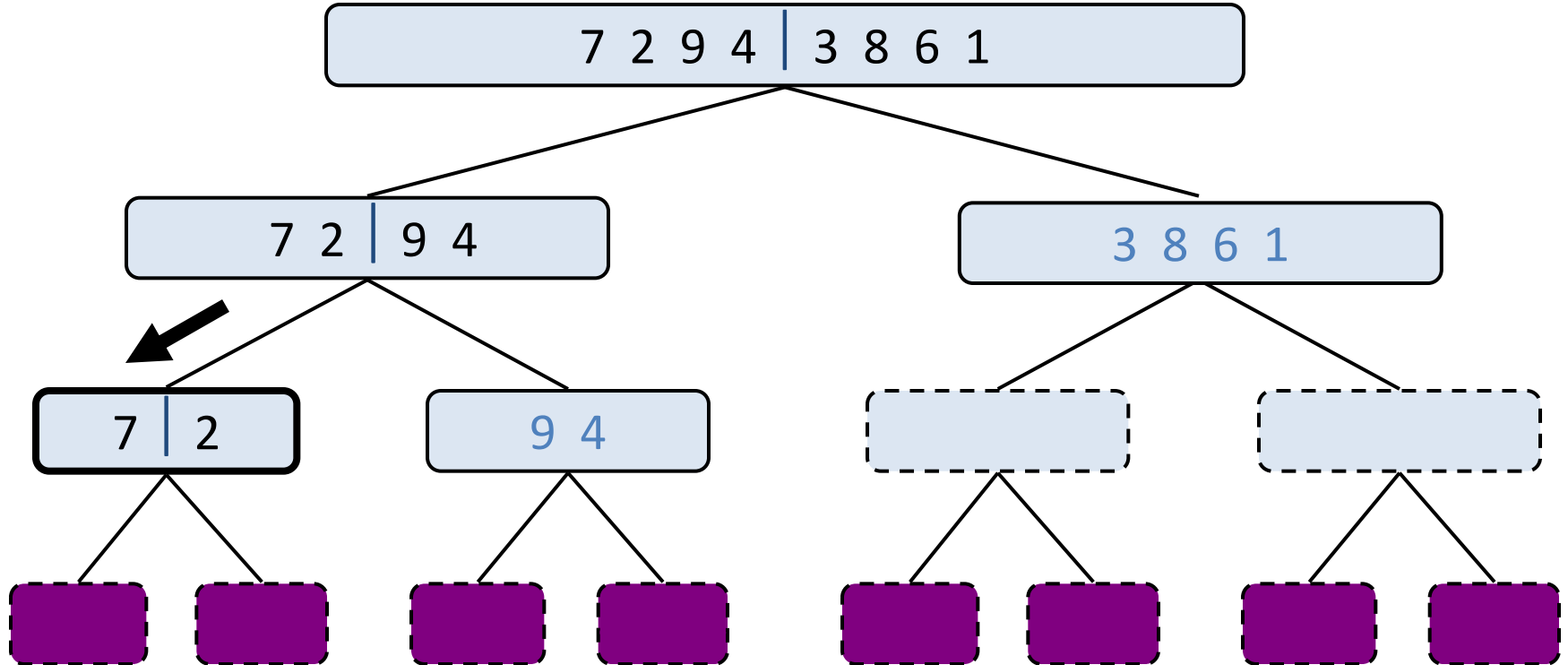
Esempio di esecuzione (cont.)

- Chiamata ricorsiva, divide



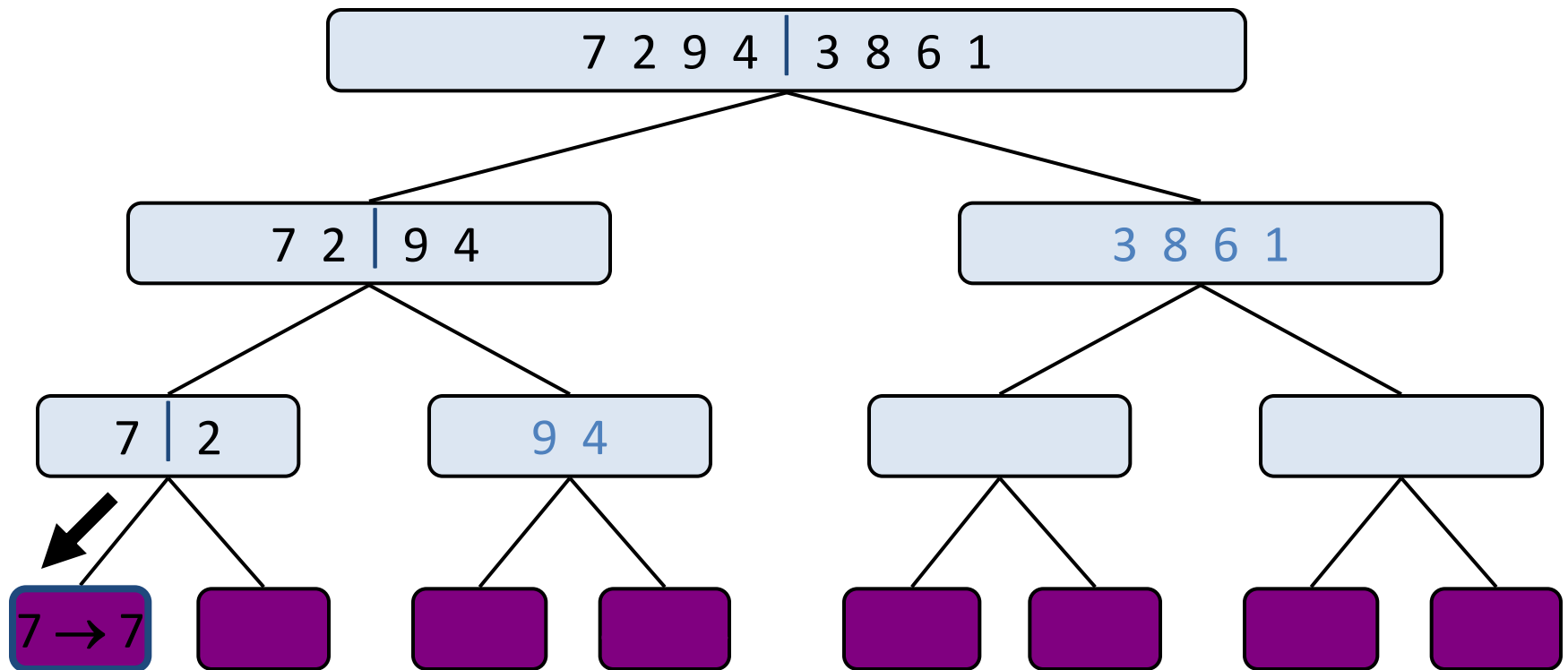
Esempio di esecuzione (cont.)

- Chiamata ricorsiva, divide



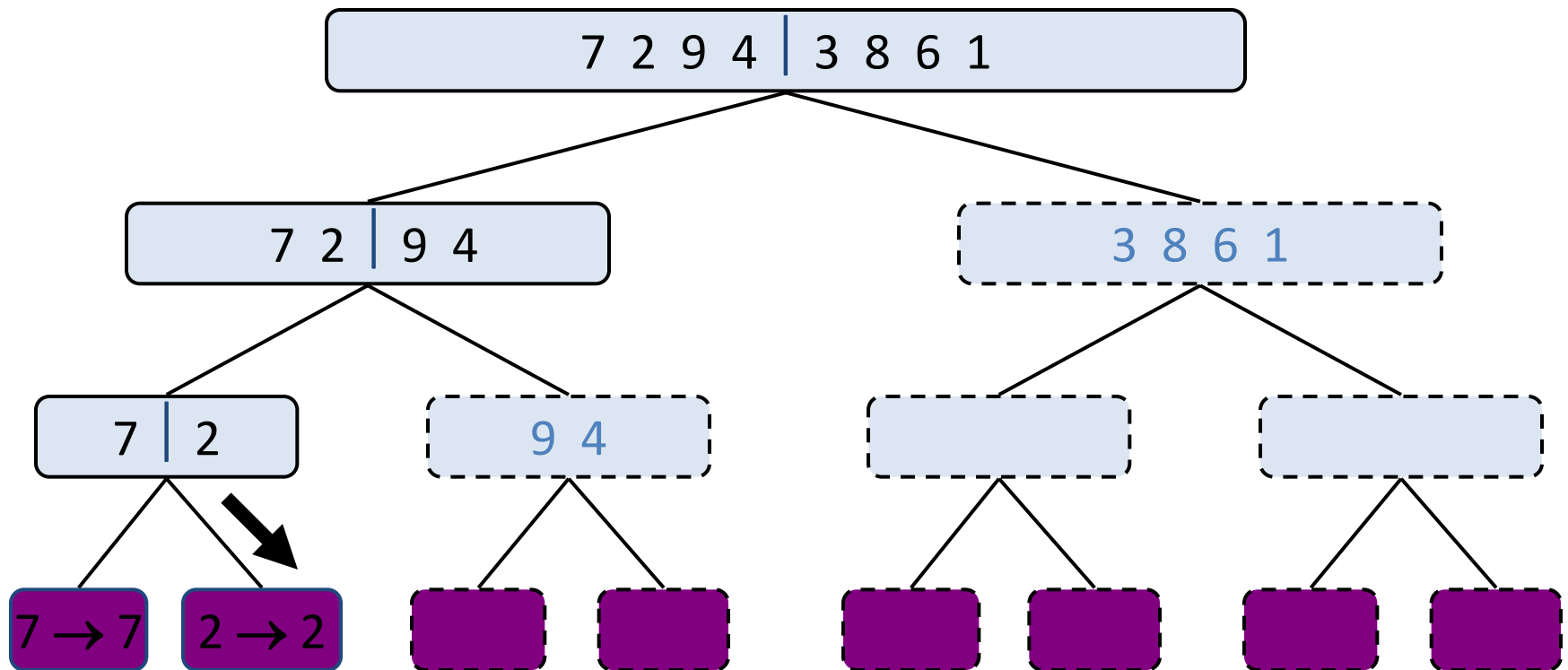
Esempio di esecuzione (cont.)

- Chiamata ricorsiva: caso base



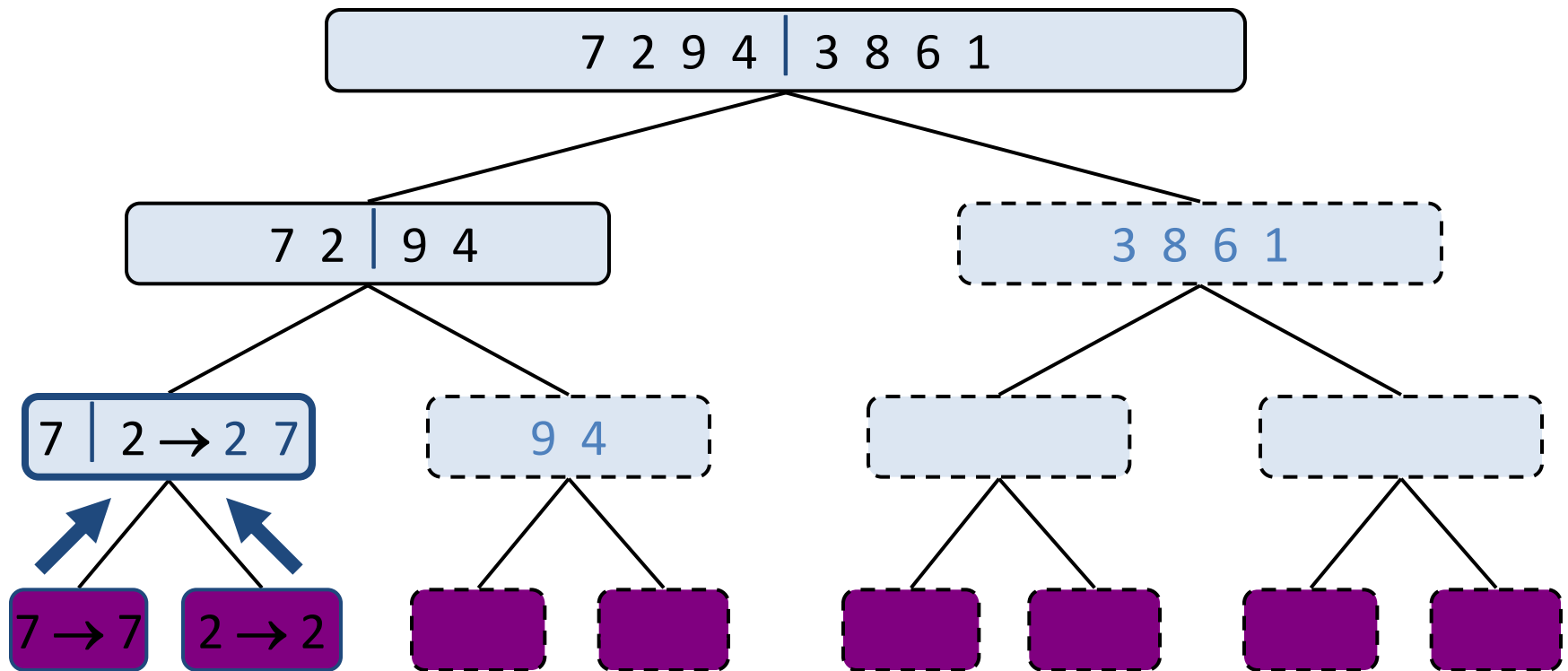
Esempio di esecuzione (cont.)

- Chiamata ricorsiva: caso base



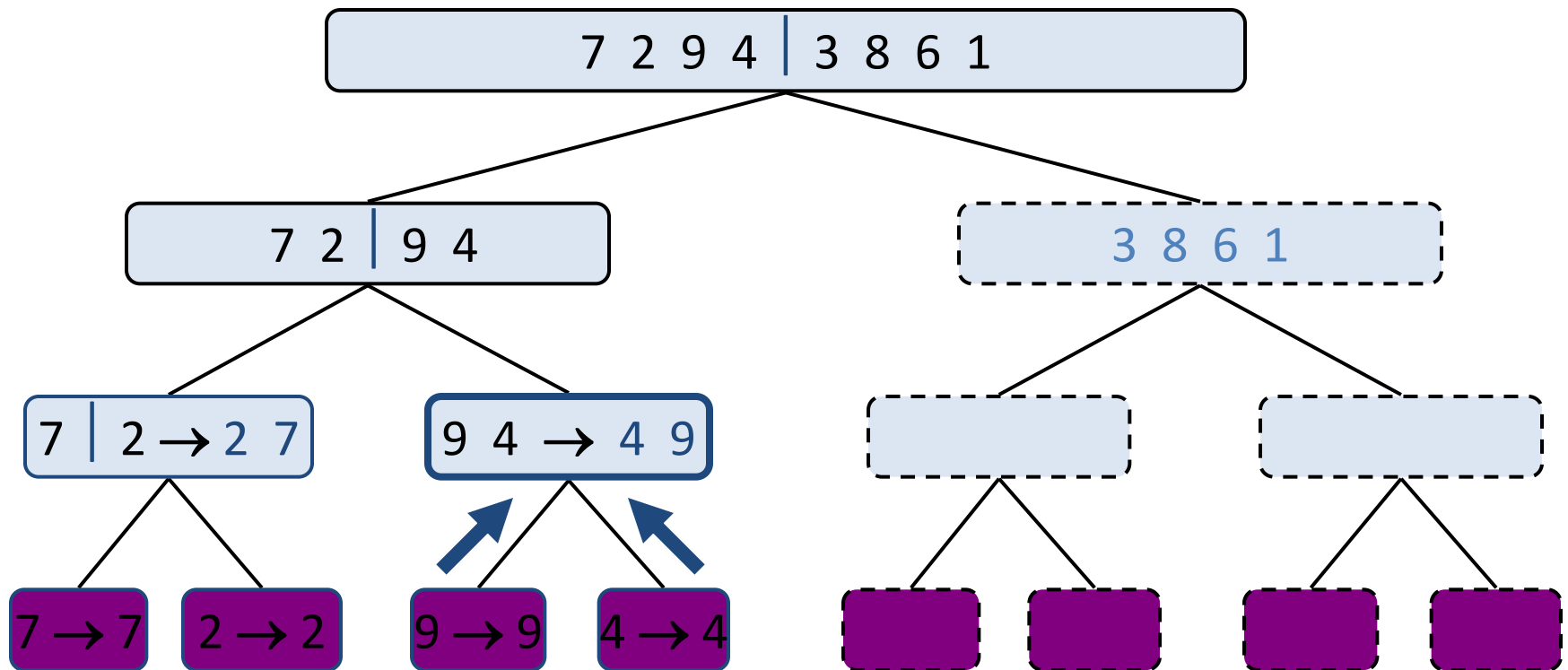
Esempio di esecuzione (cont.)

- Merge



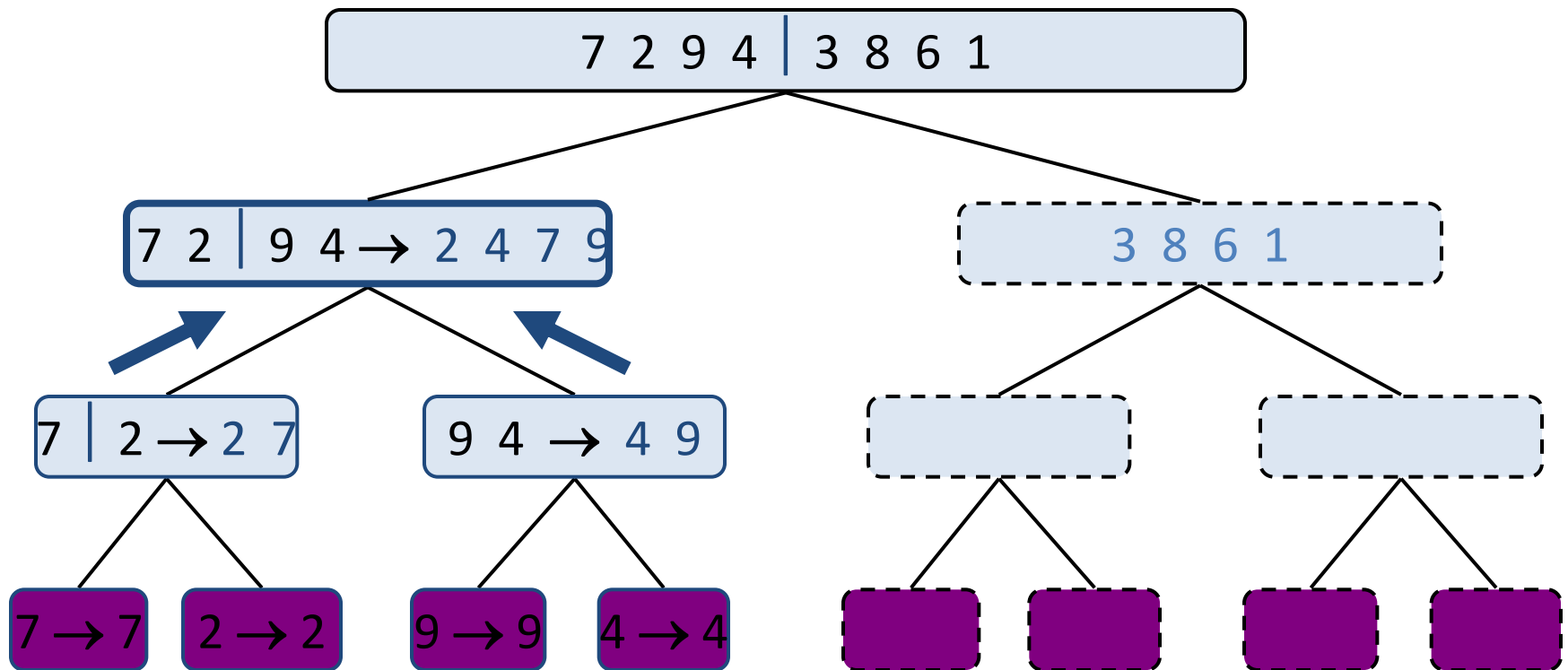
Esempio di esecuzione (cont.)

- Chiamata ricorsiva, ..., caso base, merge



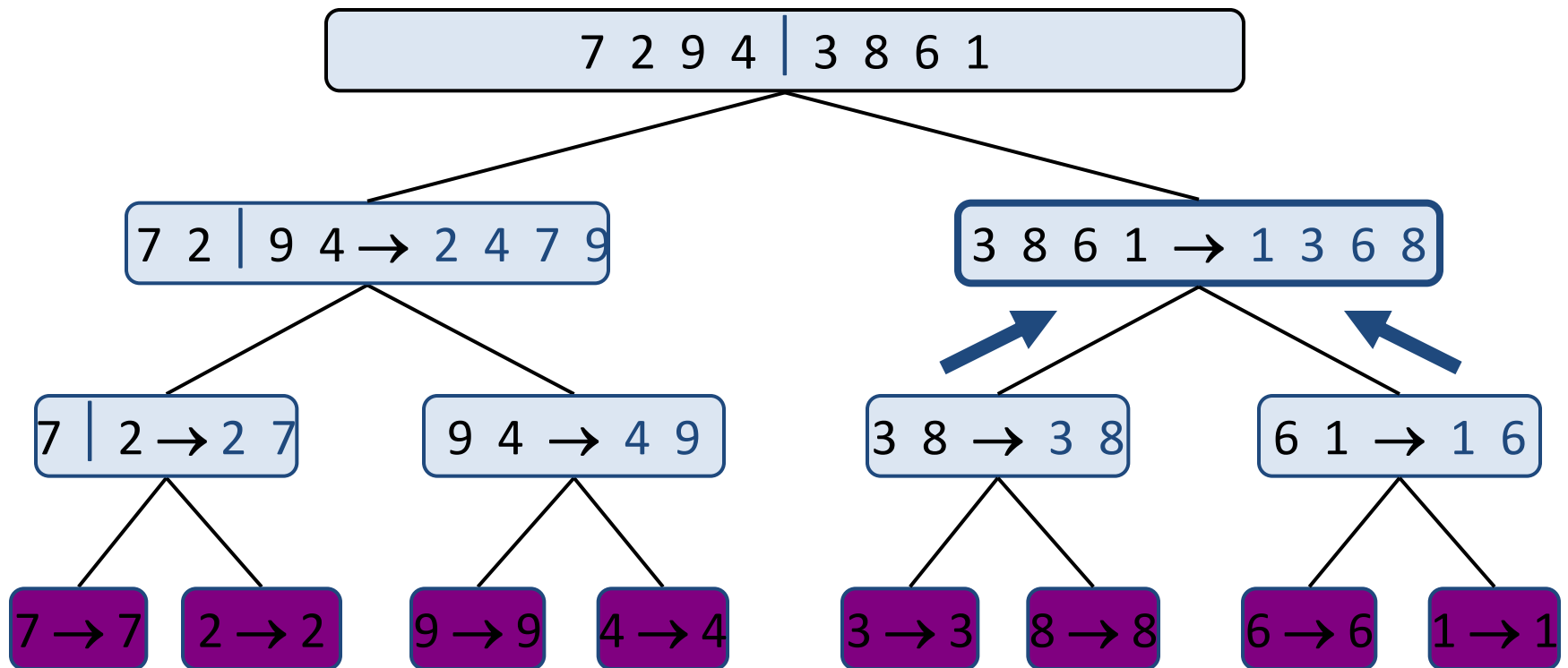
Esempio di esecuzione (cont.)

- Merge



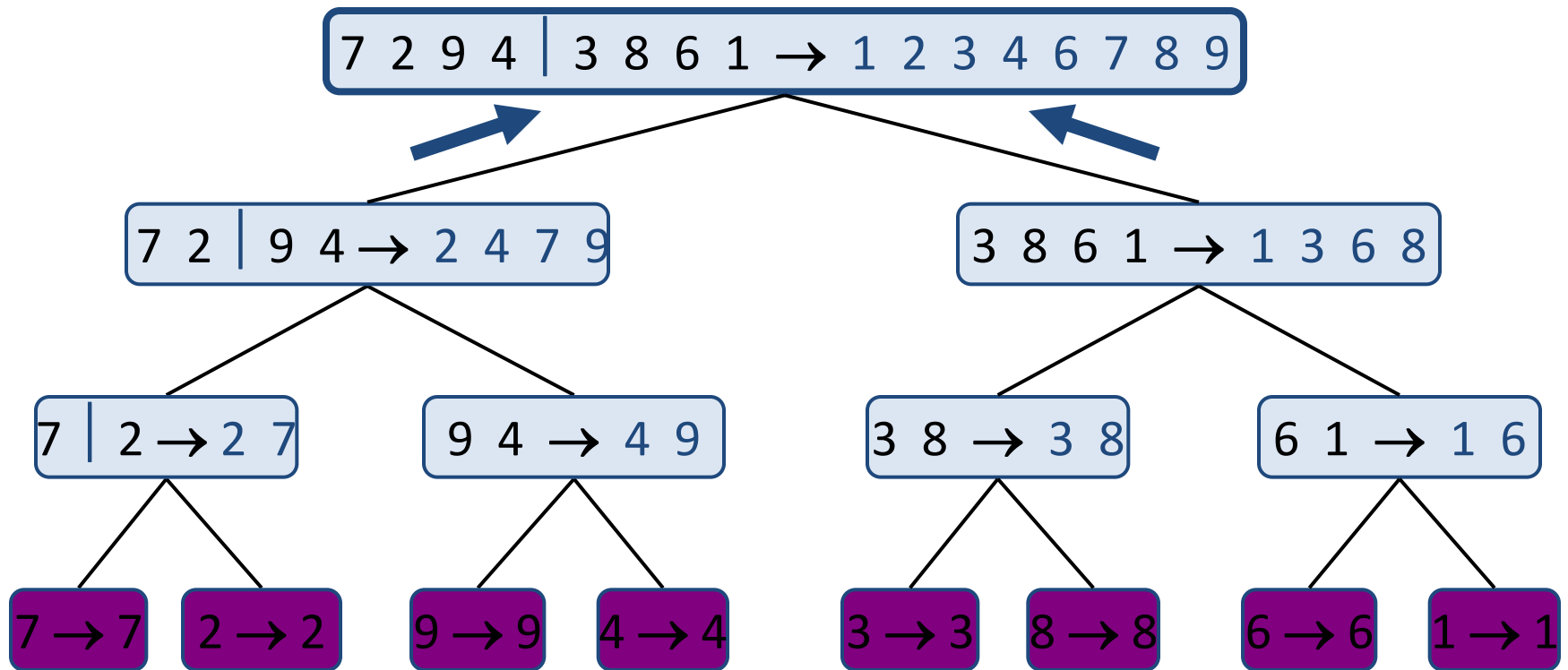
Esempio di esecuzione (cont.)

- Chiamata ricorsiva, ..., merge, merge



Esempio di esecuzione (fine)

- Ultimo Merge



Tempo di esecuzione di Mergesort

```
MERGE-SORT (A, p, r)
1  if p < r
2      then q ← ⌊ (p + r) / 2 ⌋
3          MERGE-SORT (A, p, q)
4          MERGE-SORT (A, q + 1, r)
5          MERGE (A, p, q, r)
```

Come si calcola il tempo di esecuzione di un algoritmo ricorsivo?

E in particolare:

Come si calcola il tempo di esecuzione di un algoritmo Divide et Impera?

A Recurrence Relation for Mergesort

Def. $T(n)$ = number of comparisons to mergesort an input of size n .

Mergesort recurrence.

$$T(n) \leq \begin{cases} \Theta(1) & \text{if } n = 1 \\ \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right half}} + \underbrace{\Theta(n)}_{\text{merging}} & \text{otherwise} \end{cases}$$

Solution. $T(n) = O(n \log_2 n)$.

Assorted proofs. We will describe several ways to prove this recurrence. Initially we assume n is a power of 2 and replace \leq with $=$.

A Recurrence Relation for Binary Search

Def. $T(n)$ = number of comparisons to run Binary Search on an input of size n .

Binary Search recurrence.

$$T(n) \leq \begin{cases} \Theta(1) & \text{if } n = 1 \\ \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve left or right half}} + \underbrace{\Theta(1)}_{\text{comparison}} & \text{otherwise} \end{cases}$$

Solution. $T(n) = O(\log_2 n)$ (constant in the best case).

Algoritmi ricorsivi

Schema di un algoritmo ricorsivo (su un'istanza \mathcal{I}):

ALGO (\mathcal{I})

If «caso base» then «esegui certe operazioni»

else «esegui delle operazioni fra le quali

ALGO(\mathcal{I}_1), ... , ALGO(\mathcal{I}_k) »

Relazioni di ricorrenza per algoritmi ricorsivi

$$T(n) = \begin{cases} c & \text{se } n = n_0 \\ aT(f(n)) + g(n) & \text{altrimenti} \end{cases}$$

- n_0 = base ricorsione, c = tempo di esecuzione per la base
- a = numero di volte che le chiamate ricorsive sono effettuate
- $f(n)$ = taglia dei problemi risolti nelle chiamate ricorsive
- $g(n)$ = tutto il tempo di calcolo non incluso nelle chiamate ricorsive

Relazioni di ricorrenza per algoritmi Divide-et-Impera

- **Dividi** il problema di taglia n in a sotto-problemi di taglia n/b
- **Ricorsione** sui sottoproblemi
- **Combinazione** delle soluzioni

$T(n)$ = tempo di esecuzione su input di taglia n

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

Nelle prossime lezioni....

Impareremo dei metodi per risolvere le relazioni di ricorrenza

6.5 RNA Secondary Structure

Struttura di una biomolecola

Biomolecola: DNA, RNA

Struttura primaria: descrizione esatta della sua composizione atomica e dei legami presenti fra gli atomi

Struttura secondaria: capacità di assumere una struttura **spaziale** regolare e ripetitiva

DNA

Struttura secondaria: doppia elica (Watson e Crick).

Ogni catena è composta da nucleotidi: *A, C, G, T*
A (adenina), *C* (citosina), *G* (guanina), *T* (timina)

Le catene sono connesse da basi complementari: *A-T, C-G*

Ribonucleic acid (RNA)

Simile al DNA.

Singola catena con 4 nucleotidi: adenine (A), cytosine (C), guanine (G), uracil (U).

RNA. Stringa $B = b_1b_2\dots b_n$ su alfabeto $\{ A, C, G, U \}$.

Struttura secondaria. RNA è una singola catena e tende a formare coppie di basi con se stessa. Questa struttura è essenziale per capire il comportamento delle molecole.

coppie di base complementari: A-U, C-G

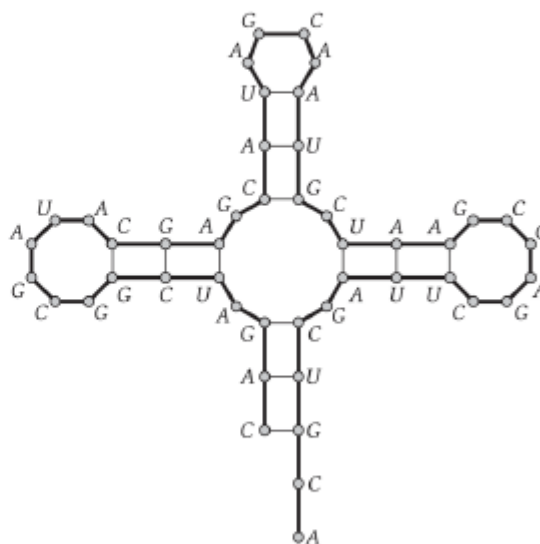


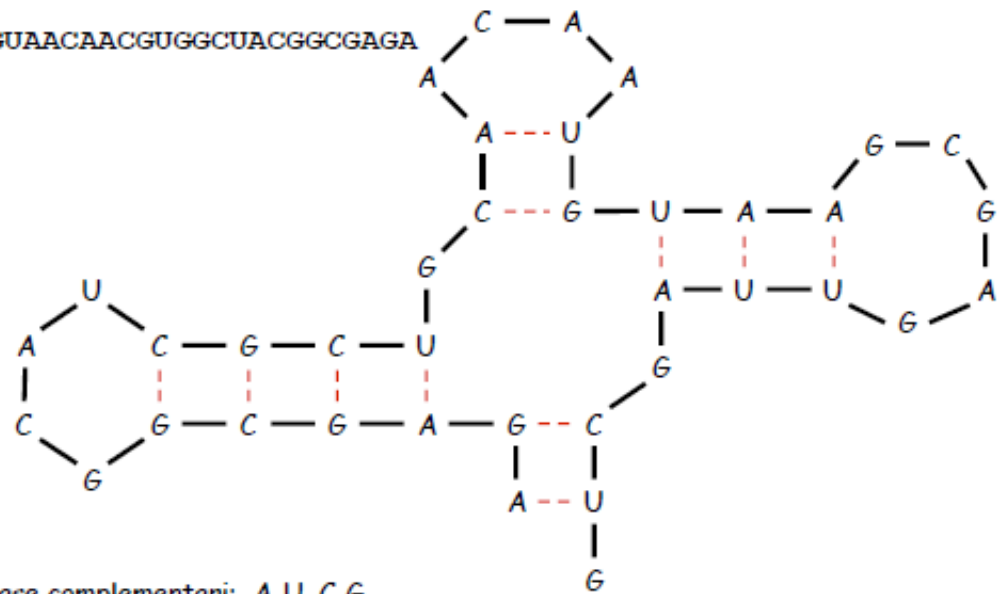
Figure 6.13 An RNA secondary structure. Thick lines connect adjacent elements of the sequence; thin lines indicate pairs of elements that are matched.

RNA Secondary Structure

RNA. Stringa $B = b_1b_2\dots b_n$ su alfabeto $\{A, C, G, U\}$.

Struttura secondaria. RNA è una singola catena e tende a formare coppie di basi con se stessa. Questa struttura è essenziale per capire il comportamento delle molecole.

Esempio: GUCGAUUGAGCGAAUGUAACAACGUGGCUACGGCGAGA



Per una stessa stringa di RNA possono esistere più strutture secondarie

Two views of RNA secondary structure

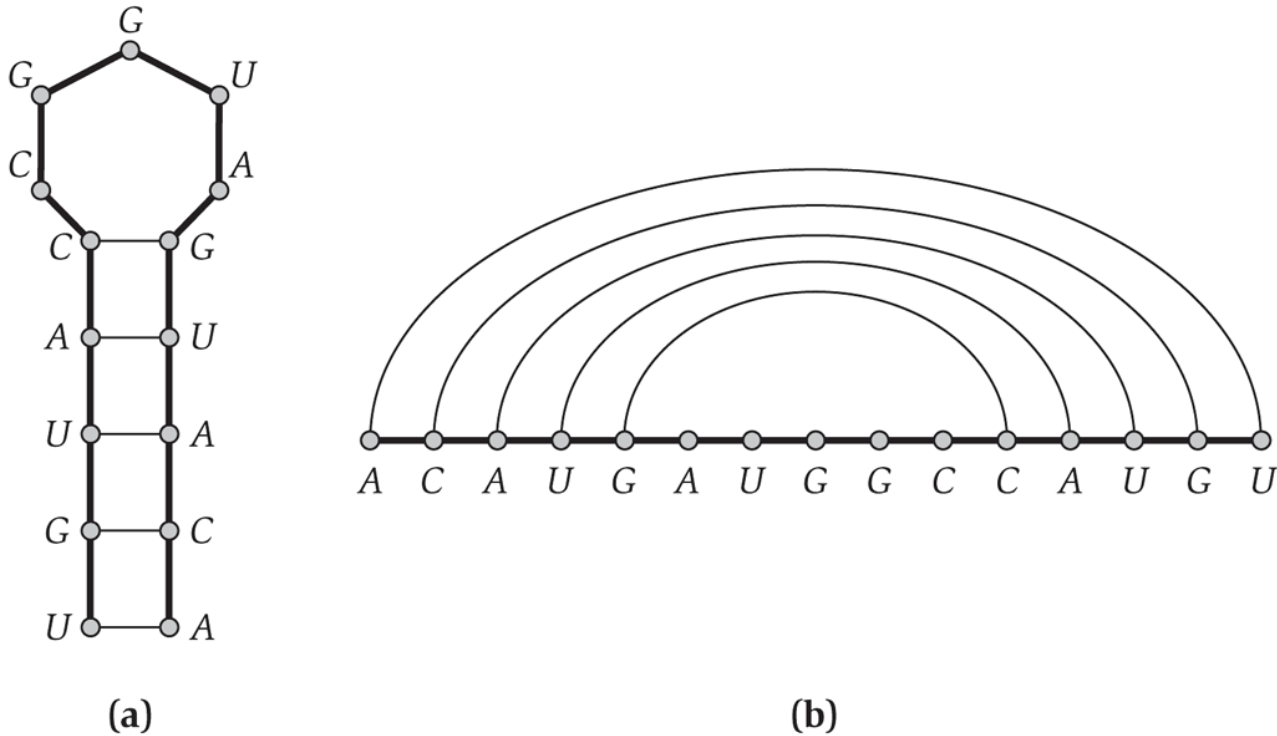
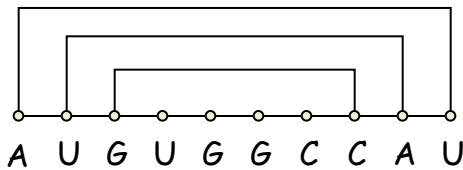
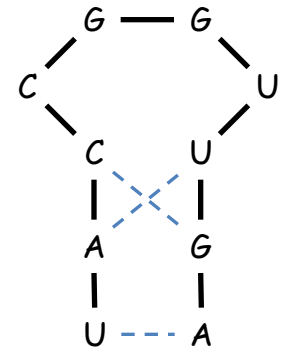
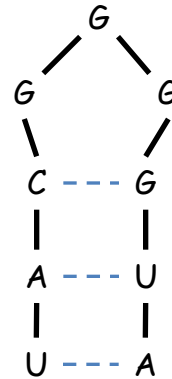
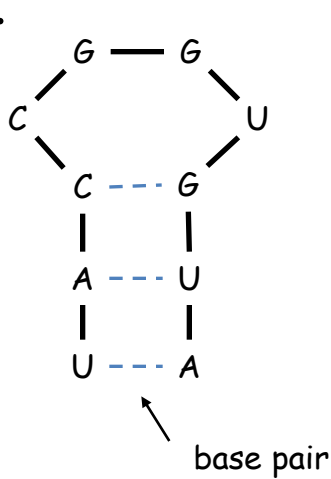


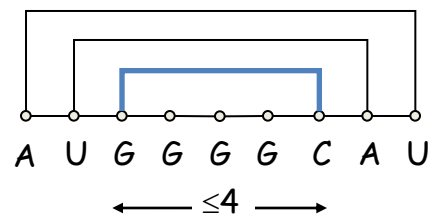
Figure 6.14 Two views of an RNA secondary structure. In the second view, (b), the string has been “stretched” lengthwise, and edges connecting matched pairs appear as noncrossing “bubbles” over the string.

RNA Secondary Structure: Examples

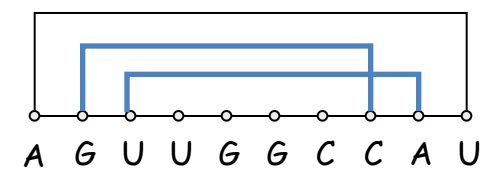
Examples.



ok



sharp turn: no!



crossing: no!

RNA Secondary Structure

Secondary structure. A set of pairs $S = \{ (b_i, b_j) \}$ that satisfy:

[Watson-Crick.] S is a matching and each pair in S is a Watson-Crick complement: A-U, U-A, C-G, or G-C.

[No sharp turns.] The ends of each pair are separated by at least 4 intervening bases. If $(b_i, b_j) \in S$, then $i < j - 4$.

[Non-crossing.] If (b_i, b_j) and (b_k, b_l) are two pairs in S , then we cannot have $i < k < j < l$.

Free energy. Usual hypothesis is that an RNA molecule will form the secondary structure with the optimum total free energy.

↑
approximate by number of base pairs

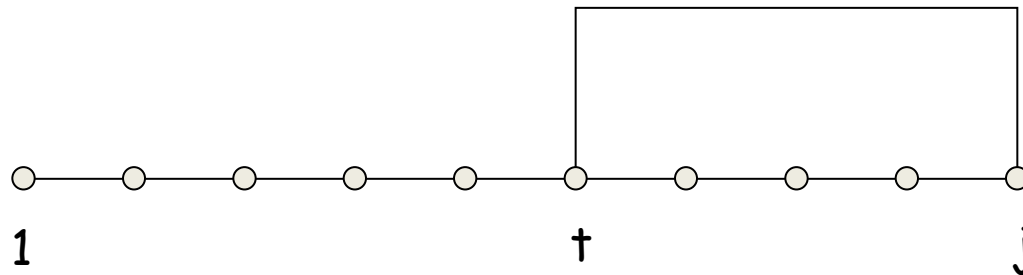
Goal. Given an RNA molecule $B = b_1b_2\dots b_n$, find a secondary structure S that **maximizes** the number of base pairs.

RNA Secondary Structure: Subproblems

First attempt. $OPT(j)$ = maximum number of base pairs in a secondary structure of the substring $b_1b_2\dots b_j$.

Case 1: b_j is not involved in a pair: $OPT(j)=OPT(j-1)$

Case 2: b_j matches b_t for some $1 \leq t < j-4$



Difficulty. Results in two sub-problems:

Finding secondary structure in: $b_1b_2\dots b_{t-1}$. ← $OPT(t-1)$

Finding secondary structure in: $b_{t+1}b_{t+2}\dots b_{j-1}$. ← need different sub-problems

Dynamic Programming Over Intervals

Notation. $OPT(i, j)$ = maximum number of base pairs in a secondary structure of the substring $b_i b_{i+1} \dots b_j$.

Case 1. If $i \geq j - 4$.

$OPT(i, j) = 0$ by no-sharp turns condition.

Case 2. Base b_j is not involved in a pair.

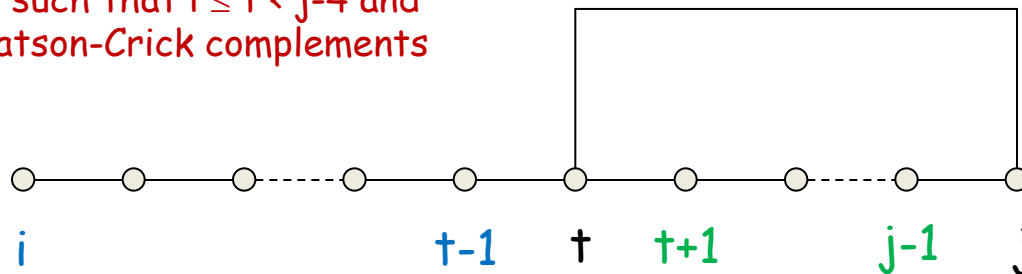
$OPT(i, j) = OPT(i, j-1)$

Case 3. Base b_j pairs with b_t for some $i \leq t < j - 4$.

non-crossing constraint (no match over t) decouples resulting sub-problems

$OPT(i, j) = 1 + \max_t \{ OPT(i, t-1) + OPT(t+1, j-1) \}$

↑
take max over t such that $i \leq t < j-4$ and b_t and b_j are Watson-Crick complements



Relazione di ricorrenza

$OPT(i, j)$ = maximum number of base pairs in a secondary structure of the substring $b_i b_{i+1} \dots b_j$
with $i, j = 1, 2, \dots, n$

$$OPT(i, j) = \begin{cases} 0 & i \geq j - 4 \\ \max \left\{ \begin{array}{l} OPT(i, j-1) \\ \max_{\substack{i \leq t < j-4 \\ b, b_t \text{ complementi}}} \{ 1 + OPT(i, t-1) + OPT(t+1, j-1) \} \end{array} \right\} & \text{altrimenti} \end{cases}$$

Caso generale: $i < j - 4$ ovvero $i \leq j - 5$, ovvero $j \geq i + 5$

$i = 1$ allora $j \geq 6$

$i = 2$ allora $j \geq 7$

...

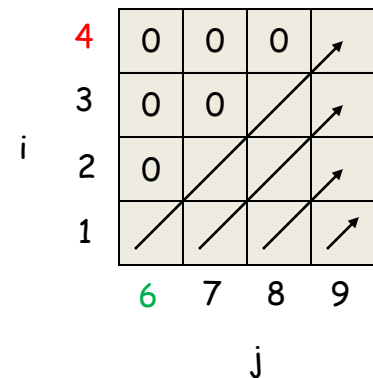
$i = n - 5$ allora $j \geq n$

$i = n - 4$ allora $j \geq n + 1$

...

$i = n$ allora $j \geq n + 5$

$n = 9$ allora $i \leq 9 - 5 = 4$, $j \geq 6$



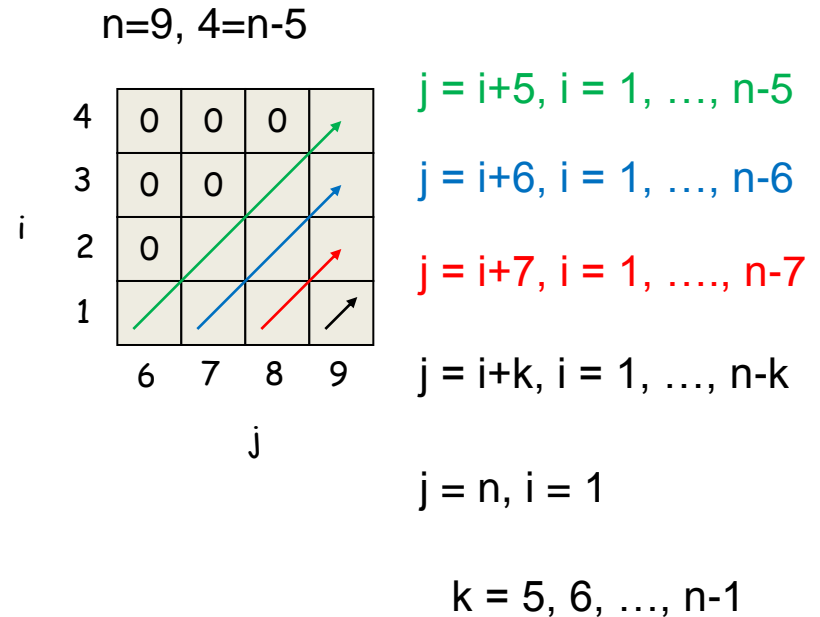
Bottom Up Dynamic Programming Over Intervals

Q. What order to solve the sub-problems?

A. Do **shortest** intervals first (from length 5 up to $n-1$).

```

RNA( $b_1, \dots, b_n$ ) {
  for  $k = 5, 6, \dots, n-1$ 
    for  $i = 1, 2, \dots, n-k$ 
       $j = i + k$ 
      Compute  $M[i, j]$ 
      ↑
      using recurrence
  return  $M[1, n]$ 
}
    
```



Running time. $O(n^3)$.

Esempio

RNA sequence *ACCGGUAGU*

4	0	0	0	
3	0	0		
2	0			
$i = 1$				

$j = 6 \quad 7 \quad 8 \quad 9$

Initial values

4	0	0	0	0
3	0	0	1	
2	0	0		
$i = 1$	1			

$j = 6 \quad 7 \quad 8 \quad 9$

**Filling in the values
for $k = 5$**

4	0	0	0	0
3	0	0	1	1
2	0	0	1	
$i = 1$	1	1		

$j = 6 \quad 7 \quad 8 \quad 9$

**Filling in the values
for $k = 6$**

4	0	0	0	0
3	0	0	1	1
2	0	0	1	1
$i = 1$	1	1	1	

$j = 6 \quad 7 \quad 8 \quad 9$

**Filling in the values
for $k = 7$**

4	0	0	0	0
3	0	0	1	1
2	0	0	1	1
$i = 1$	1	1	1	2

$j = 6 \quad 7 \quad 8 \quad 9$

**Filling in the values
for $k = 8$**

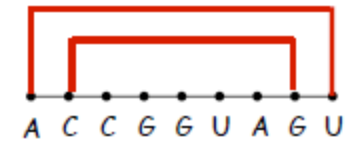
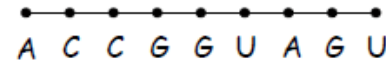


Figure 6.16 The iterations of the algorithm on a sample instance of the RNA Secondary Structure Prediction Problem.

RNA sequence ACCGGUAGU



4	0	0	0	
3	0	0		
2	0			
$i = 1$				
	$j = 6$	7	8	9

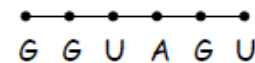
Initial values

4	0	0	0	0
3	0	0	1	
2	0	0		
$i = 1$	1			
	$j = 6$	7	8	9

Filling in the values
for $k = 5$

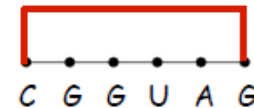
$$OPT(i, j) = \max \begin{cases} OPT(i, j-1) \\ \max_{\substack{1 \leq t < j-4 \\ b_t, b_{t+5} \text{ complement}}} \{ 1 + OPT(i, t-1) + OPT(t+1, j-1) \} \end{cases}$$

$$OPT(4, 9) = \max \begin{cases} OPT(4, 8) = 0 \\ \max_{\substack{4 \leq t < 5 \\ b_t, b_{t+5} \text{ complement}}} \{ 1 + OPT(4, t-1) + OPT(t+1, 8) \} = 0 \end{cases}$$

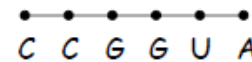


$b_4 = G$ does not match $b_9 = U$

$$OPT(3, 8) = \max \begin{cases} OPT(3, 7) = 0 \\ \max_{\substack{3 \leq t < 4 \\ b_t, b_{t+5} \text{ complement}}} \{ 1 + OPT(1, t-1) + OPT(t+1, 7) \} = 1 \end{cases}$$

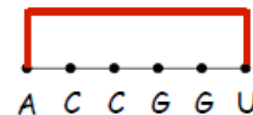


$$OPT(2, 7) = \max \begin{cases} OPT(2, 6) = 0 \\ \max_{\substack{2 \leq t < 3 \\ b_t, b_{t+5} \text{ complement}}} \{ 1 + OPT(2, t-1) + OPT(t+1, 6) \} = 0 \end{cases}$$



$b_2 = C$ does not match $b_7 = A$

$$OPT(1, 6) = \max \begin{cases} OPT(1, 5) = 0 \\ \max_{\substack{1 \leq t < 2 \\ b_t, b_{t+5} \text{ complement}}} \{ 1 + OPT(1, t-1) + OPT(t+1, 5) \} = 1 \end{cases}$$



RNA sequence ACCGGUAGU



4	0	0	0	
3	0	0		
2	0			
= 1				
	j = 6	7	8	9

Initial values

4	0	0	0	0
3	0	0	1	
2	0	0		
i = 1	1			
	j = 6	7	8	9

Filling in the values
for $k = 5$

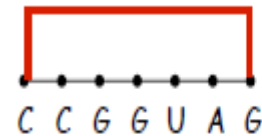
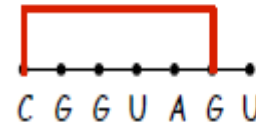
4	0	0	0	0
3	0	0	1	1
2	0	0	1	
i = 1	1	1		
	j = 6	7	8	9

Filling in the values
for $k = 6$

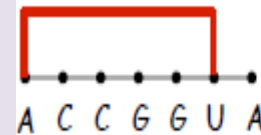
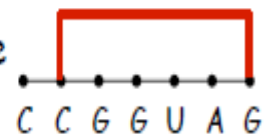
$$OPT(3,9) = \max \left\{ \begin{array}{l} OPT(3,8) = 1 \\ \max_{\substack{3 \leq t < 9 \\ b_t = U, b_{t+1} = A, \text{ compl.}}} \{1 + OPT(3, t-1) + OPT(t+1, 8)\} = 0 \end{array} \right.$$

$$OPT(2,8) = \max \left\{ \begin{array}{l} OPT(2,7) = 0 \\ \max_{\substack{2 \leq t < 8 \\ b_t = G, b_{t+1} = G, \text{ compl.}}} \{1 + OPT(2, t-1) + OPT(t+1, 7)\} = 1 \end{array} \right.$$

$$OPT(1,7) = \max \left\{ \begin{array}{l} OPT(1,6) = 1 \\ \max_{\substack{1 \leq t < 7 \\ b_t = A, b_{t+1} = A, \text{ compl.}}} \{1 + OPT(1, t-1) + OPT(t+1, 6)\} = 0 \end{array} \right.$$



oppure



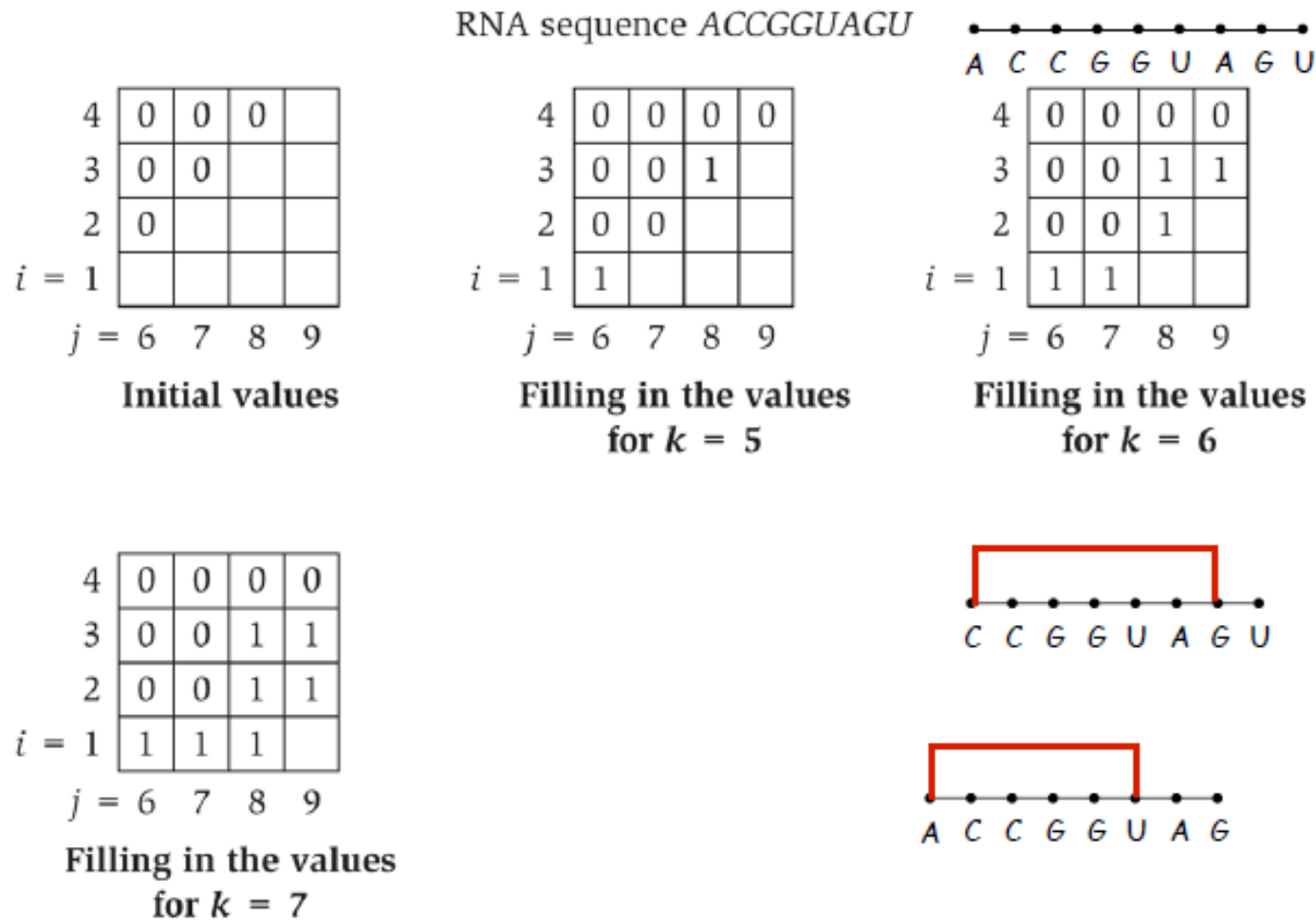


Figure 6.16 The iterations of the algorithm on a sample instance of the RNA Secondary Structure Prediction Problem.

RNA sequence *ACCGGUAGU*

4	0	0	0	
3	0	0		
2	0			
$i = 1$				

$j = 6 \quad 7 \quad 8 \quad 9$

Initial values

4	0	0	0	0
3	0	0	1	
2	0	0		
$i = 1$	1			

$j = 6 \quad 7 \quad 8 \quad 9$

**Filling in the values
for $k = 5$**

4	0	0	0	0
3	0	0	1	1
2	0	0	1	
$i = 1$	1	1		

$j = 6 \quad 7 \quad 8 \quad 9$

**Filling in the values
for $k = 6$**

4	0	0	0	0
3	0	0	1	1
2	0	0	1	1
$i = 1$	1	1	1	

$j = 6 \quad 7 \quad 8 \quad 9$

**Filling in the values
for $k = 7$**

4	0	0	0	0
3	0	0	1	1
2	0	0	1	1
$i = 1$	1	1	1	2

$j = 6 \quad 7 \quad 8 \quad 9$

**Filling in the values
for $k = 8$**

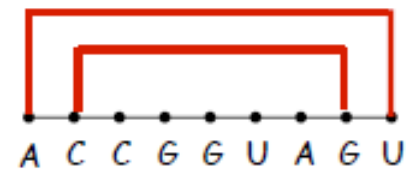


Figure 6.16 The iterations of the algorithm on a sample instance of the RNA Secondary Structure Prediction Problem.

Trovare una soluzione

RNA sequence *ACCGGUAGU*

4	0	0	0	
3	0	0		
2	0			
$i = 1$				

$j = 6 \quad 7 \quad 8 \quad 9$

Initial values

4	0	0	0	0
3	0	0	1	
2	0	0		
$i = 1$	1			

$j = 6 \quad 7 \quad 8 \quad 9$

**Filling in the values
for $k = 5$**

4	0	0	0	0
3	0	0	1	1
2	0	0	1	
$i = 1$	1	1		

$j = 6 \quad 7 \quad 8 \quad 9$

**Filling in the values
for $k = 6$**

4	0	0	0	0
3	0	0	1	1
2	0	0	1	1
$i = 1$	1	1	1	

$j = 6 \quad 7 \quad 8 \quad 9$

**Filling in the values
for $k = 7$**

4	0	0	0	0
3	0	0	1	1
2	0	0	1	1
$i = 1$	1	1	1	2

$j = 6 \quad 7 \quad 8 \quad 9$

**Filling in the values
for $k = 8$**

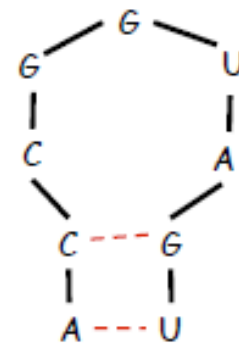


Figure 6.16 The iterations of the algorithm on a sample instance of the RNA Secondary Structure Prediction Problem.

Dynamic Programming Summary

Recipe.

- Characterize structure of problem.
- Recursively define value of optimal solution.
- Compute value of optimal solution.
- Construct optimal solution from computed information.

Dynamic programming techniques.

- Binary choice: weighted interval scheduling.
- Multi-way choice: segmented least squares.
- Adding a new variable: knapsack.
- Dynamic programming over intervals: RNA secondary structure.

Top-down vs. bottom-up: different people have different intuitions.

Calcolo di Combinazioni

Sia $\binom{n}{r}$ il numero di modi con cui possiamo scegliere r oggetti da un insieme di n elementi. Come possiamo calcolare $\binom{n}{r}$?

n elementi, possiamo o

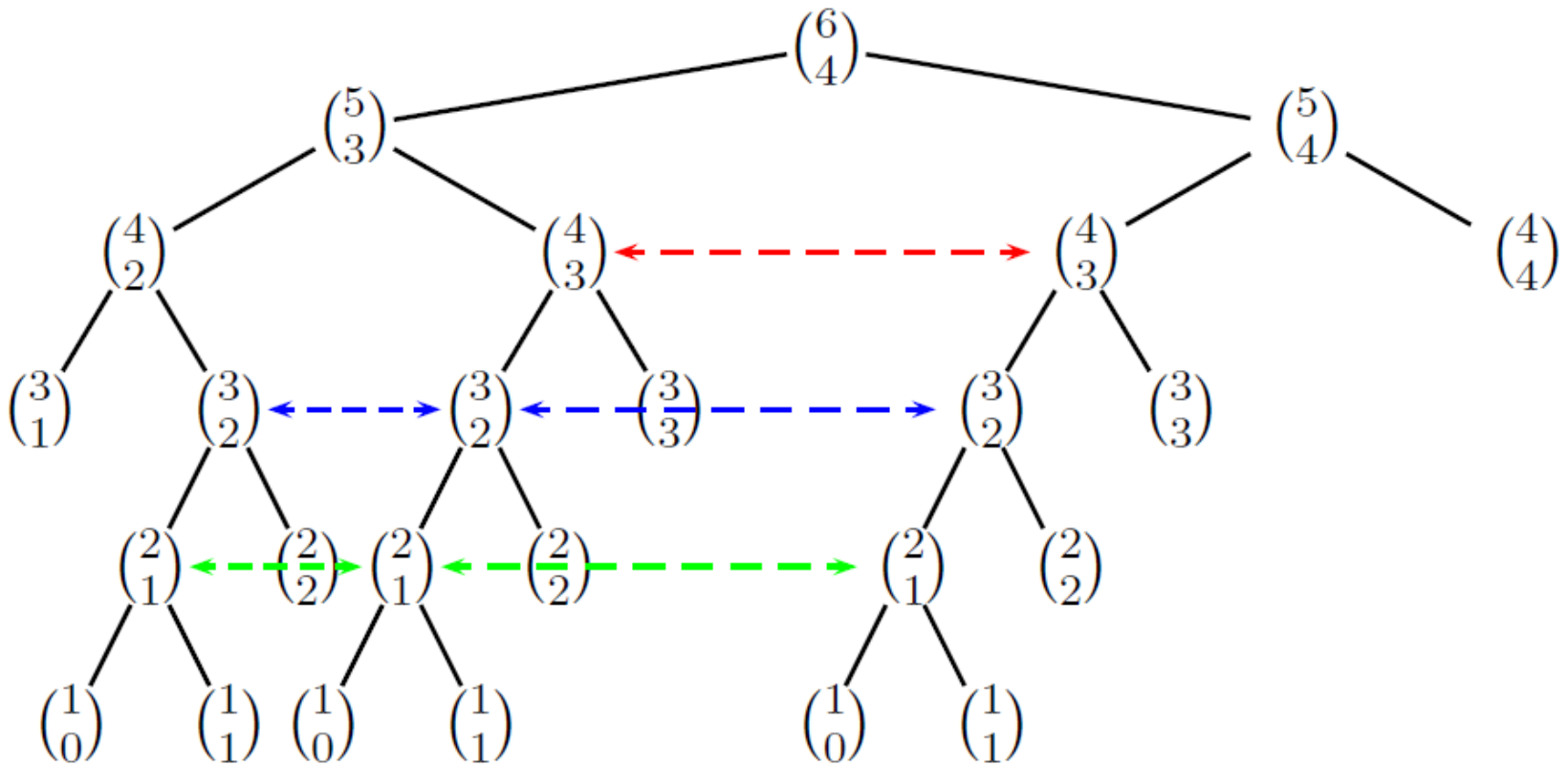
- scegliere di prendere il primo oggetto dall'insieme (in questo caso ci resterà il problema di scegliere i restanti $r - 1$ oggetti da un insieme di $n - 1$ elementi, e questo si potrà fare in $\binom{n-1}{r-1}$ modi), oppure
- scegliere di non prendere il primo oggetto dall'insieme (in questo secondo caso ci resterà il problema di scegliere tutti gli r oggetti da un insieme di $n - 1$ elementi, e questo si potrà fare in $\binom{n-1}{r}$ modi).

Ciò equivale a dire che

$$\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r}$$

Algoritmo ricorsivo CHOOSE

Albero delle chiamate ricorsive di CHOOSE(6, 4))



Algoritmo di programmazione dinamica

ITERCHOOSE(n, r)

for $i \leftarrow 0$ **to** $n - r$ **do** $T[i, 0] \leftarrow 1$

for $i \leftarrow 0$ **to** r **do** $T[i, i] \leftarrow 1$

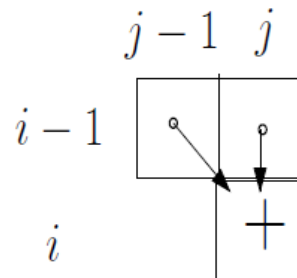
for $j \leftarrow 1$ **to** r **do**

for $i \leftarrow j + 1$ **to** $n - r + j$ **do**

$T[i, j] \leftarrow T[i - 1, j - 1] + T[i - 1, j]$

return ($T[n, r]$)

Al momento dell' assegnazione $T[i, j] \leftarrow T[i - 1, j - 1] + T[i - 1, j]$ occorre che $T[i - 1, j - 1]$ e $T[i - 1, j]$ siano stati già calcolati (e infatti l'algoritmo correttamente procede in questo modo)

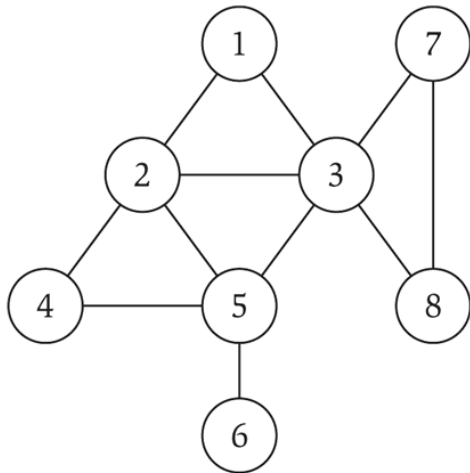


Grafi: **definizioni e visite**

Grafi (non orientati)

Grafo (non orientato): $G = (V, E)$

- V = nodi (o vertici)
- E = archi fra coppie di nodi distinti.
- Modella relazioni fra coppie di oggetti.
- Parametri della taglia di un grafo: $n = |V|$, $m = |E|$.



$$V = \{ 1, 2, 3, 4, 5, 6, 7, 8 \}$$

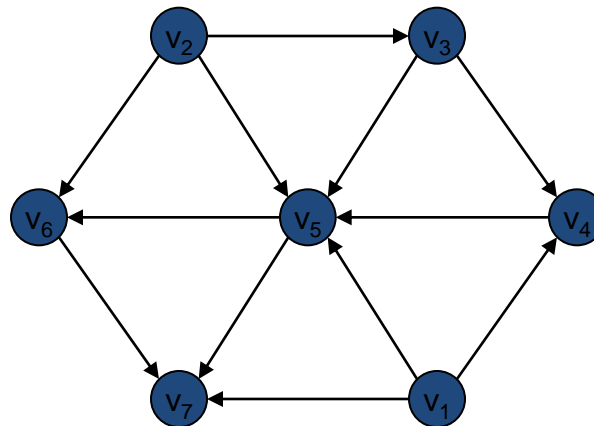
$$E = \{ (1,2), (1,3), (2,3), (2,4), (2,5), (3,5), (3,7), (3,8), (4,5), (5,6) \}$$

$$n = 8$$

$$m = 11$$

Grafi diretti (o orientati)

- Grafo diretto: $G = (V, E)$
- V = nodi (o vertici)
- E = archi diretti da un nodo (coda) in un altro (testa)
- Modella relazioni non simmetriche fra coppie di oggetti.
- Parametri della taglia di un grafo: $n = |V|$, $m = |E|$.



Grafi

Il grafo è una delle strutture più espressive e fondamentali della matematica discreta.

Semplice modo di **modellare relazioni a coppie** in un insieme di oggetti.

Innumerevoli applicazioni.

"Più si lavora coi grafi, più si tende a vederli ovunque".

Qualche applicazione

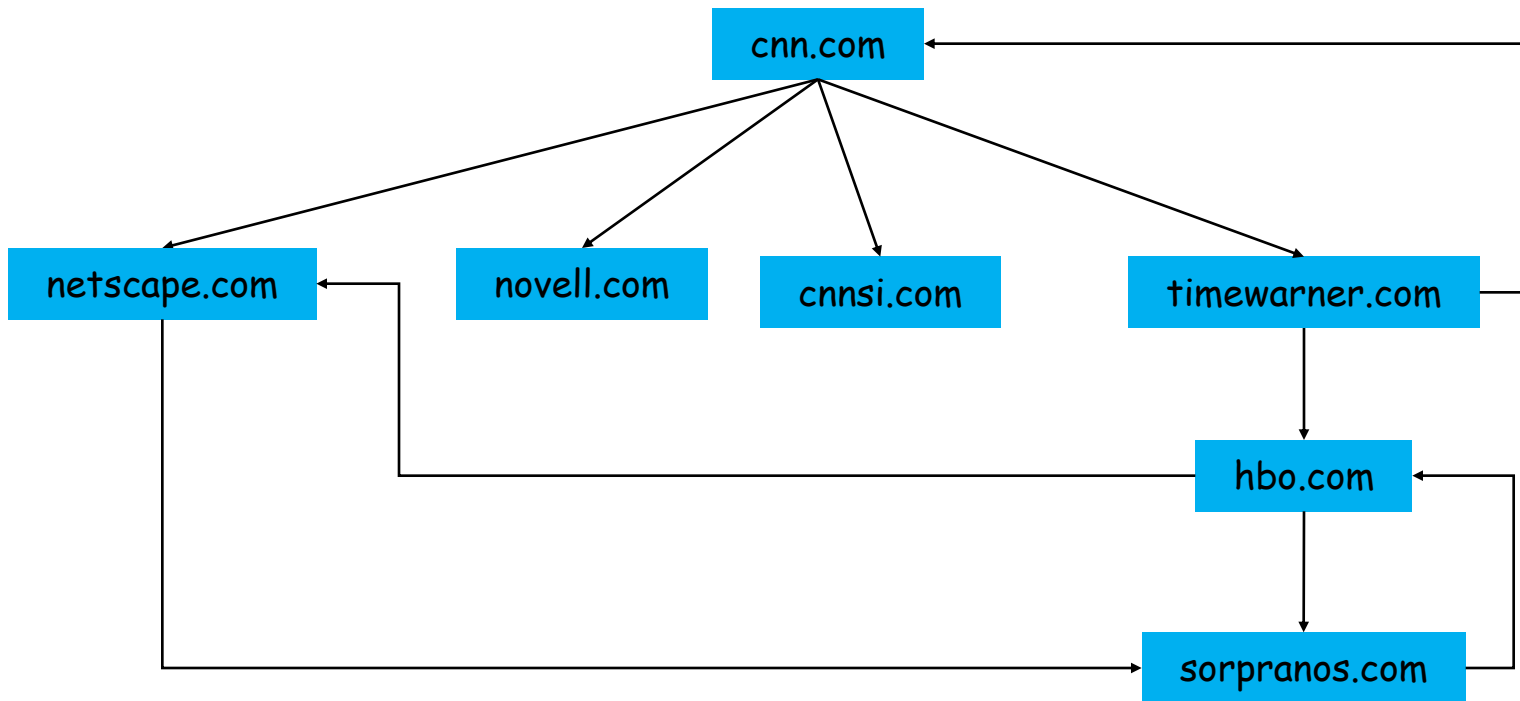
<i>Grafo</i>	<i>Nodi</i>	<i>Archi</i>
trasporti	Incroci di strade /città	Strade
	aeroporti	Rotte aeree
comunicazioni	computers	cavi in fibre ottiche
World Wide Web	pagine web	hyperlinks
social network	persone	relazioni
food web	specie	predatore-preda
software systems	funzioni	chiamate di funzioni
scheduling	tasks	vincoli di precedenza
circuiti	porte	fili

World Wide Web

Grafo del Web (orientato).

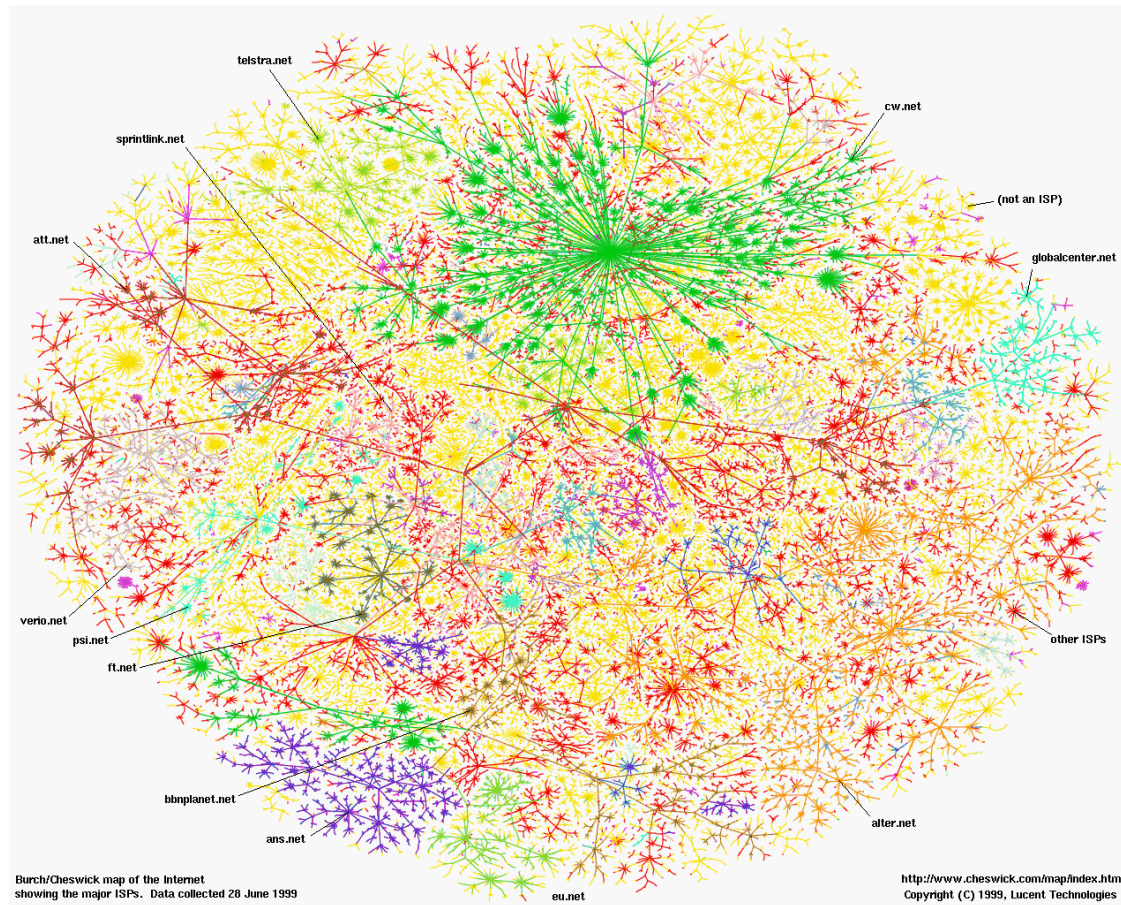
Nodo: pagina web

Arco: hyperlink da una pagina all'altra.



World Wide Web

Albero con circa 100.000 nodi



<http://www.cheswick.com/ches/map/>

Rete terroristi dell'11 settembre

Grafo di una social network.

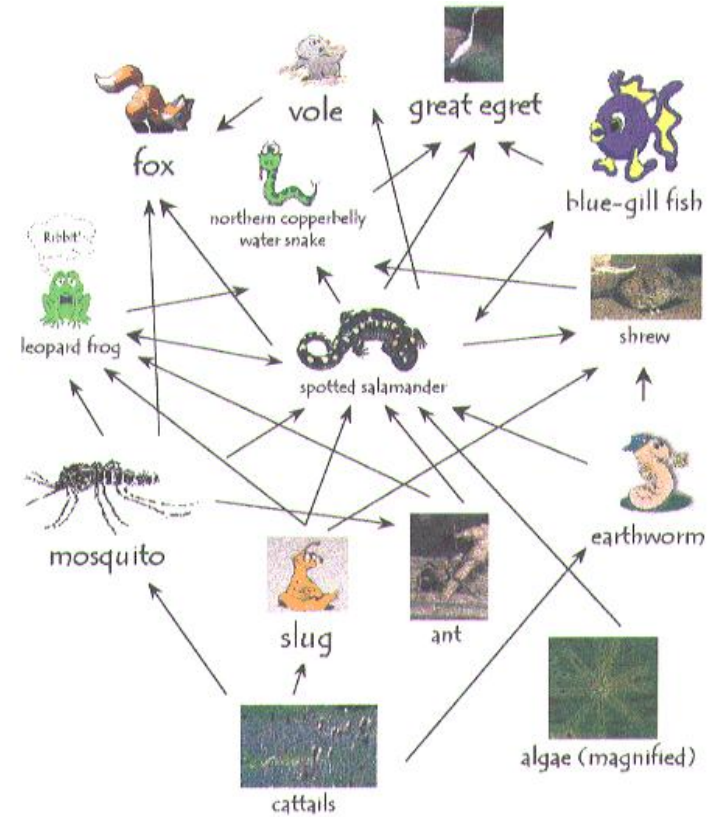
- Nodo: persona.
- Arco: relazione fra 2 persone



Ecological Food Web

Food web (grafo orientato).

- nodo = specie.
- arco = dalla preda al predatore



Reference: <http://www.twingroves.district96.k12.il.us/Wetlands/Salamander/Salgrafoics/salfoodweb.giff>

Lezioni su grafi

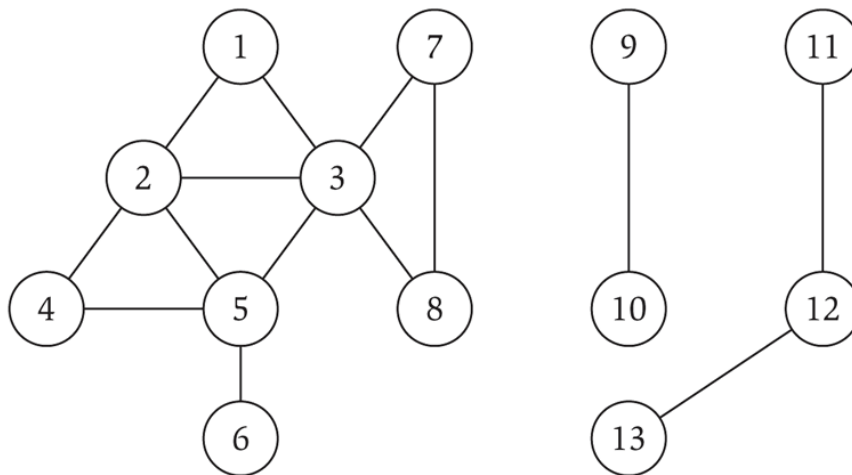
- Definizioni. Problemi di connettività e sulle componenti connesse. Visite BFS e DFS.
- Problemi su componenti connesse in grafi orientati. Test per grafi bipartiti. Ordine topologico in DAG.
- Calcolo di cammini minimi: algoritmo di Dijkstra (*greedy*) (par. 4.4). Calcolo di cammini minimi con costi anche negativi: algoritmo di Bellman-Ford (*programmazione dinamica*) (par. 6.8)
- Albero di ricoprimento minimo: algoritmi di Prim e di Kruskal (*greedy*) (par. 4.5). Applicazione di MST: clustering (par. 4.7)
(4 lezioni)
- Flusso (parr. 7.1, 7.2, 7.3) e applicazione a matching su grafi bipartiti (par. 7.5)
(2 lezioni)

Un pò di terminologia

Def. In un grafo $G = (V, E)$ se l'arco $(u,v) \in E$ allora diremo che:

- l'arco è **incidente** u e v
- u e v sono **adiacenti**
- (u,v) è un arco **uscente** da u.

Def. Il **grado** ("degree") di un nodo u, indicato $\text{deg}(u)$, è il numero di archi uscenti da u.



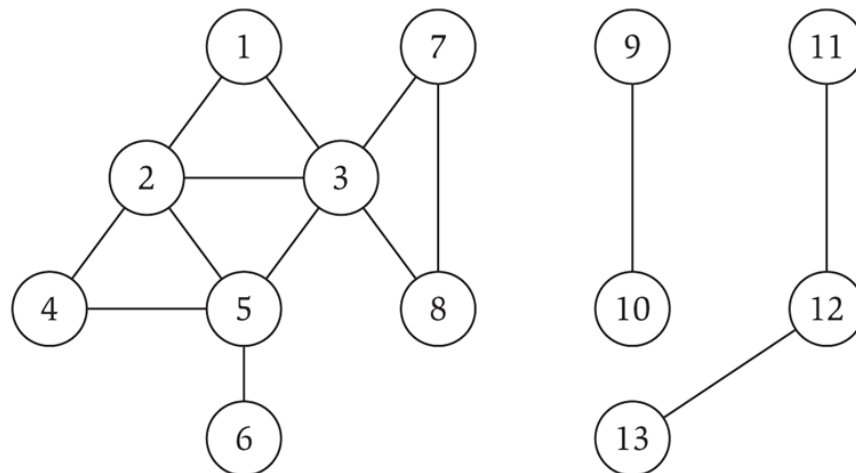
Cammini

Def. Un **cammino** in un grafo $G = (V, E)$ è una sequenza P di nodi $v_1, v_2, \dots, v_{k-1}, v_k$ con la proprietà che ogni coppia consecutiva v_i, v_{i+1} è collegata da un arco in E .

La sua **lunghezza** è $k-1$

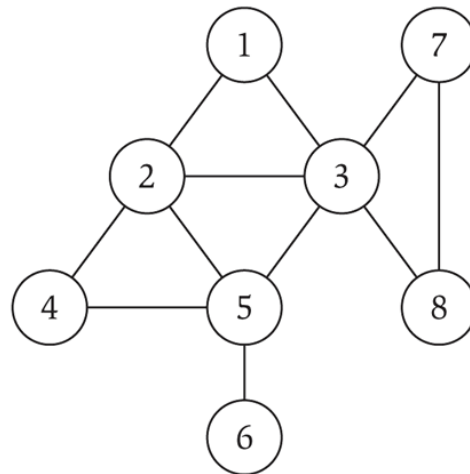
Def. Un cammino è **semplice** se tutti nodi sono distinti.

Def. Un grafo è **connesso** se per ogni coppia di nodi u e v , c'è un cammino fra u e v .



Cicli

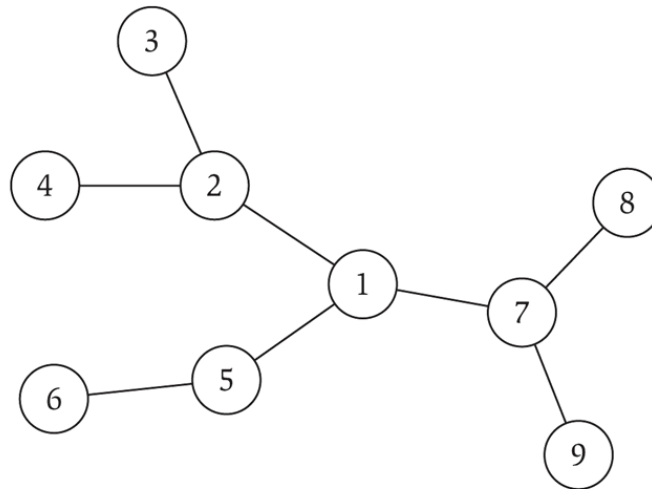
Def. Un **ciclo** è un cammino $v_1, v_2, \dots, v_{k-1}, v_k$ nel quale $v_1 = v_k$, $k > 2$, e i primi $k-1$ nodi sono tutti distinti.



ciclo $C = 1-2-4-5-3-1$

Alberi

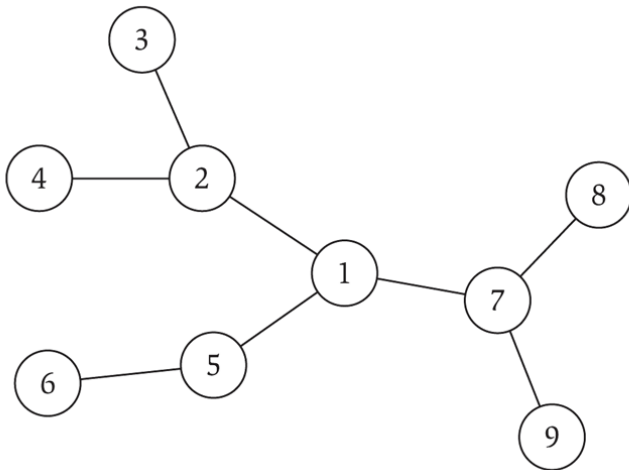
Def. Un grafo è un **albero** se è connesso e non contiene un ciclo.



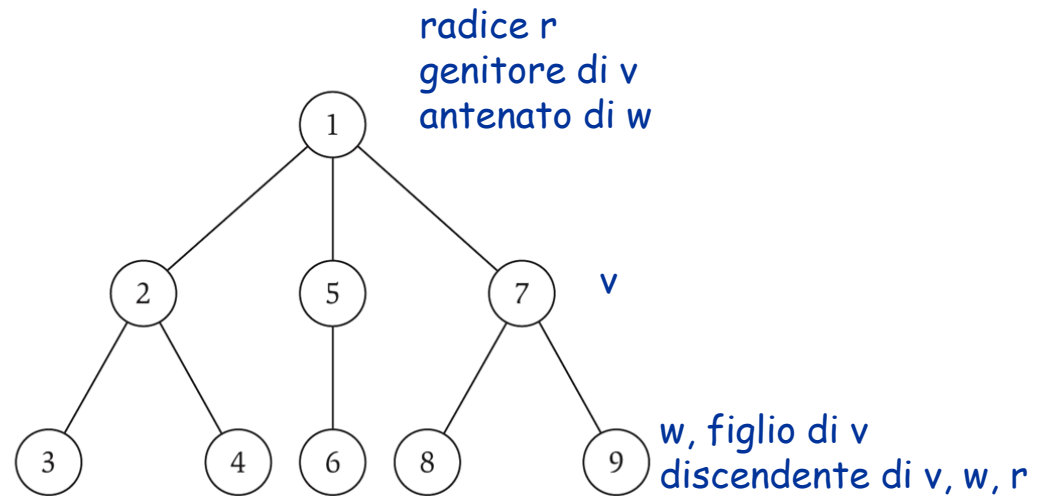
Alberi radicati

Albero radicato. Dato un albero T , scegliere un nodo **radice** r e orientare ogni arco rispetto ad r .

Importanza: Modella una struttura gerarchica



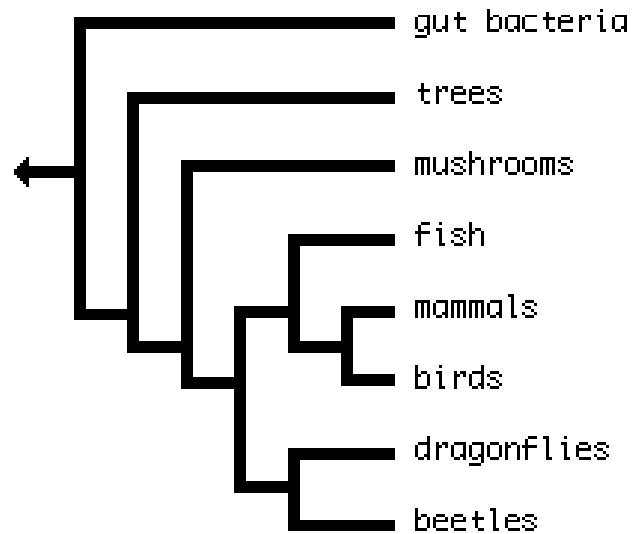
un albero



lo stesso albero, radicato in 1

Alberi filogenetici

Alberi filogenetici: Descrivono la storia evolutiva delle specie.



Proprietà degli alberi

Lemma. Ogni albero con n nodi ha $n-1$ archi

(conta gli archi da ogni nodo non radice al padre)

Teorema. Sia G un grafo con n nodi.

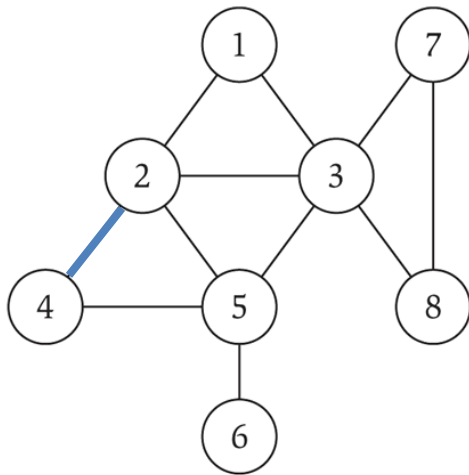
Due qualsiasi delle seguenti affermazioni implicano la terza:

- G è connesso.
- G non contiene un ciclo.
- G ha $n-1$ archi.

Rappresentazione di un grafo: Matrice di adiacenza

Matrice di adiacenza: matrice $n \times n$ con $A_{uv} = 1$ se (u, v) è un arco.

- Due rappresentazioni di ogni arco.
- Spazio proporzionale ad n^2 .
- Verificare se (u, v) è un arco richiede tempo $\Theta(1)$.
- Elencare tutti gli archi richiede tempo $\Theta(n^2)$.
- Elencare tutti gli archi incidenti su un fissato nodo richiede tempo $\Theta(n)$.

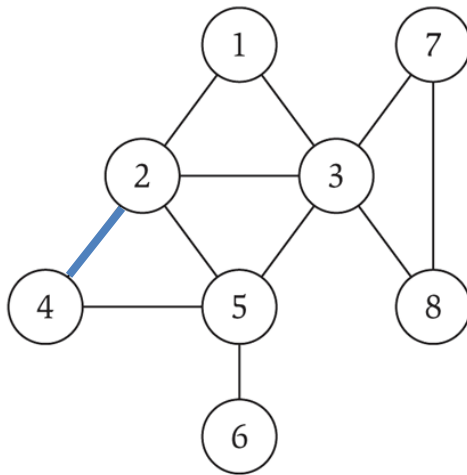


	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

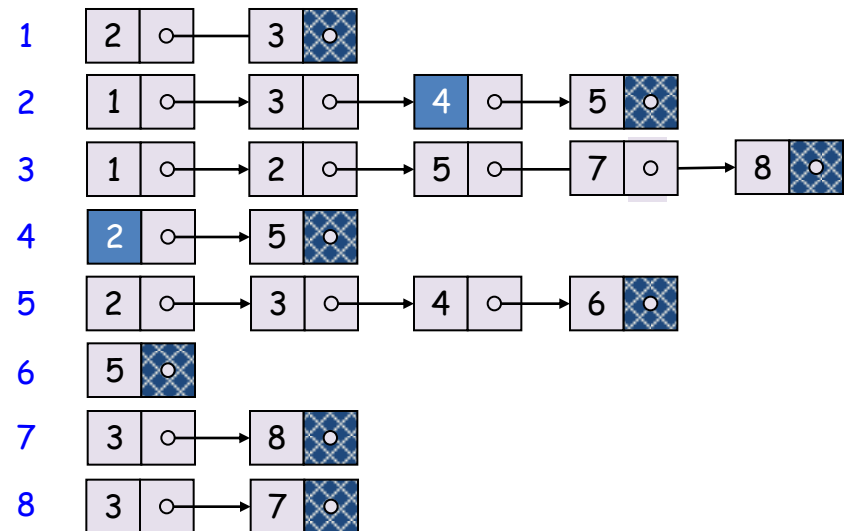
Rappresentazione di un grafo: Lista di adiacenza

Lista di adiacenza: array di n liste indicizzate dai nodi.

- Due rappresentazioni di ogni arco.
- Spazio proporzionale ad $m + n$. (ogni arco compare 2 volte)
- Verificare se (u, v) è un arco richiede tempo $O(\text{deg}(u))$.
- Elencare tutti gli archi richiede tempo $\Theta(m + n)$.
- Elencare tutti gli archi incidenti su un fissato nodo u richiede tempo $\Theta(\text{deg}(u))$.



$m=11$



Somma lunghezze liste = $2m = 22$

Confronto fra le rappresentazioni

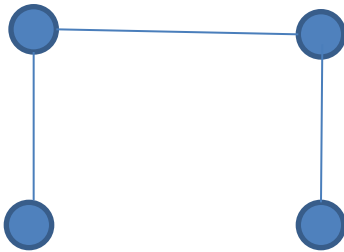
$$|V|=n, |E|=m$$

E' maggiore n^2 o $m+n$?

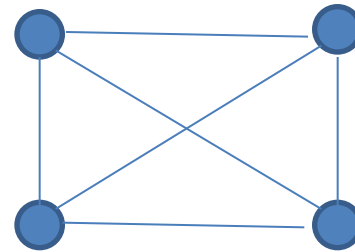
In un grafo connesso con n vertici:
quanti archi posso avere, al minimo e al massimo?

$$n-1 \leq m \leq n(n-1)/2$$

$$|E| = \Omega(n)$$



$$|E| = O(n^2)$$

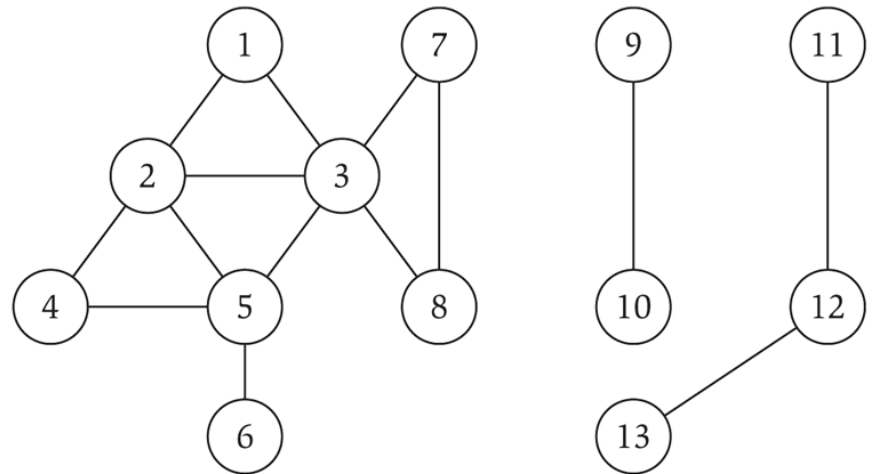


Problemi di connettività in grafi non orientati

- Problema della **connettività** s-t:
Dati due nodi s e t, esiste un cammino fra s e t?
- Problema del **cammino minimo** s-t:
Dati due nodi s e t, qual è la lunghezza del cammino minimo fra s e t?

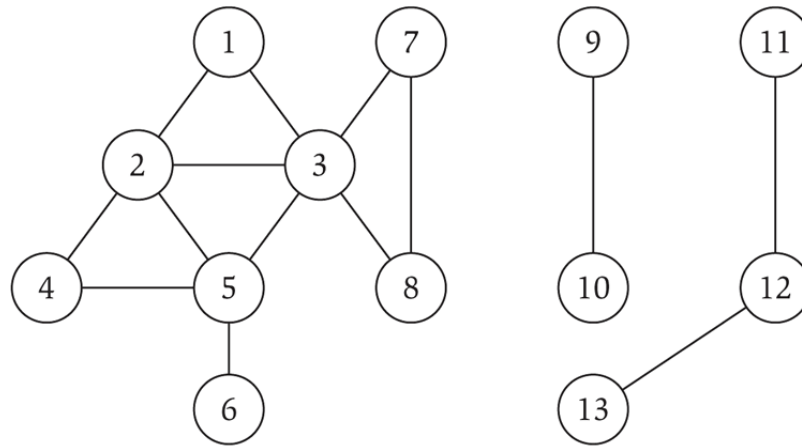
Diremo: t **raggiungibile** da s se esiste un cammino fra s e t;
distanza di s da t = lunghezza del cammino minimo fra s e t

- Vedremo:
 Breadth First Search (BFS)
 Depth First Search (DFS)



Componenti connesse

Def. La **componente connessa** di un grafo che contiene un nodo s è l'insieme dei nodi raggiungibili da s .



La componente connessa che contiene 1 è $\{ 1, 2, 3, 4, 5, 6, 7, 8 \}$.

La componente connessa che contiene 9 è $\{ 9, 10 \}$.

La componente connessa che contiene 11 è $\{ 11, 12, 13 \}$.

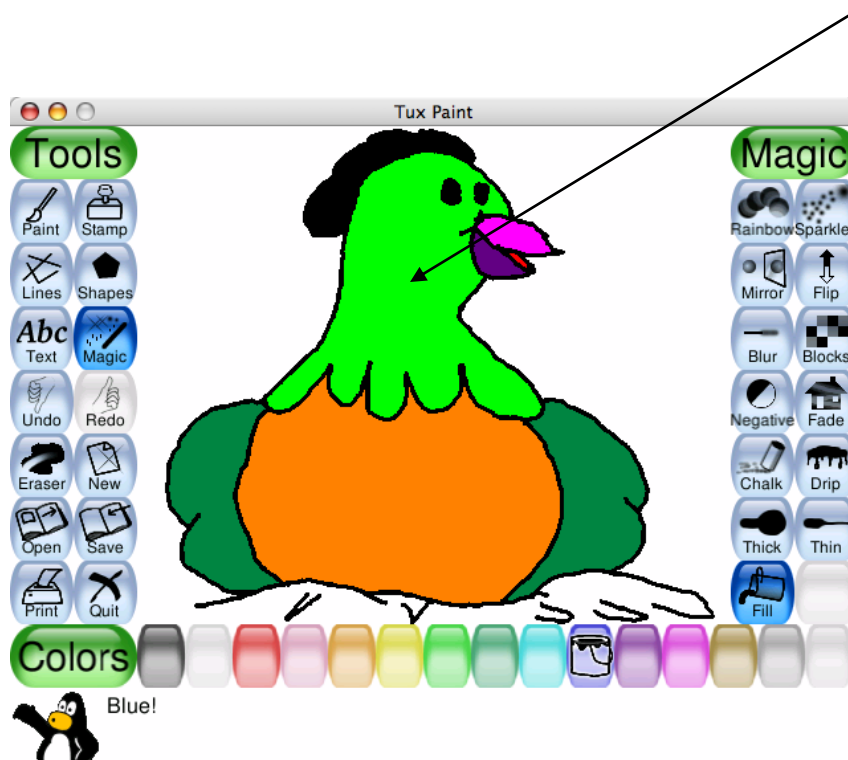
Problemi risolvibili con BFS o DFS

- Problema della **connettività** s-t:
Dati due nodi s e t , esiste un cammino fra s e t ?
- Problema del **cammino minimo** s-t:
Dati due nodi s e t , qual è la lunghezza del cammino minimo fra s e t ?
- Problema della **componente connessa** di s : trovare tutti i nodi raggiungibili da s
- Problema di tutte le **componenti connesse** di un grafo G : trovare tutte le componenti connesse di G

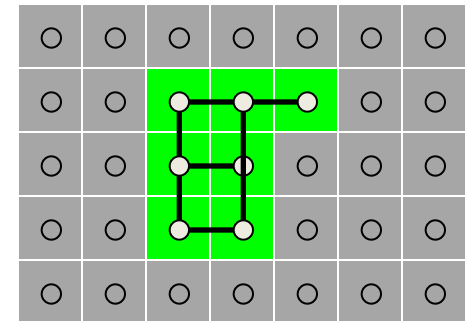
Applicazione componenti connesse: Riempimento aree

Riempimento aree. Per un fissato pixel verde in un'immagine, cambia colore a tutti i pixel adiacenti

- nodo: pixel.
- arco: fra due pixel adiacenti (nella stessa area)
- L'area da cambiare è una componente connessa



Ricolora da verde a blu

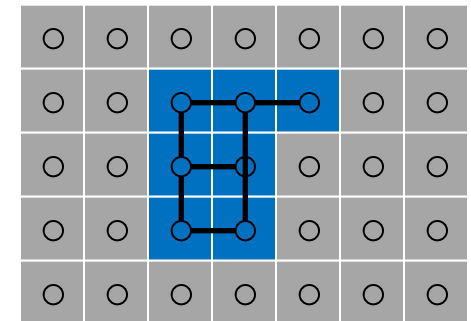


Applicazione componenti connesse: Riempimento aree

Riempimento aree. Per un fissato pixel verde in un'immagine, cambia colore a tutti i pixel adiacenti

- nodo: pixel.
- arco: fra due pixel adiacenti (nella stessa area)
- L'area da cambiare è una componente connessa

Ricolora da verde a blu

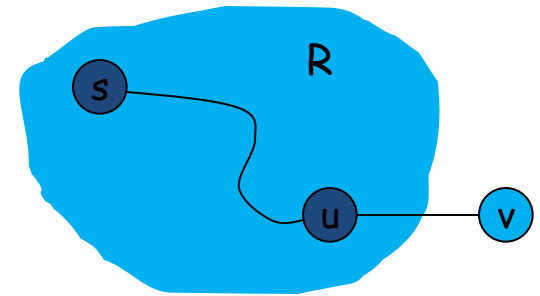


Componente connessa

Problema: Trova tutti i nodi raggiungibili da s .

Algoritmo generico:

R will consist of nodes to which s has a path
Initially $R = \{s\}$
While there is an edge (u, v) where $u \in R$ and $v \notin R$
 Add v to R
Endwhile



È possibile aggiungere v

Teorema. Terminato l'algoritmo, R è la componente connessa che contiene s .

Prova: se $v \in R$, allora è raggiungibile da s ;

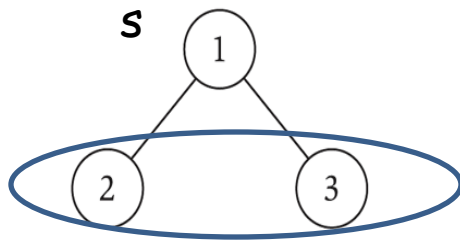
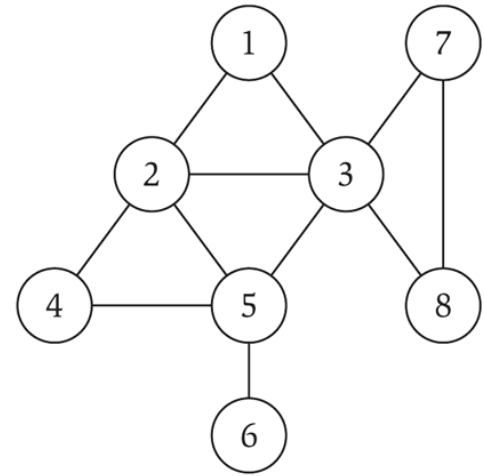
se $v \notin R$ e per assurdo esistesse un cammino $s-v$, allora esisterebbe (x, y) con $x \in R$, $y \notin R$ contro terminazione algoritmo.

Nota: BFS = esplora in ordine di distanza da s .

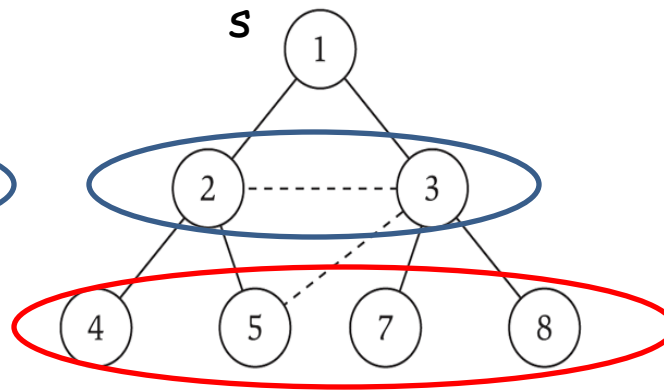
DFS = esplora come in un "labirinto".

BFS: visita in ampiezza

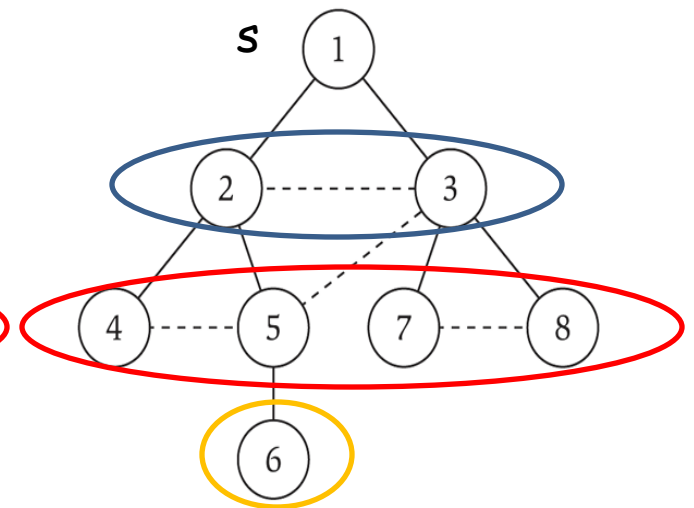
Idea della BFS: Esplorare a partire da s in tutte le possibili direzioni, aggiungendo nodi, uno strato ("layer") alla volta.



(a)



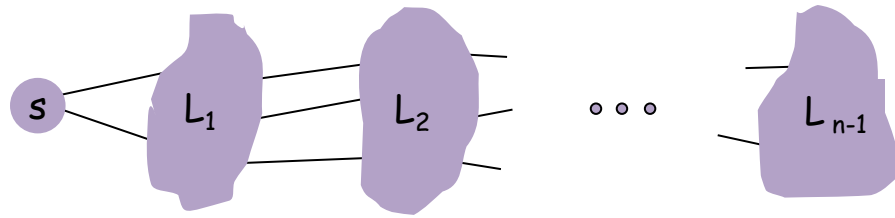
(b)



(c)

Breadth First Search

L_i sono i layers:



Algoritmo BFS:

- $L_0 = \{s\}$.
- $L_1 =$ tutti i vicini di L_0 .
- $L_2 =$ tutti i nodi che non sono in L_0 o L_1 , e che hanno un arco con un nodo in L_1 .
- ...
- $L_{i+1} =$ tutti i nodi che non sono in un layer precedente, e che hanno un arco con un nodo in L_i .

Teorema.

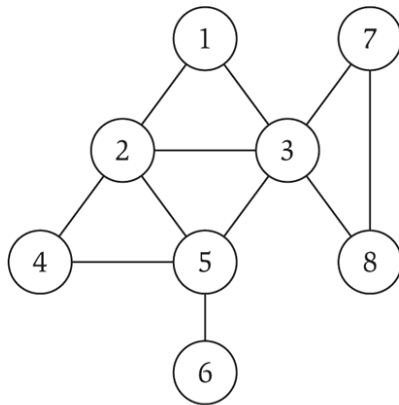
Per ogni i , L_i consiste di tutti i nodi a distanza i da s .

Esiste un cammino da s a t se e solo se t appare in qualche layer.

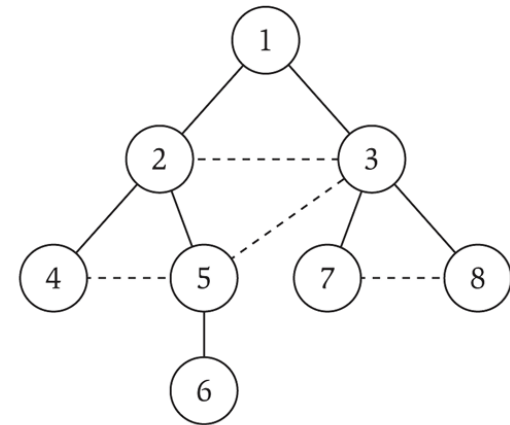
Prova: per induzione

Breadth First Search: proprietà

- Determina i nodi raggiungibili da s (insieme dei layer)
- Determina la loro distanza da s (indice del layer)
- Produce un albero radicato in s : **l'albero BFS**
(aggiungo (u,v) quando $u \in L_i$ e $v \notin L_0 \cup \dots \cup L_i$)



Grafo G

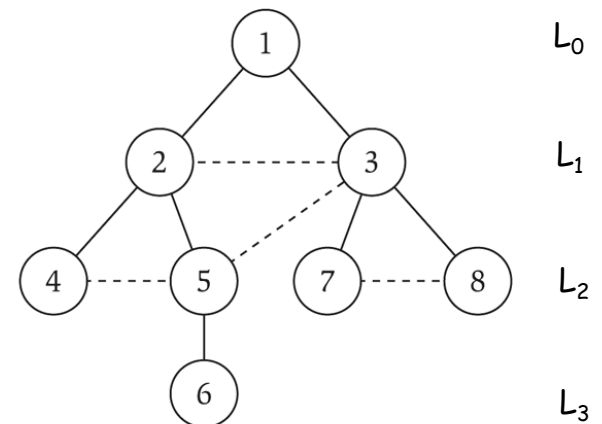
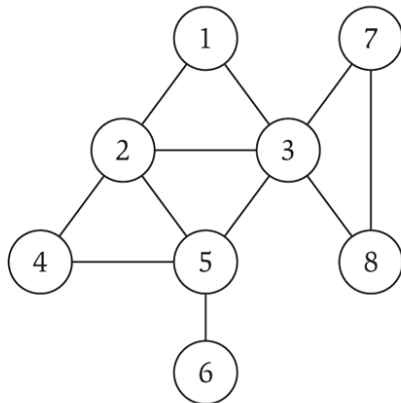


Gli archi non tratteggiati formano l'albero BFS di G

Proprietà dell'albero BFS

Proprietà: Sia T un albero BFS di un grafo G , sia (x, y) un arco di G con x appartenente ad L_i , y ad L_j .
Allora i e j differiscono di al più 1.

Prova: Supponi $i \leq j$. Quando BFS esamina gli archi incidenti x , o y viene scoperto ora (y in L_{i+1}), o è stato scoperto prima (contro $i \leq j$).



BFS implementazione

BFS(s):

Set Discovered[s] = true and Discovered[v] = false for all other v

Initialize $L[0]$ to consist of the single element s

Set the layer counter $i = 0$

Set the current BFS tree $T = \emptyset$

While $L[i]$ is not empty

 Initialize an empty list $L[i+1]$

 For each node $u \in L[i]$

 Consider each edge (u, v) incident to u

 If Discovered[v] = false then

 Set Discovered[v] = true

 Add edge (u, v) to the tree T

 Add v to the list $L[i+1]$

 Endif

 Endfor

 Increment the layer counter i by one

Endwhile

BFS: analisi

Teorema: L'implementazione di BFS richiede tempo $O(m+n)$ se il grafo è rappresentato con una *lista delle adiacenze*.

Prova:

E' facile provare un running time $O(n^2)$.

Un'analisi più accurata da $O(m+n)$.

BFS implementazione

BFS(s):

Set Discovered[s] = true and Discovered[v] = false for all other v

Initialize L[0] to consist of the single element s

Set the layer counter $i=0$

Set the current BFS tree $T = \emptyset$

While L[i] is not empty

 Initialize an empty list L[i+1]

 For each node $u \in L[i]$

 Consider each edge (u,v) incident to u

 If Discovered[v] = false then

 Set Discovered[v] = true

 Add edge (u,v) to the tree T

 Add v to the list L[i+1]

 Endif

 Endfor

 Increment the layer counter i by one

Endwhile

E' facile provare running time $O(n^2)$:

- Inizializzazione in $O(n)$
- al massimo n liste L[i] da creare in $O(n)$
- ogni nodo è presente in al più una lista: cicli for in totale iterano $\leq n$ volte
- Per un fissato nodo u, vi sono al più n archi incidenti (u,v) e spendiamo $O(1)$ per ogni arco.

BFS implementazione

BFS(s):

Set Discovered[s] = true and Discovered[v] = false for all other v

Initialize L[0] to consist of the single element s

Set the layer counter $i=0$

Set the current BFS tree $T = \emptyset$

While L[i] is not empty

 Initialize an empty list L[i+1]

 For each node $u \in L[i]$

 Consider each edge (u,v) incident to u

 If Discovered[v] = false then

 Set Discovered[v] = true

 Add edge (u,v) to the tree T

 Add v to the list L[i+1]

 Endif

 Endfor

 Increment the layer counter i by one

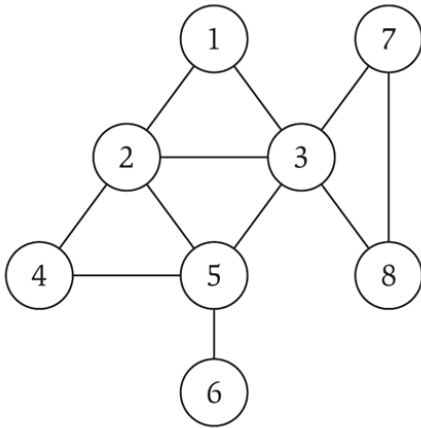
Endwhile

Un'analisi più accurata: $O(m+n)$:

- Inizializzazione in $O(n)$
- Al massimo n liste L[i] da creare in $O(n)$
- Ogni nodo è presente in al più una lista: per un fissato nodo u vi sono $\text{deg}(u)$ archi incidenti (u,v)
- Tempo totale per processare gli archi è $\sum_{u \in V} \text{deg}(u) = 2m$

DFS: visita in profondità

Idea di DFS: Esplorare quanto più in profondità possibile e tornare indietro ("backtrack") solo quando è necessario (*come in un labirinto...*)



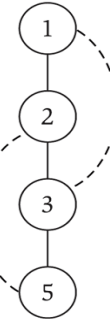
G



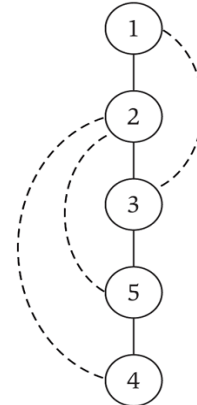
(a)



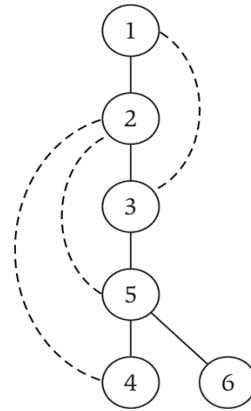
(b)



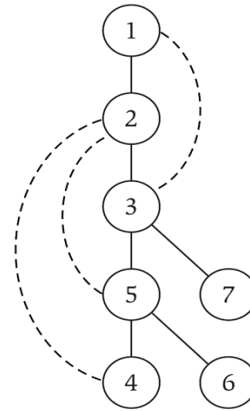
(c)



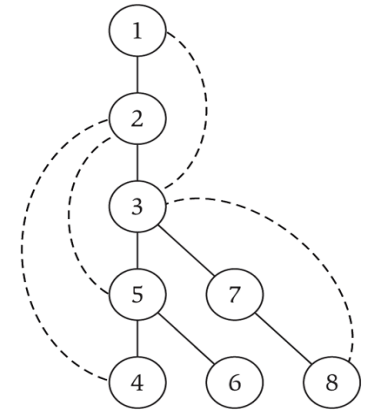
(d)



(e)



(f)



(g)

Gli archi non tratteggiati in (g) formano l'albero DFS di G

Algoritmo DFS

Idea di DFS: Esplorare quanto più in profondità possibile e tornare indietro ("backtrack") solo quando è necessario (*come in un labirinto...*)

Algoritmo ricorsivo

```
DFS( $u$ ):  
  Mark  $u$  as "Explored" and add  $u$  to  $R$   
  For each edge  $(u, v)$  incident to  $u$   
    If  $v$  is not marked "Explored" then  
      Recursively invoke DFS( $v$ )  
    Endif  
  Endfor
```

Algoritmo DFS

DFS(u):

Mark u as "Explored" and add u to R

For each edge (u, v) incident to u

If v is not marked "Explored" then

Add (u, v) to T

 Recursively invoke DFS(v)

Endif

Endfor

Proprietà 1: Per una fissata chiamata ricorsiva DFS(u), tutti i nodi che sono marcati `Explored` tra l'inizio e la fine della chiamata ricorsiva sono discendenti di u in T .

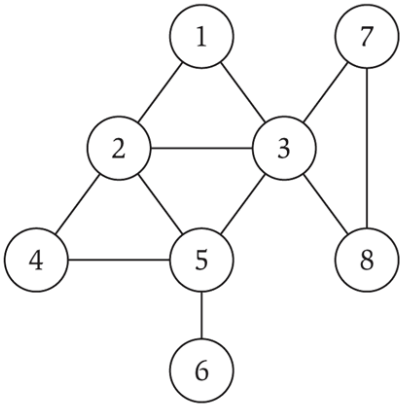
Proprietà 2: Sia T un DFS, siano x e y nodi in T si supponga (x, y) non in T . Allora x è antenato di y o viceversa.

Prova:

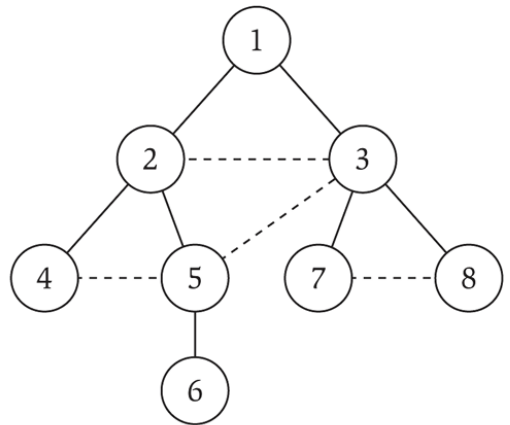
Esempio: $x=1, y=3$. DFS(1) chiama DFS(2) e DFS(3). Poiché (x, y) non in T , quando chiama DFS(3), 3 è già `Explored` quindi è stato marcato durante la chiamata DFS(1) + Proprietà 1.

Alberi BFS e DFS

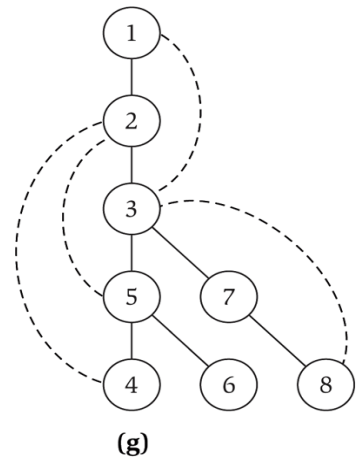
Grafo G



Albero BFS di G



Albero DFS di G



(g)

Applicazioni di BFS e DFS

- Problema della **connettività** s - t :
Dati due nodi s e t , esiste un cammino fra s e t ?
- Problema del **cammino minimo** s - t :
Dati due nodi s e t , qual è la lunghezza del cammino minimo fra s e t ?
- Problema della **componente connessa** di s : trovare tutti i nodi raggiungibili da s

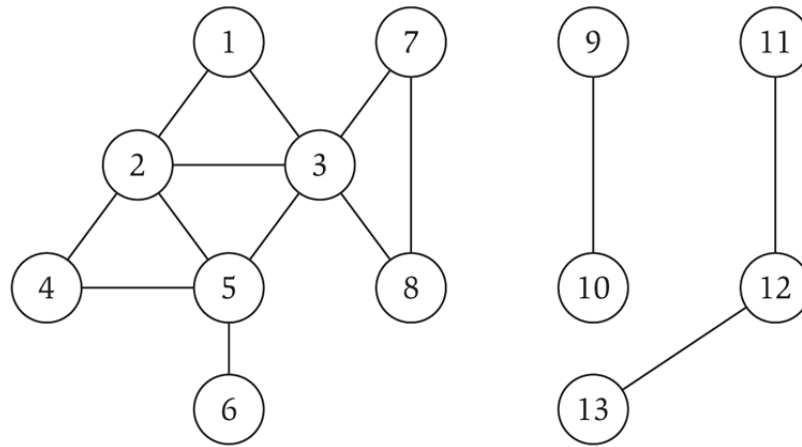
Resta:

- Problema di **tutte le componenti connesse** di un grafo G : trovare tutte le componenti connesse di G

Relazioni fra componenti connesse

Def. La **componente connessa** di un grafo che contiene un nodo s è l'insieme dei nodi raggiungibili da s .

Problema: Che relazione c'è fra due componenti connesse?



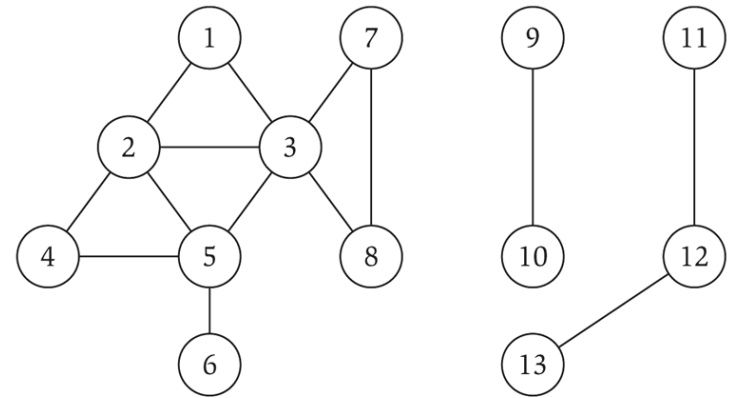
Proprietà: Siano x ed y due nodi in un grafo G . Allora le componenti connesse sono identiche oppure disgiunte

Tutte le componenti connesse

Proprietà: Siano x ed y due nodi in un grafo G . Allora le componenti connesse sono identiche oppure disgiunte

Prova . Due casi:

- Esiste un cammino tra x e y :
(ogni nodo v raggiungibile da x è raggiungibile da y)
- Non esiste un cammino tra x e y
(idem)



Algoritmo per trovare tutte le componenti connesse:

Partendo da s qualsiasi usa BFS (o DFS) per generare la componente connessa di s . Trova un nodo v non visitato. Se esiste, usa BFS da v per calcolare la componente connessa di v (che è disgiunta). Ripeti.

Tempo: $O(m+n)$

In realtà $BFS(s)$ richiede tempo lineare nel numero di nodi e archi della componente connessa di s .

Altre applicazioni di BFS e DFS

- Problema della verifica se un grafo è **bipartito** (par. 3.4)
- Problema della **connettività** nei grafi diretti (par. 3.5)

... e altre ancora.

Shortest paths

4.4 Shortest Paths in a Graph (only with positive costs):

Dijkstra Algorithm: Greedy

6.8 Shortest Paths in a Graph:

Bellman & Ford Algorithm: Dynamic Programming

Shortest Path Problem

Shortest path network.

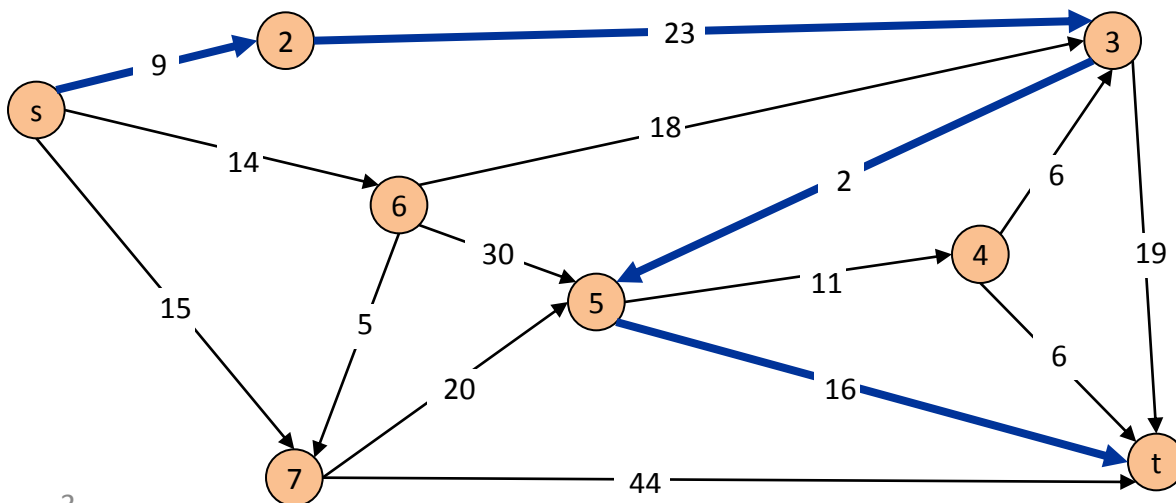
Directed graph $G = (V, E)$.

Source s , destination t .

Length ℓ_e = length/cost/weight of edge e .

Shortest path problem: find shortest (min cost) directed path from s to t .

cost of path = sum of edge costs in path



Cost of path s -2-3-5- t
= $9 + 23 + 2 + 16$
= 48.

Dijkstra's Algorithm

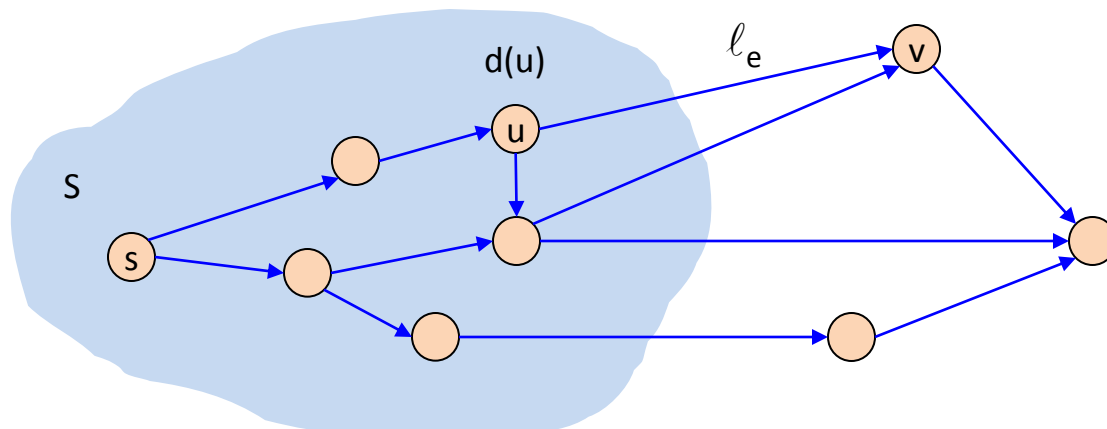
Dijkstra's algorithm (greedy approach).

- Maintain a set of **explored nodes** S for which we have determined the shortest path distance $d(u)$ from s to u .
- Initialize $S = \{s\}$, $d(s) = 0$.
- Repeatedly choose unexplored node v which minimizes

$$\pi(v) = \min_{e = (u,v) : u \in S} d(u) + \ell_e$$

- Add v to S , and set $d(v) = \pi(v)$.

shortest path to some u in explored part, followed by a single edge (u, v)



Dijkstra's Algorithm

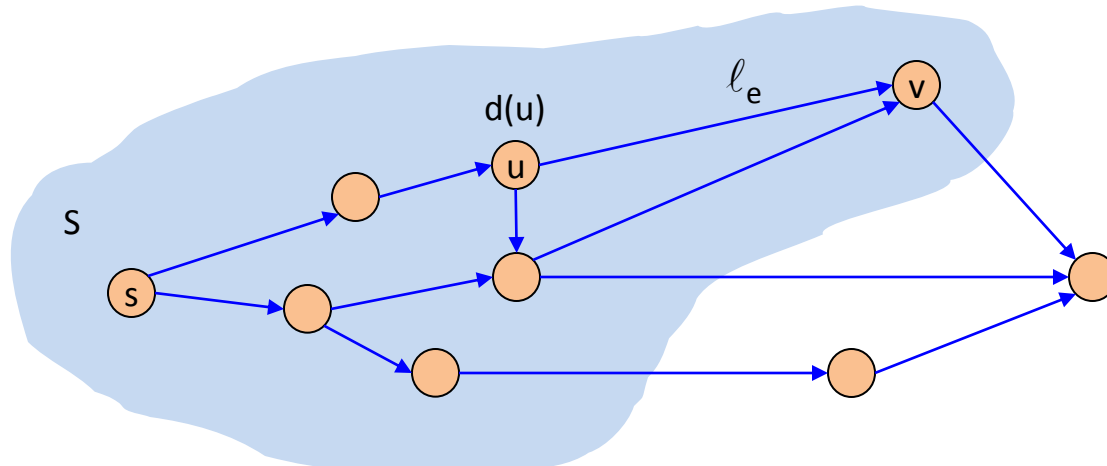
Dijkstra's algorithm.

- Maintain a set of **explored nodes** S for which we have determined the shortest path distance $d(u)$ from s to u .
- Initialize $S = \{s\}$, $d(s) = 0$.
- Repeatedly choose unexplored node v which minimizes

$$\pi(v) = \min_{e=(u,v): u \in S} d(u) + \ell_e,$$

- Add v to S , and set $d(v) = \pi(v)$.

shortest path to some u in explored part, followed by a single edge (u, v)



Dijkstra's Algorithm: Proof of Correctness

Invariant. For each node $u \in S$, $d(u)$ is the length of the shortest s - u path.

Pf. (by induction on $|S|$)

Base case: $|S| = 1$ is trivial.

Inductive hypothesis: Assume true for $|S| = k \geq 1$.

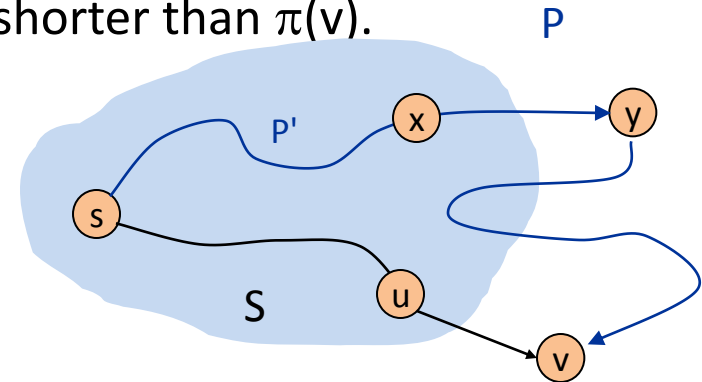
Let v be next node added to S , and let u - v be the chosen edge.

The shortest s - u path plus (u, v) is an s - v path of length $\pi(v)$.

Consider any s - v path P . We'll see that it's no shorter than $\pi(v)$.

Let x - y be the first edge in P that leaves S , and let P' be the subpath to x .

P is already too long as soon as it leaves S .



$$\begin{array}{ccccccc}
 \ell(P) & \geq & \ell(P') + \ell(x, y) & \geq & d(x) + \ell(x, y) & \geq & \pi(y) \geq \pi(v) \\
 \uparrow & & \uparrow & \nearrow & \uparrow & & \uparrow \\
 \text{nonnegative} & & \text{inductive} & & \text{defn of } \pi(y) & & \text{Dijkstra chose } v \\
 \text{weights} & & \text{hypothesis} & & & & \text{instead of } y \\
 & & \text{on } x & & & &
 \end{array}$$

Dijkstra's Algorithm: Implementation

For each unexplored node, explicitly maintain

$$\pi(v) = \min_{e=(u,v): u \in S} d(u) + \ell_e .$$

Next node to explore = node with minimum $\pi(v)$.

When exploring v , for each incident edge $e = (v, w)$, update

$$\pi(w) = \min \{ \pi(w), \pi(v) + \ell_e \} .$$

Efficient implementation. Maintain a priority queue of unexplored nodes, prioritized by $\pi(v)$.

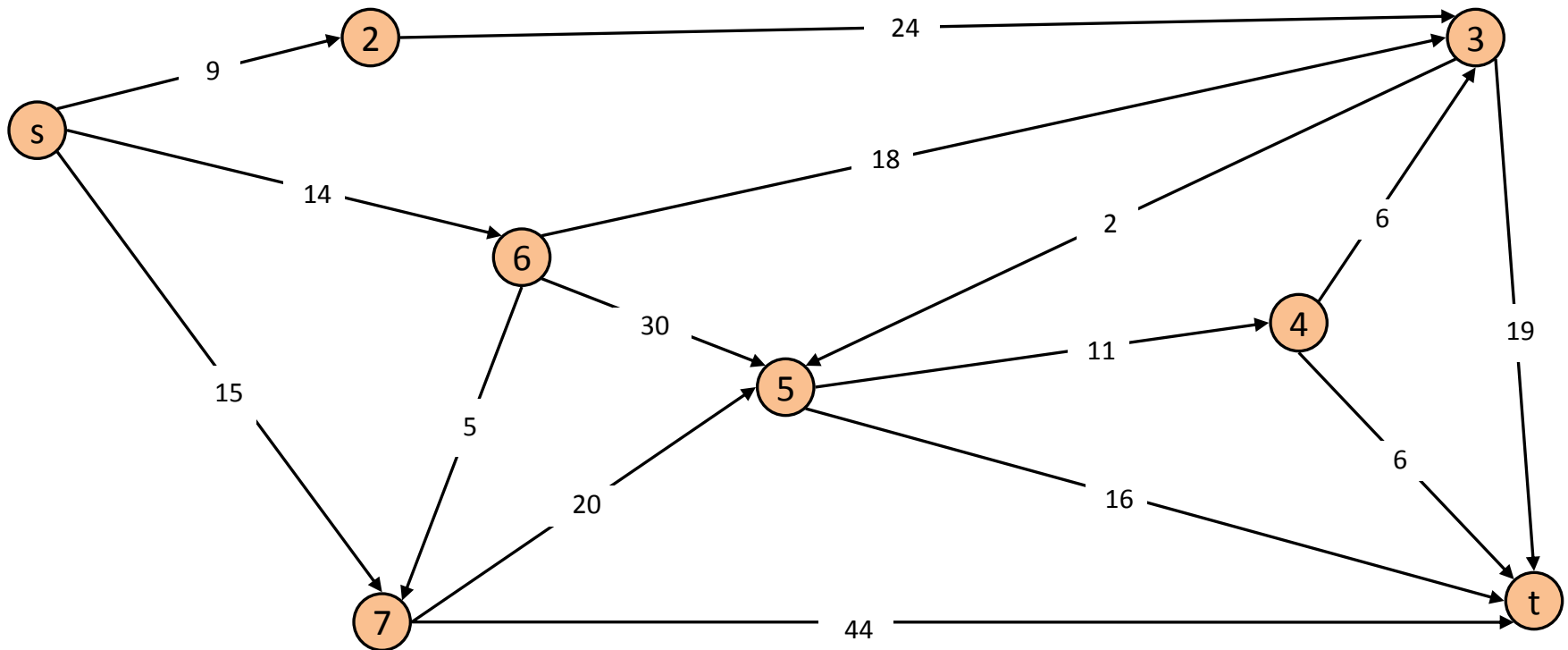
Priority Queue

PQ Operation	Dijkstra	Array	Binary heap	d-way Heap	Fib heap [†]
Insert	n	1	log n	d log _d n	1
ExtractMin	n	n	log n	d log _d n	log n
ChangeKey	m	1	log n	log _d n	1
IsEmpty	n	1	1	1	1
Total		n ²	m log n	m log _{m/n} n	m + n log n

[†] Individual ops are amortized bounds

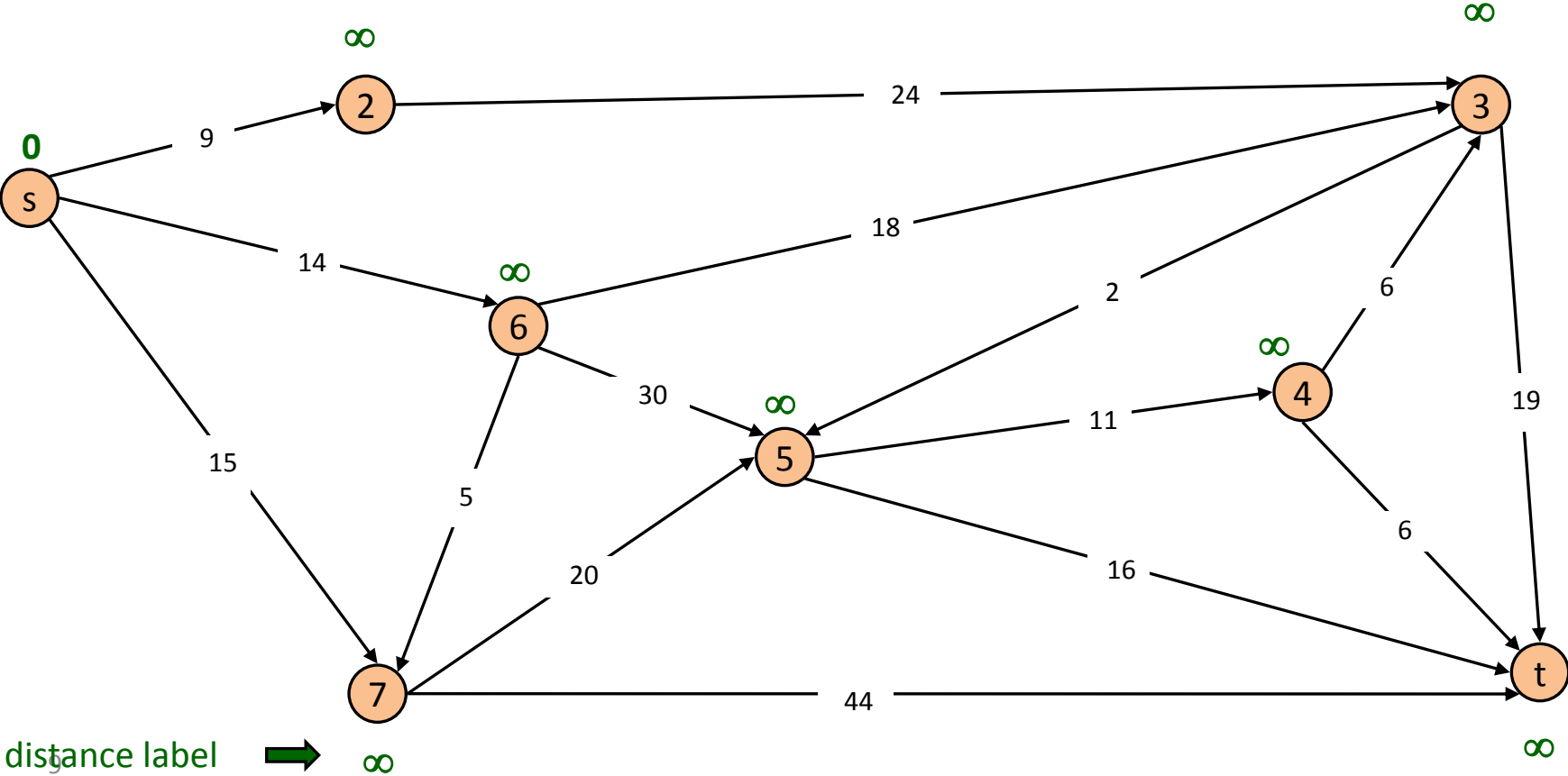
Dijkstra's Shortest Path Algorithm

Find shortest path from s to t.



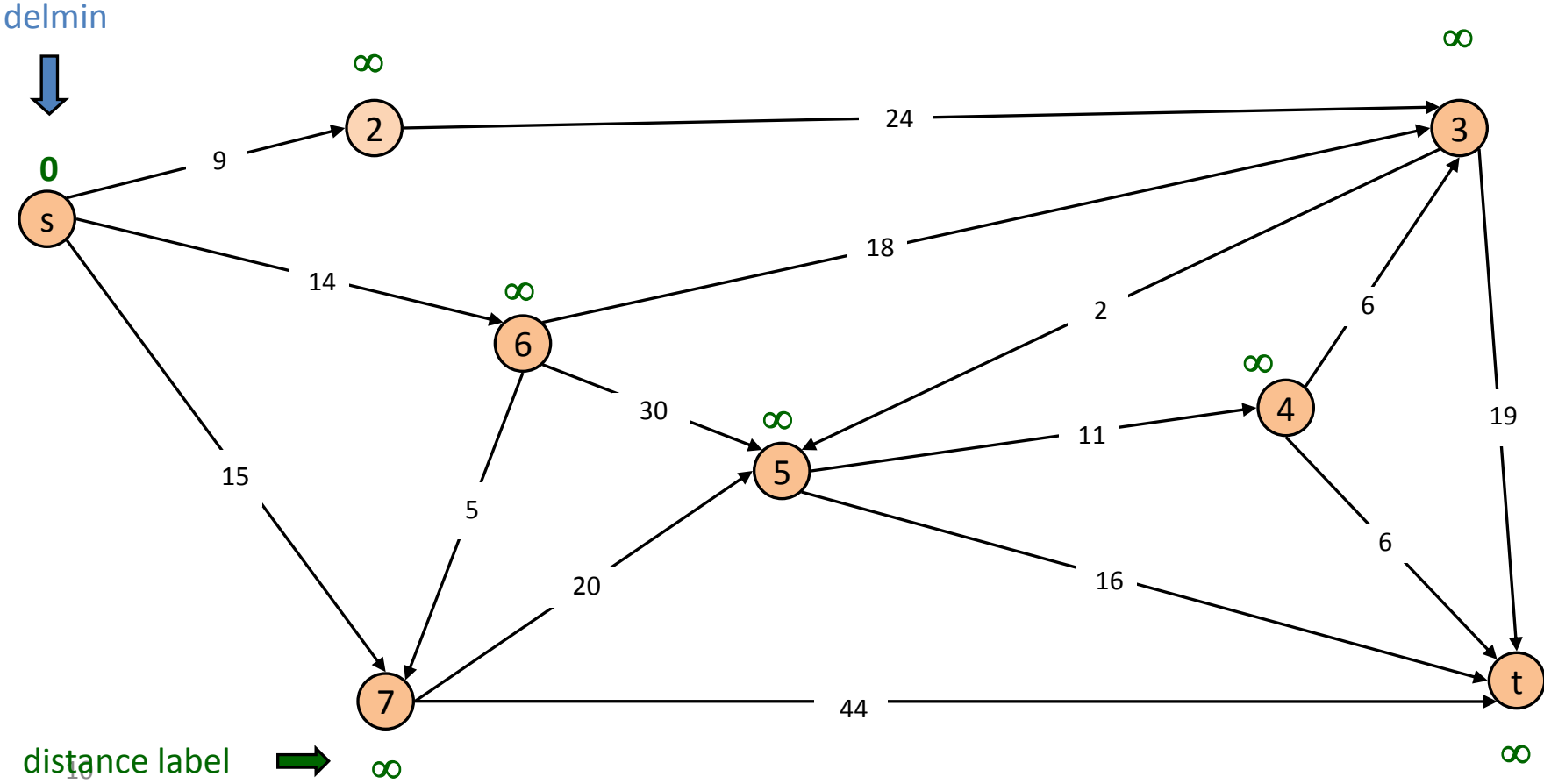
Dijkstra's Shortest Path Algorithm

$S = \{ \}$
 $PQ = \{s, 2, 3, 4, 5, 6, 7, t\}$



Dijkstra's Shortest Path Algorithm

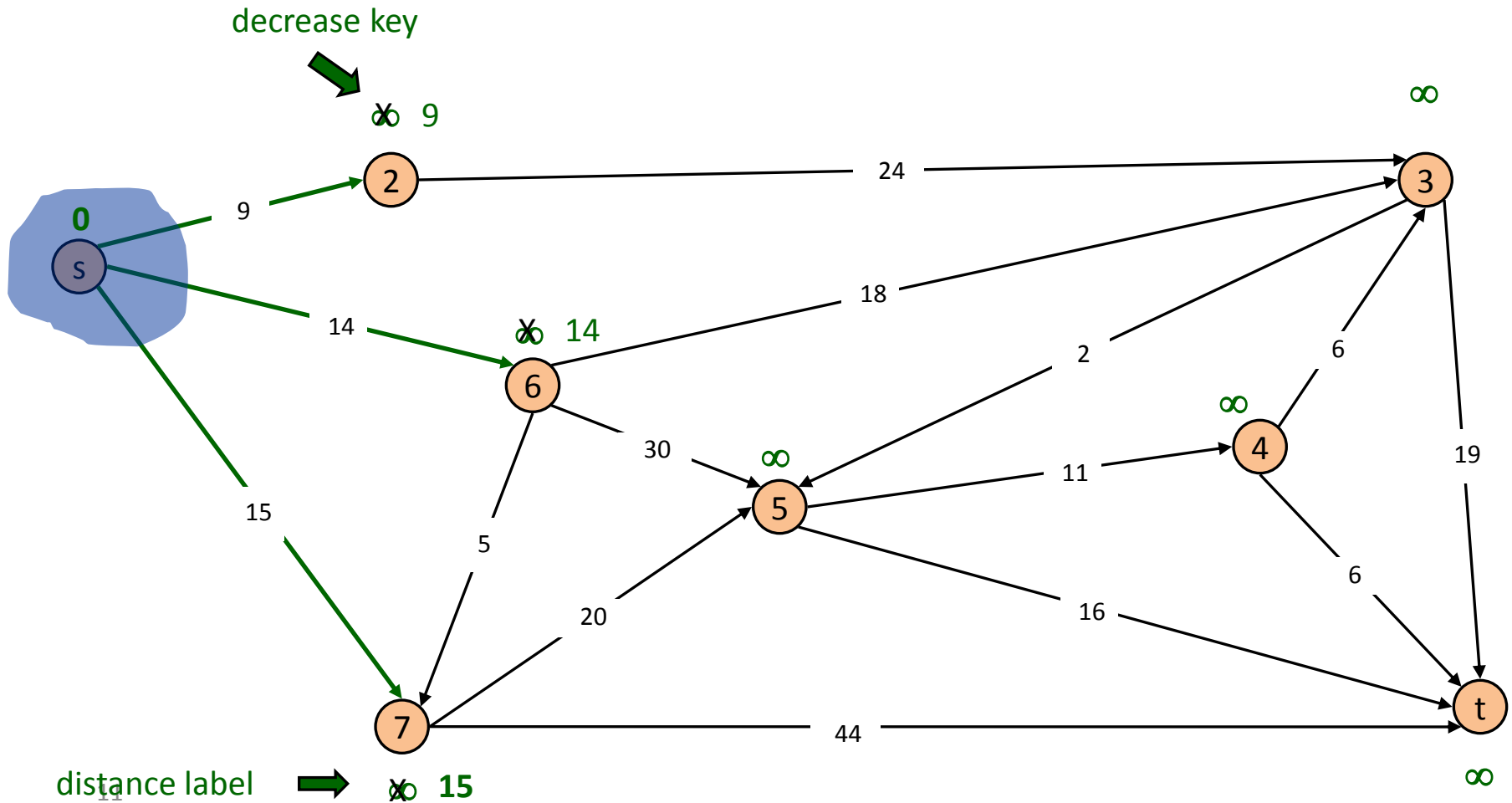
$S = \{ \}$
 $PQ = \{s, 2, 3, 4, 5, 6, 7, t\}$



Dijkstra's Shortest Path Algorithm

$S = \{s\}$

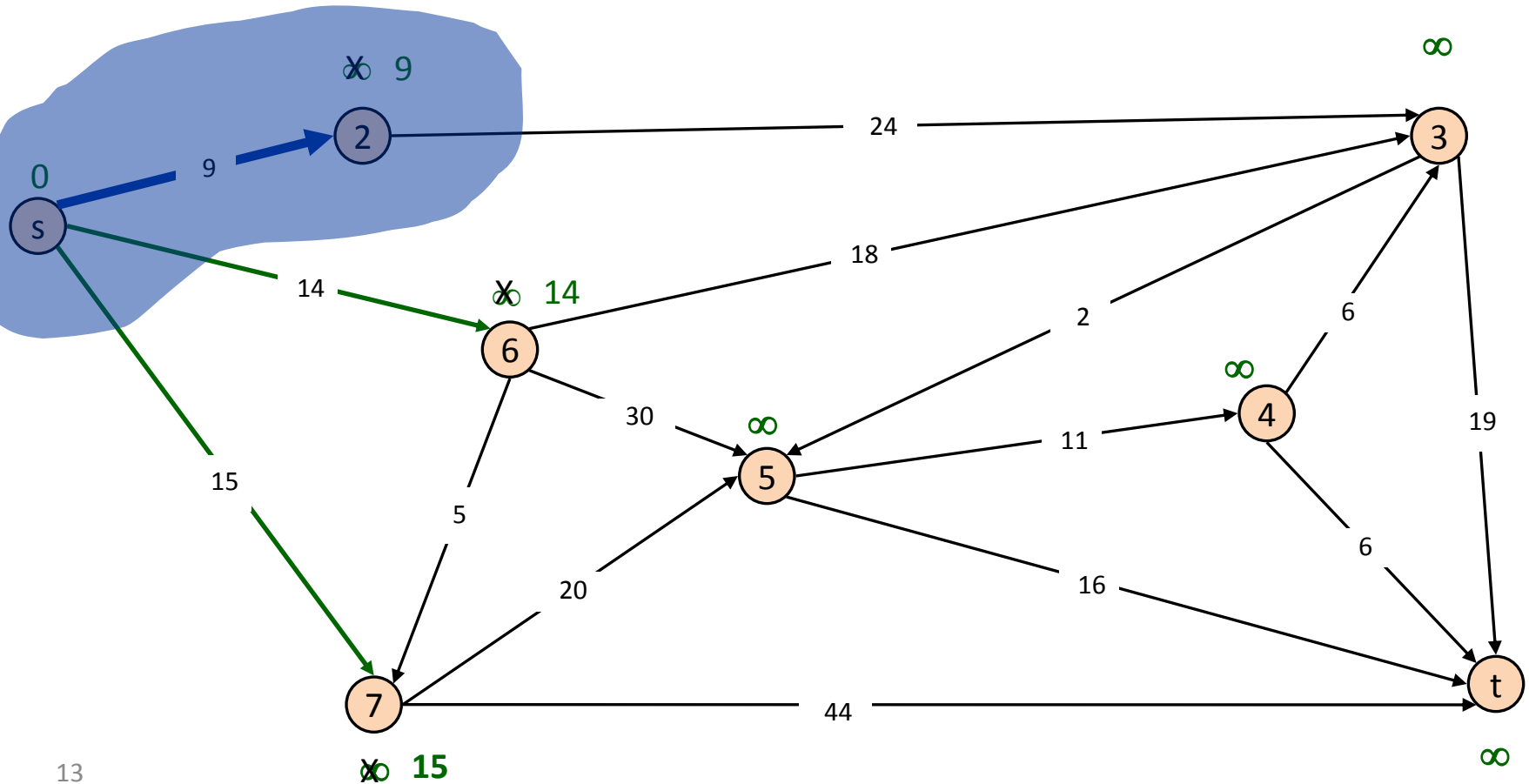
$PQ = \{2, 3, 4, 5, 6, 7, t\}$



Dijkstra's Shortest Path Algorithm

$S = \{s, 2\}$

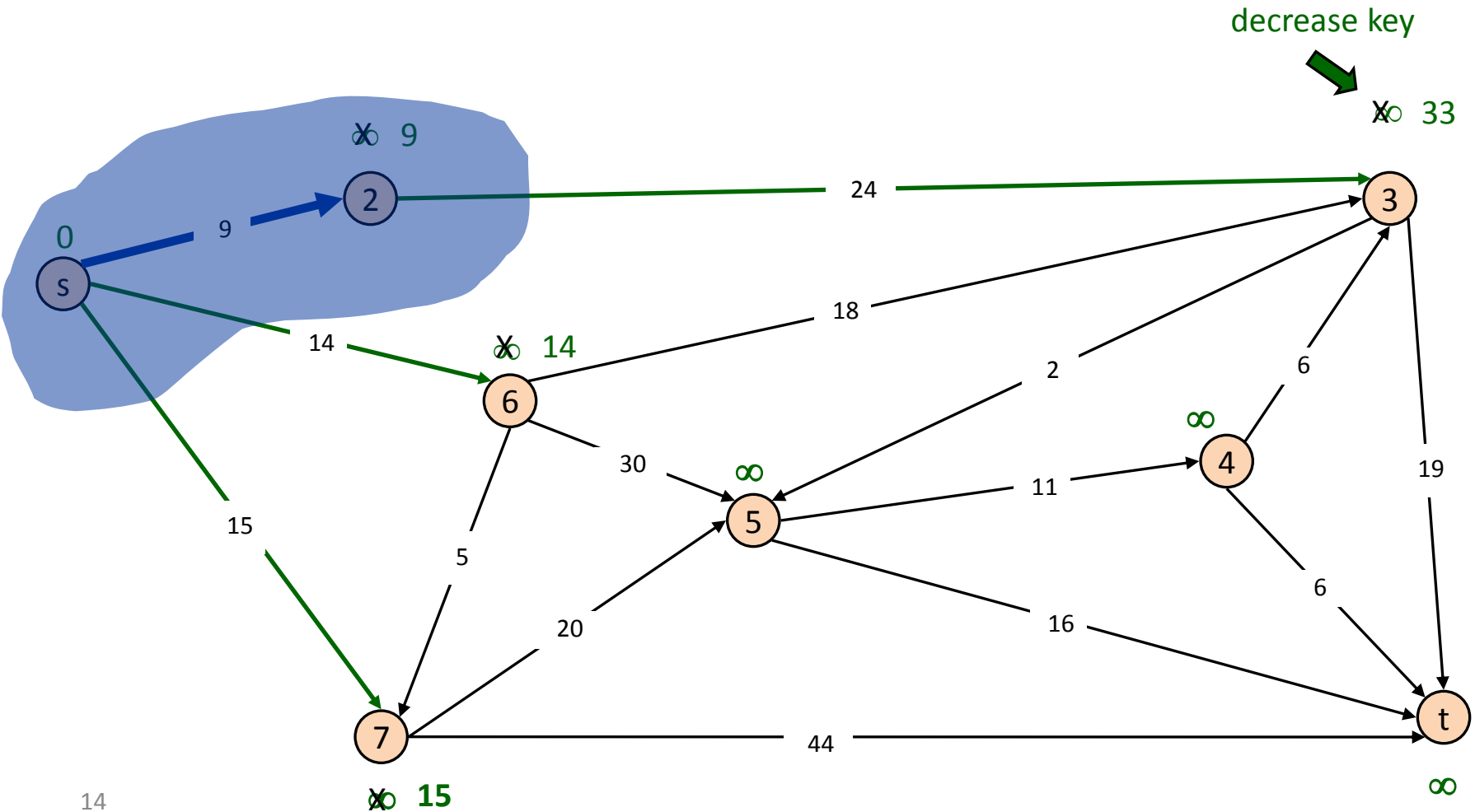
$PQ = \{3, 4, 5, 6, 7, t\}$



Dijkstra's Shortest Path Algorithm

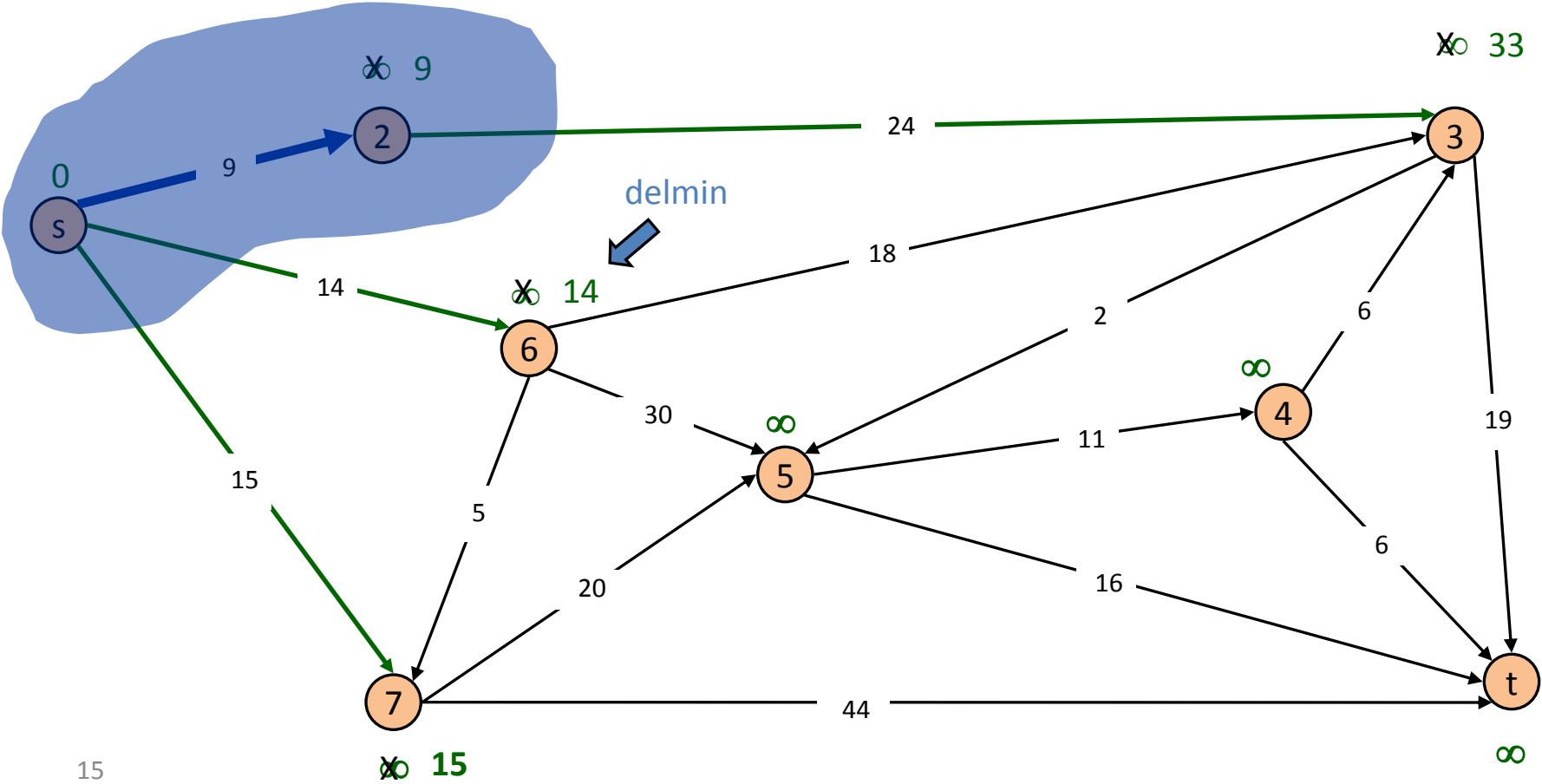
$S = \{s, 2\}$

$PQ = \{3, 4, 5, 6, 7, t\}$



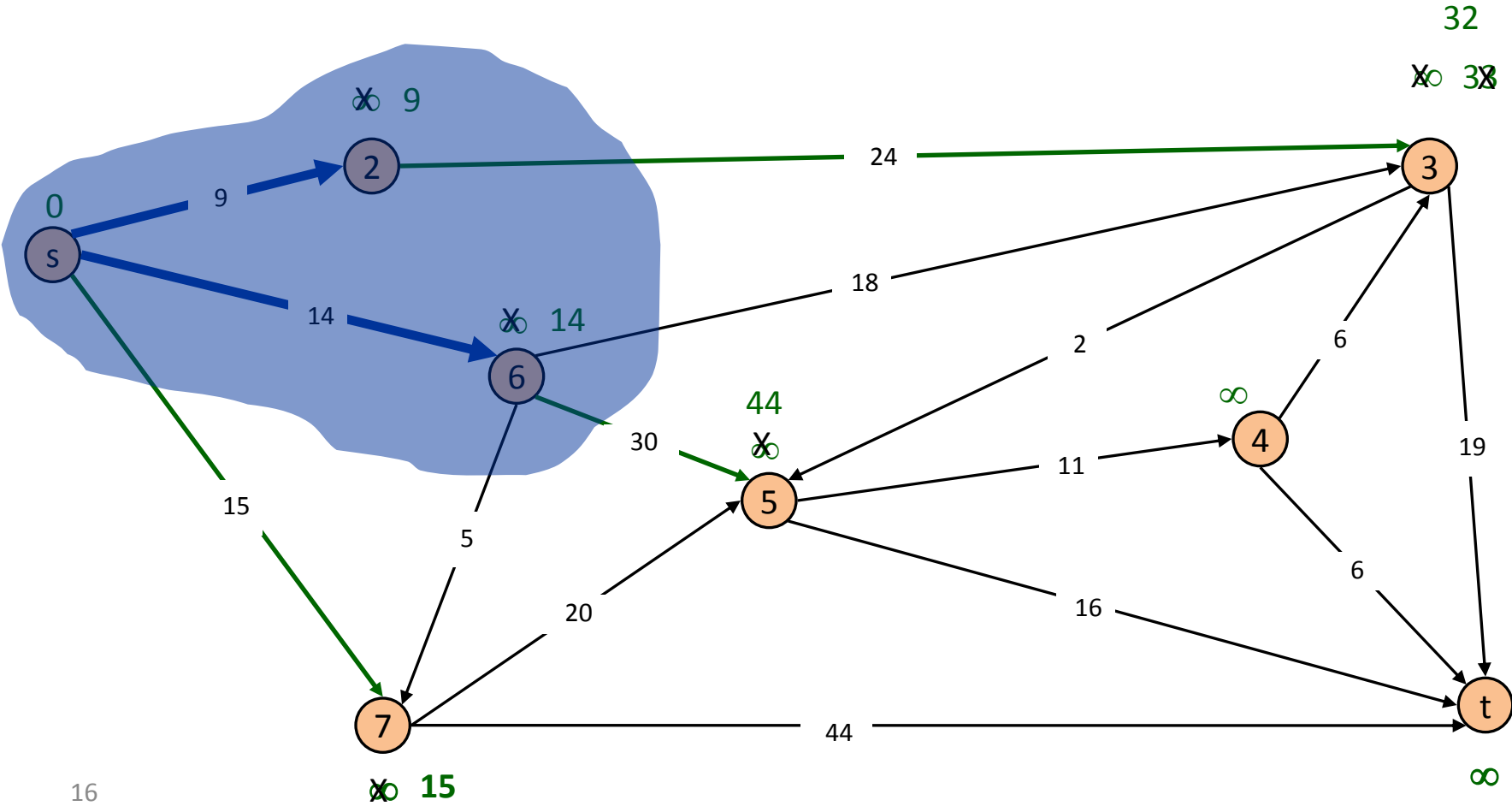
Dijkstra's Shortest Path Algorithm

$S = \{s, 2\}$
 $PQ = \{3, 4, 5, 6, 7, t\}$



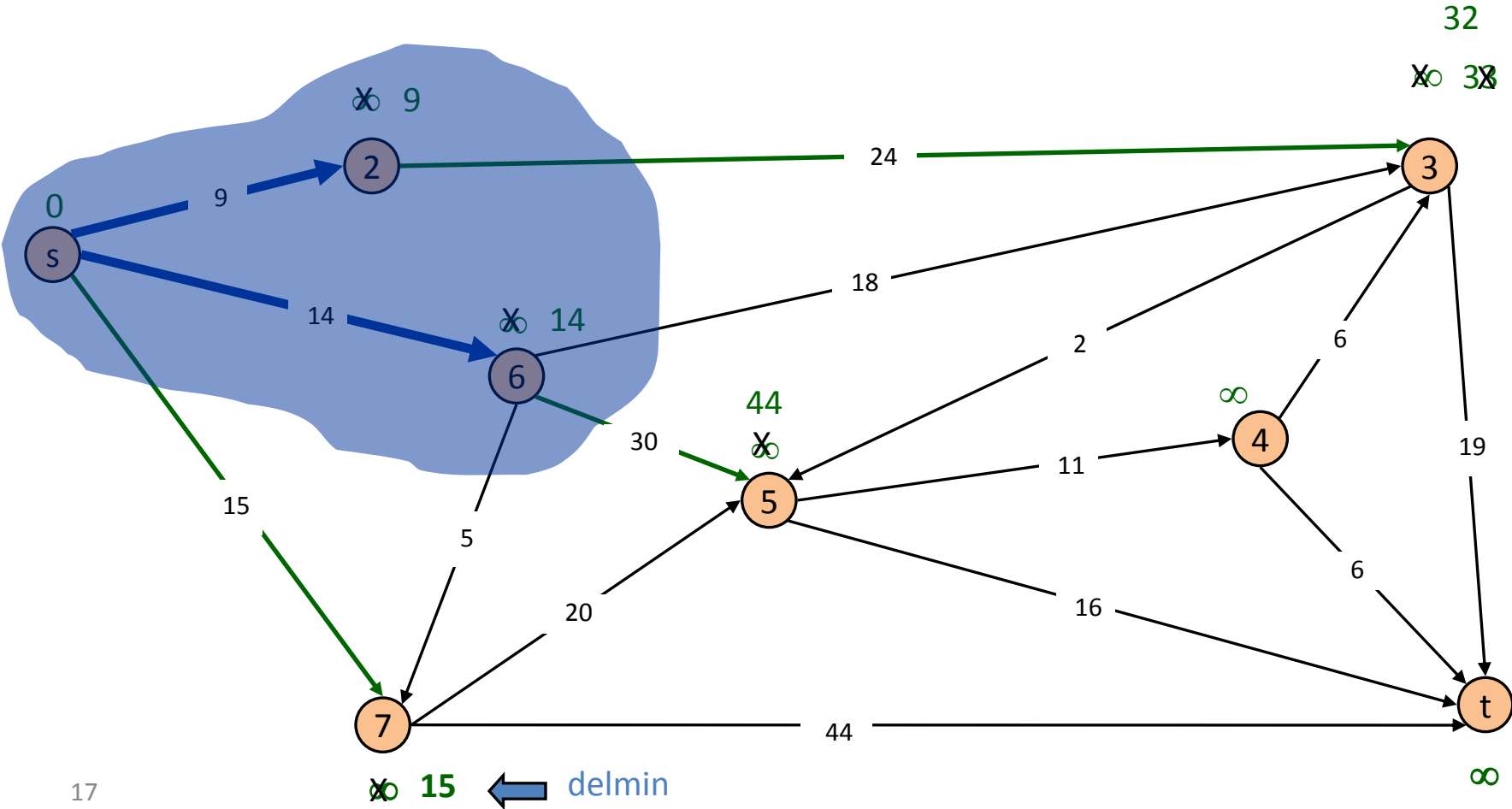
Dijkstra's Shortest Path Algorithm

$S = \{s, 2, 6\}$
 $PQ = \{3, 4, 5, 7, t\}$



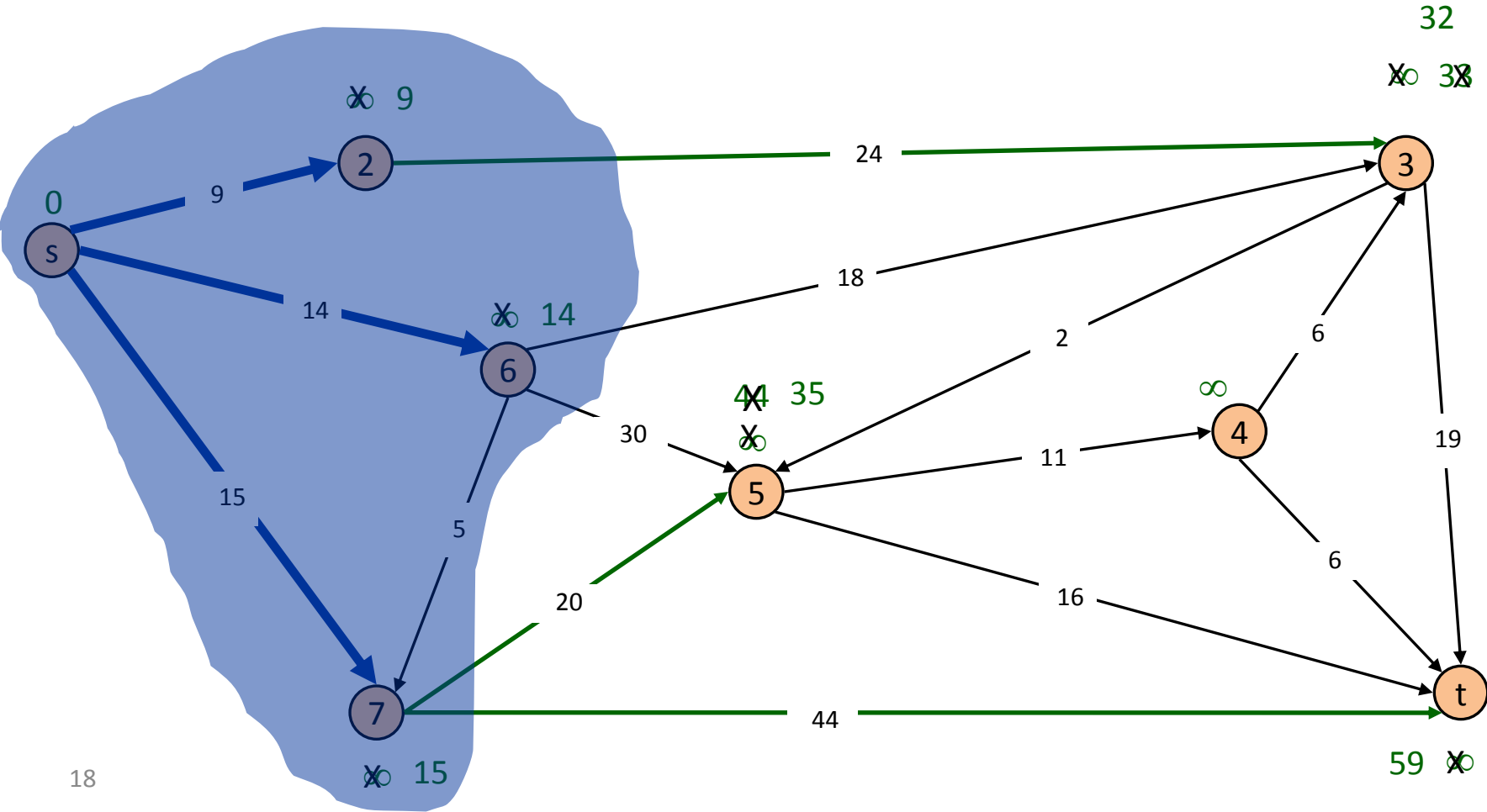
Dijkstra's Shortest Path Algorithm

$S = \{s, 2, 6\}$
 $PQ = \{3, 4, 5, 7, t\}$



Dijkstra's Shortest Path Algorithm

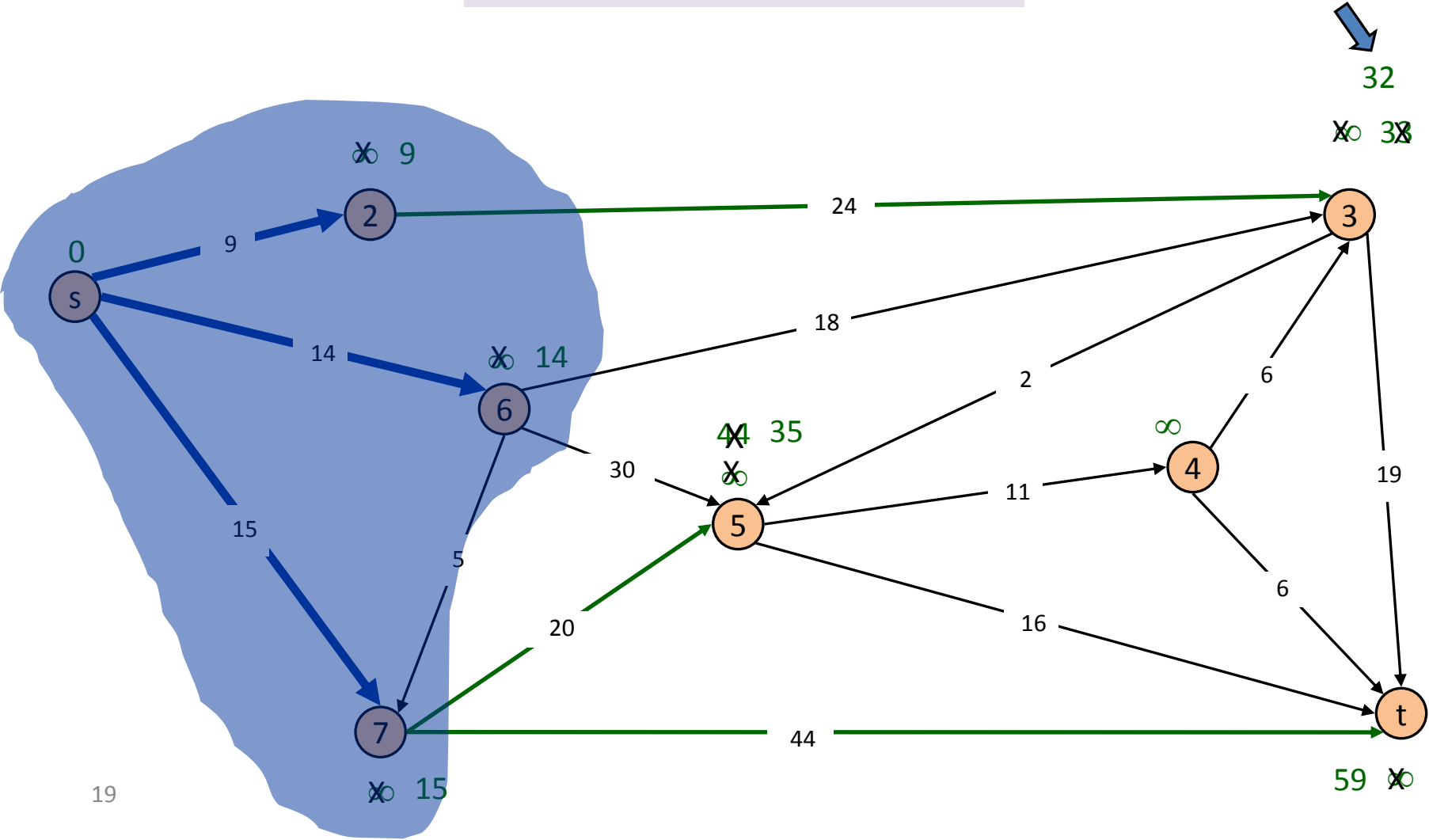
$S = \{s, 2, 6, 7\}$
 $PQ = \{3, 4, 5, t\}$



Dijkstra's Shortest Path Algorithm

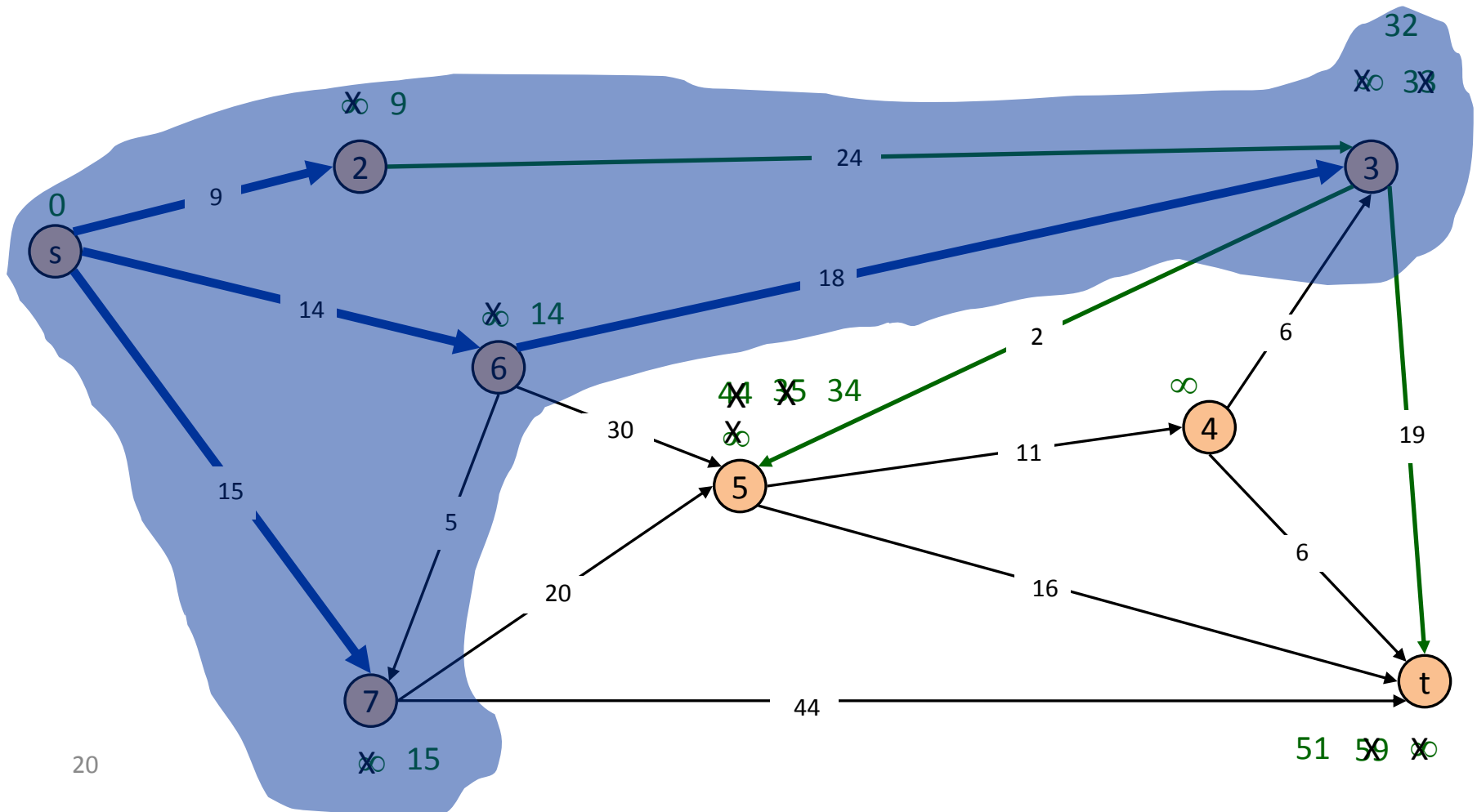
$S = \{s, 2, 6, 7\}$
 $PQ = \{3, 4, 5, t\}$

delmin



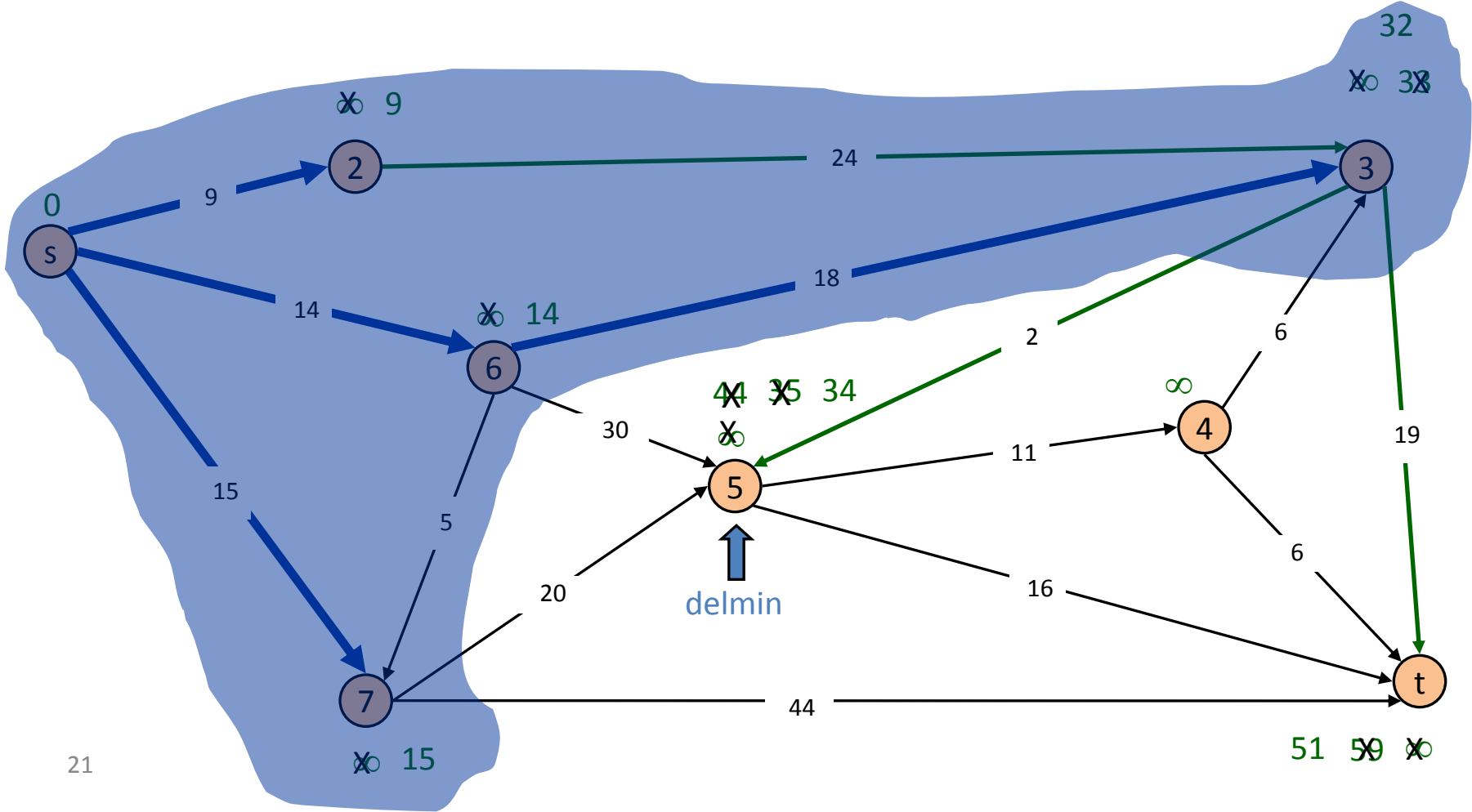
Dijkstra's Shortest Path Algorithm

$S = \{s, 2, 3, 6, 7\}$
 $PQ = \{4, 5, t\}$



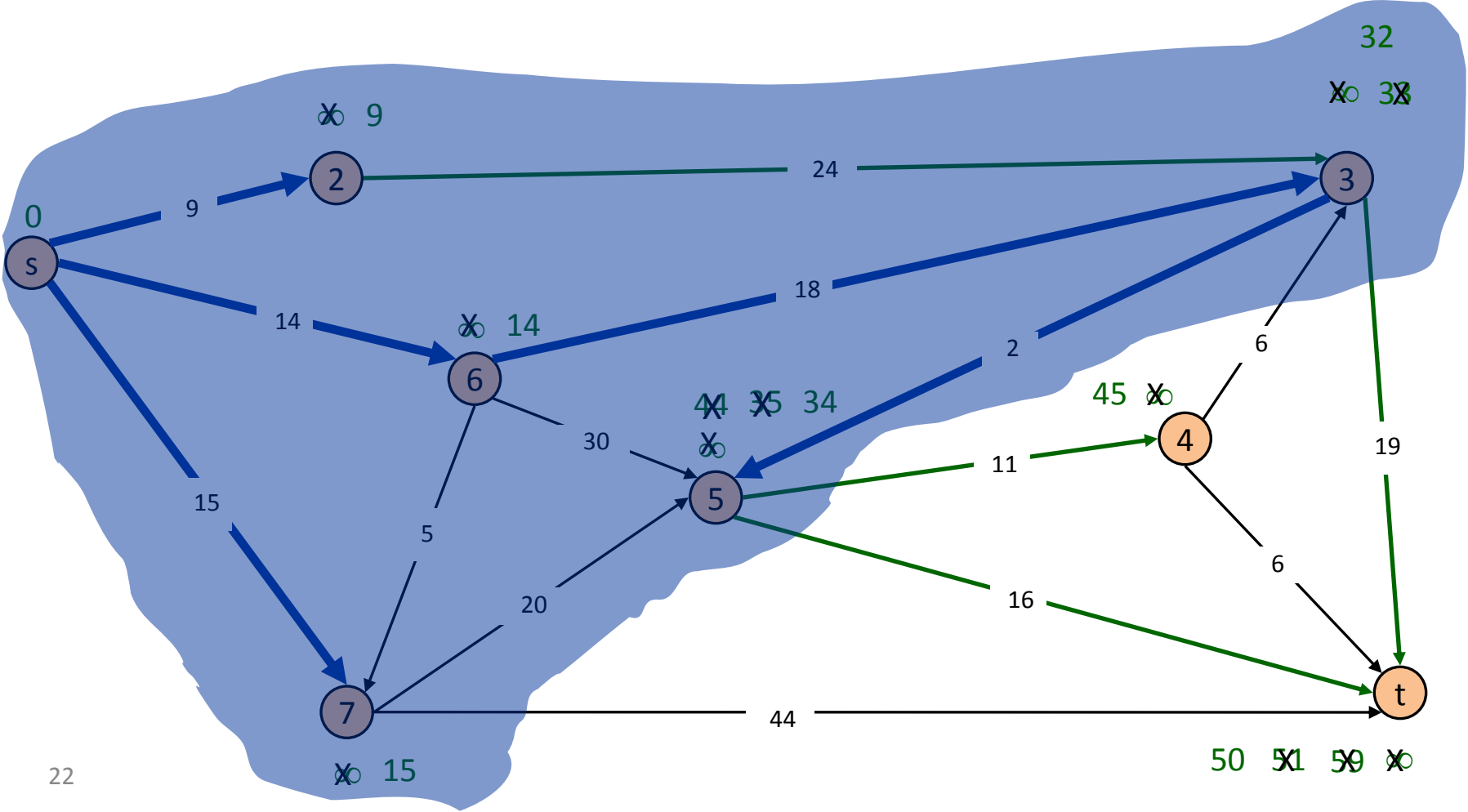
Dijkstra's Shortest Path Algorithm

$S = \{s, 2, 3, 6, 7\}$
 $PQ = \{4, 5, t\}$



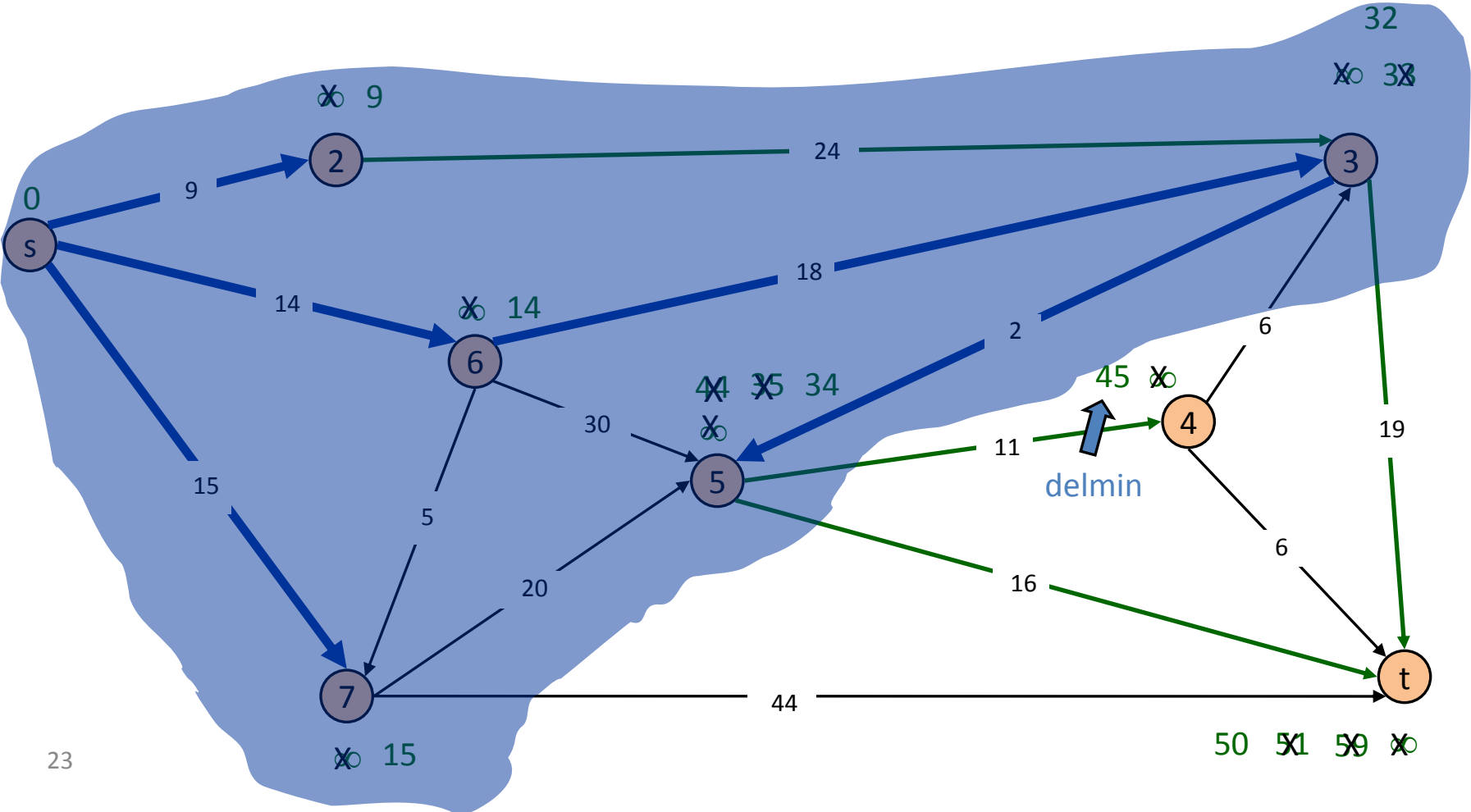
Dijkstra's Shortest Path Algorithm

$S = \{s, 2, 3, 5, 6, 7\}$
 $PQ = \{4, t\}$



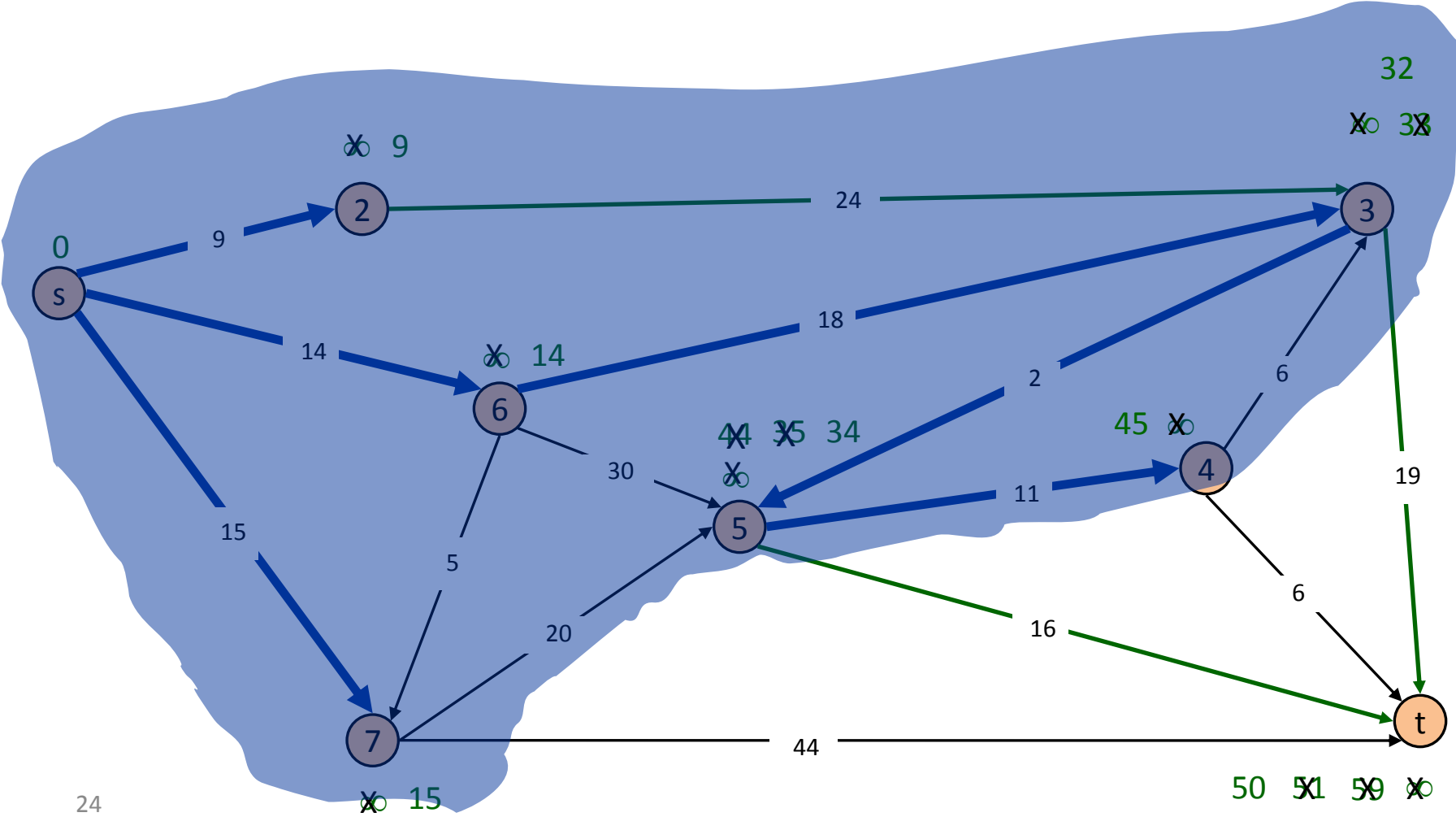
Dijkstra's Shortest Path Algorithm

$S = \{s, 2, 3, 5, 6, 7\}$
 $PQ = \{4, t\}$



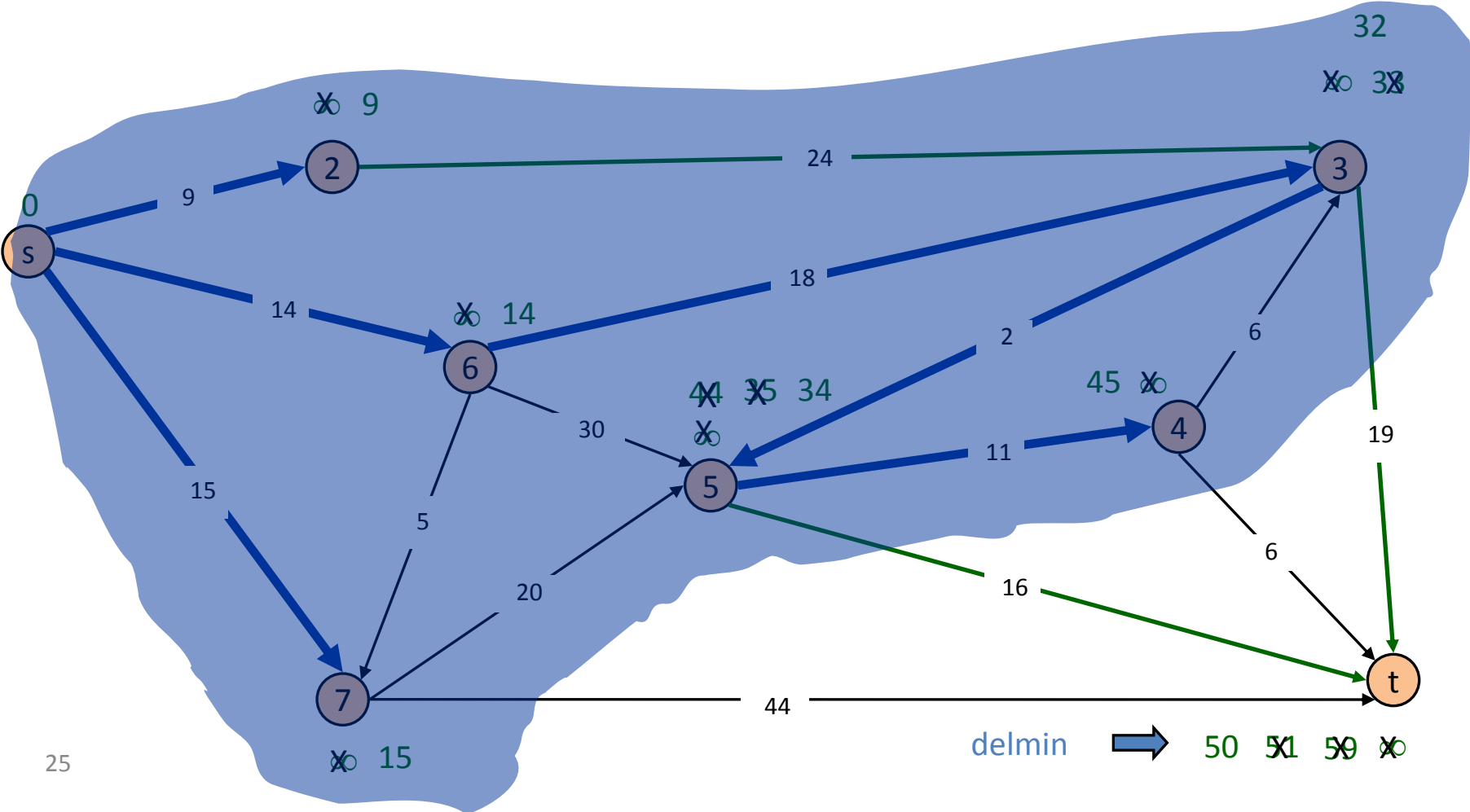
Dijkstra's Shortest Path Algorithm

$S = \{s, 2, 3, 4, 5, 6, 7\}$
 $PQ = \{t\}$



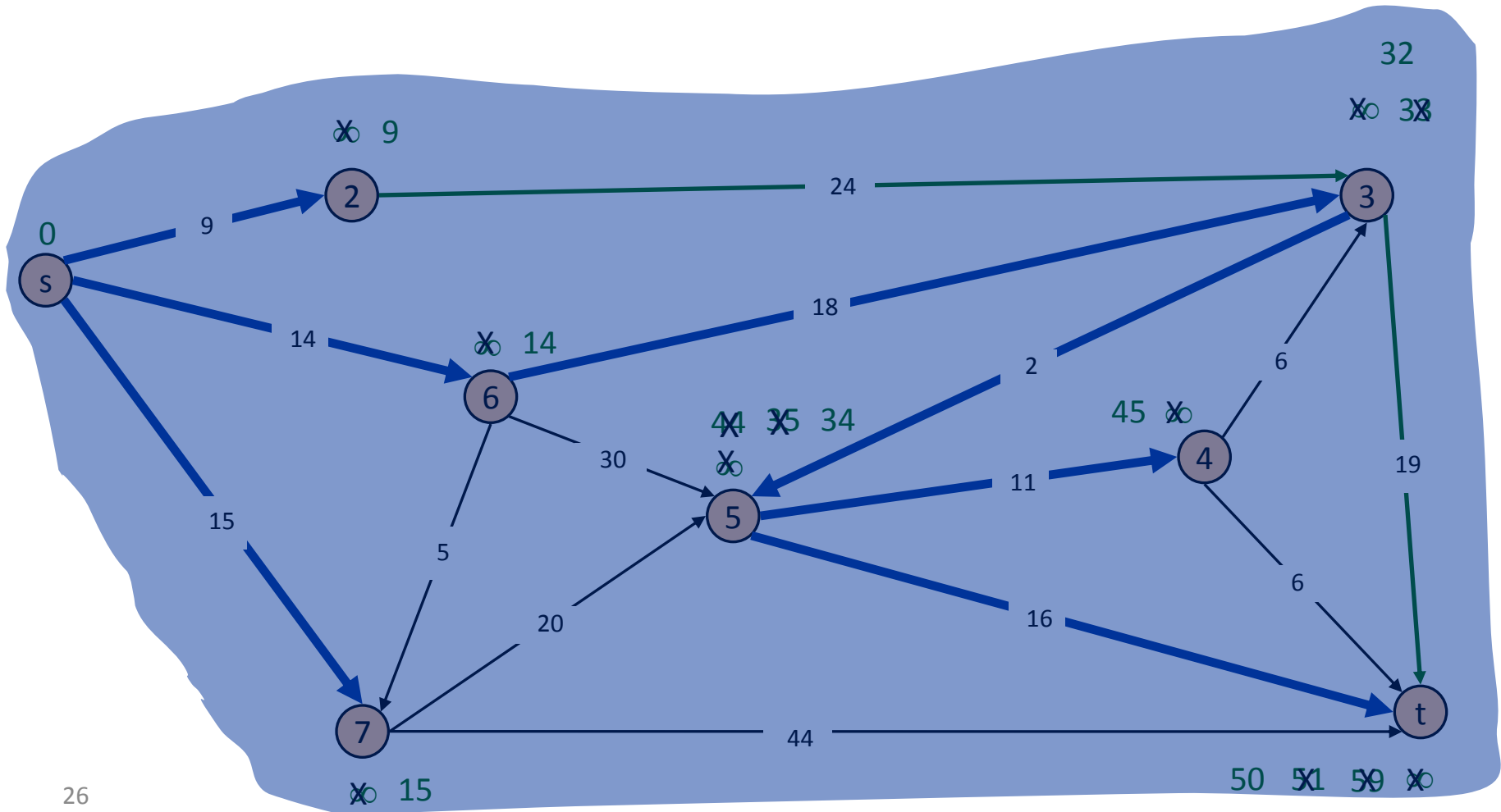
Dijkstra's Shortest Path Algorithm

$S = \{s, 2, 3, 4, 5, 6, 7\}$
 $PQ = \{t\}$



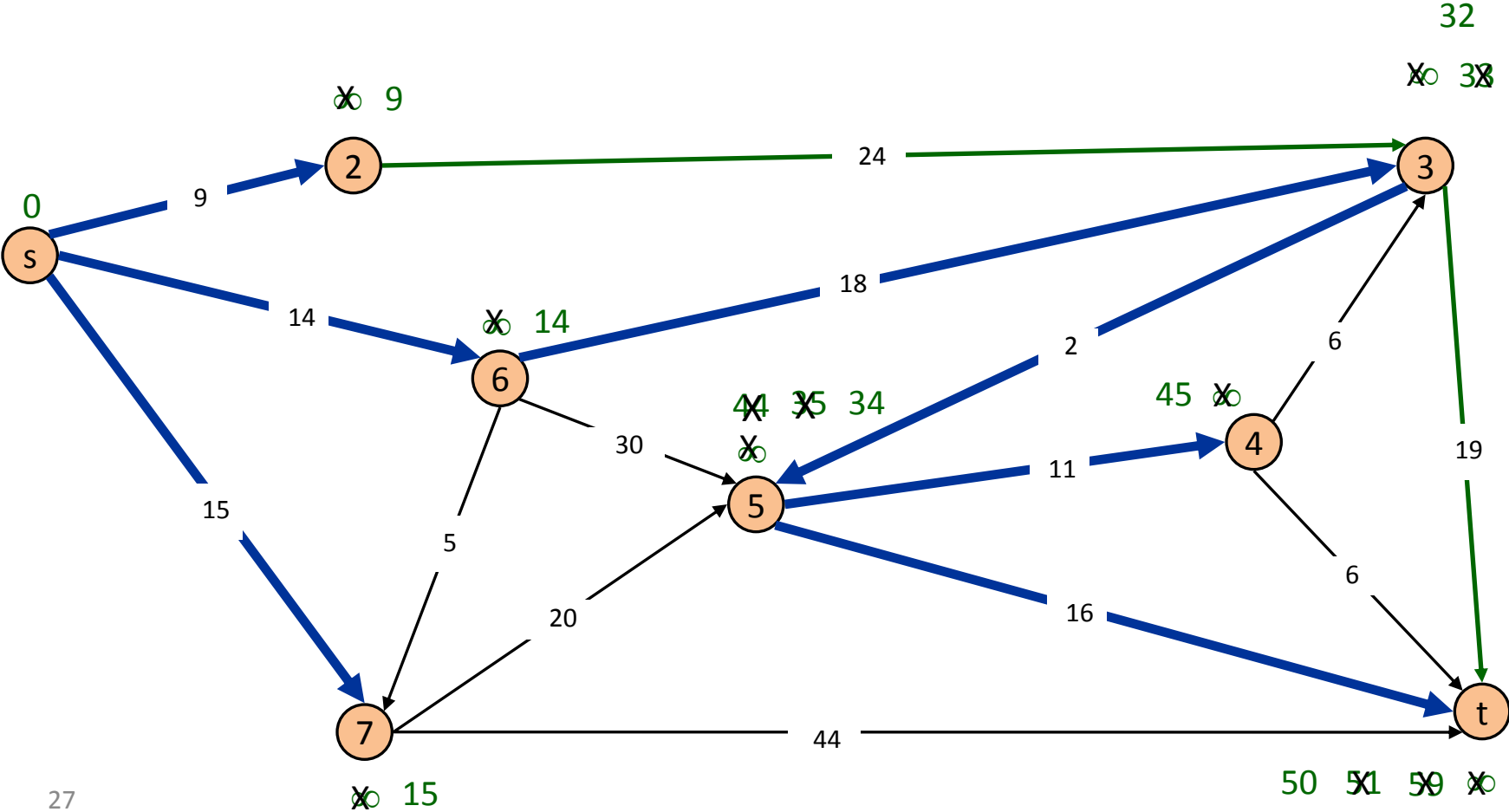
Dijkstra's Shortest Path Algorithm

$S = \{s, 2, 3, 4, 5, 6, 7, t\}$
 $PQ = \{\}$



Dijkstra's Shortest Path Algorithm

$S = \{s, 2, 3, 4, 5, 6, 7, t\}$
 $PQ = \{\}$



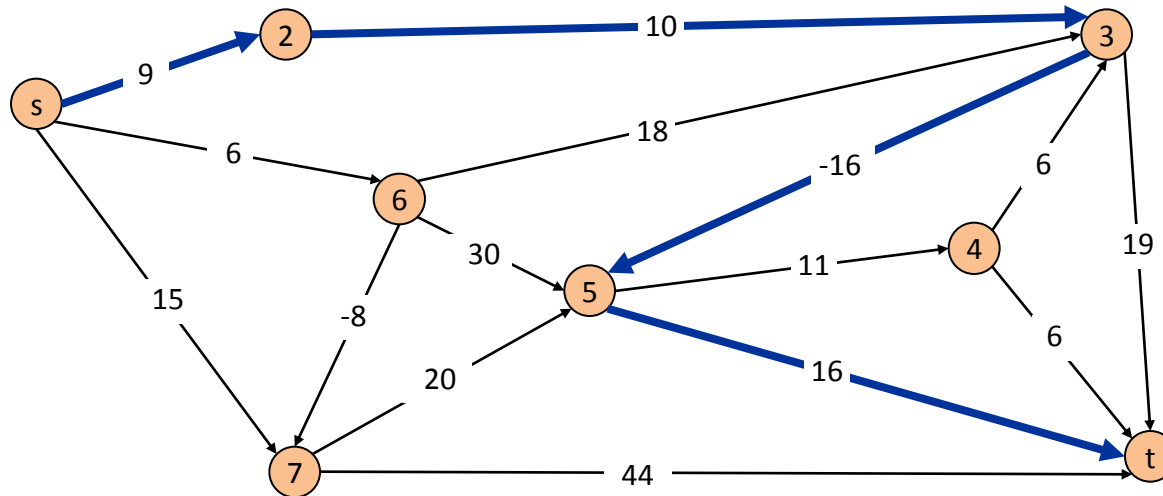
6.8 Shortest Paths: Bellman & Ford Algorithm Dynamic Programming

Shortest Paths

Shortest path problem. Given a directed graph $G = (V, E)$, with edge costs c_{vw} find shortest (min cost) path from node s to node t .

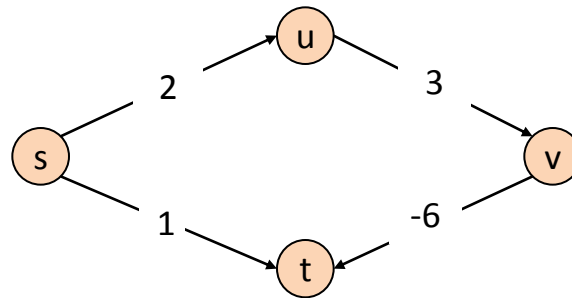
↙
allow negative costs

Application. Nodes represent agents in a financial setting and c_{vw} is cost of transaction in which we buy from agent v and sell immediately to w .

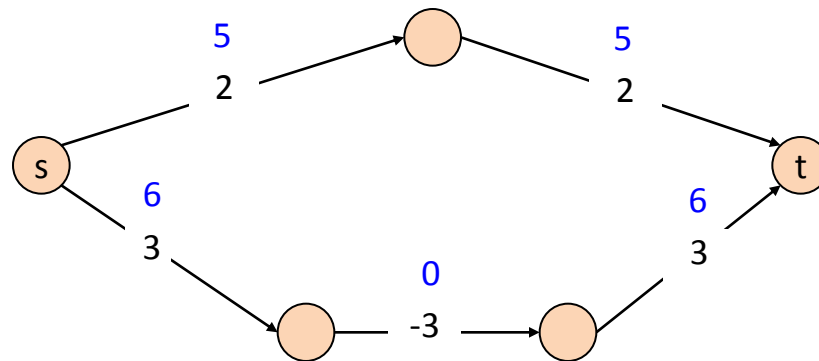


Shortest Paths: Failed Attempts

Dijkstra. Can fail if negative edge costs /weights.

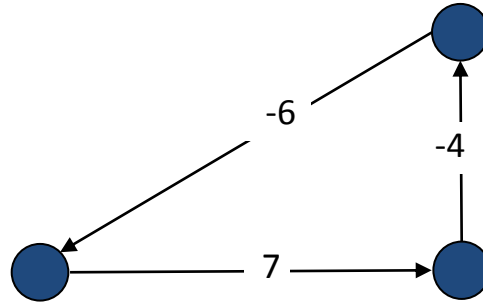


Re-weighting. Adding a constant to every edge cost can fail.



Shortest Paths: Negative Cost Cycles

Negative cost **cycle**.



Observation. If some path from s to t contains a negative cost cycle, there does not exist a shortest s - t path.

If no negative cost cycle, there exists a shortest s - t path that is **simple**.

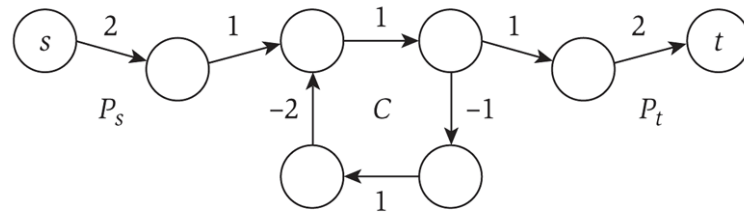
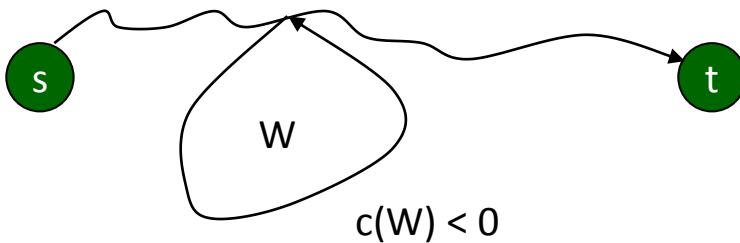


Figure 6.20 In this graph, one can find s - t paths of arbitrarily negative cost (by going around the cycle C many times).

Shortest Paths: Dynamic Programming

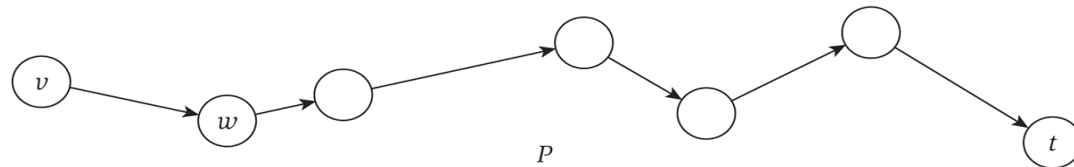
Def. $OPT(i, v)$ = cost of shortest v - t path P using at most i edges.

Case 1: P uses at most $i-1$ edges.

$$OPT(i, v) = OPT(i-1, v)$$

Case 2: P uses exactly i edges.

if (v, w) is first edge, then OPT uses (v, w) , and then selects best w - t path using at most $i-1$ edges



$$OPT(i, v) = \begin{cases} 0 & \text{se } v = t \\ \min \left\{ OPT(i-1, v), \min_{(v,w) \in E} \{ OPT(i-1, w) + c_{vw} \} \right\} & \text{altrimenti} \\ \infty & \text{se } i = 0, v \neq t \end{cases}$$

Remark. By previous observation, if no negative cycles, then $OPT(n-1, v)$ = cost of shortest v - t path.

Shortest Paths: Implementation

```
Shortest-Path(G, t) {  
    foreach node v ∈ V  
        M[0, v] ← ∞  
    M[0, t] ← 0  
  
    for i = 1 to n-1  
        foreach node v ∈ V  
            M[i, v] ← M[i-1, v]  
            foreach edge (v, w) ∈ E  
                M[i, v] ← min { M[i, v], M[i-1, w] + cvw }  
}
```

Analysis. $\Theta(n^2)$ space, $\Theta(mn)$ time

(la somma del numero di archi uscenti da v, per ogni v, è m)

Finding the shortest paths. Maintain a "successor" for each table entry.

Shortest Paths: Practical Improvements

(FACOLTATIVO)

Practical improvements.

Maintain only one array $M[v]$ = shortest v - t path that we have found so far.

No need to check edges of the form (v, w) unless $M[w]$ changed in previous iteration.

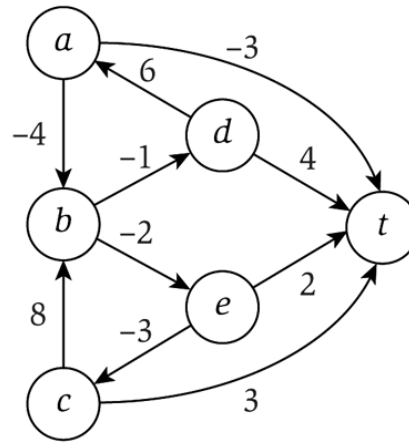
Theorem. Throughout the algorithm, $M[v]$ is length of some v - t path, and after i rounds of updates, the value $M[v]$ is no larger than the length of shortest v - t path using $\leq i$ edges.

Overall impact.

Memory: $O(m + n)$.

Running time: $O(mn)$ worst case, but substantially faster in practice.

Esempio



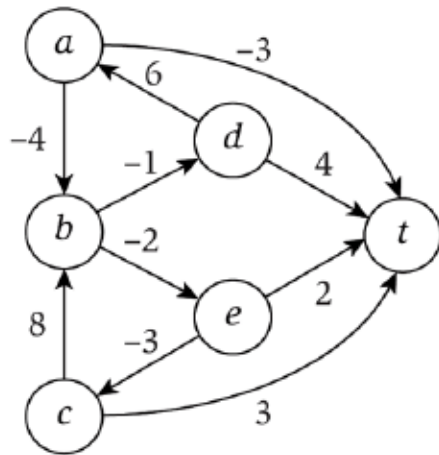
(a)

	0	1	2	3	4	5
<i>t</i>	0	0	0	0	0	0
<i>a</i>	∞	-3	-3	-4	-6	-6
<i>b</i>	∞	∞	0	-2	-2	-2
<i>c</i>	∞	3	3	3	3	3
<i>d</i>	∞	4	3	3	2	0
<i>e</i>	∞	2	0	0	0	0

(b)

Figure 6.23 For the directed graph in (a), the Shortest-Path Algorithm constructs the dynamic programming table in (b).

Shortest Paths: Esempio



(a)

	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4	-6	-6
b	∞	∞	0	-2	-2	-2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

(b)

$$OPT(i, v) = \begin{cases} 0 & \text{se } v = t \\ \min \left\{ OPT(i-1, v), \min_{(v,w) \in E} \{ OPT(i-1, w) + c_{vw} \} \right\} & \text{altrimenti} \\ \infty & \text{se } i = 0, v \neq t \end{cases}$$

$$OPT(5, a) = \min(OPT(4, a), \min(OPT(4, t) + c_{at}, OPT(4, b) + c_{ab})) \\ = \min(-6, \min(0 - 3, -2 - 4))$$

$$OPT(4, a) = \min(OPT(3, a), \min(OPT(3, t) + c_{at}, OPT(3, b) + c_{ab})) \quad \text{a-b} \\ = \min(-4, \min(0 - 3, -2 - 4))$$

$$OPT(3, b) = \min(OPT(2, b), \min(OPT(2, e) + c_{be}, OPT(2, d) + c_{bd})) \quad \text{b-e} \\ = \min(0, \min(0 - 2, 3 - 1))$$

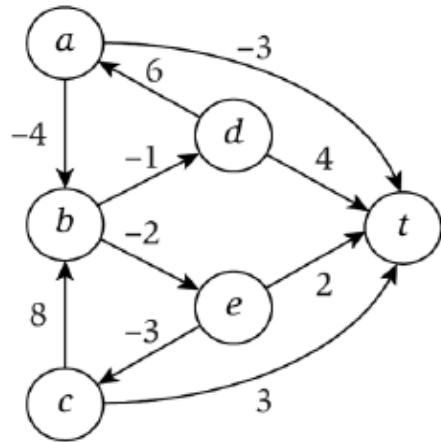
$$OPT(2, e) = \min(OPT(1, e), \min(OPT(1, c) + c_{ec}, OPT(1, t) + c_{et})) \quad \text{e-c} \\ = \min(2, \min(3 - 3, 0 + 2))$$

$$OPT(1, c) = \min(OPT(0, c), \min(OPT(0, b) + c_{cb}, OPT(0, t) + c_{ct})) \quad \text{c-t} \\ = \min(\infty, \min(\infty + 8, 0 + 3))$$

$$\text{a - b - e - c - t} \\ -4 \quad -2 \quad -3 \quad +3 \quad = -6$$

Figure 6.23 For the directed graph in (a), the Shortest-Path Algorithm constructs the dynamic programming table in (b).

Shortest Paths: Esempio



(a)

Trovare il cammino più corto.
Qual'è l'informazione che possiamo ricordare?

	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4	-6	-6
b	∞	∞	0	-2	-2	-2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

(b)

$$\begin{aligned} \text{OPT}(5,a) &= \min(\text{OPT}(4,a), \min(\text{OPT}(4,t)+c_{at}, \text{OPT}(4,b)+c_{ab})) \\ &= \min(-6, \min(0-3, -2-4)) \end{aligned}$$

$$\begin{aligned} \text{OPT}(4,a) &= \min(\text{OPT}(3,a), \min(\text{OPT}(3,t)+c_{at}, \text{OPT}(3,b)+c_{ab})) && \mathbf{a-b} \\ &= \min(-4, \min(0-3, -2-4)) \end{aligned}$$

$$\begin{aligned} \text{OPT}(3,b) &= \min(\text{OPT}(2,b), \min(\text{OPT}(2,e)+c_{be}, \text{OPT}(2,d)+c_{bd})) && \mathbf{b-e} \\ &= \min(0, \min(0-2, 3-1)) \end{aligned}$$

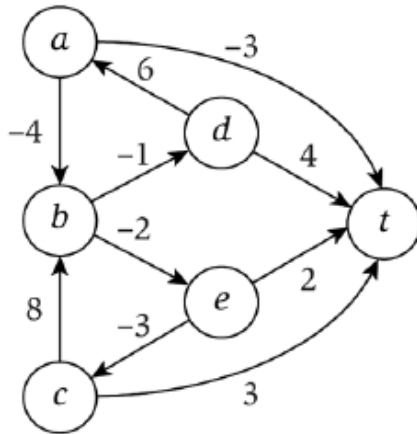
$$\begin{aligned} \text{OPT}(2,e) &= \min(\text{OPT}(1,e), \min(\text{OPT}(1,c)+c_{ec}, \text{OPT}(1,t)+c_{et})) && \mathbf{e-c} \\ &= \min(2, \min(3-3, 0+2)) \end{aligned}$$

$$\begin{aligned} \text{OPT}(1,c) &= \min(\text{OPT}(0,c), \min(\text{OPT}(0,b)+c_{cb}, \text{OPT}(0,t)+c_{ct})) && \mathbf{c-t} \\ &= \min(\infty, \min(\infty+8, 0+3)) \end{aligned}$$

$$\begin{aligned} \mathbf{a - b - e - c - t} \\ \mathbf{-4 \quad -2 \quad -3 \quad +3 \quad = -6} \end{aligned}$$

Figure 6.23 For the directed graph in (a), the Shortest-Path Algorithm constructs the dynamic programming table in (b).

Shortest Paths: Esempio



(a)

Trovare il cammino più corto.
Qual'è l'informazione che possiamo ricordare?

Mantenere un "successore" per ogni valore della tabella.

	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4	-6	-6
b	∞	∞	0	-2	-2	-2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

(b)

$$\begin{aligned} \text{OPT}(5,a) &= \min(\text{OPT}(4,a), \min(\text{OPT}(4,t)+c_{at}, \text{OPT}(4,b)+c_{ab})) \\ &= \min(-6, \min(0-3, -2-4)) \end{aligned}$$

$$\begin{aligned} \text{OPT}(4,a) &= \min(\text{OPT}(3,a), \min(\text{OPT}(3,t)+c_{at}, \text{OPT}(3,b)+c_{ab})) && \mathbf{a-b} \\ &= \min(-4, \min(0-3, -2-4)) \end{aligned}$$

$$\begin{aligned} \text{OPT}(3,b) &= \min(\text{OPT}(2,b), \min(\text{OPT}(2,e)+c_{be}, \text{OPT}(2,d)+c_{bd})) && \mathbf{b-e} \\ &= \min(0, \min(0-2, 3-1)) \end{aligned}$$

$$\begin{aligned} \text{OPT}(2,e) &= \min(\text{OPT}(1,e), \min(\text{OPT}(1,c)+c_{ec}, \text{OPT}(1,t)+c_{et})) && \mathbf{e-c} \\ &= \min(2, \min(3-3, 0+2)) \end{aligned}$$

$$\begin{aligned} \text{OPT}(1,c) &= \min(\text{OPT}(0,c), \min(\text{OPT}(0,b)+c_{cb}, \text{OPT}(0,t)+c_{ct})) && \mathbf{c-t} \\ &= \min(\infty, \min(\infty+8, 0+3)) \end{aligned}$$

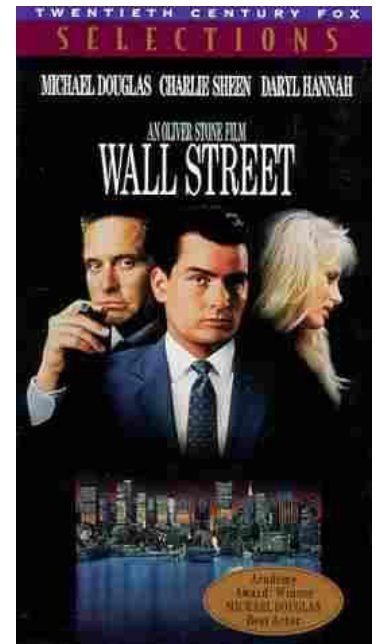
$$\begin{aligned} \mathbf{a - b - e - c - t} \\ -4 \quad -2 \quad -3 \quad +3 \quad = -6 \end{aligned}$$

Figure 6.23 For the directed graph in (a), the Shortest-Path Algorithm constructs the dynamic programming table in (b).

Coin Changing

Greed is good. Greed is right. Greed works.
Greed clarifies, cuts through, and captures the
essence of the evolutionary spirit.

- *Gordon Gecko (Michael Douglas)*



Coin Changing

Goal. Given currency denominations: 1, 5, 10, 25, 100, devise a method to pay amount to customer using fewest number of coin



Ex: 34¢.

Cashier's algorithm. At each iteration, add coin of the largest value that does not take us past the amount to be paid.



Ex: \$2.89.

Coin-Changing: Greedy Algorithm

Cashier's algorithm. At each iteration, add coin of the largest value that does not take us past the amount to be paid.

```
Sort coins denominations by value:  $c_1 < c_2 < \dots < c_n$ .
```

```
coins selected  
S ←  $\phi$   
while (x ≠ 0) {  
    let k be largest integer such that  $c_k \leq x$   
    if (k = 0)  
        return "no solution found"  
    x ← x -  $c_k$   
    S ← S ∪ {k}  
}  
return S
```

Q. Is cashier's algorithm optimal?

Coin-Changing: Analysis of Greedy Algorithm

Theorem. Greed is optimal for U.S. coinage: 1, 5, 10, 25, 100.
(penny, nickel, dime, quarter, dollar)

Pf. (by induction on x)

Consider optimal way to change $c_k \leq x < c_{k+1}$: greedy takes coin k .

We claim that any optimal solution must also take coin k .

if not, it needs enough coins of type c_1, \dots, c_{k-1} to add up to x

table below indicates no optimal solution can do this

Problem reduces to coin-changing $x - c_k$ cents, which, by induction, is optimally solved by greedy algorithm. ■

k	c_k	All optimal solutions must satisfy	Max value of coins 1, 2, ..., $k-1$ in any OPT
1	1	$P \leq 4$	-
2	5	$N \leq 1$	4
3	10	$N + D \leq 2$	$4 + 5 = 9$
4	25	$Q \leq 3$	$20 + 4 = 24$
5	100	no limit	$75 + 24 = 99$

Cambio monete: Analisi algoritmo greedy

Teorema. Greedy è ottimale per il conio U.S.A. : 1, 5, 10, 25, 100.

Prova. (induzione su x)

- Considera il modo ottimale per $c_k \leq x < c_{k+1}$: greedy sceglie moneta k .
- Ogni soluzione ottimale contiene la moneta k .
 - altrimenti, ci sarebbero monete di tipo c_1, \dots, c_{k-1} che sommano ad x
 - nessuna soluzione ottimale può farlo, come si vede dalla tabella

k	c_k	Tutte le soluzioni ottimali soddisfano	Massimo valore delle monete 1, 2, ..., $k-1$ in una sol. ottimale
1	1	penny ≤ 4	-
2	5	nickel ≤ 1	4 ←
3	10	nickel + dime ≤ 2	$4 + 5 = 9$
4	25	quarter ≤ 3	$20 + 4 = 24$
5	100	Senza limiti	$75 + 24 = 99$

$5 \leq x < 10$
Al massimo
• 4 penny

$5 \leq x < 10$
Se ci fossero 5 penny potrei migliorare con "1 nickel"

Cambio monete: Analisi algoritmo greedy

Teorema. Greedy è ottimale per il conio U.S.A. : 1, 5, 10, 25, 100.

Prova. (induzione su x)

- Considera il modo ottimale per $c_k \leq x < c_{k+1}$: greedy sceglie moneta k .
- Ogni soluzione ottimale contiene la moneta k .
 - altrimenti, ci sarebbero monete di tipo c_1, \dots, c_{k-1} che sommano ad x
 - nessuna soluzione ottimale può farlo, come si vede dalla tabella

k	c_k	Tutte le soluzioni ottimali soddisfano	Massimo valore delle monete 1, 2, ..., $k-1$ in una sol. ottimale
1	1	penny ≤ 4	-
2	5	nickel ≤ 1	4
3	10	nickel + dime ≤ 2	4 + 5 = 9 ←
4	25	quarter ≤ 3	20 + 4 = 24
5	100	Senza limiti	75 + 24 = 99

$10 \leq x < 25$
 Al massimo
 • 4 penny e
 • 1 nickel

$10 \leq x < 25$

Se ci fossero 5 penny potrei migliorare con "1 nickel"

Se ci fossero 2 nickel potrei migliorare con "1 dime"

Cambio monete: Analisi algoritmo greedy

Teorema. Greedy è ottimale per il conio U.S.A. : 1, 5, 10, 25, 100.

Prova. (induzione su x)

- Considera il modo ottimale per $c_k \leq x < c_{k+1}$: greedy sceglie moneta k .
- Ogni soluzione ottimale contiene la moneta k .
 - altrimenti, ci sarebbero monete di tipo c_1, \dots, c_{k-1} che sommano ad x
 - nessuna soluzione ottimale può farlo, come si vede dalla tabella

k	c_k	Tutte le soluzioni ottimali soddisfano	Massimo valore delle monete 1, 2, ..., $k-1$ in una sol. ottimale
1	1	penny ≤ 4	-
2	5	nickel ≤ 1	4
3	10	nickel + dime ≤ 2	$4 + 5 = 9$
4	25	quarter ≤ 3	$20 + 4 = 24$ ←
5	100	Senza limiti	$75 + 24 = 99$

$25 \leq x < 100$
 Al massimo
 • 4 penny e
 • 2 dime

$25 \leq x < 100$

Se ci fossero 3 dime potrei migliorare con "1 quarter + 1 nickel"

Se ci fossero 2 dime e 1 nickel potrei migliorare con "1 quarter"

Se ci fossero 1 dime e 1 nickel avrebbe valore inferiore a "2 dime"

nickel + dime ≤ 2

Cambio monete: Analisi algoritmo greedy

Teorema. Greedy è ottimale per il conio U.S.A. : 1, 5, 10, 25, 100.

Prova. (induzione su x)

- Considera il modo ottimale per $c_k \leq x < c_{k+1}$: greedy sceglie moneta k .
- Ogni soluzione ottimale contiene la moneta k .
 - altrimenti, ci sarebbero monete di tipo c_1, \dots, c_{k-1} che sommano ad x
 - nessuna soluzione ottimale può farlo, come si vede dalla tabella

k	c_k	Tutte le soluzioni ottimali soddisfano	Massimo valore delle monete 1, 2, ..., $k-1$ in una sol. ottimale
1	1	penny ≤ 4	-
2	5	nickel ≤ 1	4
3	10	nickel + dime ≤ 2	$4 + 5 = 9$
4	25	quarter ≤ 3	$20 + 4 = 24$
5	100	Senza limiti	$75 + 24 = 99 \leftarrow$

$100 \leq x$
Al massimo
• 3 quarter

$100 \leq x$
Se ci fossero 4 quarter potrei migliorare con "1 dollaro"

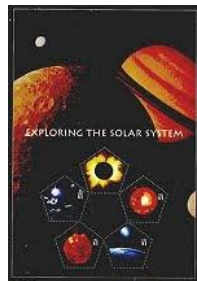
Coin-Changing: Analysis of Greedy Algorithm

Observation. Greedy algorithm is **sub-optimal** for US postal denominations: 1, 10, 21, 34, 70, 100, 350, 1225, 1500.

Counterexample. 140¢.

Greedy: 100, 34, 1, 1, 1, 1, 1, 1.

Optimal: 70, 70.



Cambio monete: Esercizio

Conio euro: 1, 2, 5, 10, 20, 50, 100, 200.



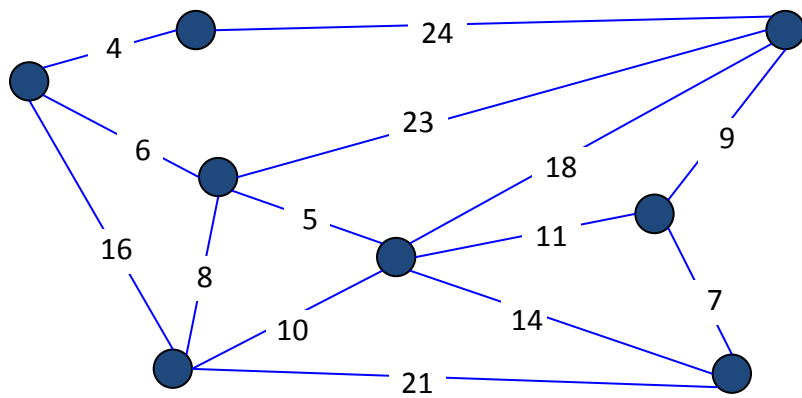
Greedy è ottimale?

4.5 Minimum Spanning Tree

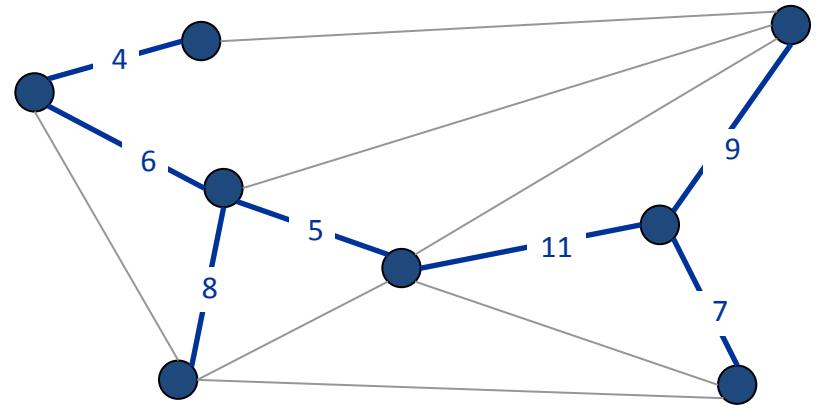
Minimo albero di copertura o ricoprimento

Minimum Spanning Tree (MST)

Minimum spanning tree. Given a connected graph $G = (V, E)$ with real-valued edge weights c_e , an **MST** is a subset of the edges $T \subseteq E$ such that (V, T) is a tree (connected and acyclic), denoted **spanning tree**, whose sum of edge weights is **minimized**.



$G = (V, E)$



$T, \sum_{e \in T} c_e = 50$

Recall: a tree with n nodes has $n-1$ edges.

Cayley's Theorem. There are n^{n-2} spanning trees of K_n : **can't solve by brute force!**

Applications

MST is fundamental problem with diverse applications.

Network design.

telephone, electrical, hydraulic, TV cable, computer, road

Approximation algorithms for NP-hard problems.

traveling salesperson problem, Steiner tree

Indirect applications.

max bottleneck paths

LDPC codes for error correction

image registration with Renyi entropy

learning salient features for real-time face verification

reducing data storage in sequencing amino acids in a protein

model locality of particle interactions in turbulent fluid flows

autoconfig protocol for Ethernet bridging to avoid cycles in a network

Cluster analysis.

Greedy Algorithms: possible choices

1. **Sort by increasing edge costs:** Start with some root node s and greedily grow a tree T from s outward. At each step, add the cheapest edge e to T that has exactly one endpoint in T .
2. **Sort by increasing edge costs:** Start with $T = \phi$. Consider edges in ascending order of cost. Insert edge e in T unless doing so would create a cycle.
3. **Sort by decreasing edge costs:** Start with $T = E$. Consider edges in descending order of cost. Delete edge e from T unless doing so would disconnect T .

Quale può funzionare?

Greedy Algorithms

All three algorithms produce an MST!!!

Prim's algorithm. Start with some root node s and greedily grow a tree T from s outward. At each step, add the cheapest edge e to T that has exactly one endpoint in T .

Kruskal's algorithm. Start with $T = \phi$. Consider edges in ascending order of cost. Insert edge e in T unless doing so would create a cycle.

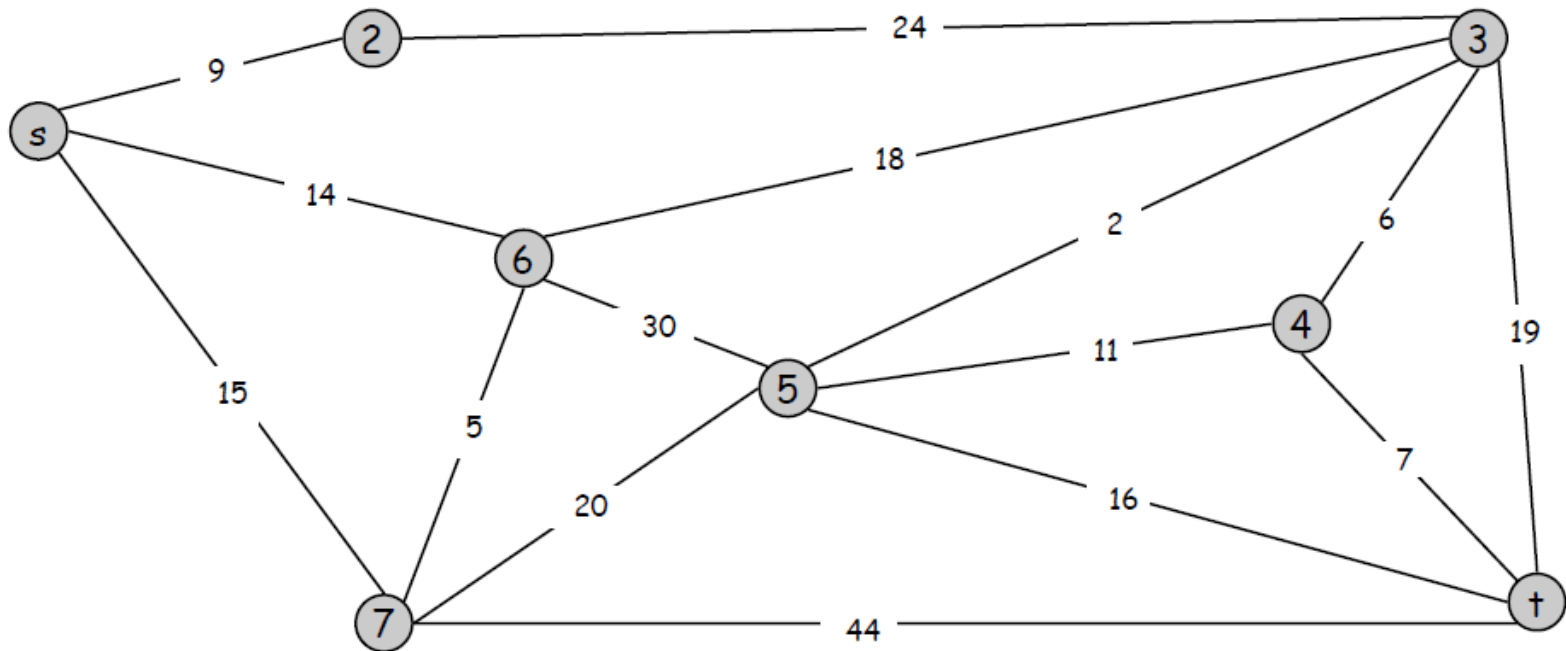
Reverse-Delete algorithm. Start with $T = E$. Consider edges in descending order of cost. Delete edge e from T unless doing so would disconnect T .

Algoritmo di Prim

Trovare Minimo Spanning Tree.

- Inizia con $T = \phi$ e con un nodo radice s .
- Aggiungi arco a T che è incidente solo su un nodo in T e con costo minimo.

$T = \phi$



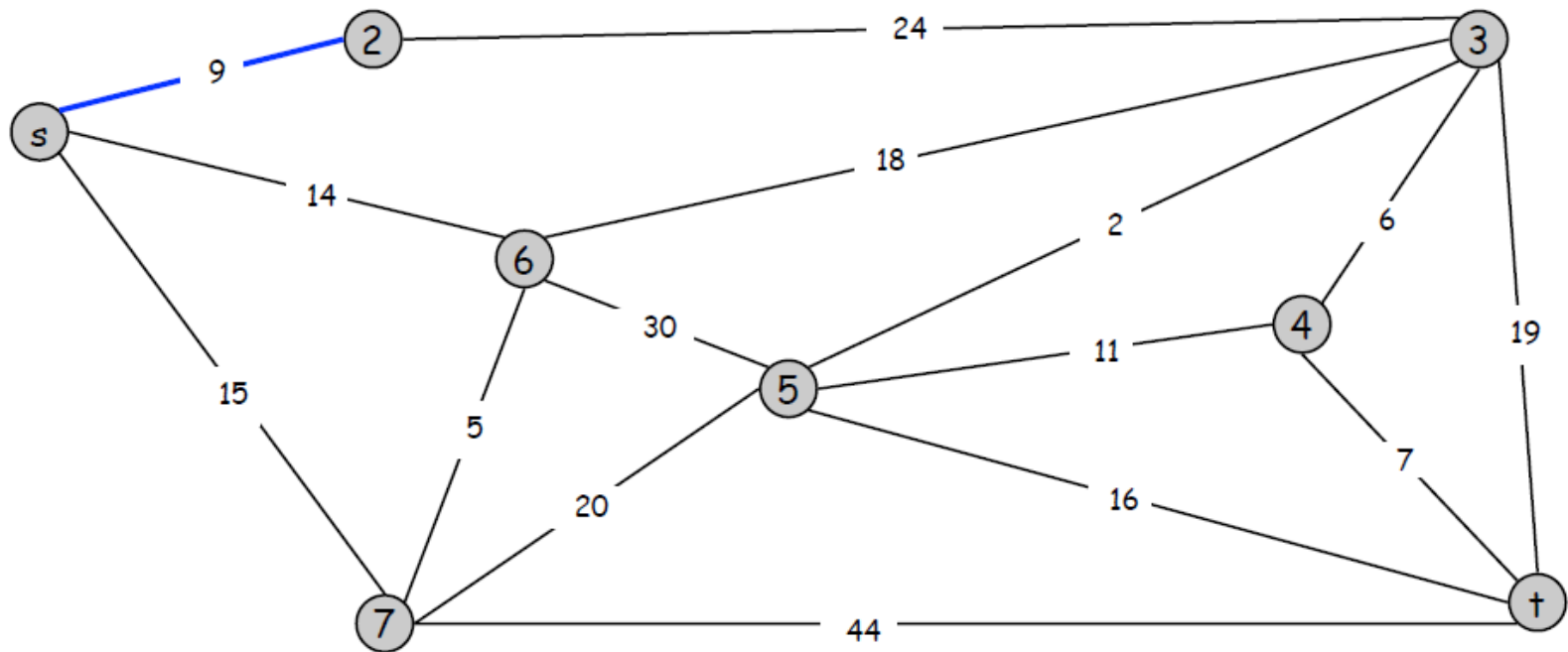
Nota: simile ad algoritmo di Dijkstra

Algoritmo di Prim

Trovare *Minimo Spanning Tree*.

- Inizia con $T = \phi$ e con un nodo radice s .
- Aggiungi arco a T che è incidente solo su un nodo in T e con costo minimo.

$T = \{ \{s, 2\} \}$

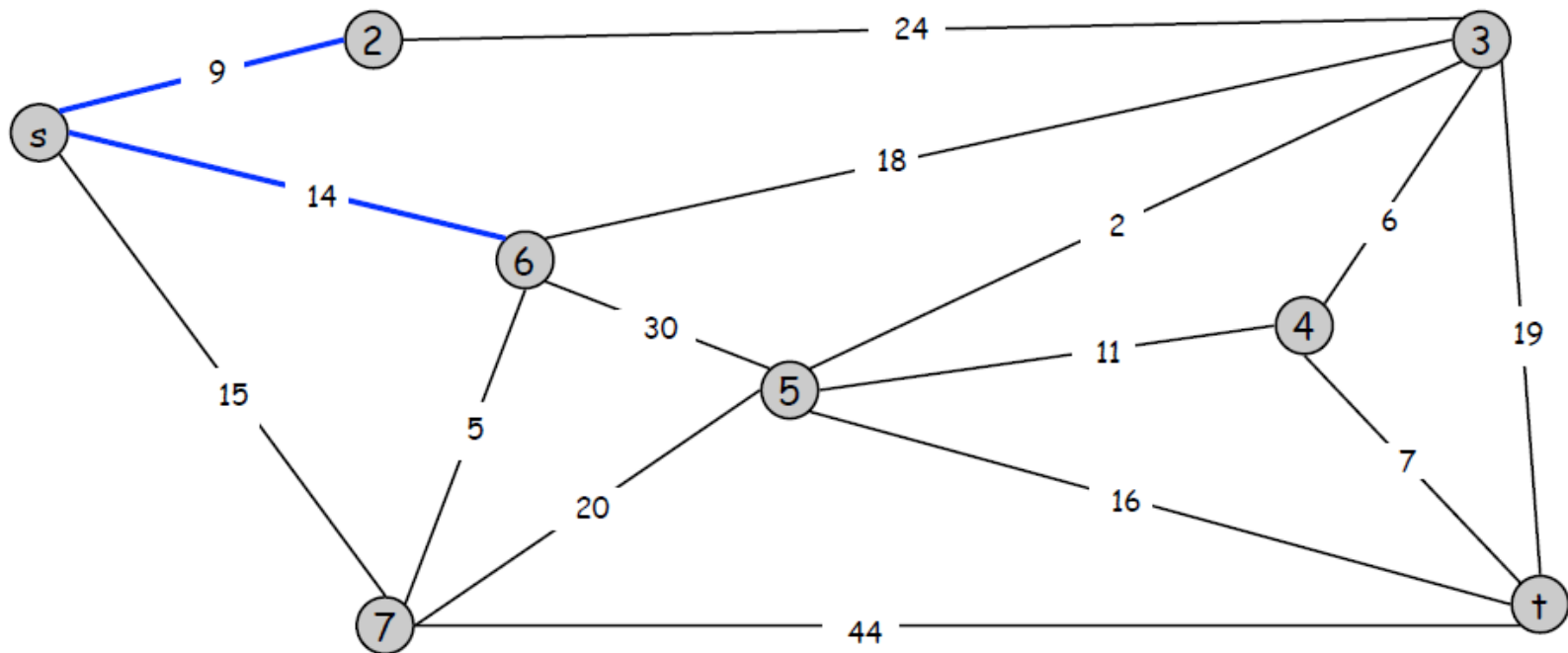


Algoritmo di Prim

Trovare Minimo Spanning Tree.

- Inizia con $T = \phi$ e con un nodo radice s .
- Aggiungi arco a T che è incidente solo su un nodo in T e con costo minimo.

$$T = \{ \{s,2\}, \{s,6\} \}$$

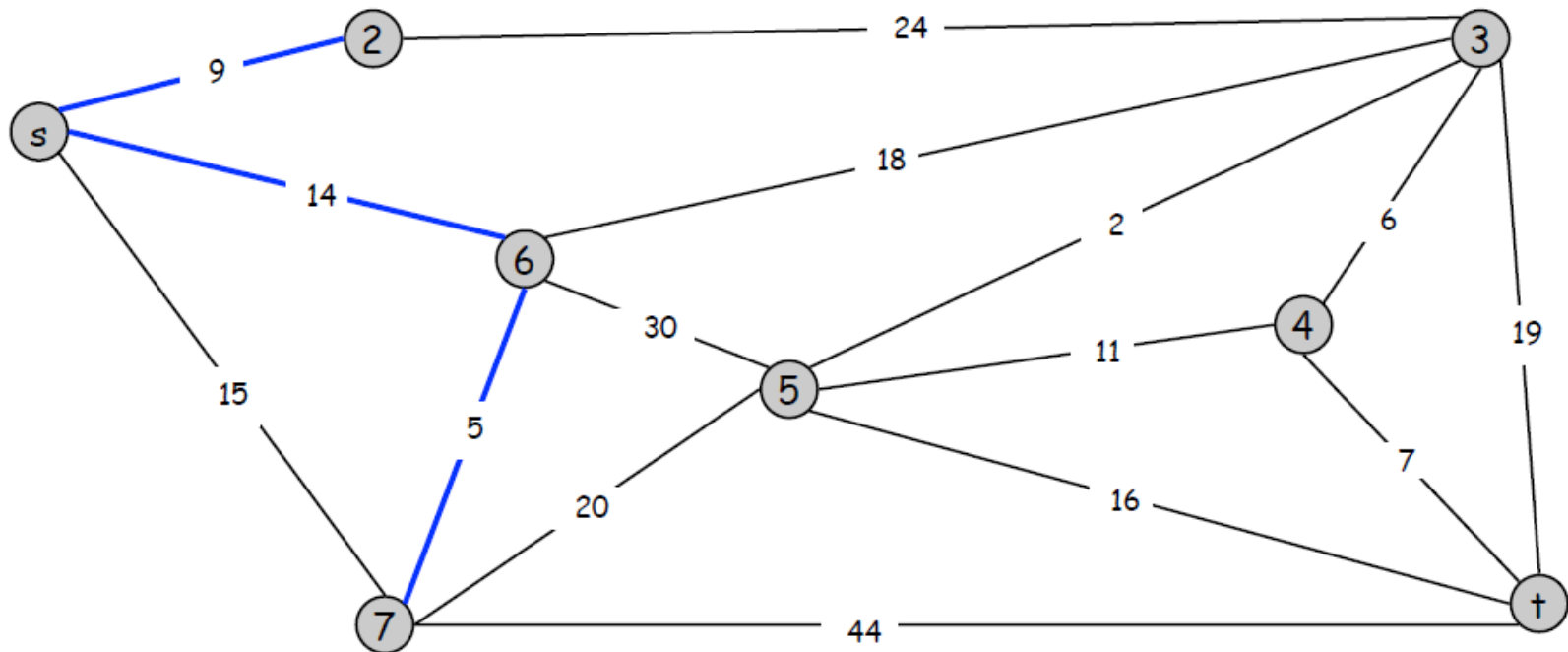


Algoritmo di Prim

Trovare Minimo Spanning Tree.

- Inizia con $T = \phi$ e con un nodo radice s .
- Aggiungo arco a T che è incidente solo su un nodo in T e con costo minimo.

$T = \{ \{s,2\}, \{s,6\}, \{6,7\} \}$

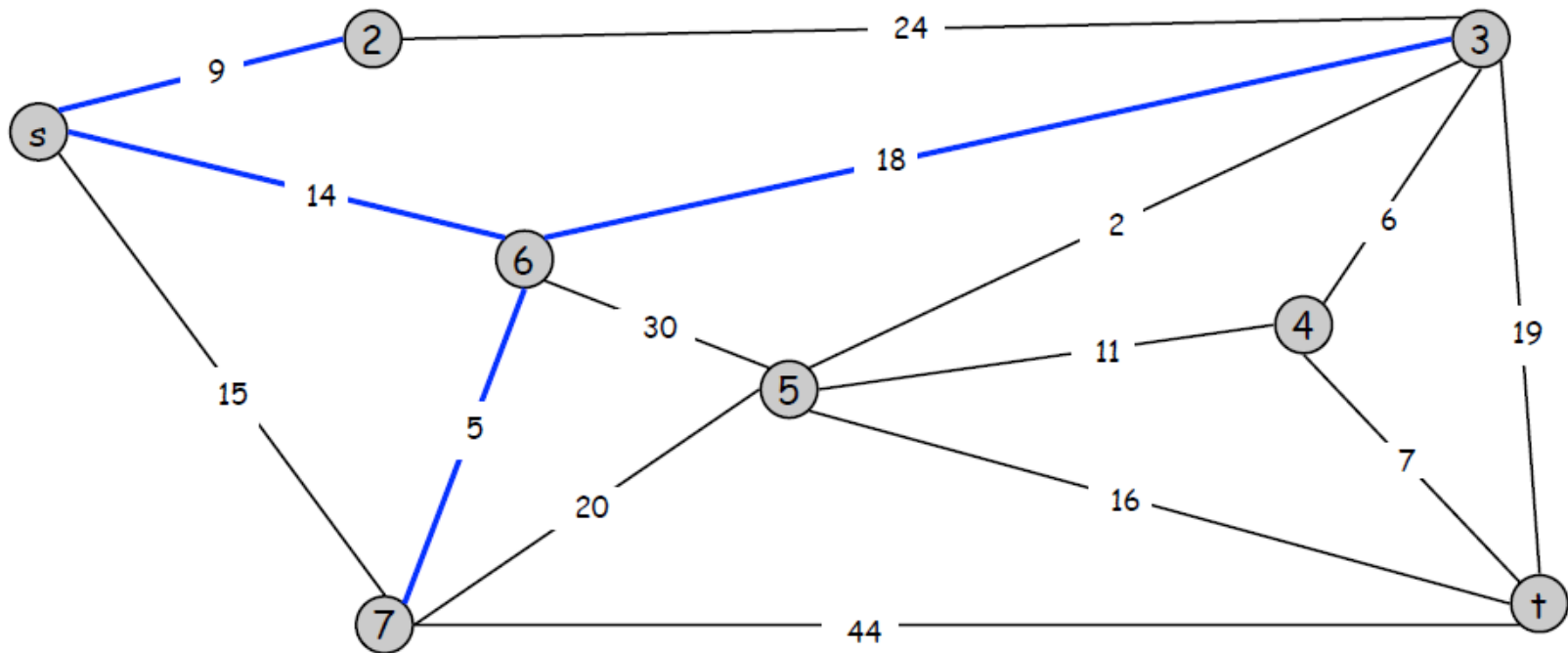


Algoritmo di Prim

Trovare Minimo Spanning Tree.

- Inizia con $T = \phi$ e con un nodo radice s .
- Aggiungi arco a T che è incidente solo su un nodo in T e con costo minimo.

$T = \{ \{s,2\}, \{s,6\}, \{6,7\}, \{6,3\} \}$

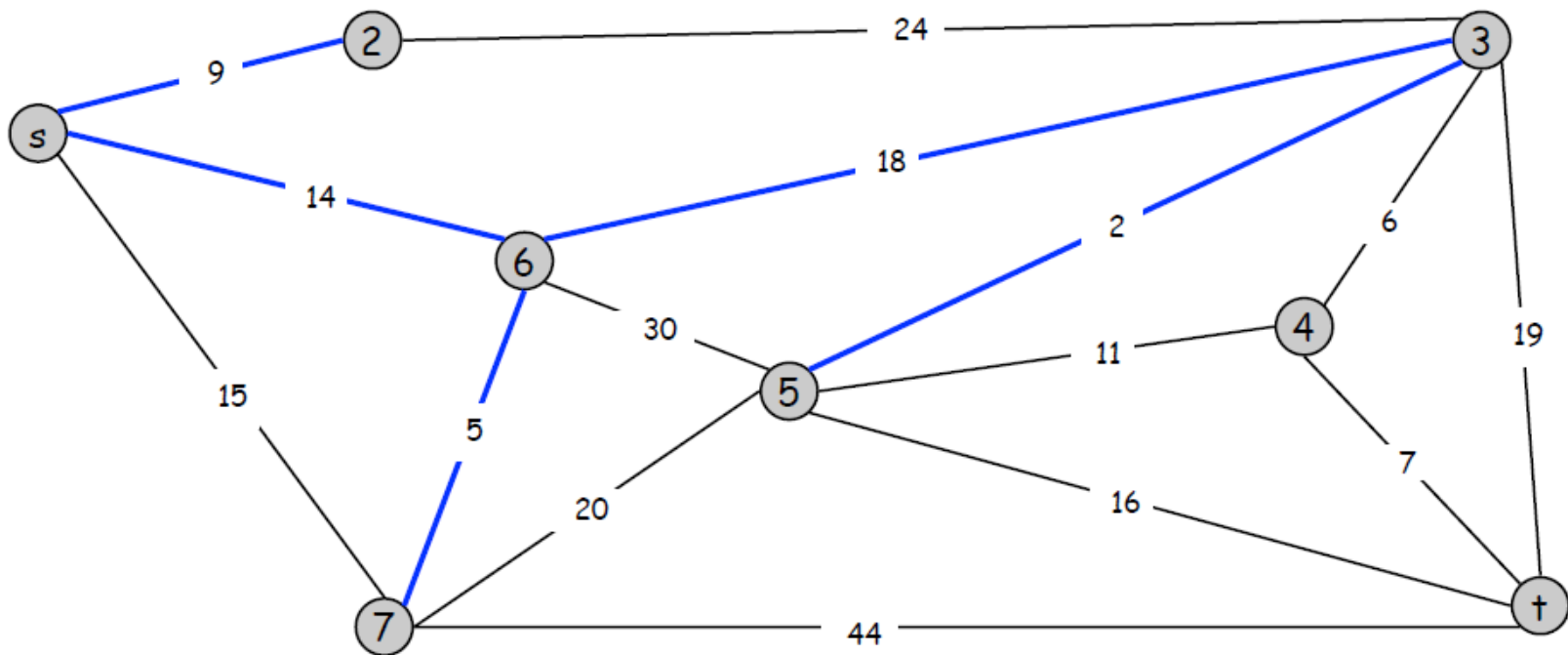


Algoritmo di Prim

Trovare Minimo Spanning Tree.

- Inizia con $T = \phi$ e con un nodo radice s .
- Aggiungi arco a T che è incidente solo su un nodo in T e con costo minimo.

$T = \{ \{s,2\}, \{s,6\}, \{6,7\}, \{6,3\}, \{3,5\} \}$

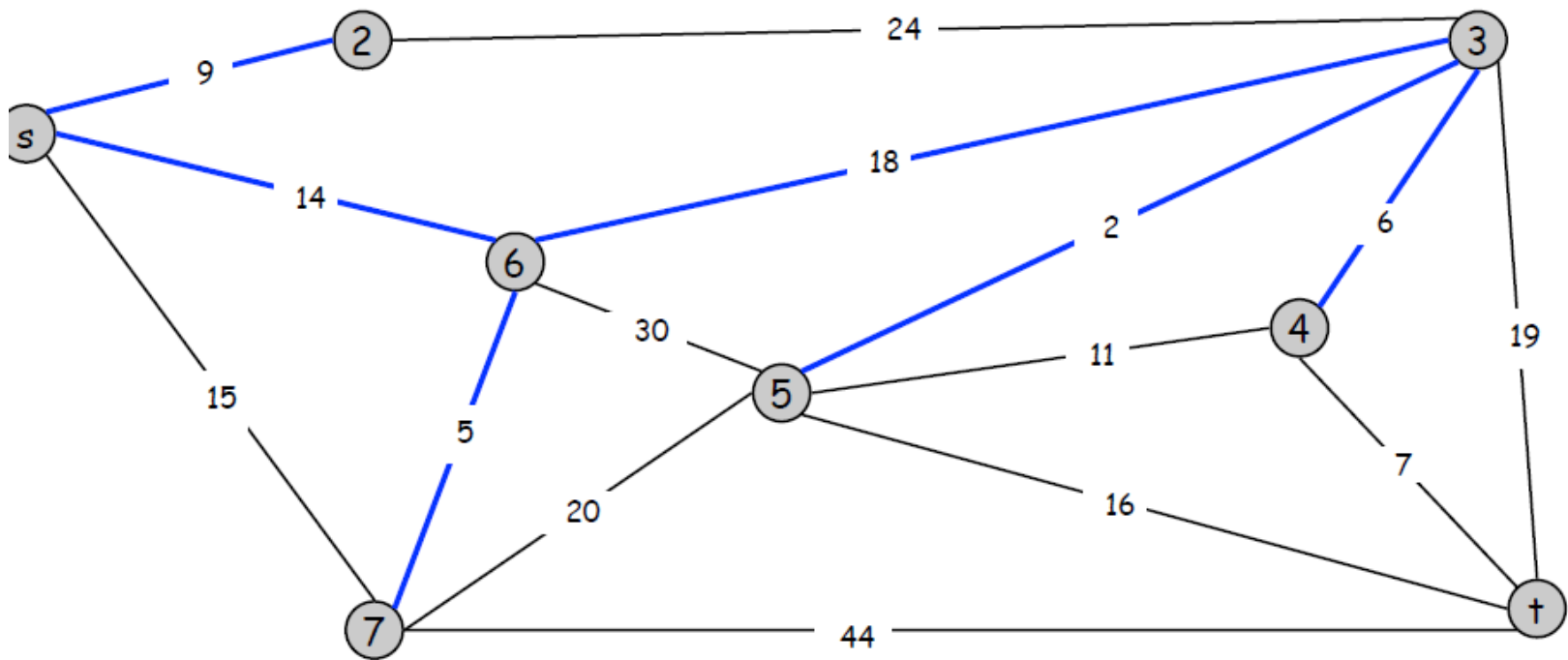


Algoritmo di Prim

Trovare Minimo Spanning Tree.

- Inizia con $T = \phi$ e con un nodo radice s .
- Aggiungi arco a T che è incidente solo su un nodo in T e con costo minimo.

$T = \{ \{s,2\}, \{s,6\}, \{6,7\}, \{6,3\}, \{3,5\}, \{3,4\} \}$

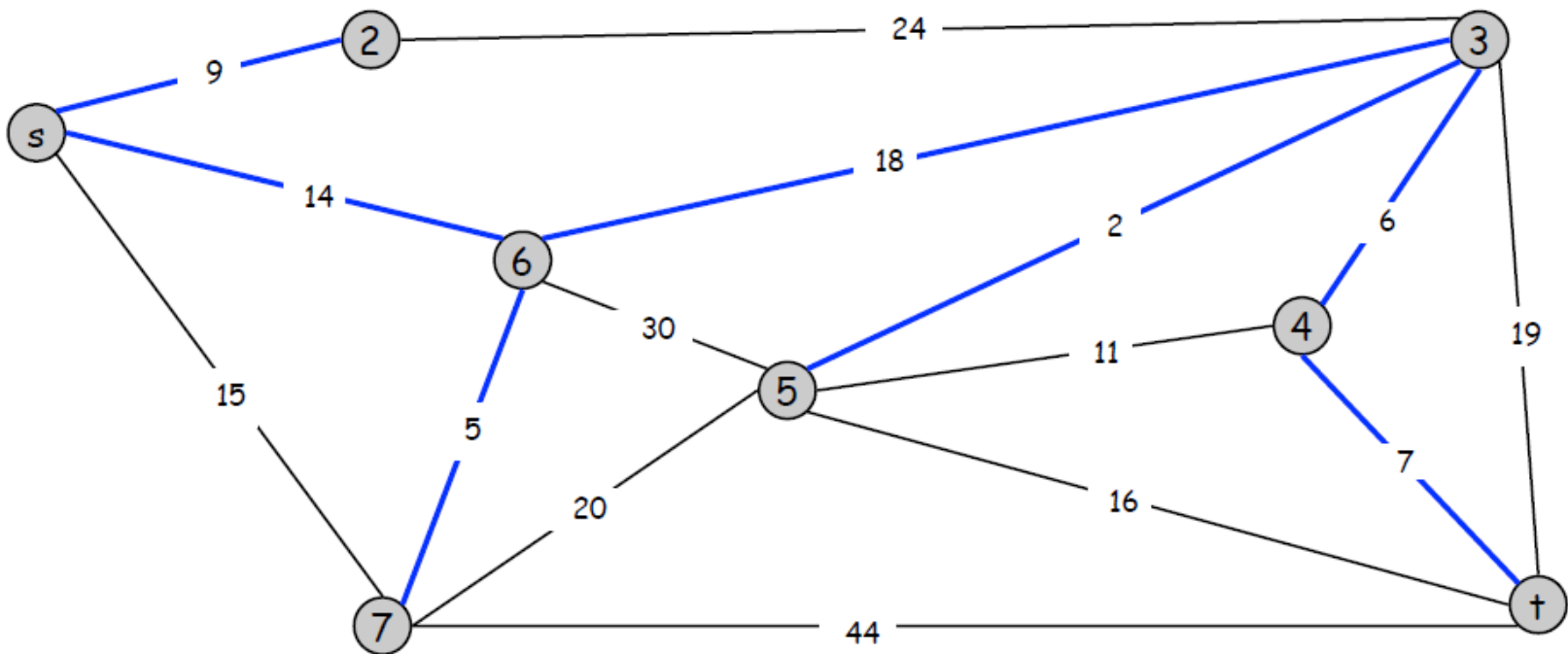


Algoritmo di Prim

Trovare Minimo Spanning Tree.

- Inizia con $T = \phi$ e con un nodo radice s .
- Aggiungi arco a T che è incidente solo su un nodo in T e con costo minimo.

$T = \{ \{s,2\}, \{s,6\}, \{6,7\}, \{6,3\}, \{3,5\}, \{3,4\}, \{4,t\} \}$ è MST di costo $9+14+5+18+2+6+7= 61$

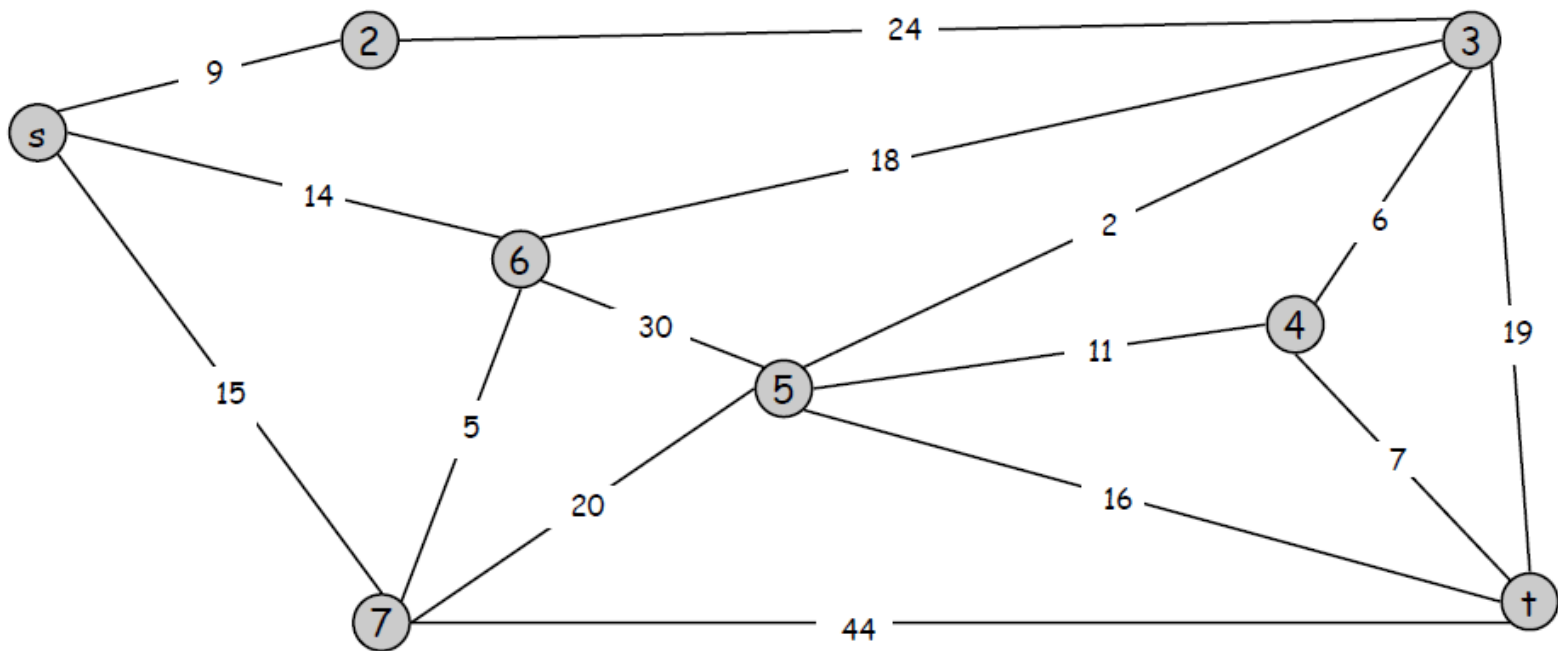


Algoritmo di Kruskal

Trovare Minimo Spanning Tree.

- Inizia con $T = \phi$.
- Considera archi in ordine crescente di costo.
- Inserisci l'arco in T se non crea un ciclo.

$T = \phi$

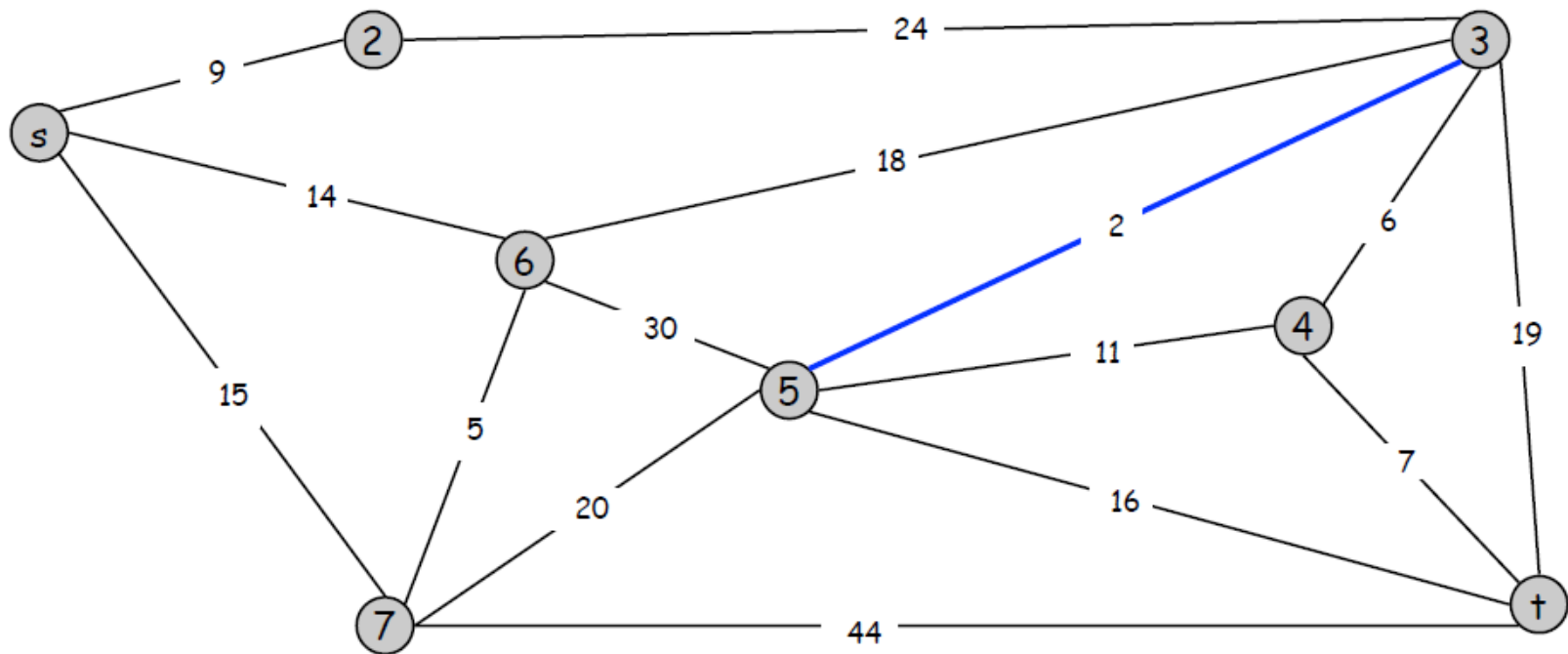


Algoritmo di Kruskal

Trovare *Minimo Spanning Tree*.

- Inizia con $T = \phi$.
- Considera archi in ordine crescente di costo.
- Inserisci l'arco in T se non crea un ciclo.

$T = \{ \{s,2\} \}$

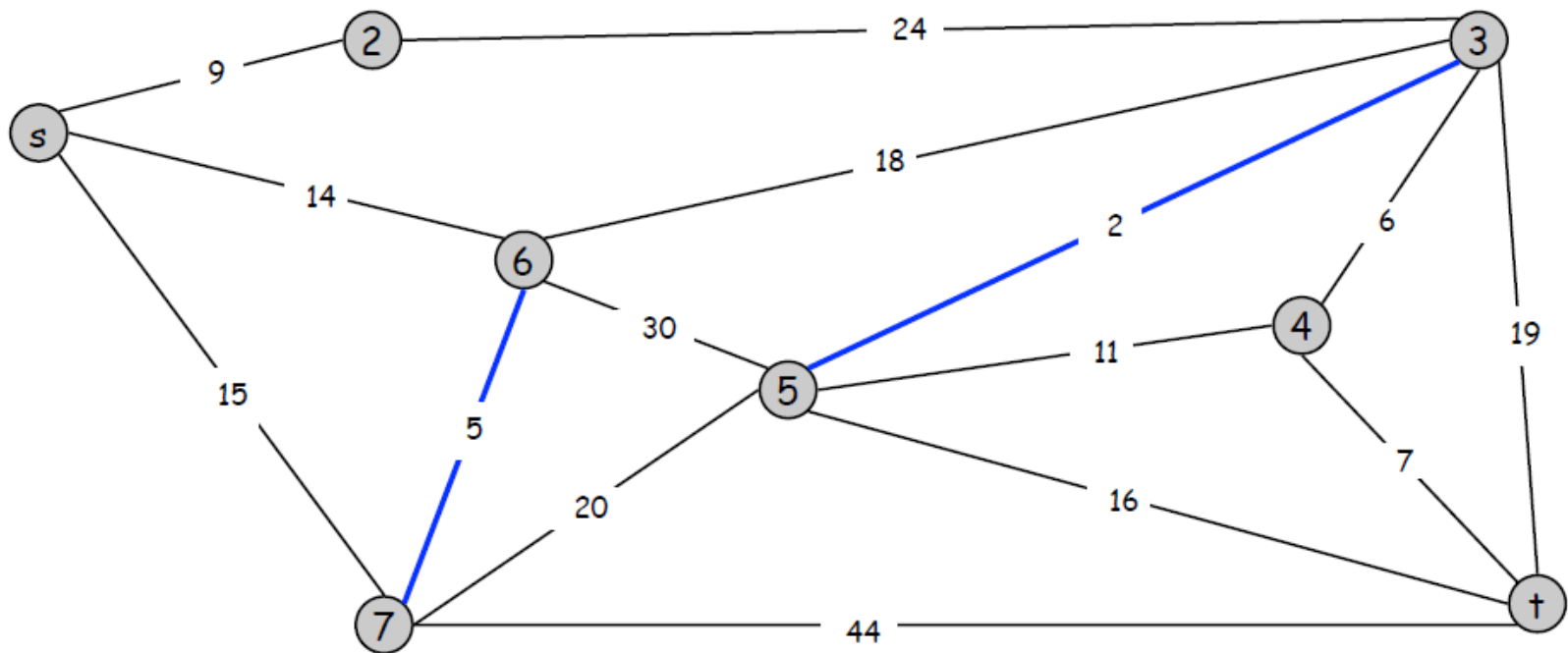


Algoritmo di Kruskal

Trovare Minimo Spanning Tree.

- Inizia con $T = \phi$.
- Considera archi in ordine crescente di costo.
- Inserisci l'arco in T se non crea un ciclo.

$T = \{ \{s,2\}, \{s,5\} \}$

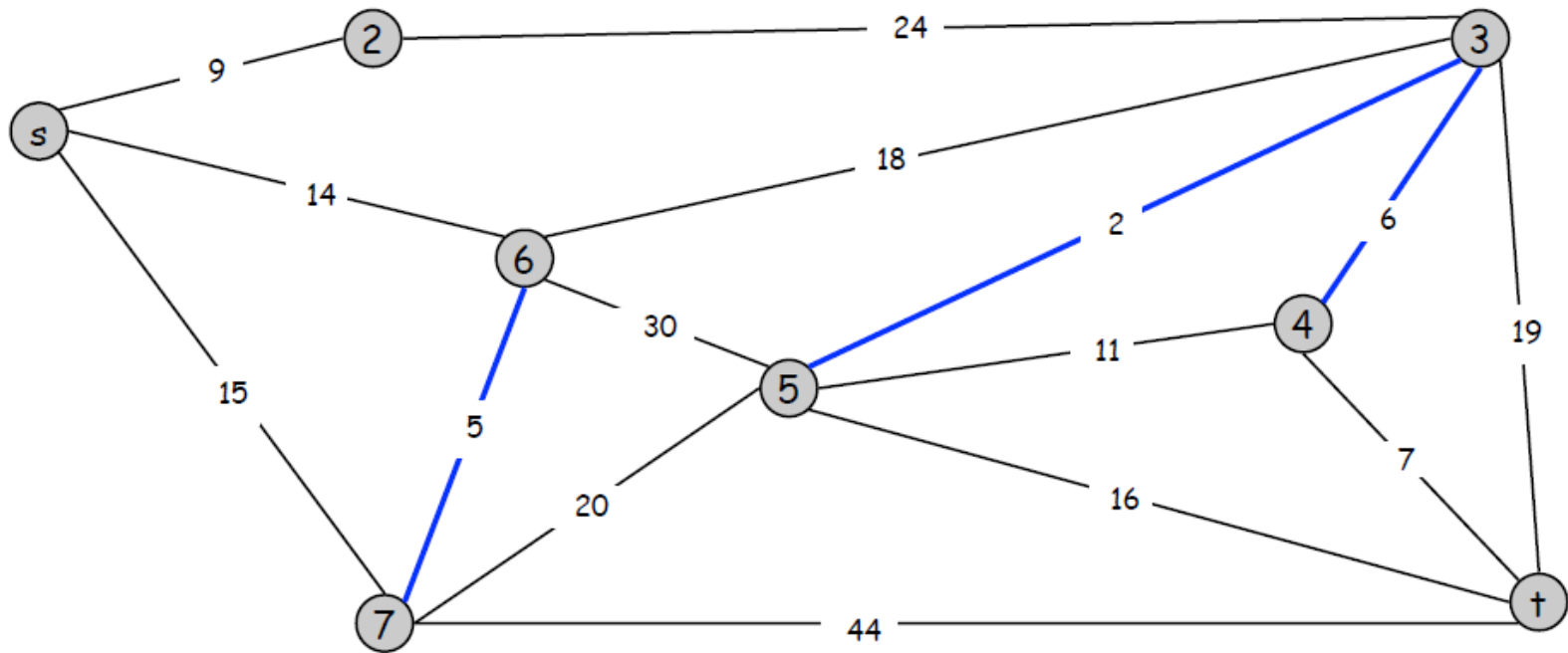


Algoritmo di Kruskal

Trovare *Minimo Spanning Tree*.

- Inizia con $T = \phi$.
- Considera archi in ordine crescente di costo.
- Inserisci l'arco in T se non crea un ciclo.

$T = \{ \{s,2\}, \{s,5\}, \{s,6\} \}$

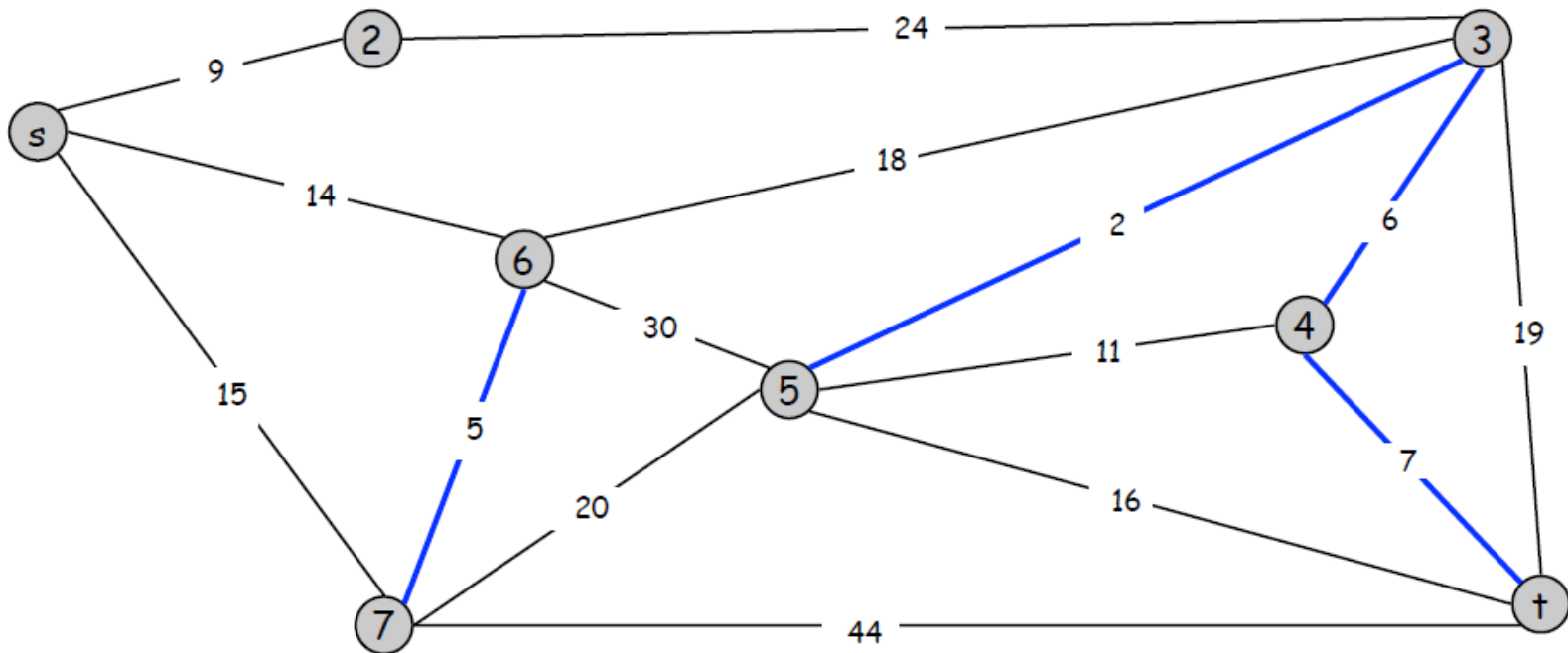


Algoritmo di Kruskal

Trovare *Minimo Spanning Tree*.

- Inizia con $T = \phi$.
- Considera archi in ordine crescente di costo.
- Inserisci l'arco in T se non crea un ciclo.

$T = \{ \{s,2\}, \{s,5\}, \{s,6\}, \{s,7\} \}$

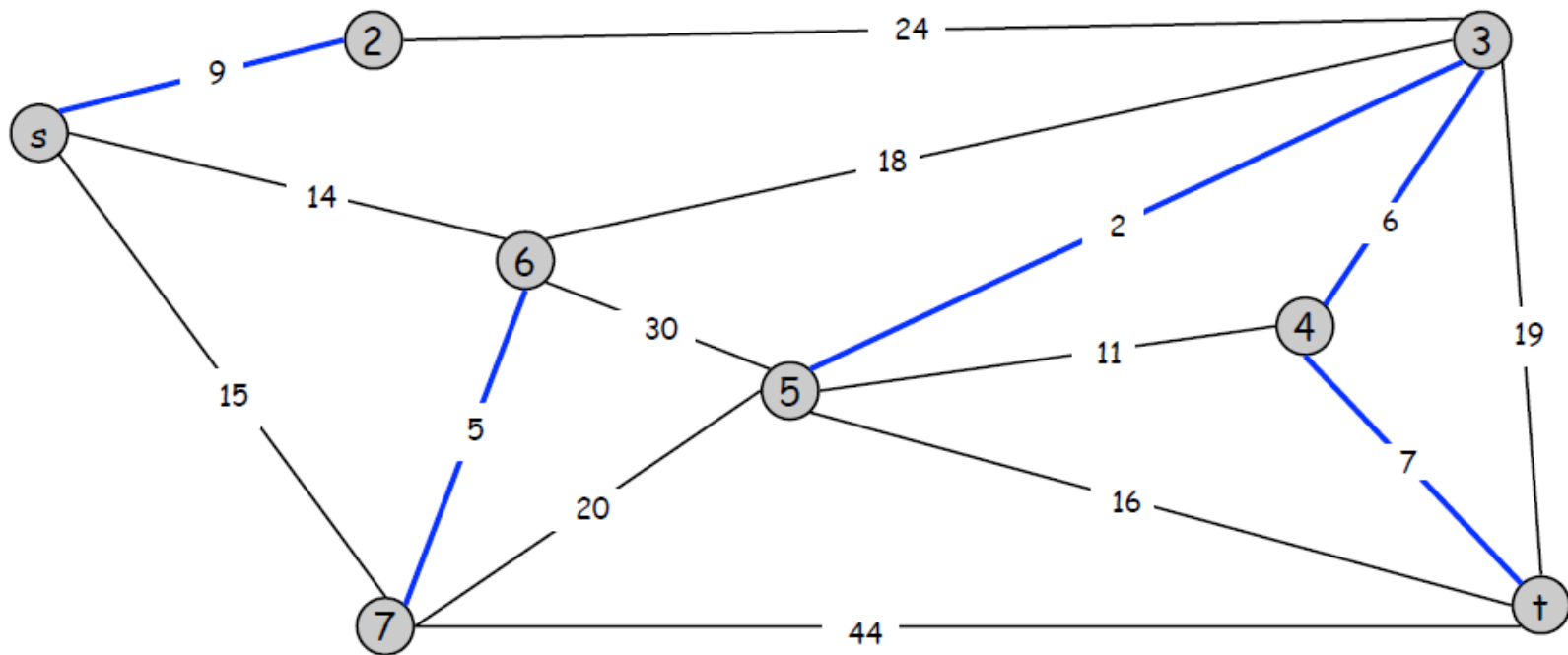


Algoritmo di Kruskal

Trovare Minimo Spanning Tree.

- Inizia con $T = \phi$.
- Considera archi in ordine crescente di costo.
- Inserisci l'arco in T se non crea un ciclo.

$T = \{ \{s,2\}, \{s,5\}, \{s,6\}, \{s,7\}, \{s,9\} \}$

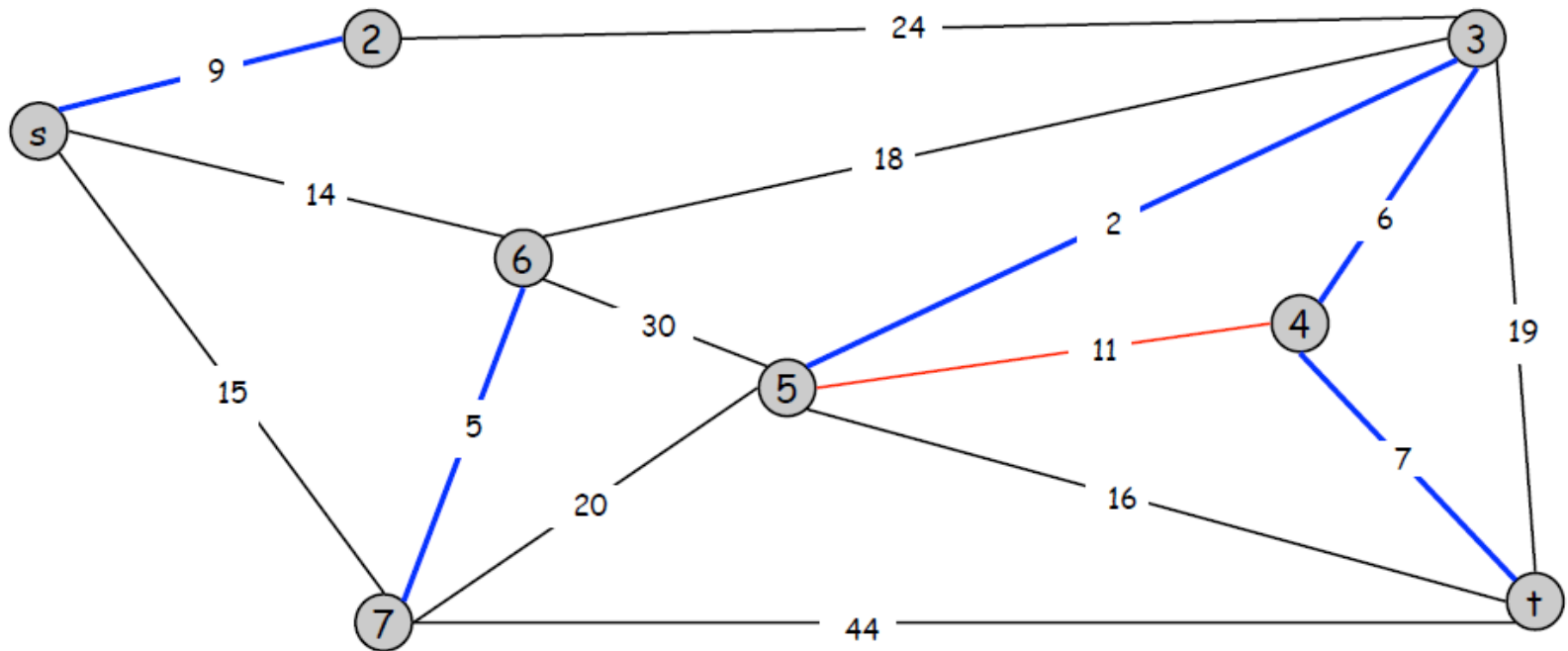


Algoritmo di Kruskal

Trovare Minimo Spanning Tree.

- Inizia con $T = \phi$.
- Considera archi in ordine crescente di costo.
- Inserisci l'arco in T se non crea un ciclo.

$T = \{ \{s,2\}, \{s,5\}, \{s,6\}, \{s,7\}, \{s,9\} \}$

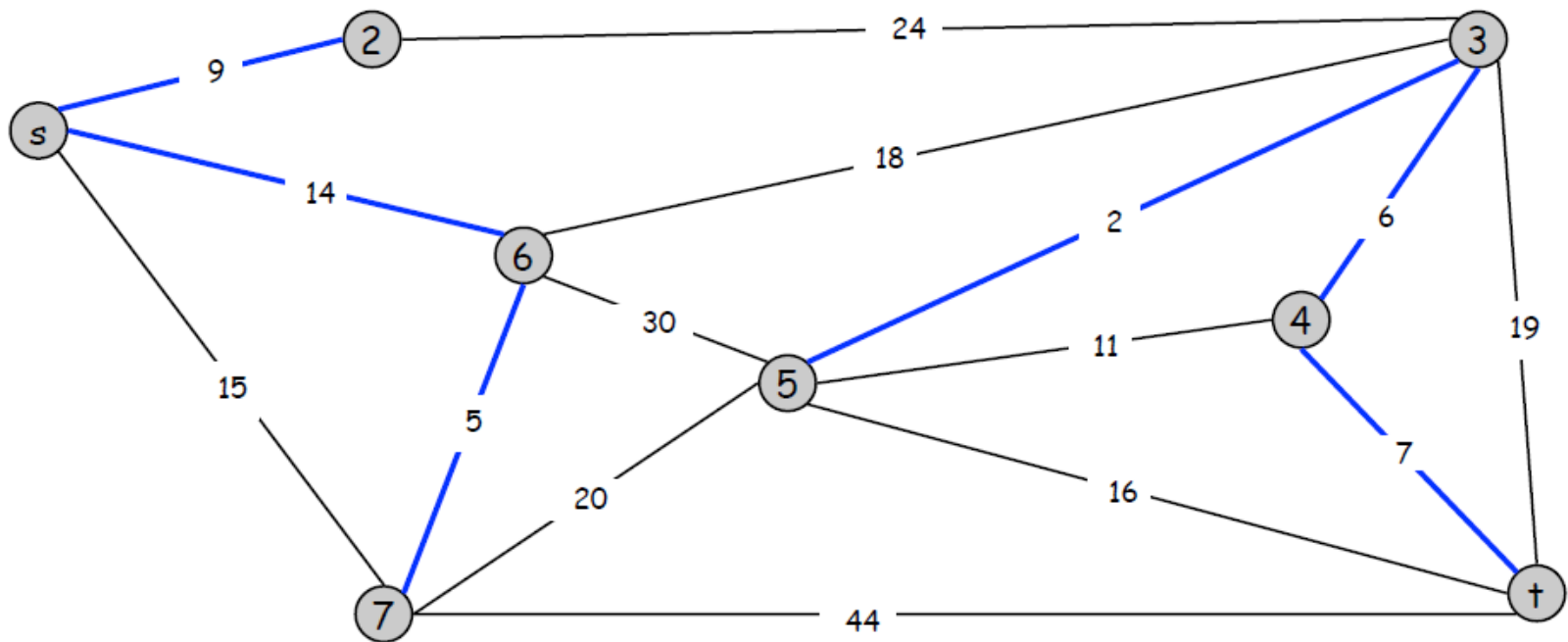


Algoritmo di Kruskal

Trovare Minimo Spanning Tree.

- Inizia con $T = \phi$.
- Considera archi in ordine crescente di costo.
- Inserisci l'arco in T se non crea un ciclo.

$T = \{ \{s,2\}, \{s,5\}, \{s,6\}, \{s,7\}, \{s,9\}, \{s,14\} \}$

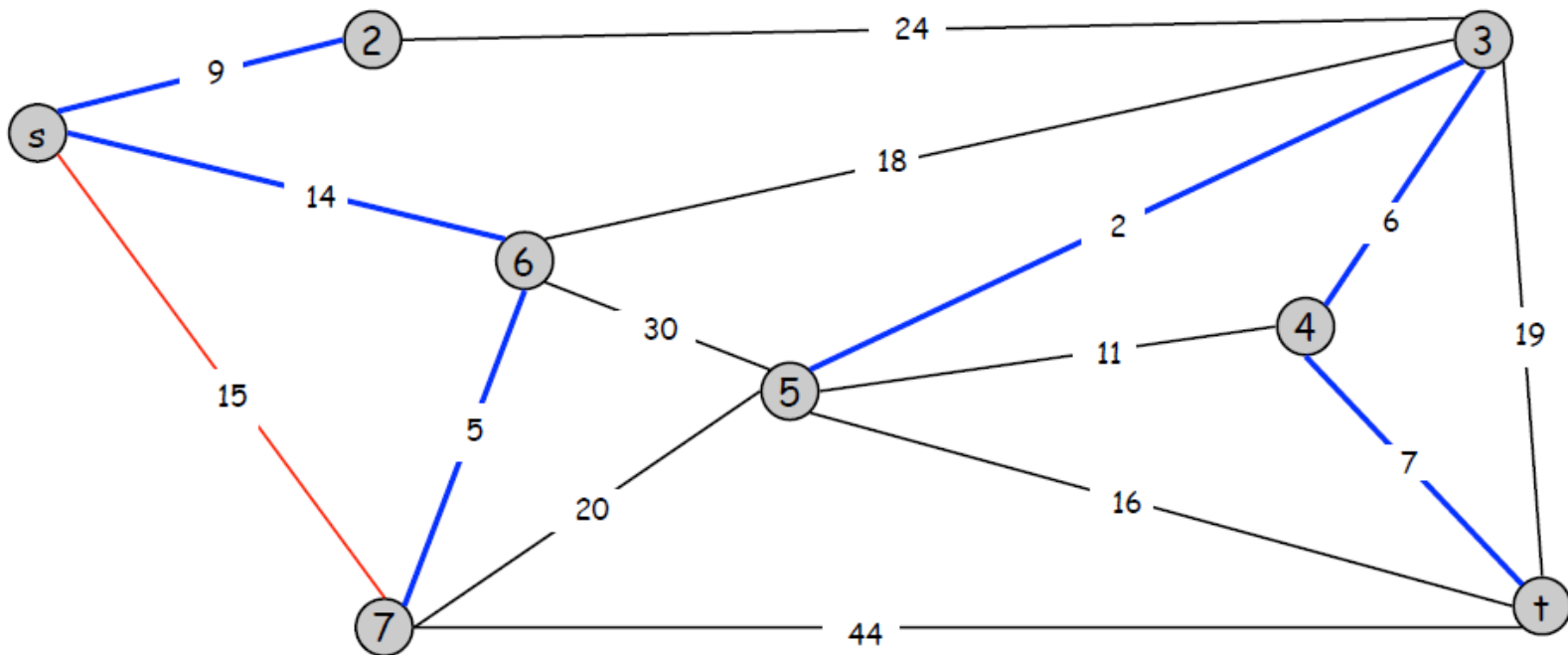


Algoritmo di Kruskal

Trovare Minimo Spanning Tree.

- Inizia con $T = \phi$.
- Considera archi in ordine crescente di costo.
- Inserisci l'arco in T se non crea un ciclo.

$T = \{ \{s,2\}, \{s,5\}, \{s,6\}, \{s,7\}, \{s,9\}, \{s,14\} \}$

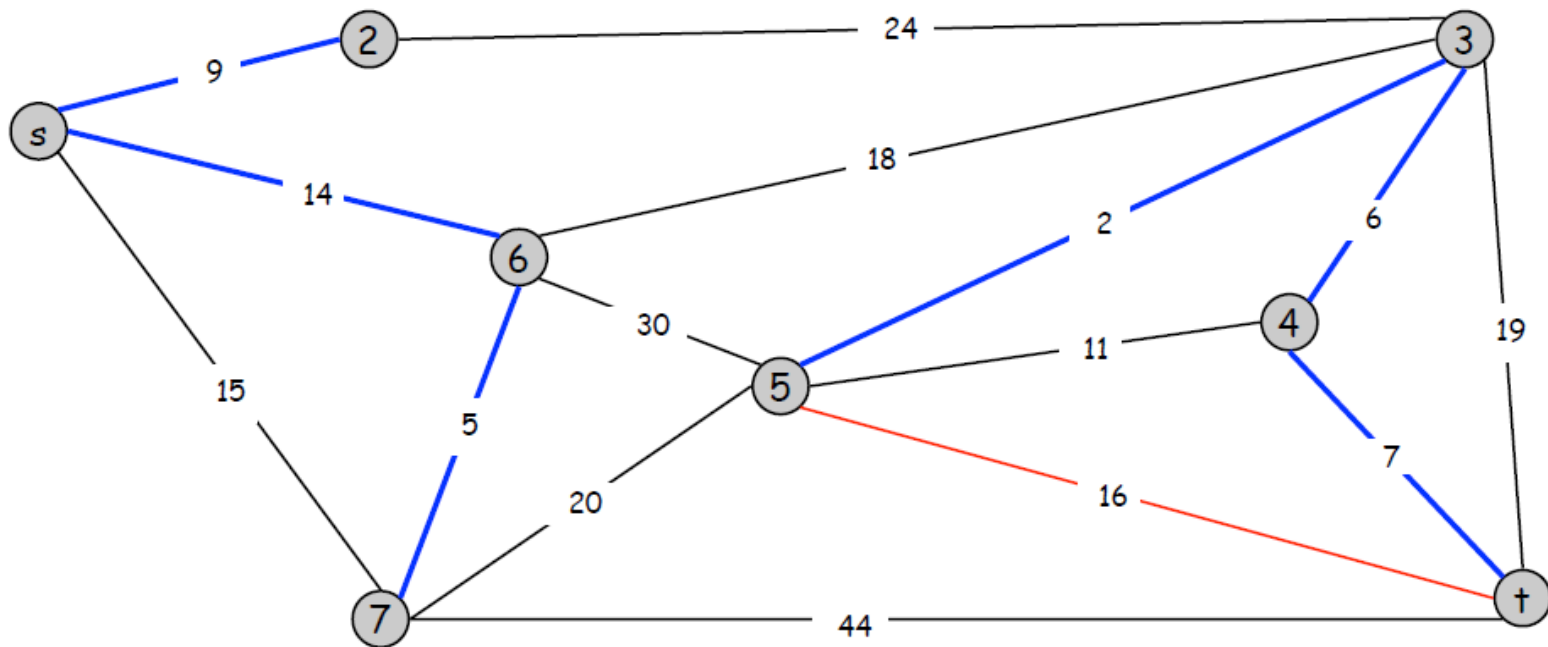


Algoritmo di Kruskal

Trovare Minimo Spanning Tree.

- Inizia con $T = \phi$.
- Considera archi in ordine crescente di costo.
- Inserisci l'arco in T se non crea un ciclo.

$T = \{ \{s,2\}, \{s,5\}, \{s,6\}, \{s,7\}, \{s,9\}, \{s,14\} \}$

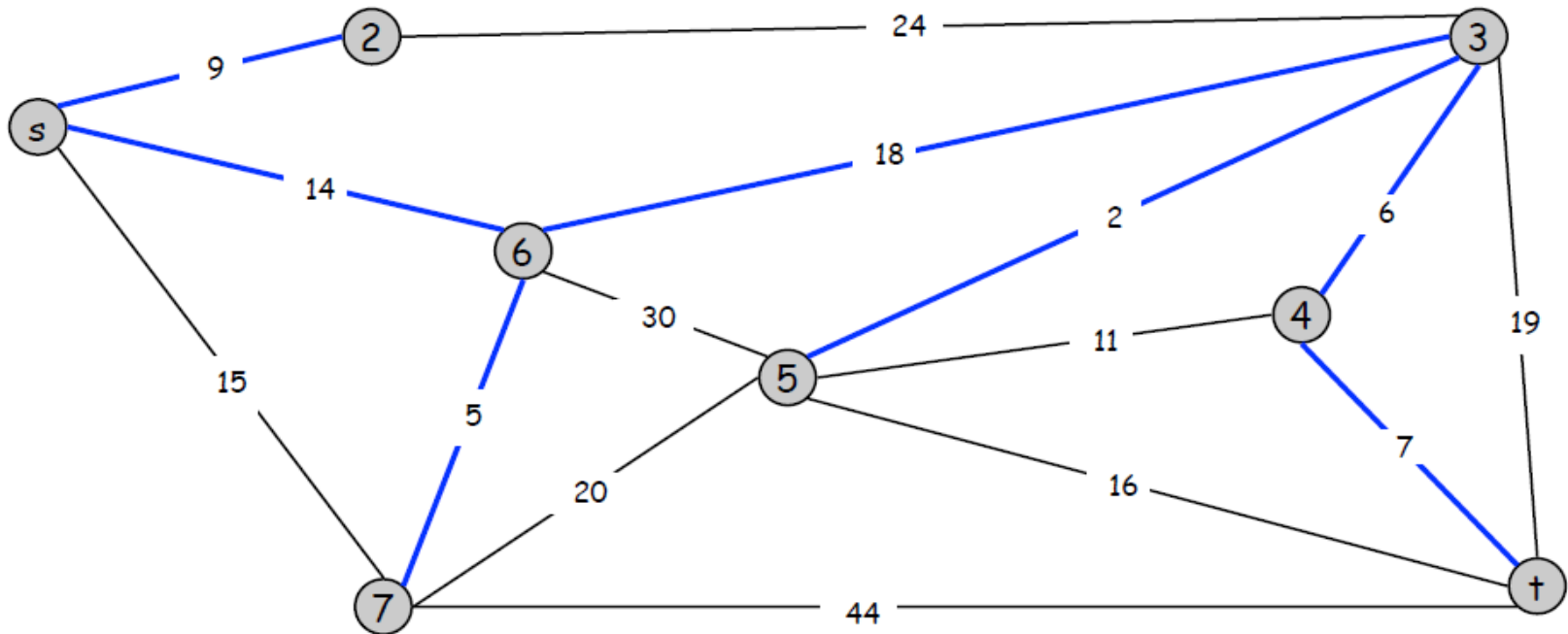


Algoritmo di Kruskal

Trovare Minimo Spanning Tree.

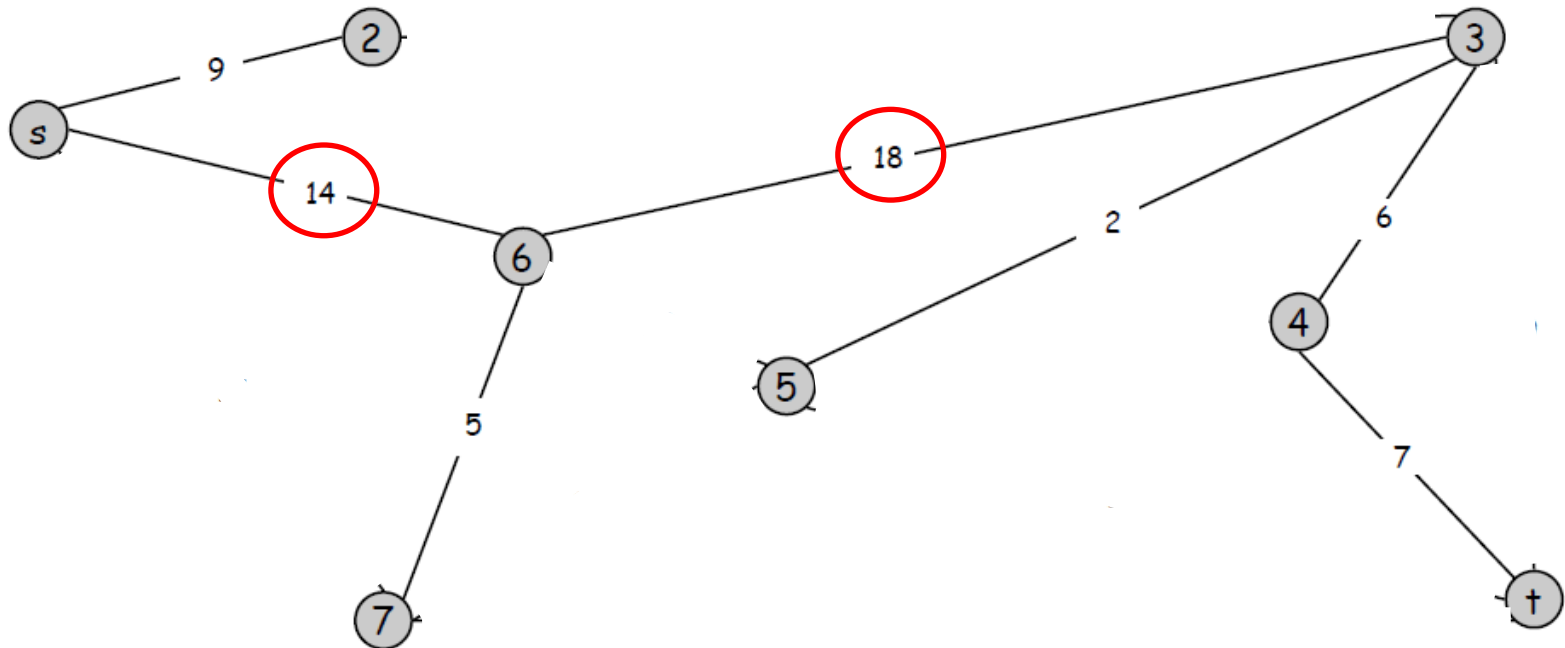
- Inizia con $T = \phi$.
- Considera archi in ordine crescente di costo.
- Inserisci l'arco in T se non crea un ciclo.

$T = \{ \{s,2\}, \{s,5\}, \{s,6\}, \{s,7\}, \{s,9\}, \{s,14\}, \{s,18\} \}$ è MST di costo $9+14+5+18+2+6+7= 61$



Reverse-Delete: an example

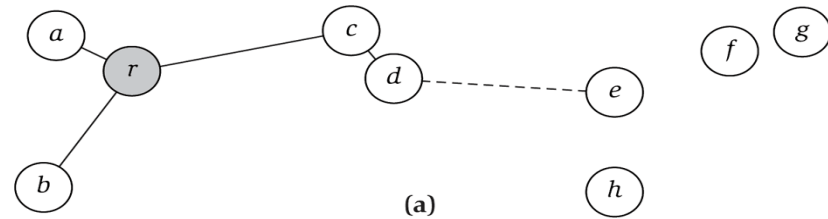
Reverse-Delete algorithm. Start with $T = E$. Consider edges in descending order of cost. Delete edge e from T unless doing so would disconnect T .



Confronto fra algoritmi di Prim e di Kruskal

Durante l'esecuzione:

l'algoritmo di Prim mantiene un singolo albero



l'algoritmo di Kruskal un insieme di alberi (foresta).

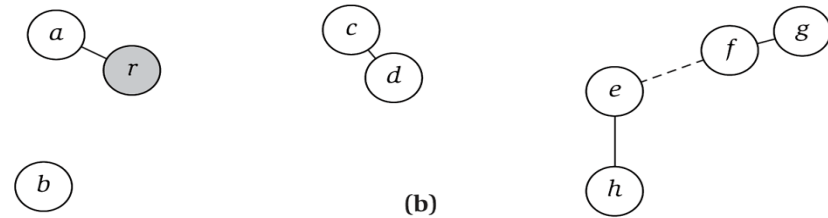


Figure 4.9 Sample run of the Minimum Spanning Tree Algorithms of (a) Prim and (b) Kruskal, on the same input. The first 4 edges added to the spanning tree are indicated by solid lines; the next edge to be added is a dashed line.

Gli alberi restituiti sono gli stessi?

Se i costi sono distinti, il MST è unico. (es. 8, p.192: usa proprietà seguenti).

In generale no.

Osservazioni (per il seguito).

In un albero: se tolgo un arco disconnetto; se aggiungo un arco creo un ciclo.

In un grafo connesso: se, togliendo un arco, non disconnetto, l'arco apparteneva ad un ciclo.

Correctness

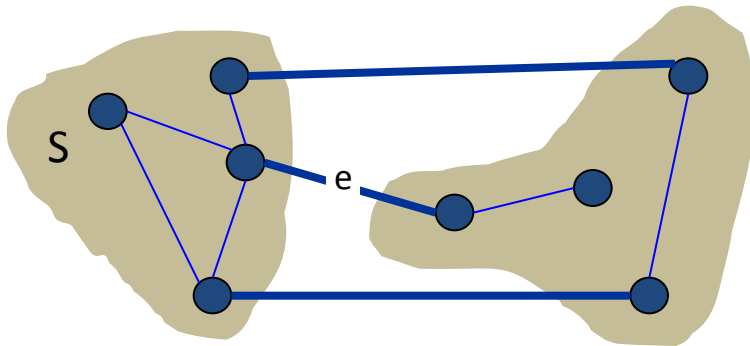
Correctness of **Prim's** and **Kruskal's** algorithms is based on Cut property. Correctness of **Reverse-Delete** on Cycle property.

Simplifying assumption. All edge costs c_e are distinct (hence the MST is unique).

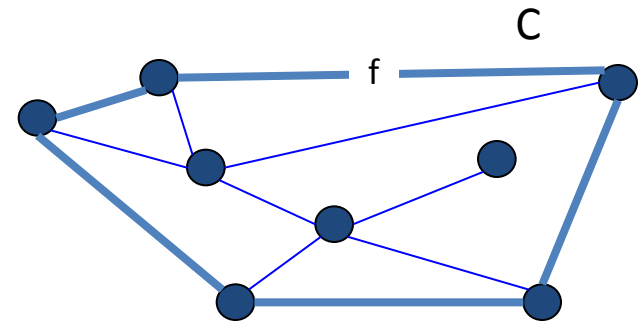
Cut property. Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S . Then the MST contains e .

(In altri testi: l'arco leggero che attraversa il taglio è sicuro per S)

Cycle property. Let C be any cycle, and let f be the max cost edge belonging to C . Then the MST does not contain f .



e is in the MST



f is not in the MST

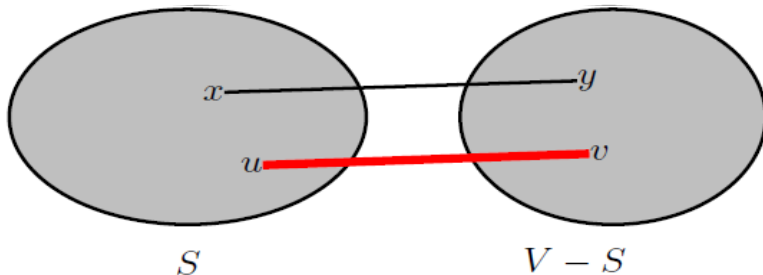
Proof of Cut Property

Simplifying assumption. All edge costs c_e are distinct.

Cut property. Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S . Then the MST contains e .

Pf. (exchange argument)

Supponiamo che ciò non sia vero, e sia T un MST che *non* contiene e . È ovvio che T dovrà contenere almeno un'arco $a = (x, y) \neq (u, v) = e$ con un'estremo in S e l'altro in $V - S$ (altrimenti come farebbe T a connettere tra di loro tutti i nodi di V ?)



Aggiungiamo a T l'arco $e = (u, v)$, (per ipotesi $c(u, v) < c(x, y)$) che succede nell'albero T ? Si crea un ciclo! Eliminiamo allora l'arco (x, y) . I nodi in S rimangono connessi

tra di loro (l'eliminazione dell'arco (x, y) non influenza i cammini in S), analoga cosa per i nodi in $V - S \Rightarrow$ da ogni nodo di S è raggiungibile ogni nodo di $V - S$, attraverso l'arco $(u, v) \Rightarrow \exists$ un nuovo albero T' che connette tutti i vertici di V , con $\text{costo}(T') < \text{costo}(T)$, contro l'ipotesi.

Proof of Cycle Property

Simplifying assumption. All edge costs c_e are distinct.

Cycle property. Let C be any cycle in G , and let e be the max cost edge belonging to C . Then the MST T^* does not contain e .

Pf. (exchange argument)

Suppose e belongs to T^* , and let's see what happens.

Deleting e from T^* disconnects T^* in S and $V \setminus S$.

In the cycle C there exists another edge, say $e'=(u',v')$, with u' in S and v' in $V \setminus S$.

$T' = T^* \cup \{e'\} - \{e\}$ is also a spanning tree.

Since $c_{e'} < c_e$ then $\text{cost}(T') < \text{cost}(T^*)$.

This is a contradiction. ■

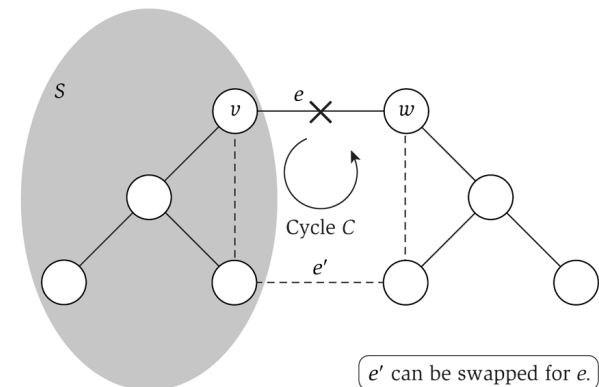


Figure 4.11 Swapping the edge e' for the edge e in the spanning tree T , as described in the proof of (4.20).

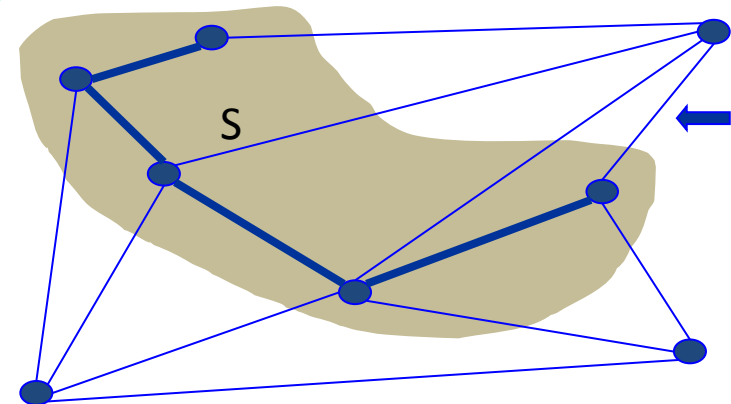
Prim's Algorithm: Proof of Correctness

Prim's algorithm. [Jarník 1930, Dijkstra 1957, Prim 1959]

Prim's algorithm. Start with some root node s and greedily grow a tree T from s outward. At each step, add the cheapest edge e to T that has exactly one endpoint in T .

Pf. Che l'algoritmo di Prim produca un albero è ovvio, visto che aggiunge archi solo da nodi già tra di loro connessi in S a nuovi nodi "fuori" di S (quindi non crea cicli). Inoltre, ad ogni passo aggiunge a T l'arco di minimo costo che ha un estremo u in S (insieme dei nodi su cui un albero ricoprente parziale è stato già costruito) ad un nodo $v \in V - S$.

Dalla proprietà prima vista, tale arco appartiene ad ogni MST del grafo (cioè, di nuovo l'algoritmo non inserisce mai archi che non appartengono a MST, quindi produce effettivamente un MST).



Implementation: Prim's Algorithm (un'idea)

Implementation. Use a priority queue Q “a la Dijkstra”.

Maintain set of explored nodes S and set of unexplored nodes Q .

For each unexplored node v in Q , maintain as priority an attachment cost $a[v] = \text{cost of cheapest edge } v \text{ to a node in } S$.

$O(n^2)$ with an array; **$O(m \log n)$** with a binary heap.

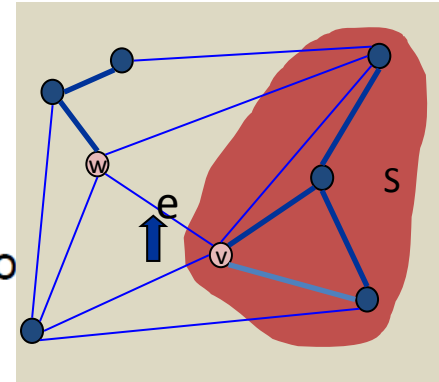
```
Prim(G, c) {
  foreach (v ∈ V) a[v] ← ∞ ; a[r] ← 0
  Initialize an empty priority queue Q
  foreach (v ∈ V) insert v onto Q
  Initialize set of explored nodes S ← ∅

  while (Q is not empty) {
    u ← delete min element from Q
    S ← S ∪ { u }
    foreach (edge e = (u, v) incident to u)
      if ((v ∉ S) and (ce < a[v]))
        decrease priority a[v] to ce
  }
```

L'algoritmo di Kruskal produce un MST

Aggiungi a T uno ad uno gli archi del grafo, in ordine di costo crescente, saltando gli archi che creano cicli con gli archi già aggiunti.

- Sia $e = (v, w)$ un generico arco inserito in T dall'algoritmo di Kruskal, e sia S l'insieme di tutti i nodi connessi a v attraverso un cammino, al momento appena prima di aggiungere (v, w) a T .
- Ovviamente vale che $v \in S$, mentre $w \notin S$, altrimenti l'arco (v, w) creerebbe un ciclo.
- Inoltre, negli istanti precedenti l'algoritmo non ha incontrato nessun arco da nodi in S a nodi in $V - S$, altrimenti un tale arco sarebbe stato aggiunto, visto che non creava cicli.
- Pertanto l'arco (v, w) è il primo arco da S a $V - S$ che l'algoritmo incontra, ovvero è l'arco di minor costo da S a $V - S$ che, abbiamo visto appartiene ad ogni MST.
- Ci rimane da mostrare che l'output dell'algoritmo di Kruskal è un albero



Facciamolo:

- Sicuramente, per costruzione, l'output (V, T) non contiene cicli.
- Potrebbe (V, T) non essere connesso? Ovvero potrebbe esistere un $\emptyset \neq S \subset V$ per cui in T non esiste alcun arco da S a $V - S$?
- Sicuramente no! Infatti, poichè il grafo G è connesso, un tale arco e esiste sicuramente in G e poichè l'algoritmo di Kruskal esamina tutti gli archi di G , prima o poi incontrerà tale arco e e lo inserirà, visto che non crea cicli.

Implementation: Kruskal's Algorithm (un'idea)

Implementation. Use the **union-find** data structure.

Build set T of edges in the MST.

Maintain set for each connected component; initially a set for each node.

$O(m \log n)$ for sorting and $O(m \alpha(m, n))$ for union-find.

$m \leq n^2 \Rightarrow \log m$ is $O(\log n)$

essentially a constant

```
Kruskal(G, c) {
  Sort edges weights so that  $c_1 \leq c_2 \leq \dots \leq c_m$ .
   $T \leftarrow \phi$ 

  foreach ( $u \in V$ ) make a set containing singleton u

  for i = 1 to m
    ( $u, v$ ) =  $e_i$ 
    if (u and v are in different sets) {
       $T \leftarrow T \cup \{e_i\}$ 
      merge the sets containing u and v
    }
  return T
}
```

are u and v in different connected components?

merge two components

Correttezza Reverse-Delete

Reverse-Delete algorithm. Start with $T = E$. Consider edges in descending order of cost. Delete edge e from T unless doing so would disconnect T .

Prova. Sia e un arco eliminato dall'algoritmo. Poiché e non disconnette il grafo, appartiene ad un ciclo. Di questo ciclo è l'arco di costo massimo (per la scelta effettuata). Per la **proprietà del ciclo** e **non appartiene** a nessun MST.

Alla fine (V, T) sarà un albero: **connesso** per costruzione dell'algoritmo; **aciclico**, altrimenti l'algoritmo non si sarebbe arrestato.

Lexicographic Tiebreaking (**facoltativo**)

To remove the assumption that all edge costs are distinct:

perturb all edge costs by tiny amounts to break any ties.

Impact. Kruskal and Prim only interact with costs via pairwise comparisons. If perturbations are sufficiently small, MST with perturbed costs is MST with original costs.

```
boolean less(i, j) {  
    if      (cost(ei) < cost(ej)) return true  
    else if (cost(ei) > cost(ej)) return false  
    else if (i < j)                 return true  
    else                             return false  
}
```

Implementation. Can handle arbitrarily small perturbations implicitly by breaking ties lexicographically, according to index.

An application: Clustering

Clustering. Given a set U of n objects labeled p_1, \dots, p_n , classify into coherent groups.

↑
photos, documents, micro-organisms

Distance function. Numeric value specifying "closeness" of two objects.

↑
number of corresponding pixels whose intensities differ by some threshold

Fundamental problem. Divide into clusters so that points in different clusters are far apart (and points in same cluster are close enough).

Routing in mobile ad hoc networks.

Identify patterns in gene expression.

Document categorization for web search.

Similarity searching in medical image databases

Skycat: cluster 10^9 sky objects into stars, quasars, galaxies.

Clustering of Maximum Spacing

k-clustering. Divide objects into k non-empty groups.

Distance function. Assume it satisfies several natural properties.

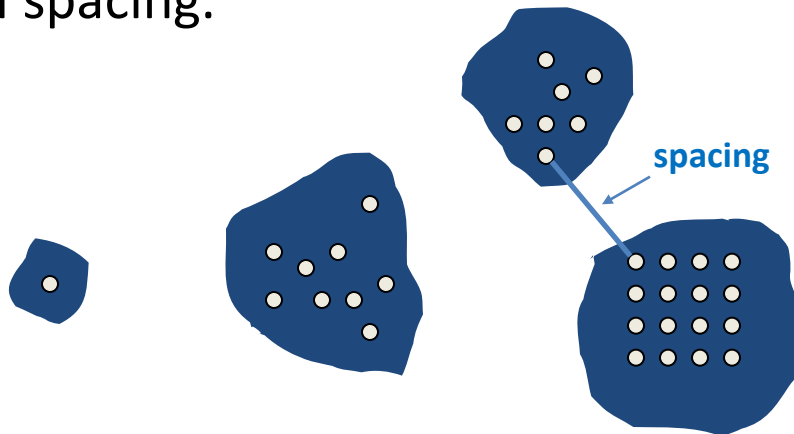
$d(p_i, p_j) = 0$ iff $p_i = p_j$ (identity of indiscernibles)

$d(p_i, p_j) \geq 0$ (nonnegativity)

$d(p_i, p_j) = d(p_j, p_i)$ (symmetry)

Spacing. Min distance between any pair of points in different clusters.

Clustering of maximum spacing. Given an integer k , find a k -clustering of maximum spacing.



Esempio

d =distanza euclidea. $k=3$

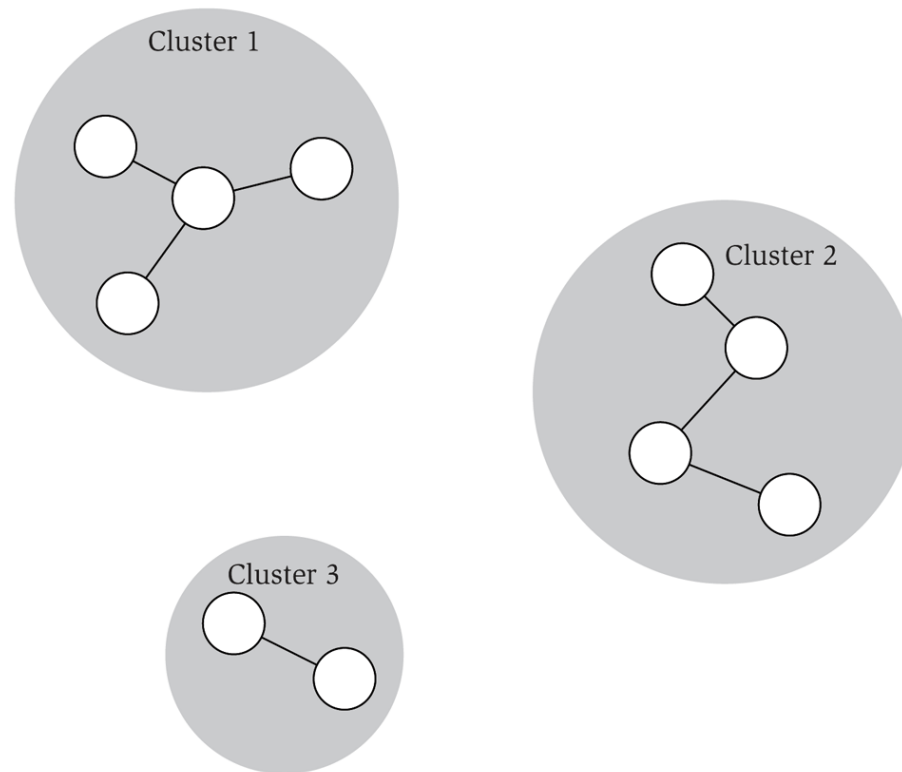


Figure 4.14 An example of single-linkage clustering with $k = 3$ clusters. The clusters are formed by adding edges between points in order of increasing distance.

Greedy Clustering Algorithm

Single-link k -clustering algorithm.

Form a graph on the vertex set U , corresponding to n clusters.

Find the closest pair of objects such that each object is in a different cluster, and add an edge between them.

Repeat $n-k$ times until there are exactly k clusters.

Key observation. This procedure is precisely Kruskal's algorithm (except we stop when there are k connected components).

Remark. Equivalent to finding an MST and deleting the $k-1$ most expensive edges.

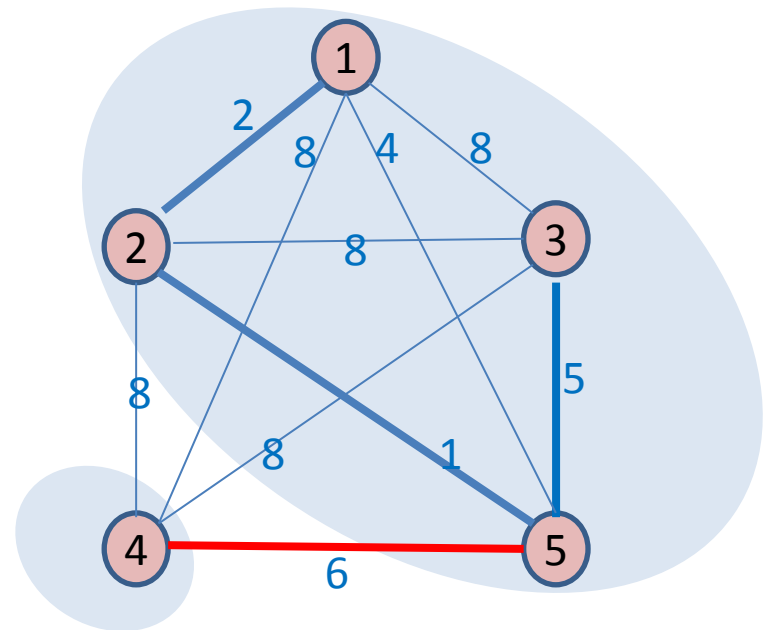
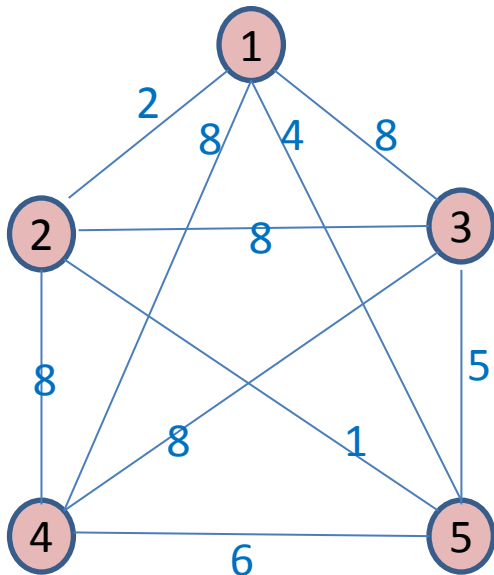
Esempio dell'algoritmo di clustering

$U = \{1,2,3,4,5\}$; $d(i,j)$ come indicata sull'arco (i,j) ; $k=2$.

Costruisco il grafo come in figura a sinistra.

Eseguo l'algoritmo di Kruskal. Seleziono in ordine:

$(2,5)$, $(1,2)$, $(3,5)$, $(4,5)$. Cancello gli ultimi $k-1=1$ archi inseriti (ovvero $(4,5)$). Ottengo il 2-clustering: $C_1=\{1,2,3,5\}$, $C_2=\{4\}$ il cui spacing è $d^*=6$



Nota: d^* = costo del $(k-1)$ -esimo arco più costoso del MST.

Greedy Clustering Algorithm: Analysis

Theorem. Let C^* denote the clustering C^*_1, \dots, C^*_k formed by deleting the $k-1$ most expensive edges of a MST. C^* is a k -clustering of max spacing.

Pf. Let C denote some other clustering C_1, \dots, C_k .

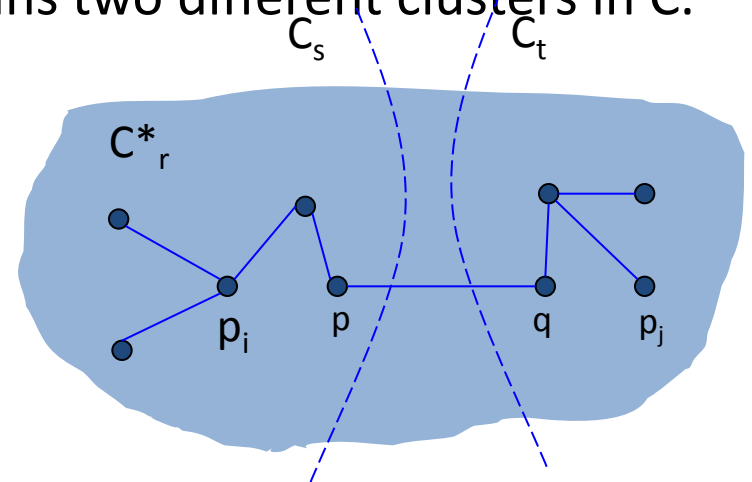
The spacing of C^* is the length d^* of the $(k-1)^{\text{st}}$ most expensive edge.

Let p_i, p_j be in the same cluster in C^* , say C^*_r , but different clusters in C , say C_s and C_t .

Some edge (p, q) on p_i - p_j path in C^*_r spans two different clusters in C .

All edges on p_i - p_j path have length $\leq d^*$ since Kruskal chose them.

Spacing of C is $\leq d^*$ since p and q are in different clusters. ■



Reti di flusso

Problema del flusso massimo

Motivazione iniziale: problemi di traffico su reti di trasporto

- Trasporti ferroviari, autostradali,...
- Trasporto di liquidi in reti idriche
- Trasporto di pacchetti di dati in una rete di comunicazione.

Applicazioni non banali / riduzioni:

- Data mining.
- Open-pit mining.
- Project selection.
- Airline scheduling.
- **Bipartite matching.**
- Baseball elimination.
- Image segmentation.
- Network connectivity.
- Network reliability.
- Distributed computing.
- Egalitarian stable matching.
- Security di statistical data.
- Network intrusion detection.
- Multi-camera scene reconstruction.
- Molte altre ancora. . .

Flusso massimo e Taglio minimo

Flusso massimo e taglio minimo

- Due problematiche molto ricche.
- Problemi importanti in ottimizzazione combinatoriale.
- Dualità matematica.

Algoritmo di Ford-Fulkerson

Algoritmo incrementale basato sulle reti residuali e i cammini aumentanti

(parr. 7.1, 7.2)

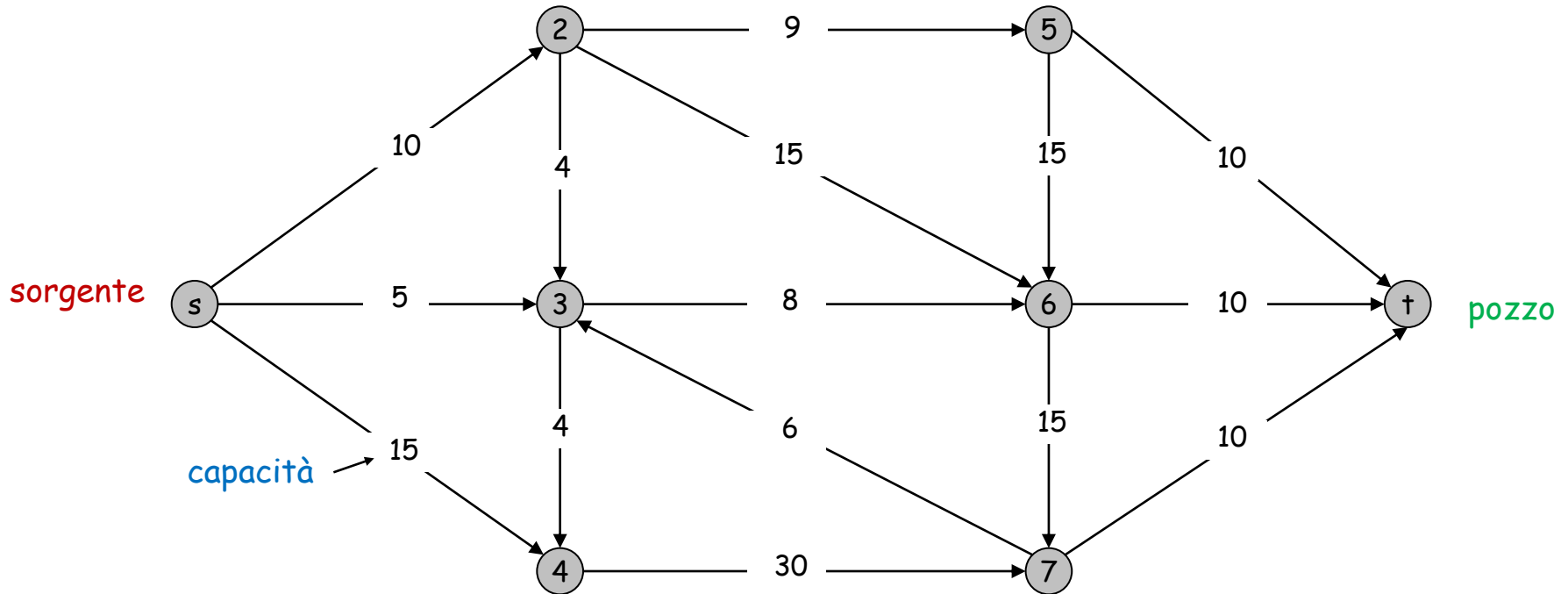
Un'applicazione: matching in un grafo bipartito (par. 7.5)

Reti di flusso

Astrazione per materiale che "scorre" attraverso gli archi (*come liquidi nei tubi*).

Una *rete di flusso* è $G = (V, E)$ = grafo orientato con

- due nodi particolari: **s = sorgente** (senza archi entranti)
t = pozzo (senza archi uscenti).
- $c(e)$ = capacità dell'arco e .

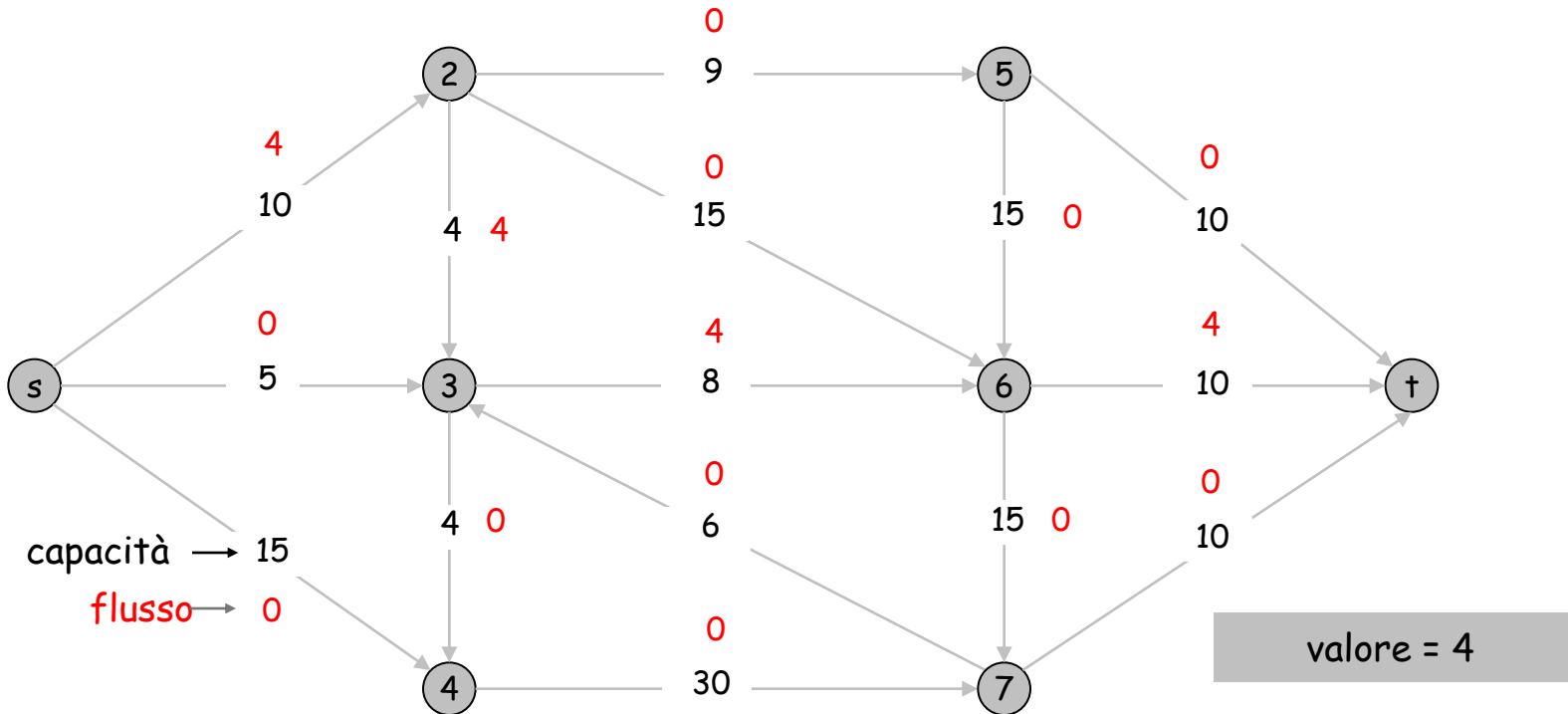


Flusso

Def. Un **flusso s-t** è una funzione $f: E \rightarrow \mathbb{R}^+$ che soddisfa:

- per ogni $e \in E$: $0 \leq f(e) \leq c(e)$ (capacità)
- per ogni $v \in V - \{s, t\}$: $\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e)$ (conservazione)

Def. Il **valore** del flusso f è: $v(f) = \sum_{e \text{ out of } s} f(e)$.



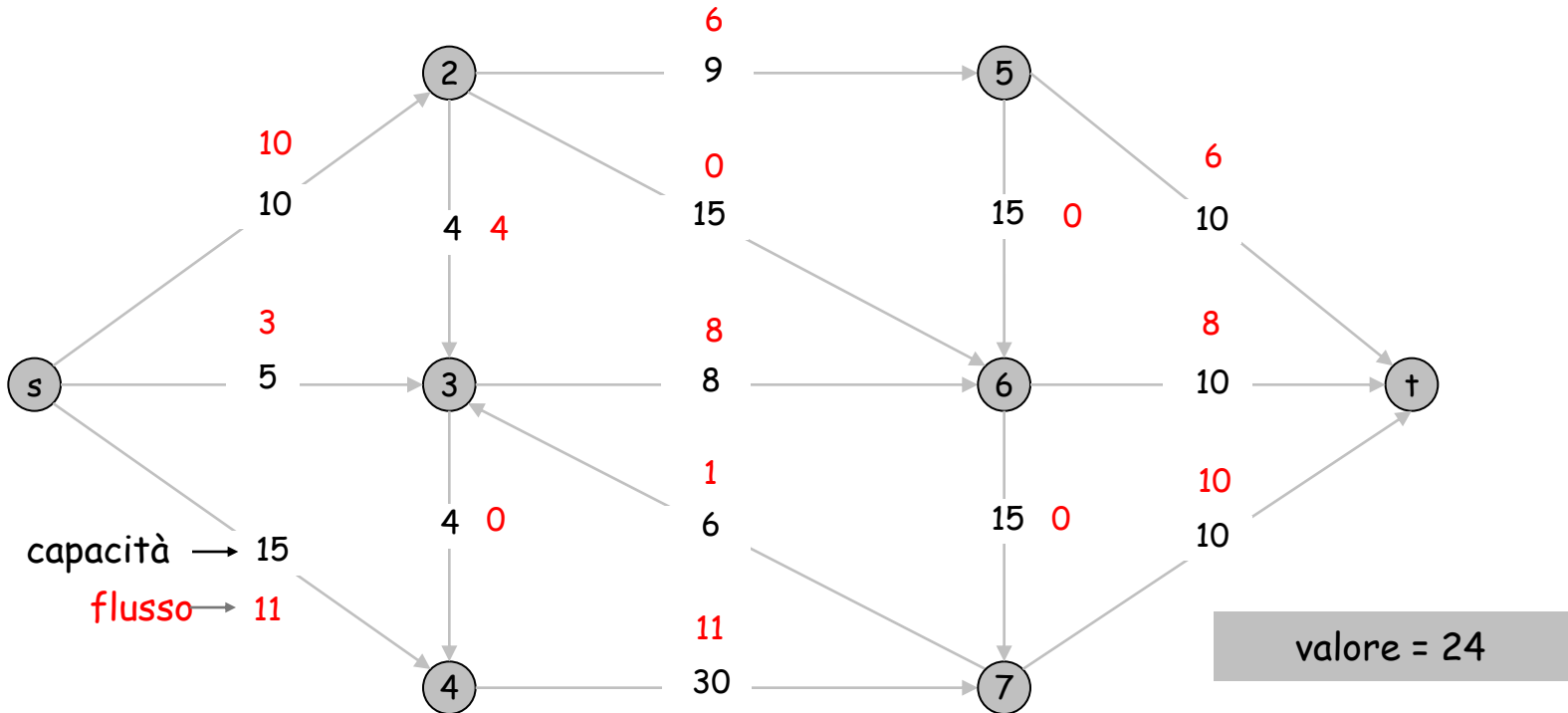
Flusso

Def. Un **flusso s-t** è una funzione $f: E \rightarrow \mathbb{R}^+$ che soddisfa:

- per ogni $e \in E$: $0 \leq f(e) \leq c(e)$ (capacità)
- per ogni $v \in V - \{s, t\}$: $\sum_{e \text{ in to } v} f(e) = \sum_{e \text{ out of } v} f(e)$ (conservazione)

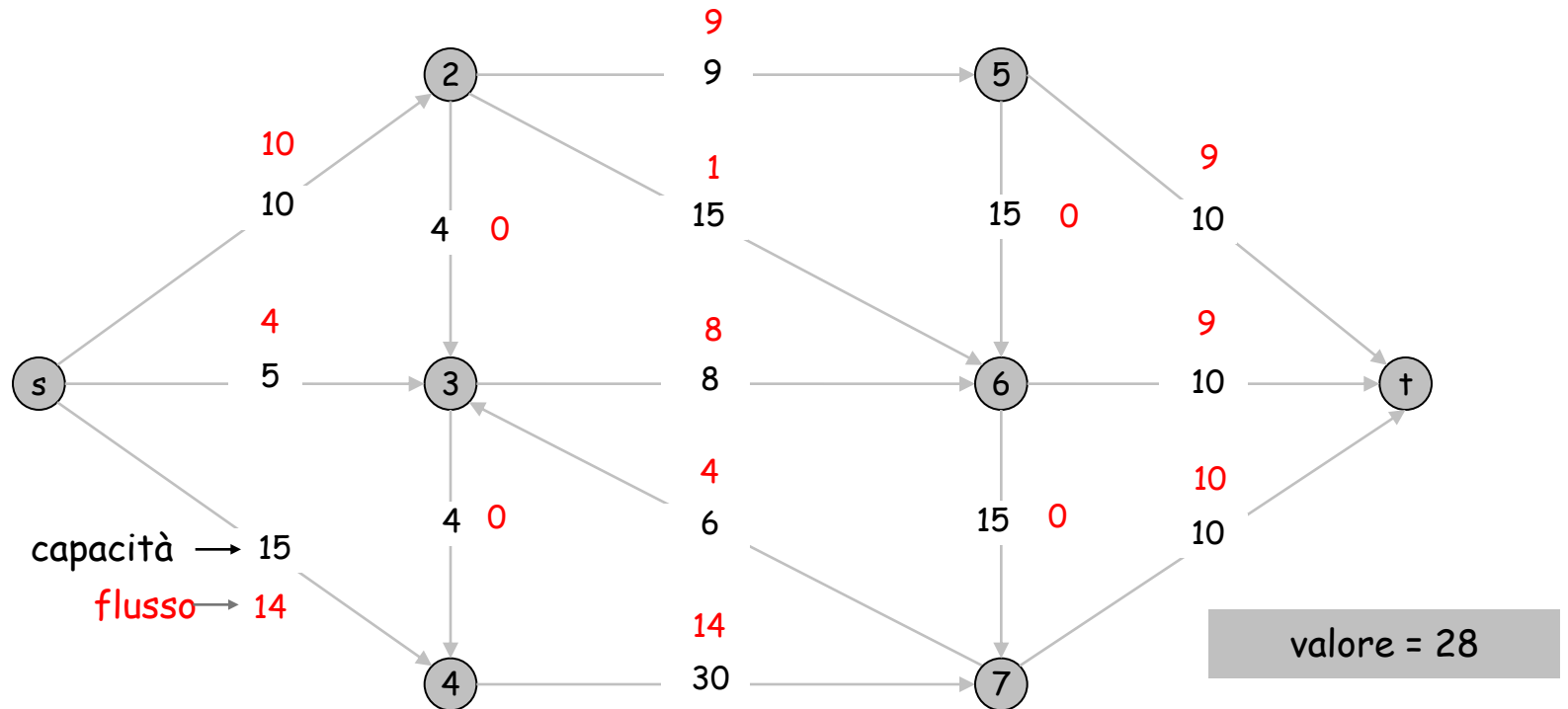
Def. Il **valore** del flusso f è:

$$v(f) = \sum_{e \text{ out of } s} f(e).$$



Problema del massimo flusso

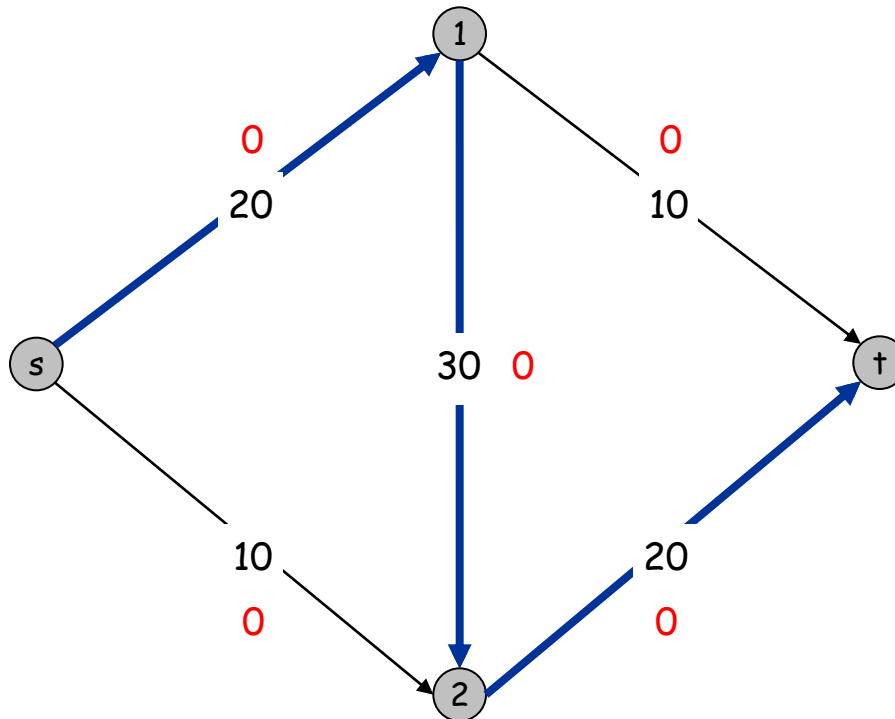
Problema del massimo flusso. Trovare il flusso s-t di massimo valore.



Verso un algoritmo per il Flusso massimo

Algoritmo greedy.

- Iniziare con $f(e) = 0$ per ogni arco $e \in E$.
- Trovare un cammino P da s a t in cui ogni arco ha $f(e) < c(e)$.
- Aumentare il flusso lungo il cammino P .
- Ripetere finchè è possibile.

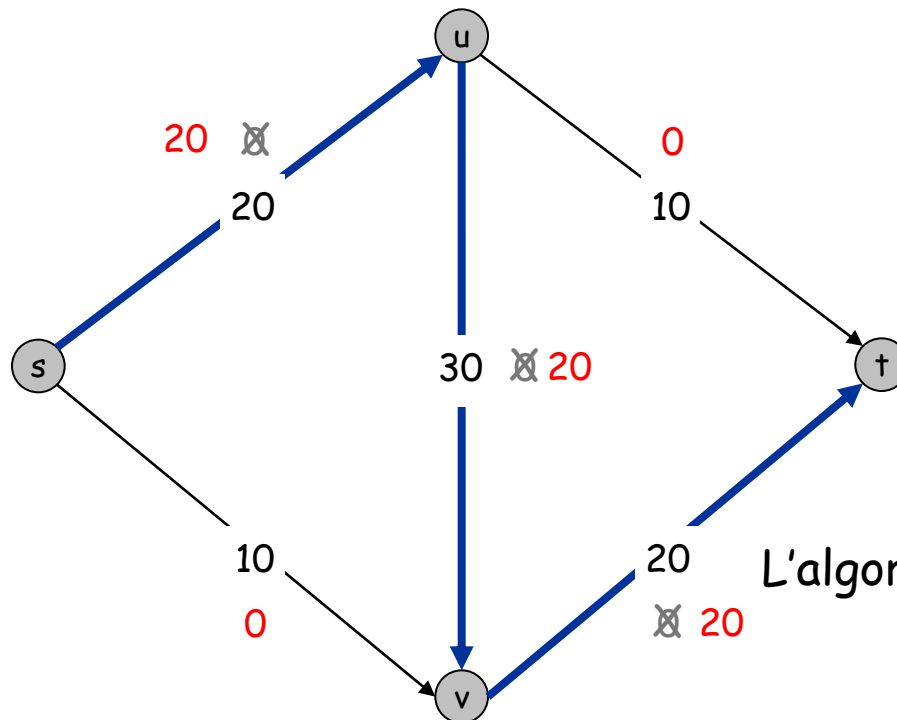


Valore flusso = 0

Verso un algoritmo per il Flusso massimo

Algoritmo greedy.

- Iniziare con $f(e) = 0$ per ogni arco $e \in E$.
- Trovare un cammino P da s a t in cui ogni arco ha $f(e) < c(e)$.
- Aumentare il flusso lungo il cammino P .
- Ripetere finchè è possibile.



Valore flusso = 20

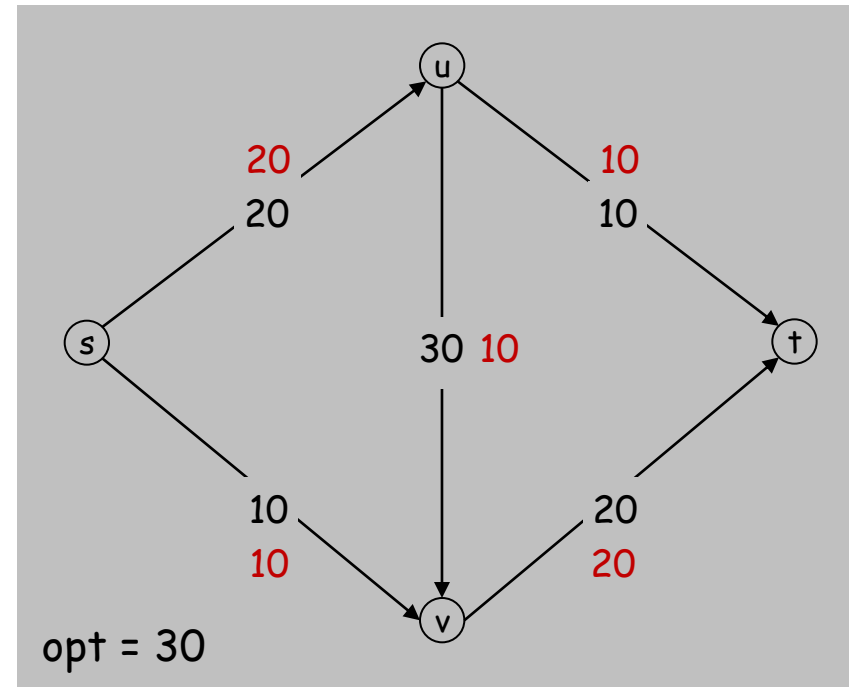
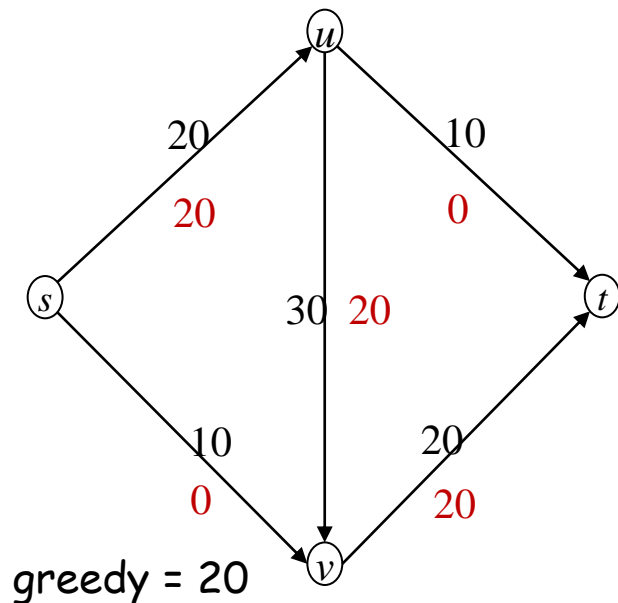
L'algoritmo greedy si fermerebbe, ma...

Verso un algoritmo per il Flusso massimo

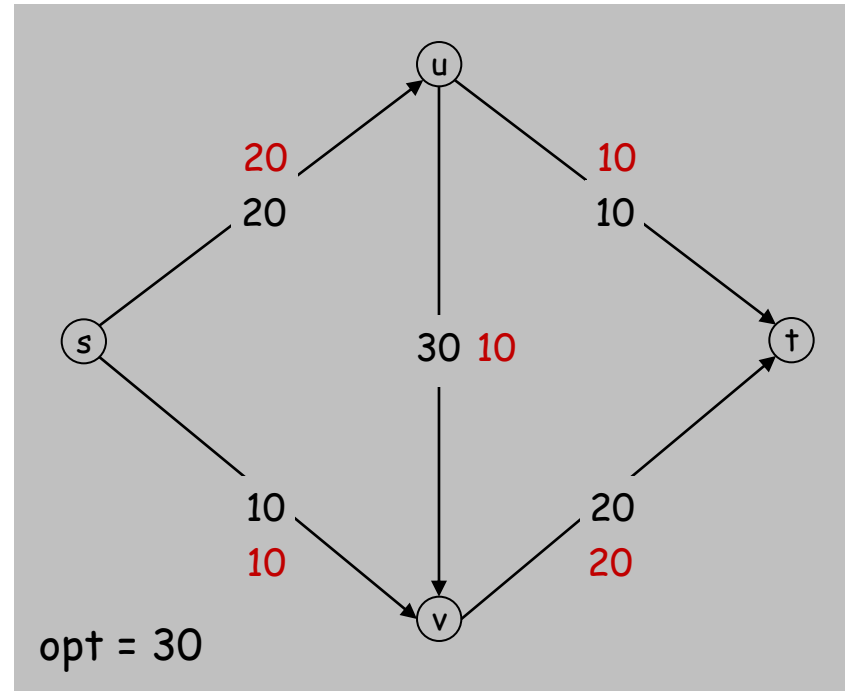
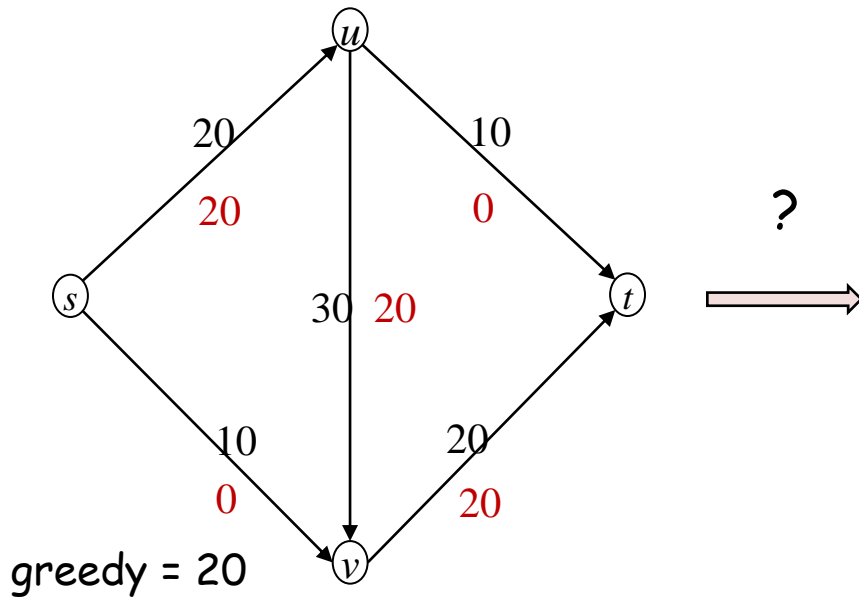
Algoritmo greedy.

- Iniziare con $f(e) = 0$ per ogni arco $e \in E$.
- Trovare un cammino s - t P in cui ogni arco ha $f(e) < c(e)$.
- Aumentare il flusso lungo il cammino P .
- Ripetere finchè è possibile.

← Ottimalità locale $\not\Rightarrow$ ottimalità globale



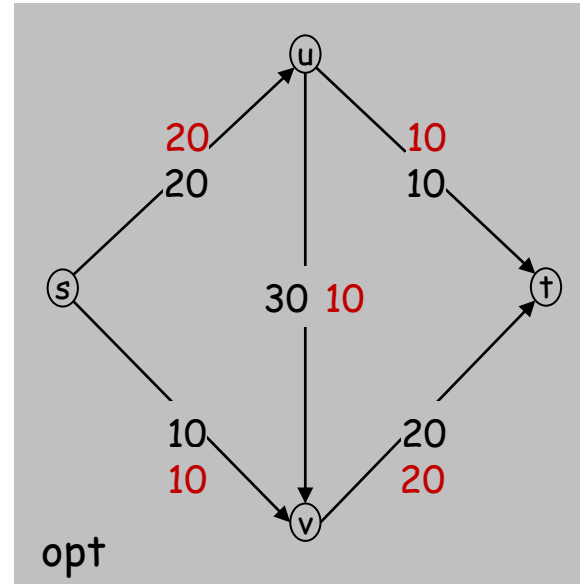
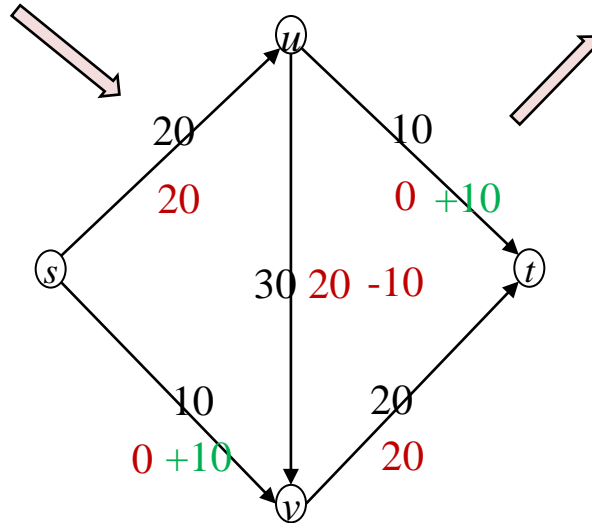
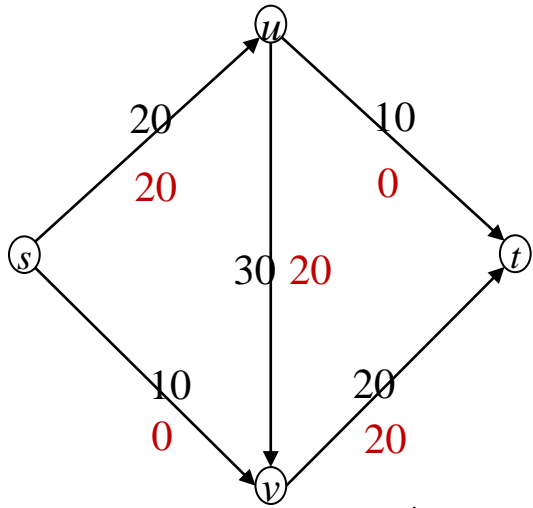
Come ottenere la soluzione ottima?



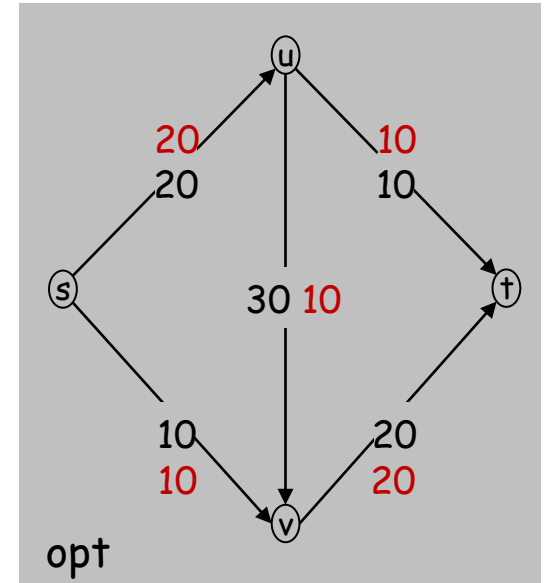
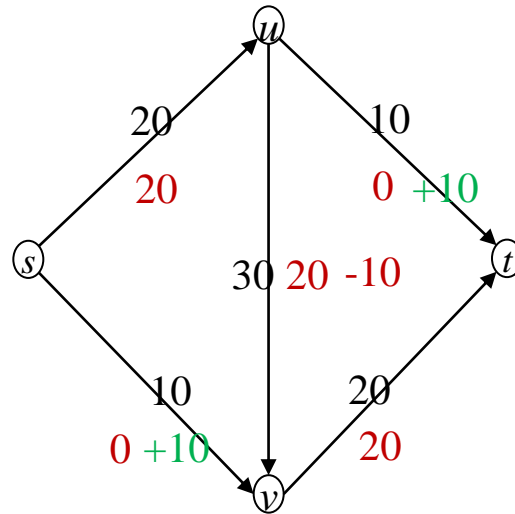
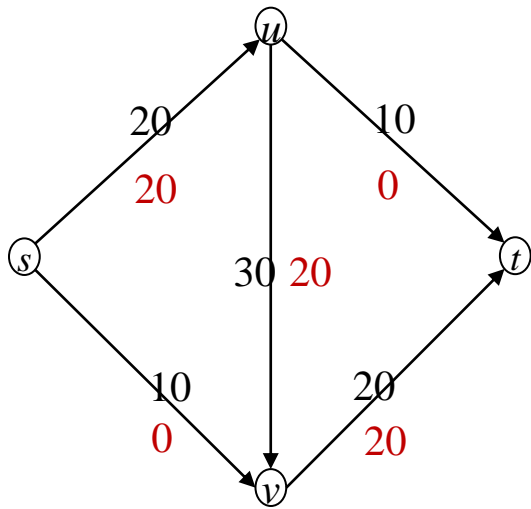
Se volessimo **aggiungere** 10 unità su (s,v) , violeremmo la proprietà di conservazione in v .

Per ristabilirla potremmo **togliere** 10 unità (delle 20) entranti in v sull'arco (u,v) , ovvero potremmo fare scorrere 10 unità "contro-senso" da v ad u ed infine **aggiungere** le 10 unità da u a t .

Come ottenere la soluzione ottima?



Verso un algoritmo per il Flusso massimo



Definiamo un nuovo grafo che tenga conto di 2 aspetti:

Per ogni arco $e = (u, v)$ con $f(e) < c(e)$, ci sono altre $c(e) - f(e)$ unità disponibili da poter fare passare da u a v (in avanti)

Per ogni arco $e = (u, v)$ con $f(e) > 0$ ci sono $f(e)$ unità che possiamo togliere/disfare facendo passare del flusso da v a u (indietro)

Grafo residuale

Dato un grafo $G=(V,E)$ e un flusso f :

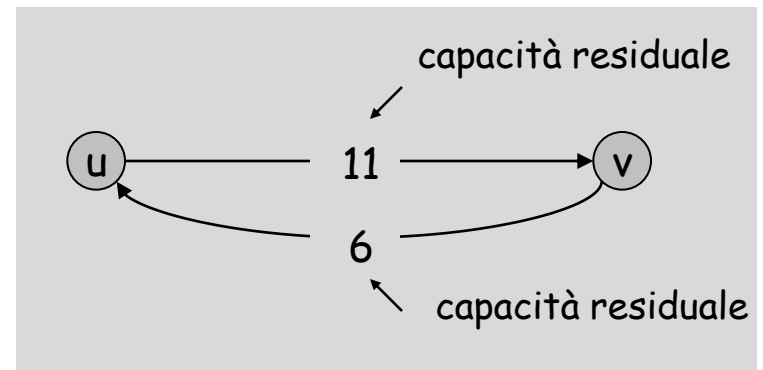
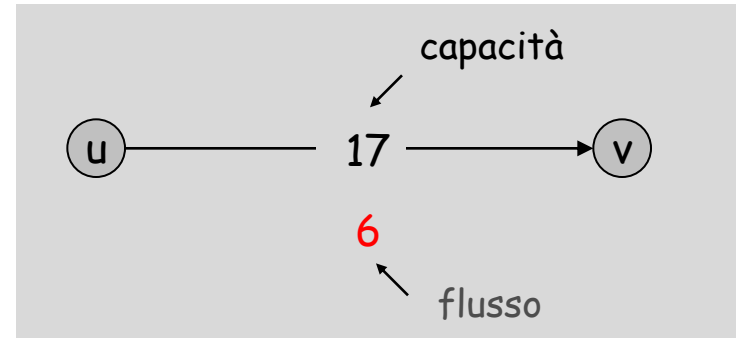
Arco originario: $e = (u, v) \in E$.

- flusso $f(e)$, capacità $c(e)$.

Arco residuale.

- $e = (u, v)$ e $e^R = (v, u)$.
- Capacità residuale:

$$c_f(e) = \begin{cases} c(e) - f(e) & \text{if } e \in E \\ f(e) & \text{if } e^R \in E \end{cases}$$



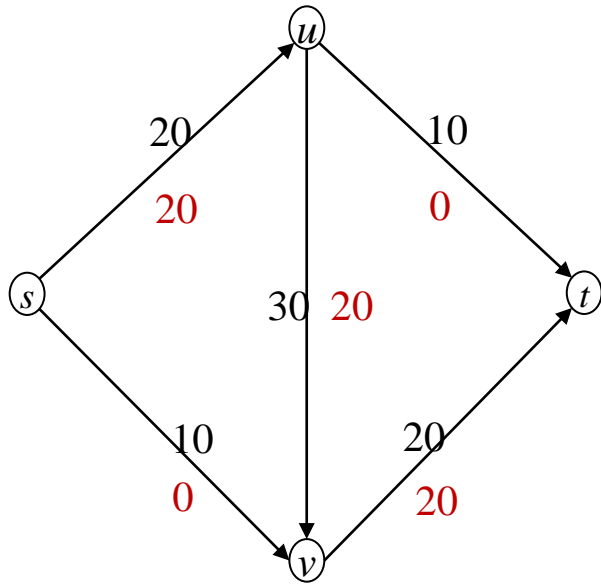
Grafo residuale : $G_f = (V, E_f)$.

- Archi residuali con capacità residuale positiva.
- $E_f = \{e : f(e) < c(e)\} \cup \{e^R : f(e) > 0\}$

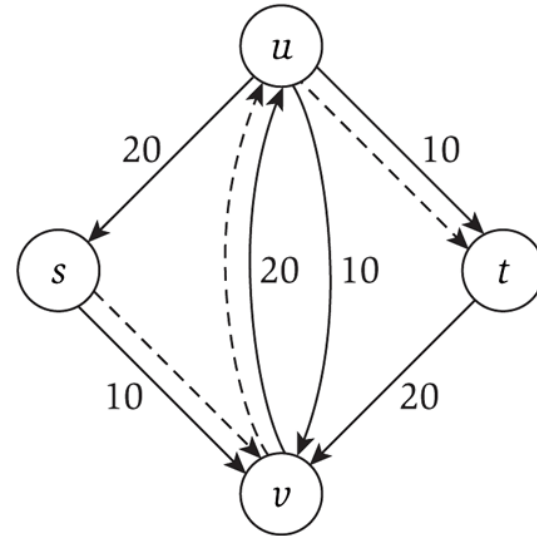
Cammino aumentante: cammino semplice da s a t in G_f

Grafo residuale: esempio

Grafo G con capacità e **flusso** f



Grafo residuale G_f



$P=s-v-u-t$ è un **cammino aumentante**

Posso mandare al più $b=10$ unità di flusso lungo P :

$$f((s,v)) = 0 + 10$$

$$f((u,v)) = 20 - 10$$

$$f((u,t)) = 0 + 10$$

Algoritmo del cammino aumentante

Sia P un cammino semplice s - t in G_f

Minima capacità residuale
di un arco di P

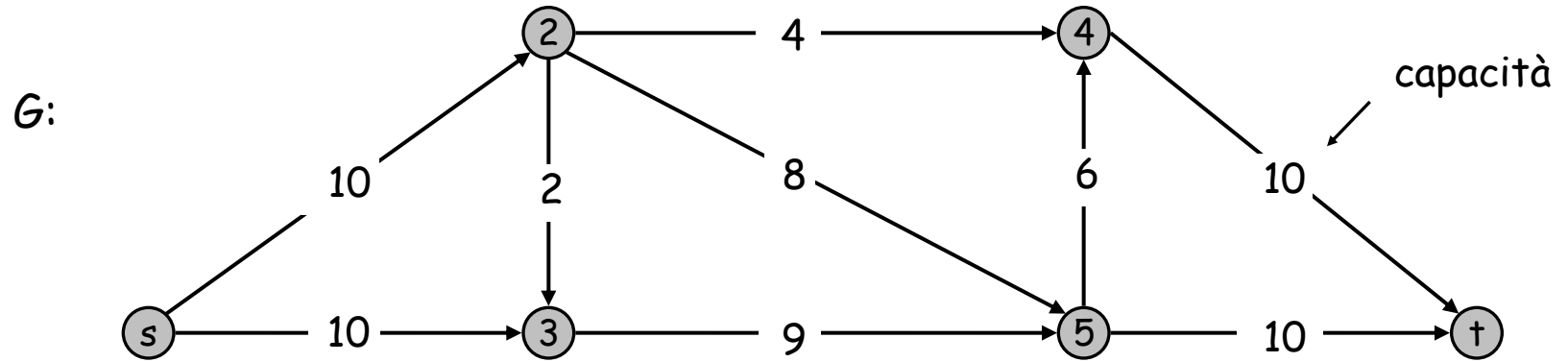
```
Augment(f, c, P) {  
  b ← bottleneck(P, f)  
  foreach e ∈ P {  
    if (e ∈ E) in G: f(e) ← f(e) + b  
    else      in G: f(eR) ← f(eR) - b  
  }  
  return f  
}
```

Arco in avanti

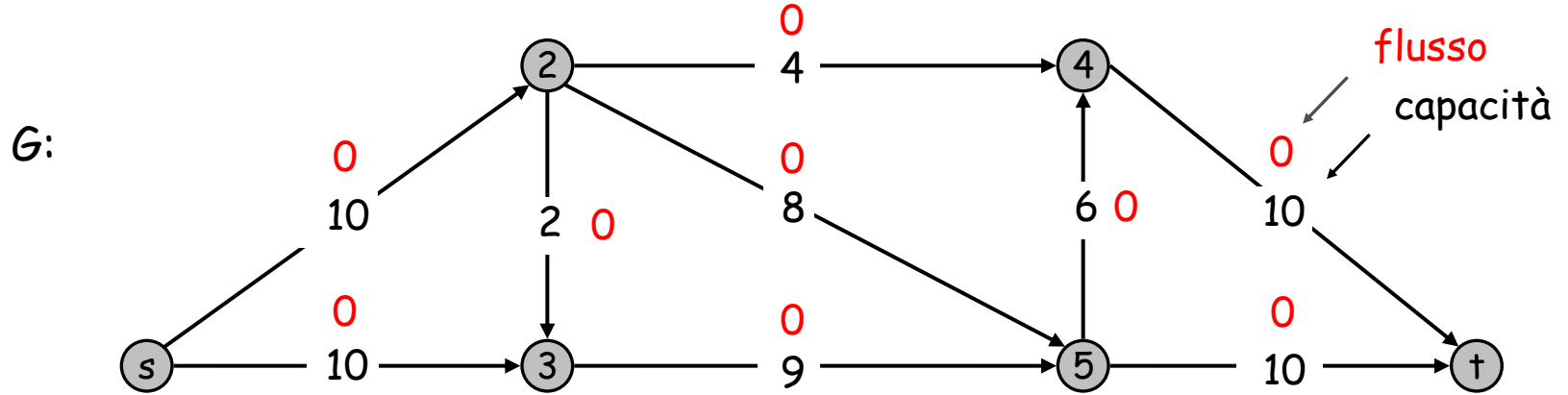
Arco inverso

```
Ford-Fulkerson(G, s, t, c) {  
  foreach e ∈ E f(e) ← 0  
  Gf ← grafo residuale  
  
  while (esiste un cammino aumentante P in Gf) {  
    f ← Augment(f, c, P)  
    aggiorna Gf  
  }  
  return f  
}
```

Algoritmo di Ford-Fulkerson



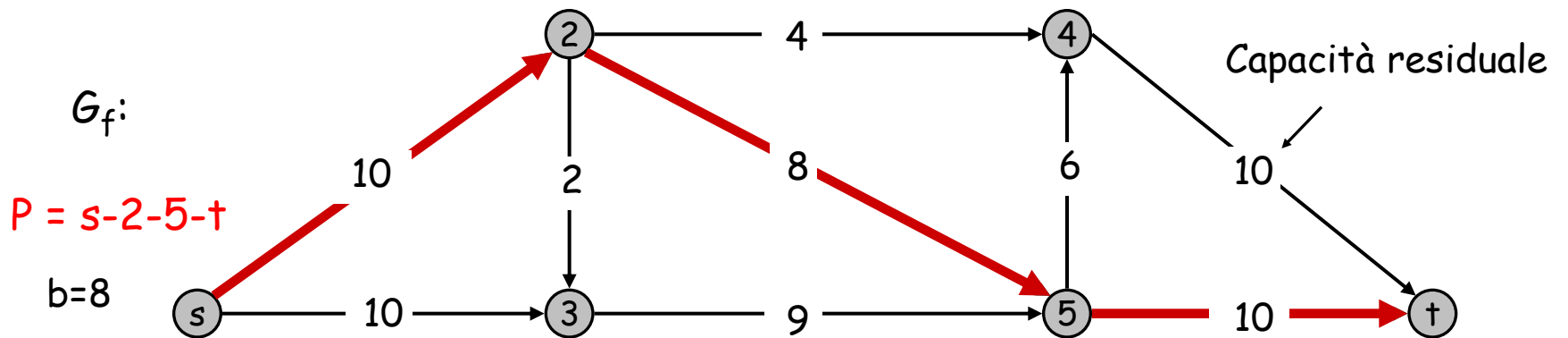
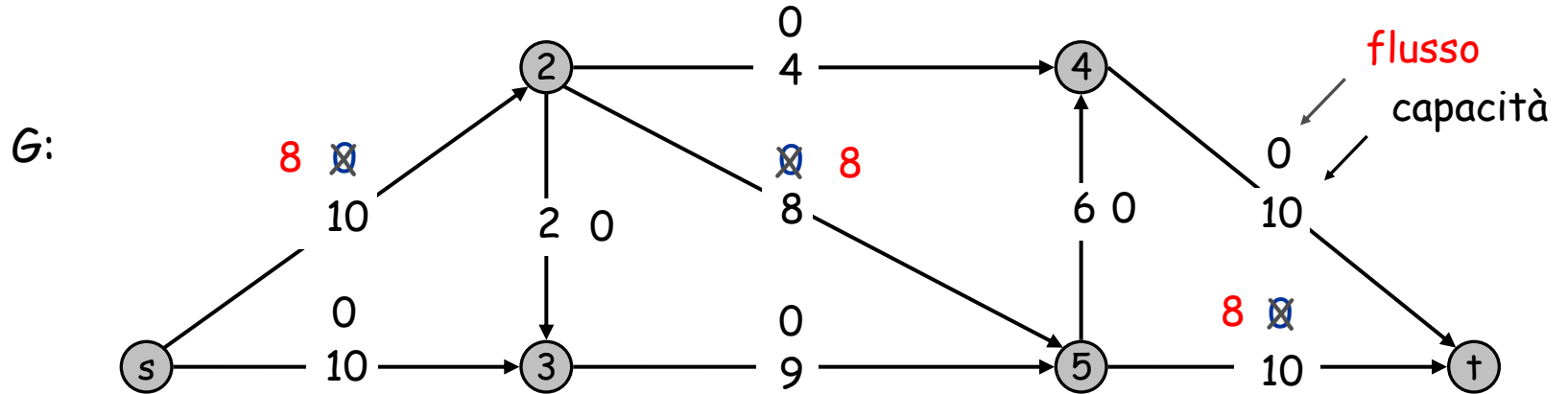
Algoritmo di Ford-Fulkerson



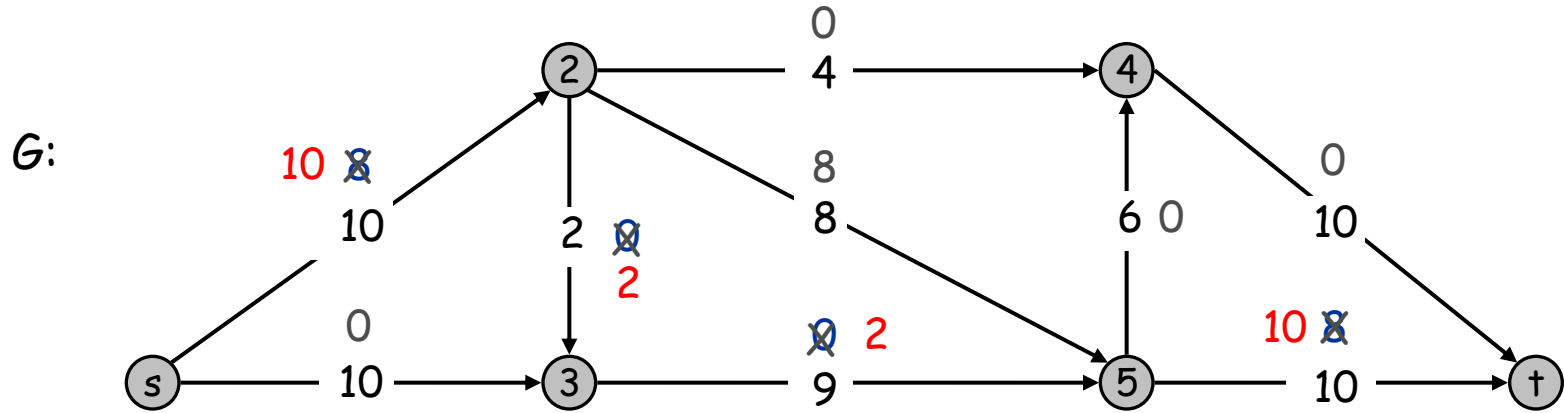
valore flusso = 0

$$G_f = G$$

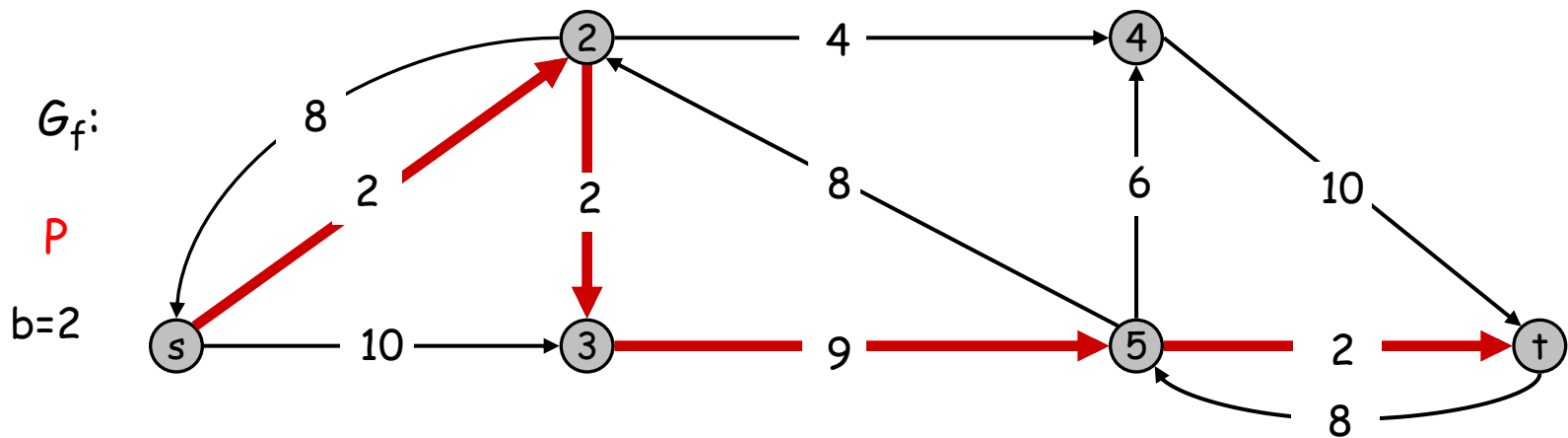
Algoritmo di Ford-Fulkerson



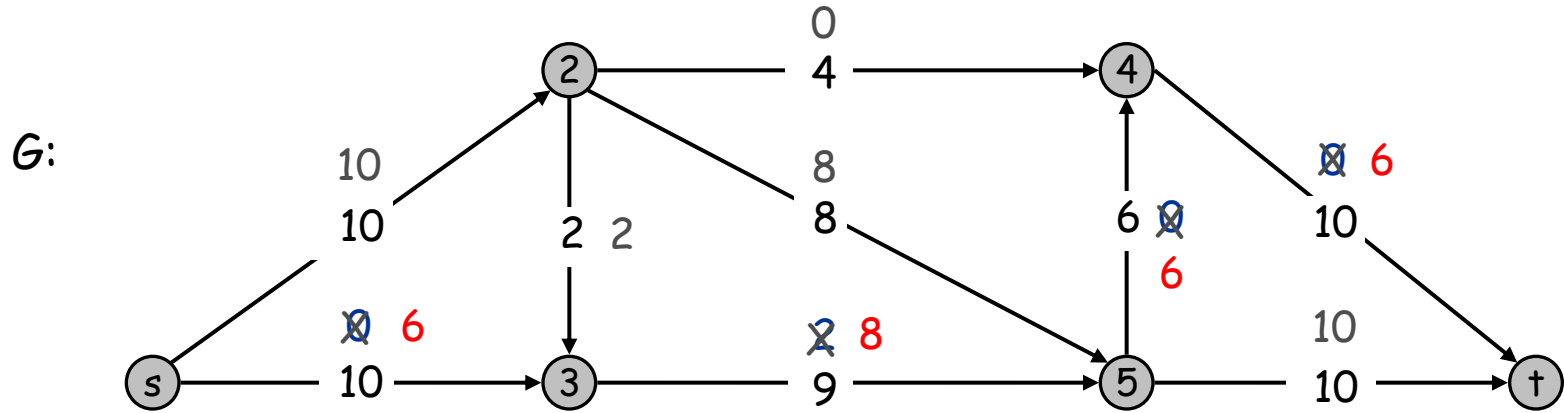
Algoritmo di Ford-Fulkerson



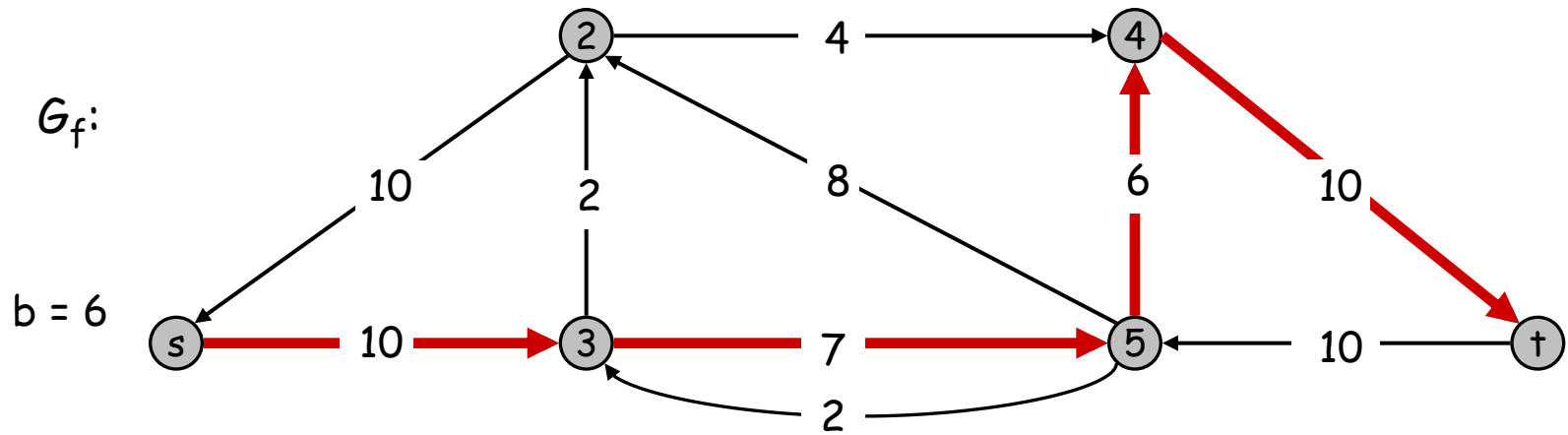
valore flusso = \otimes 10



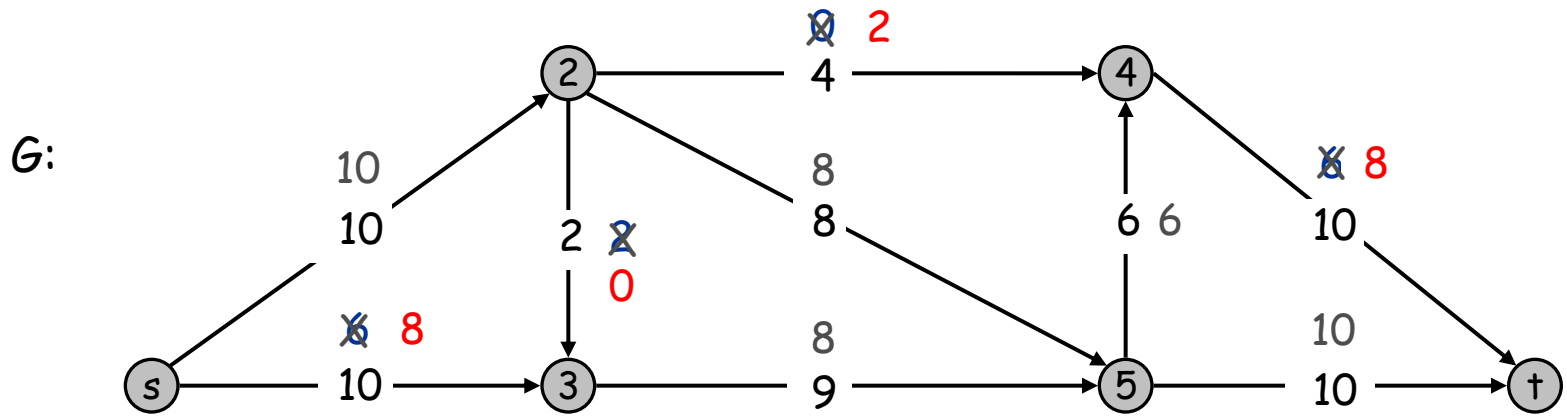
Algoritmo di Ford-Fulkerson



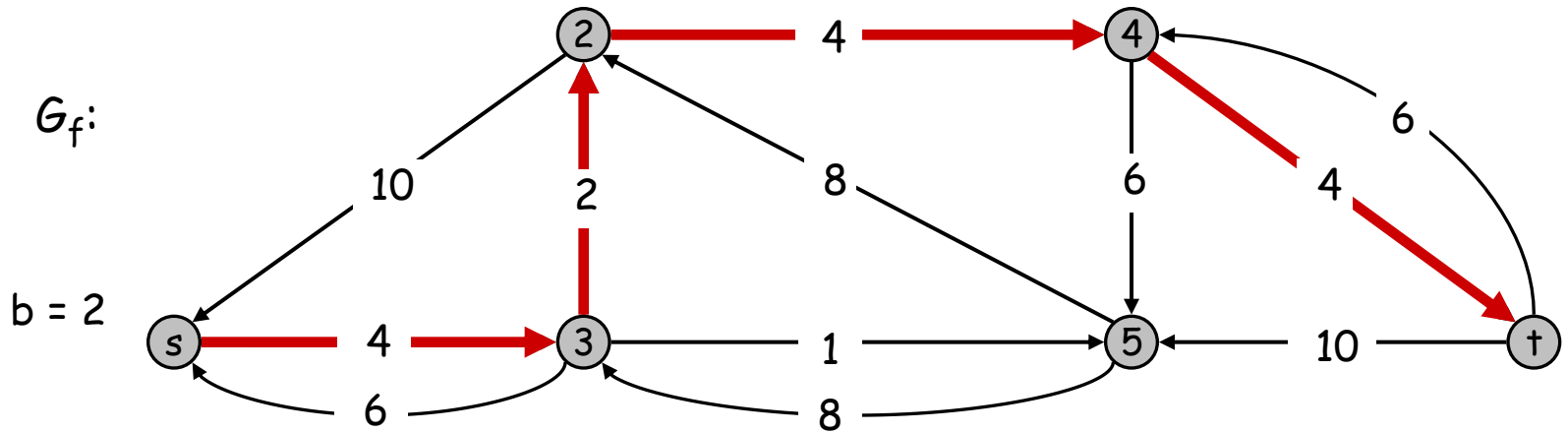
valore flusso = ~~10~~ 16



Algoritmo di Ford-Fulkerson

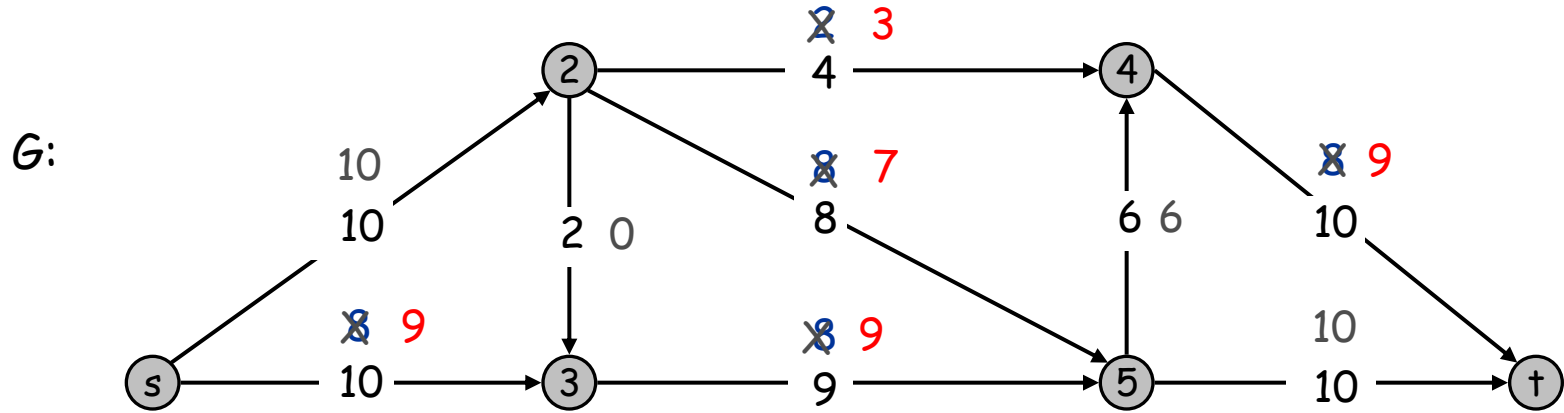


valore flusso = ~~16~~ 18

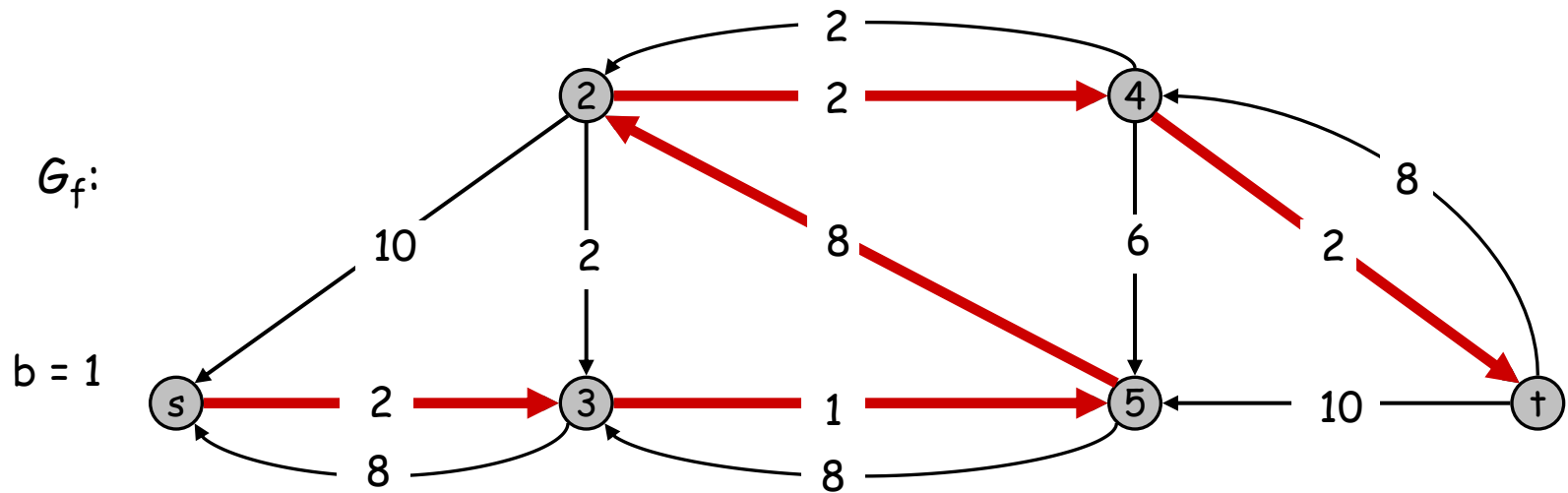


Nota: l'arco $(3,2)$ in G_f è un arco indietro: $(2,3)$ è in G , quindi $f((2,3)) \leftarrow f((2,3)) - b$

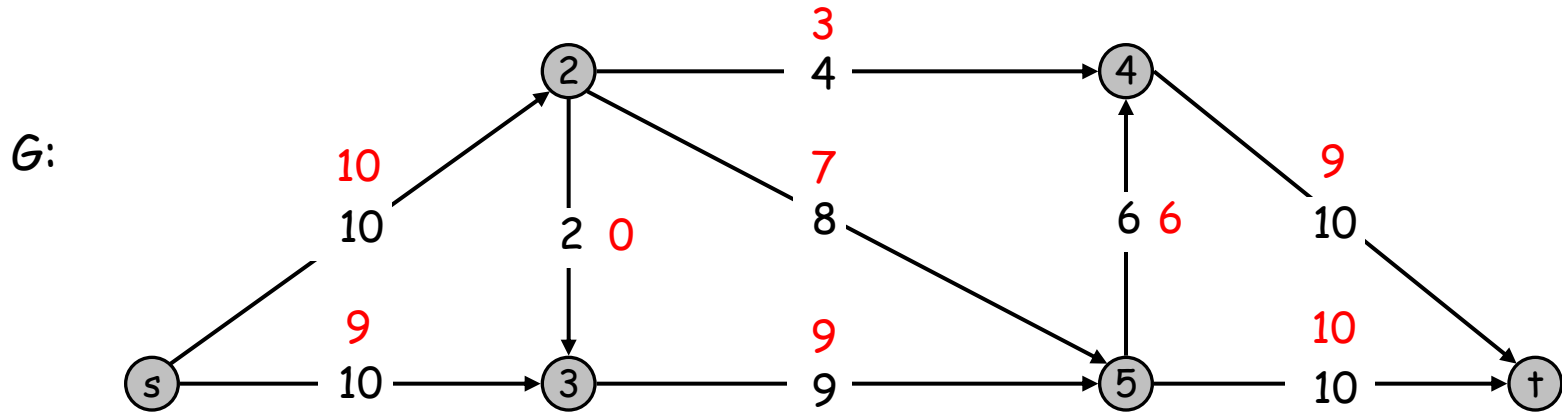
Algoritmo di Ford-Fulkerson



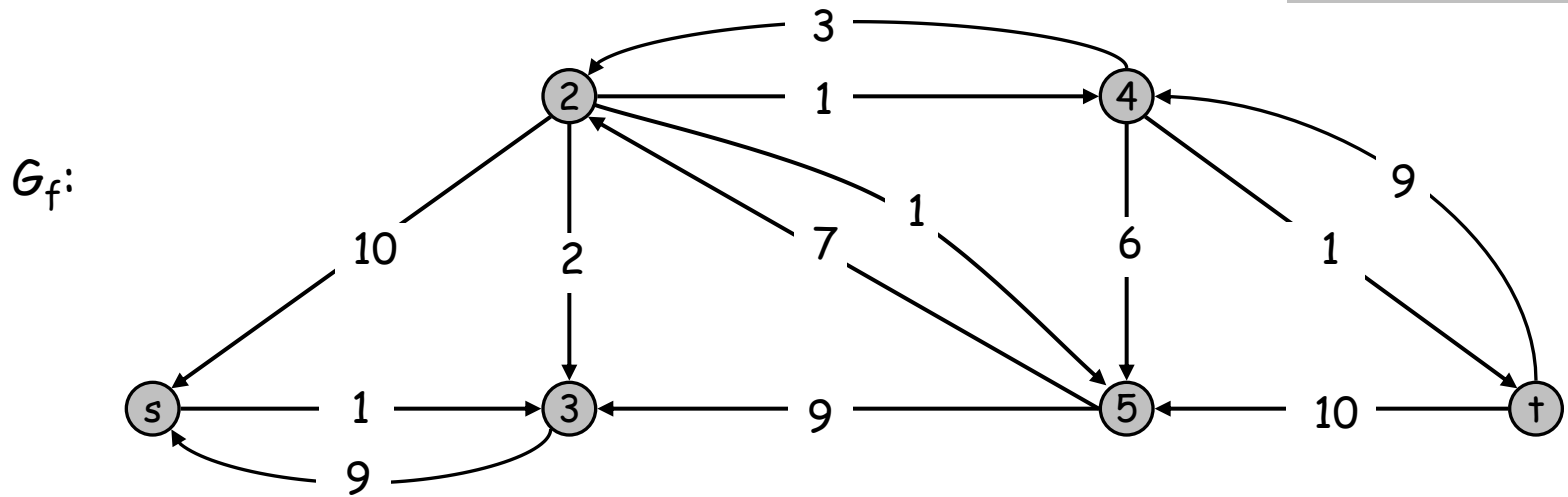
valore flusso = ~~18~~ 19



Algoritmo di Ford-Fulkerson



valore flusso = 19



Non ci sono più cammini aumentanti P in G_f : l'algoritmo termina.

Algoritmo di Ford-Fulkerson: correttezza

Vogliamo provare che il flusso fornito dall'algoritmo di Ford-Fulkerson è il massimo possibile.

Una limitazione superiore al flusso è data da $C = \sum_{e \text{ esce da } s} c(e)$

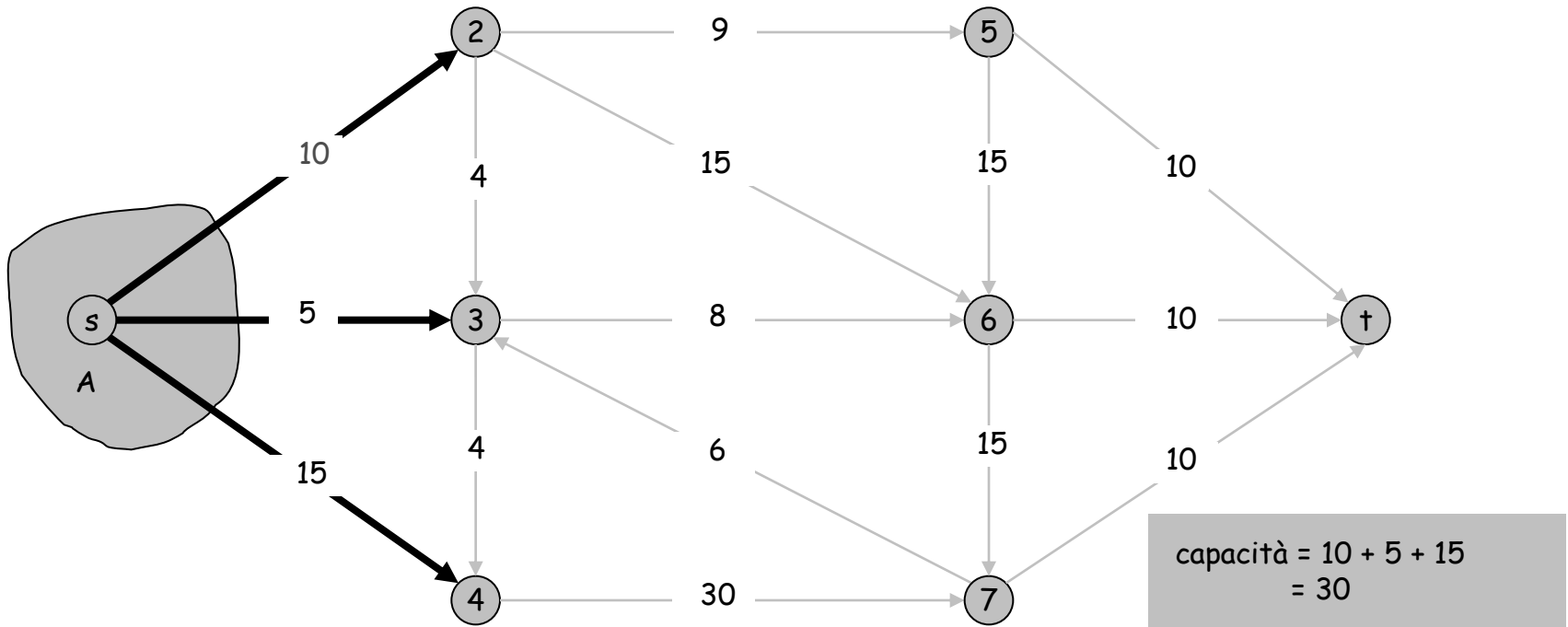
Infatti $v(f) = \sum_{e \text{ esce da } s} f(e) \leq \sum_{e \text{ esce da } s} c(e) = C$

Una limitazione superiore al flusso **più utile** si può ottenere introducendo il concetto di **taglio** in una rete di flusso.

Taglio

Def. Un **taglio s-t** è una partizione (A, B) di V con $s \in A$ e $t \in B$.

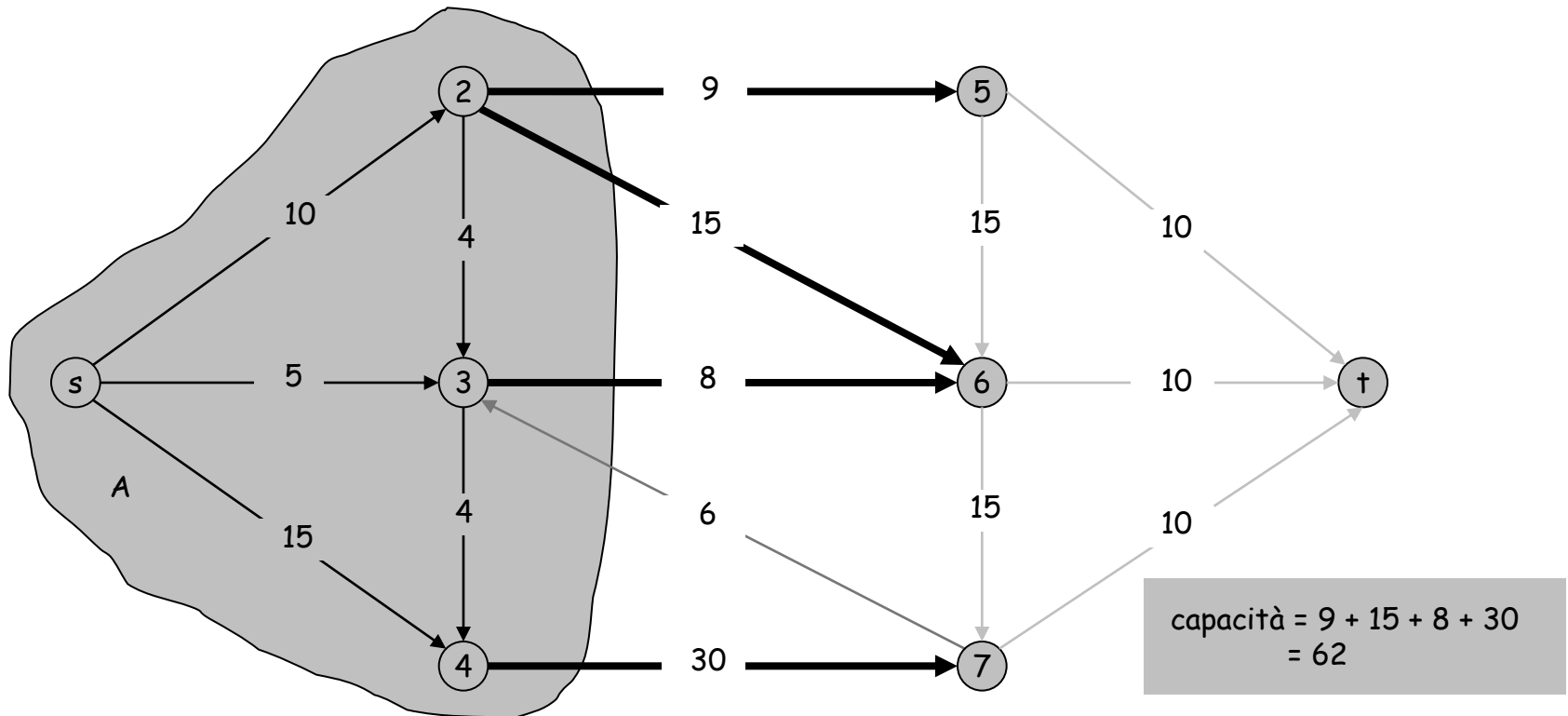
Def. La **capacità** di un taglio (A, B) è: $cap(A, B) = \sum_{e \text{ esce da } A} c(e)$



Taglio

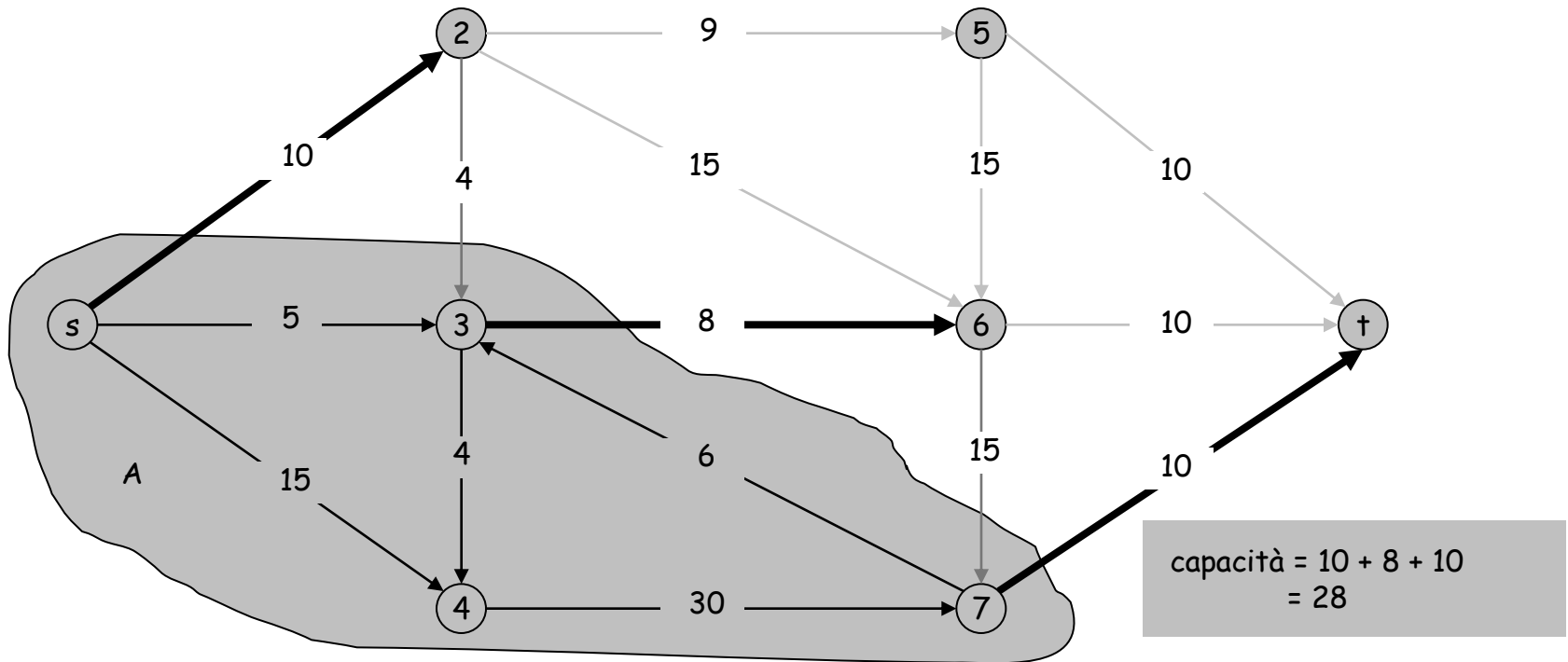
Def. Un **taglio s-t** è una partizione (A, B) di V con $s \in A$ e $t \in B$.

Def. La **capacità** di un taglio (A, B) è: $cap(A, B) = \sum_{e \text{ out of } A} c(e)$



Problema del taglio minimo

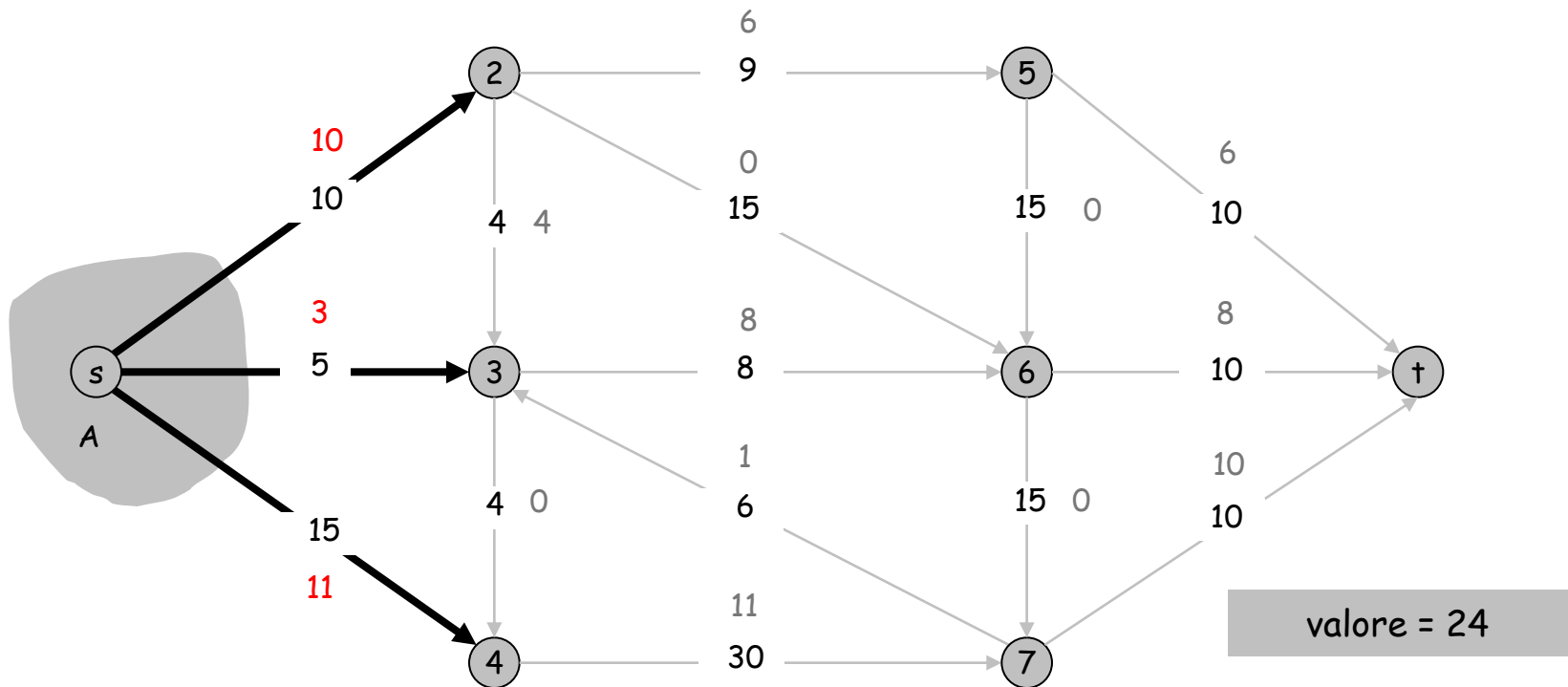
Problema del minimo taglio s-t. Trovare un taglio s-t di capacità minima in una rete di flusso.



Flusso e taglio

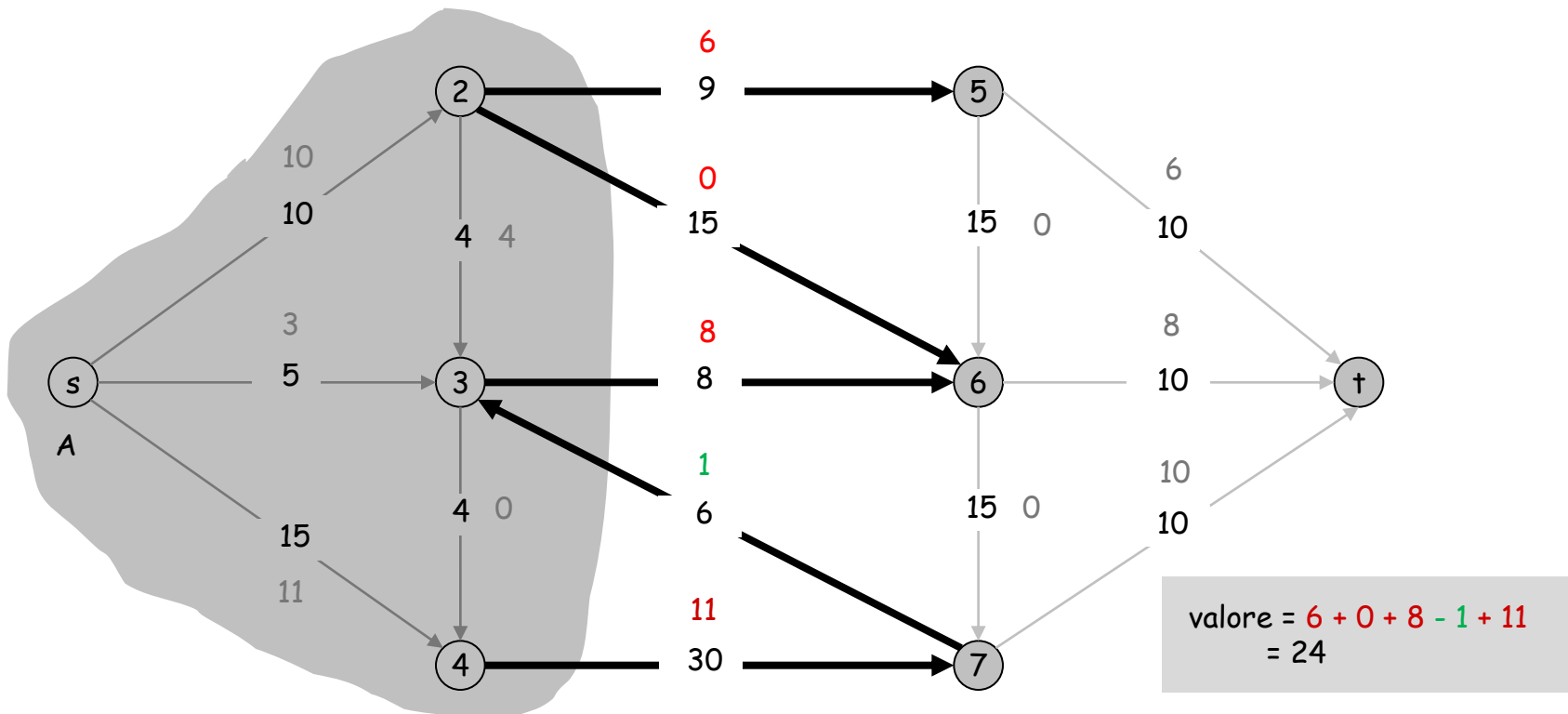
Lemma del valore del flusso. Sia f un flusso e sia (A, B) un taglio s-t. Allora, il **flusso netto** inviato attraverso il taglio è uguale alla quantità che lascia s .

$$\sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) = v(f)$$



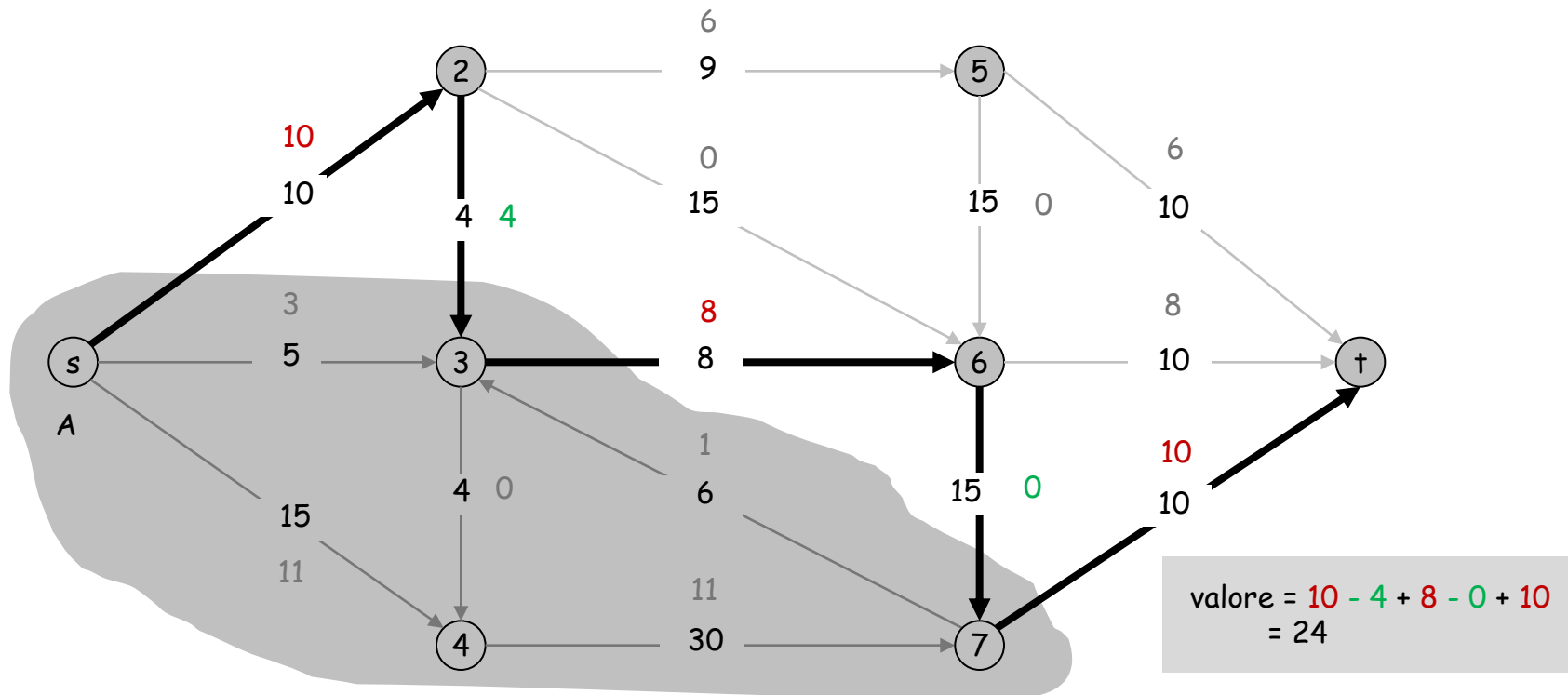
Flusso e taglio

Lemma del valore del flusso. Sia f un flusso e sia (A, B) un taglio s - t . Allora, il **flusso netto** inviato attraverso il taglio è uguale alla quantità che lascia s .



Flusso e taglio

Lemma del valore del flusso. Sia f un flusso e sia (A, B) un taglio s - t . Allora, il **flusso netto** inviato attraverso il taglio è uguale alla quantità che lascia s .



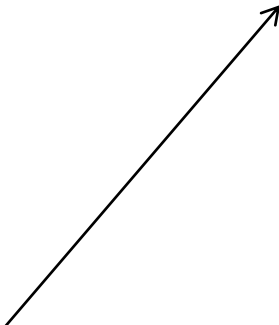
Flusso e taglio

Lemma del valore del flusso. Sia f un flusso e sia (A, B) un taglio s-t.

Allora

$$\sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) = v(f).$$

Dim.

$$\begin{aligned} v(f) &= \sum_{e \text{ out of } s} f(e) \\ &= \sum_{v \in A} \left(\sum_{e \text{ out of } v} f(e) - \sum_{e \text{ in to } v} f(e) \right) \\ &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e). \end{aligned}$$


Per la conservazione del flusso, tutti i termini eccetto $v = s$ sono 0. Inoltre il flusso entrante in s è 0.

Flusso e taglio

Proprietà di dualità debole (limitazione superiore al flusso).

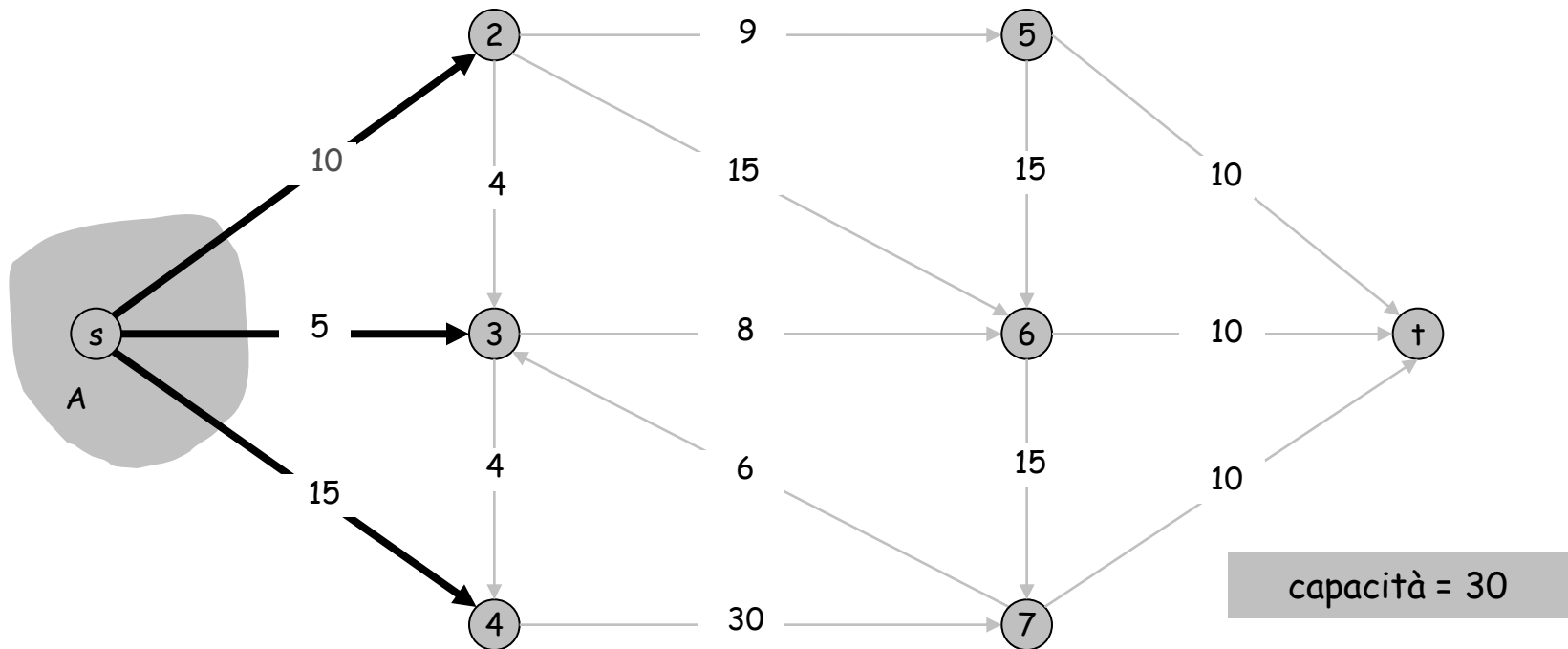
Sia f un flusso, e sia (A, B) un taglio s-t. Allora

$$v(f) \leq \text{cap}(A, B)$$

(ricorda $\text{cap}(A, B) = \sum_{e \text{ out of } A} c(e)$)

(il valore del flusso è al più la capacità di un qualsiasi taglio).

capacità taglio = 30 \Rightarrow valore flusso \leq 30



Flusso e taglio

Proprietà di dualità debole.

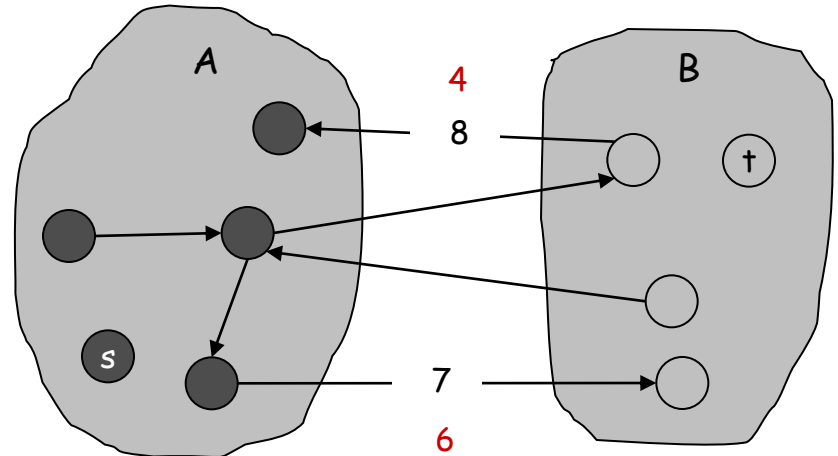
Sia f un flusso, e sia (A, B) un taglio s - t . Allora

$$v(f) \leq \text{cap}(A, B).$$

Dim.

Per il lemma del valore del flusso:

$$\begin{aligned} v(f) &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) \\ &\leq \sum_{e \text{ out of } A} f(e) \\ &\leq \sum_{e \text{ out of } A} c(e) \\ &= \text{cap}(A, B) \end{aligned}$$



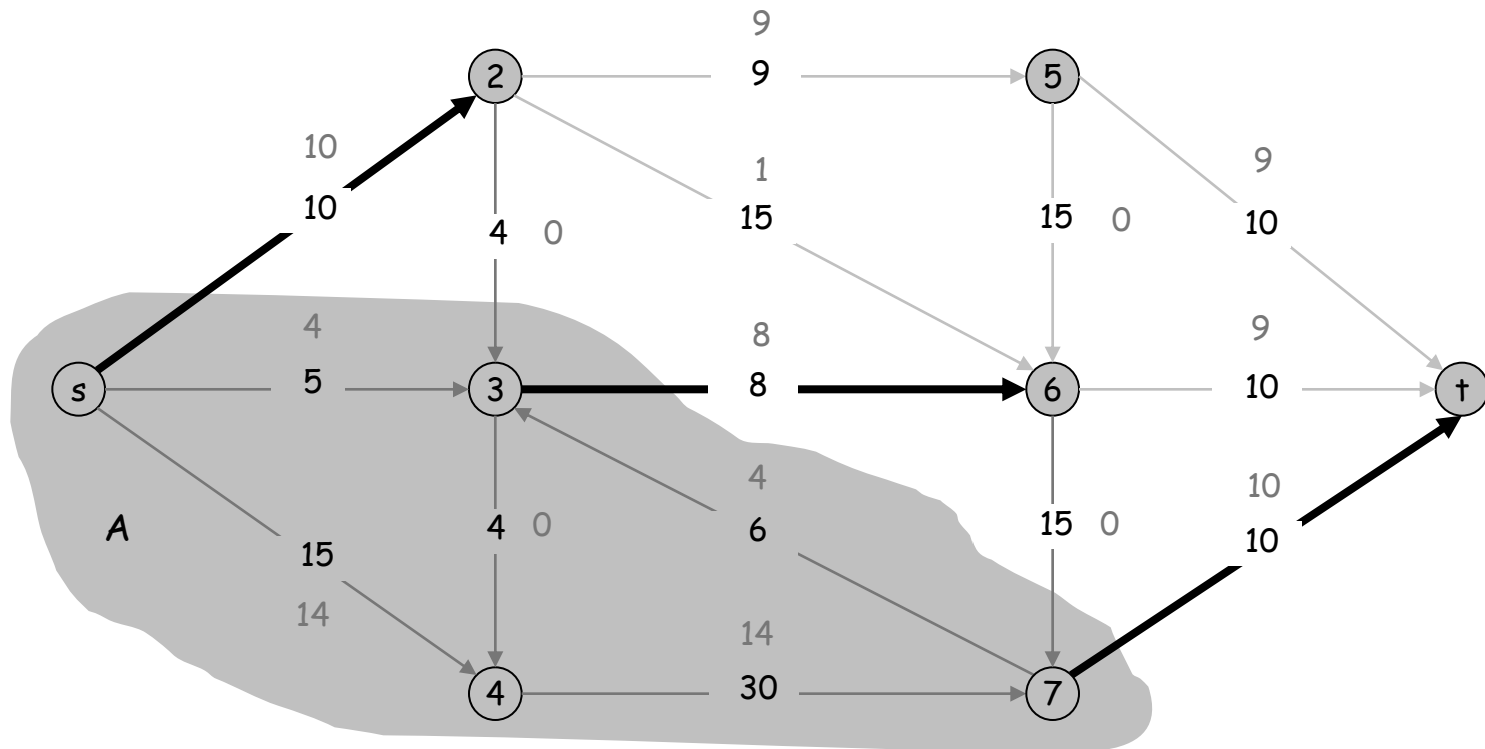
Certificato di ottimalità

Corollario Sia f un flusso, e sia (A, B) un taglio.

Se $v(f) = \text{cap}(A, B)$, allora f è un flusso massimo e (A, B) è un taglio minimo.

valore flusso = 28

capacità taglio = 28 \Rightarrow valore flusso \leq 28



Teorema Max-flusso Min-taglio

Teorema del cammino aumentante. Un flusso f è un flusso massimo sse non ci sono cammini aumentanti.

Teorema max-flusso min-taglio. [Ford-Fulkerson 1956] Il valore del flusso massimo è uguale al valore del minimo taglio.

Dim. Proviamo entrambi mostrando che sono equivalenti:

- (i) Esiste un taglio (A, B) tale che $v(f) = \text{cap}(A, B)$.
- (ii) f è un flusso massimo.
- (iii) Non ci sono cammini aumentanti nel grafo residuale G_f .

(i) \Rightarrow (ii) Questo è il corollario al lemma di dualità debole.

(ii) \Rightarrow (iii) Mostriamo per contrapposizione: non (iii) \Rightarrow non (ii).

Sia f un flusso. Se esiste un cammino aumentante, allora possiamo migliorare f mandando delle unità lungo quel cammino. Quello che ne risulta è ancora un flusso: valgono le proprietà di capacità e conservazione.

Prova del Teorema Max-flusso Min-taglio

(iii) \Rightarrow (i): Non ci sono cammini aumentanti relativi ad $f \Rightarrow$ Esiste un taglio (A, B) tale che $v(f) = \text{cap}(A, B)$.

- Sia f un flusso senza cammini aumentanti.
- Sia A^* l'insieme dei vertici raggiungibili da s nel grafo residuale e B^* tutti gli altri.
- Per definizione di A^* , $s \in A^*$.
- Per l'ipotesi (iii), $t \notin A^*$.

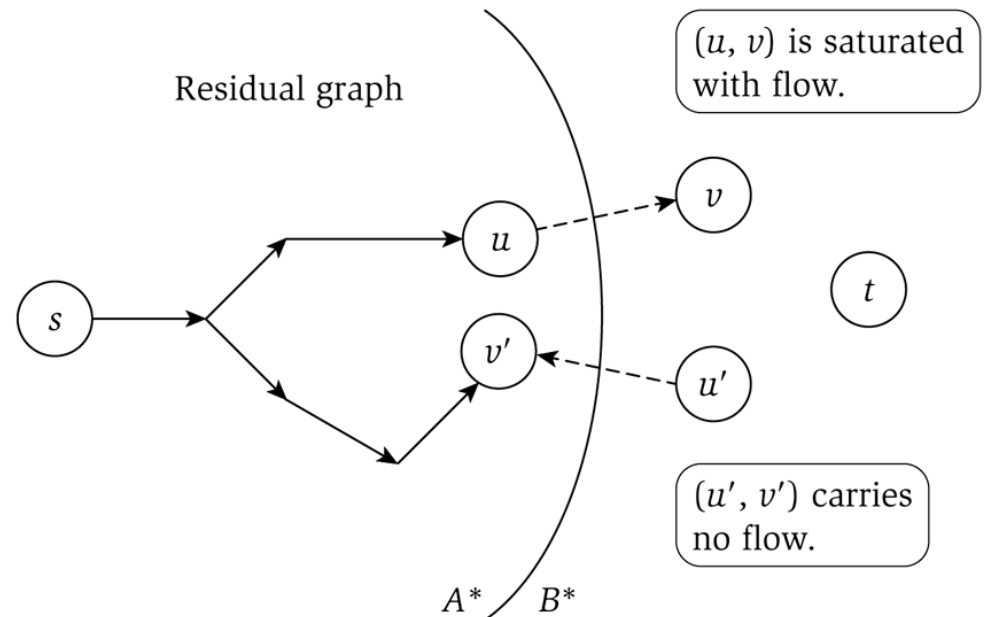
$$v(f) = \sum_{e \text{ out of } A^*} f(e) - \sum_{e \text{ into } A^*} f(e)$$

Quanto vale $f(e)$?

Per archi uscenti da A^* :

sia (u, v) tale che $u \in A^*$ e $v \in B^*$.
Allora $f(e) = c(e)$.

Se fosse $f(e) < c(e)$ allora ci sarebbe un arco (u, v) nel grafo residuale e quindi $v \in A^*$.



Prova del Teorema Max-flusso Min-taglio

(iii) \Rightarrow (i): Non ci sono cammini aumentanti relativi ad $f \Rightarrow$ Esiste un taglio (A, B) tale che $v(f) = \text{cap}(A, B)$.

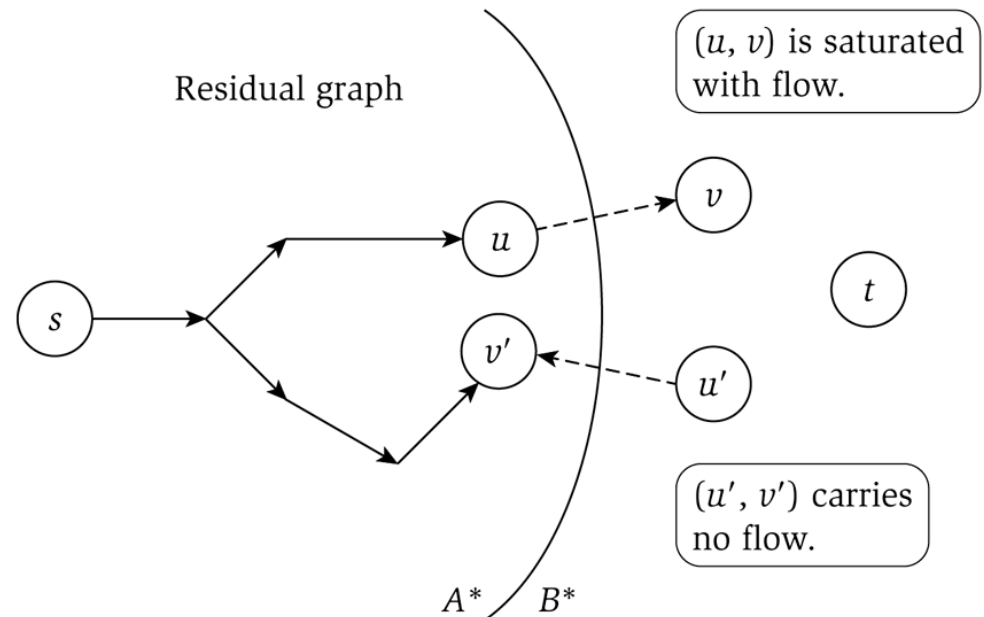
- Sia f un flusso senza cammini aumentanti.
- Sia A^* l'insieme dei vertici raggiungibili da s nel grafo residuale e B^* tutti gli altri.
- Per definizione di A^* , $s \in A^*$.
- Per l'ipotesi (iii), $t \notin A^*$.

$$v(f) = \sum_{e \text{ out of } A^*} f(e) - \sum_{e \text{ into } A^*} f(e)$$

quanto vale $f(e)$?

Per archi entranti in A^* :
sia (u', v') tale che $u' \in B^*$ e $v' \in A^*$.
Allora $f(e) = 0$.

Se fosse $f(e) > 0$ allora ci sarebbe un arco (v', u') nel grafo residuale e quindi $u' \in A^*$.

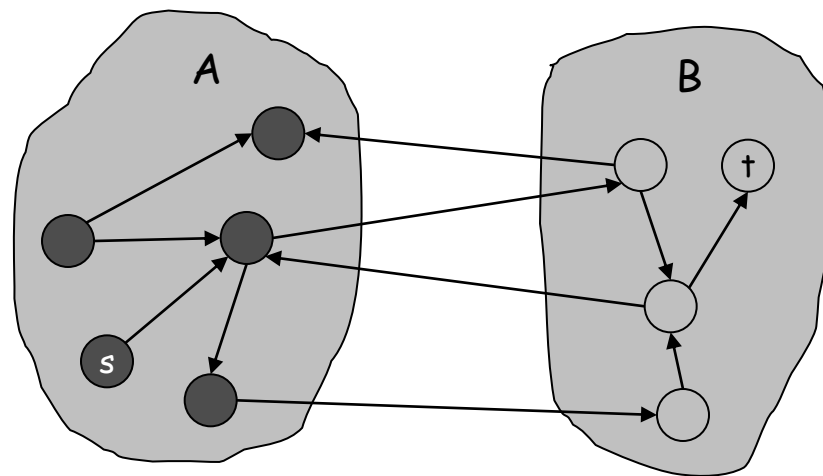


Prova del Teorema Max-flusso Min-taglio

(iii) \Rightarrow (i): Non ci sono cammini aumentanti relativi ad $f \Rightarrow$ Esiste un taglio (A, B) tale che $v(f) = \text{cap}(A, B)$.

- Sia f un flusso senza cammini aumentanti.
- Sia A^* l'insieme dei vertici raggiungibili da s nel grafo residuale.
- Per definizione di A^* , $s \in A^*$.
- Per l'ipotesi (iii), $t \notin A^*$.

$$\begin{aligned} v(f) &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) \\ &= \sum_{e \text{ out of } A} c(e) - 0 \\ &= \text{cap}(A, B) \end{aligned}$$



original network

Trovare taglio minimo

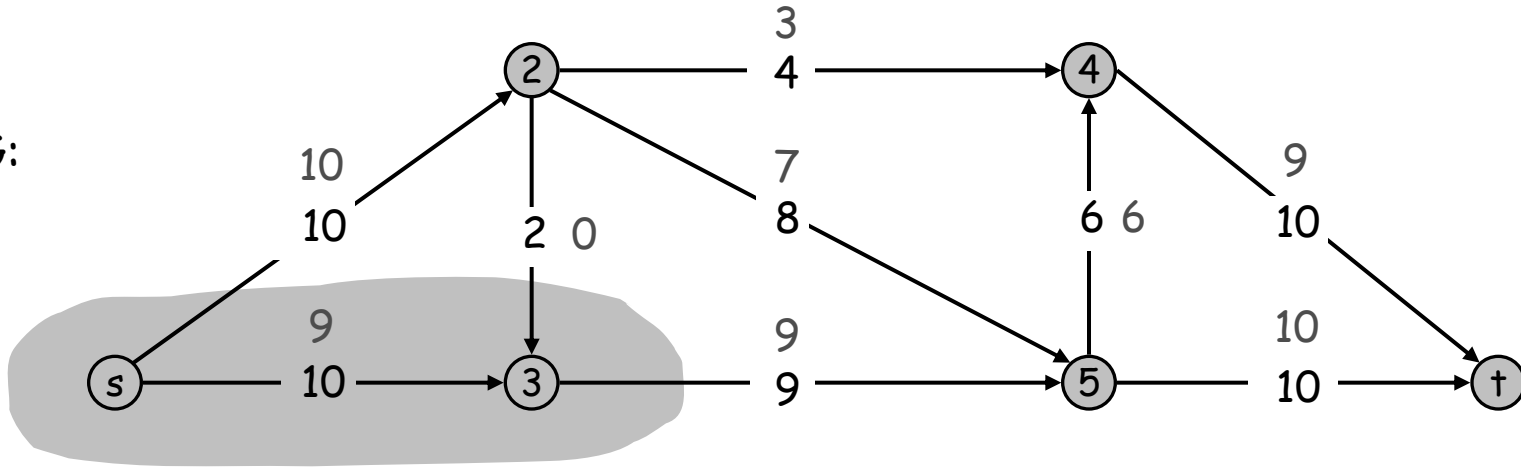
L'algoritmo di Ford-Fulkerson può essere usato anche per calcolare un taglio minimo in una rete di flusso:

- Eseguire l'algoritmo di Ford-Fulkerson
- Al termine considerare G_f e determinare l'insieme A^* dei vertici raggiungibili da s con una visita BFS o DFS.
- Restituire il taglio $(A^*, V \setminus A^*)$

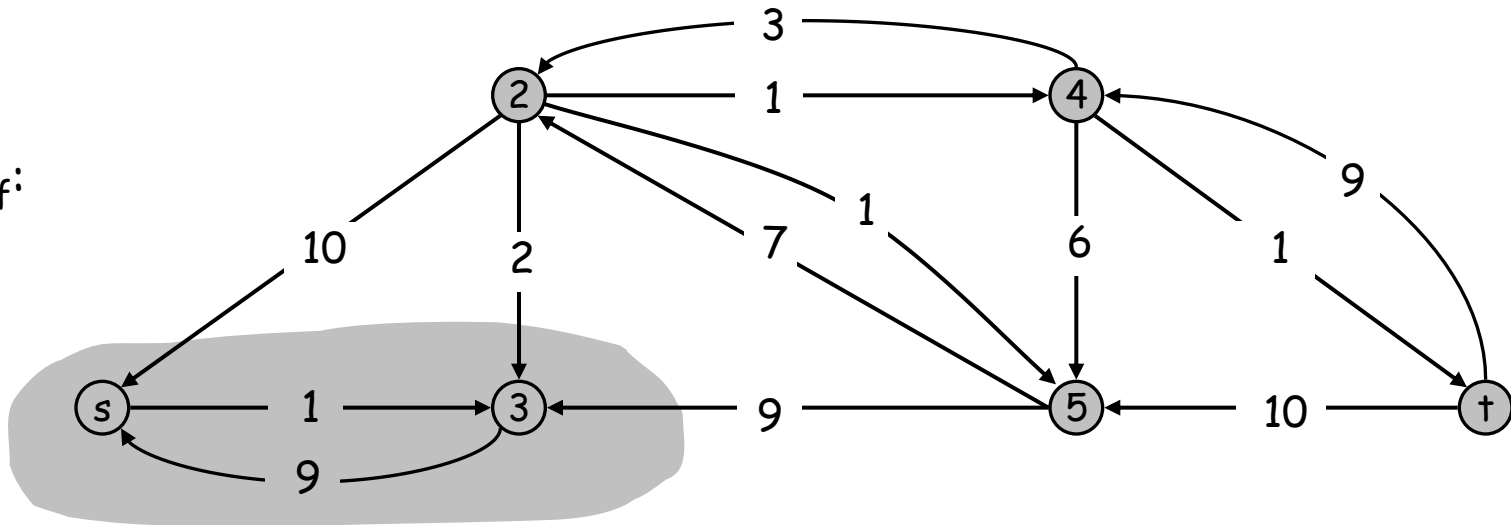
Correttezza: Prova del Teorema Max-flusso Min-taglio

Calcolo del taglio minimo

G :



G_f :



$$A^* = \{s, 3\} \text{ e } B^* = V \setminus A^*$$

$$\text{Cap}(A^*, B^*) = 19 = v(f)$$

Analisi dell'algoritmo di Ford-Fulkerson

Limitazione superiore al valore del flusso: $C = \sum_{e \text{ esce da } s} c(e)$

Infatti $v(f) = \sum_{e \text{ esce da } s} f(e) \leq \sum_{e \text{ esce da } s} c(e) = C$

Assunzione. Tutte le capacità sono **interi** fra 1 e C .

Invariante. Ogni valore del flusso $f(e)$ e ogni capacità residuale $c_f(e)$ restano interi durante l'esecuzione dell'algoritmo.

Teorema. L'algoritmo termina in al più $v(f^*) \leq C$ iterazioni
($f^* = \text{max flusso}$).

Prova.

Ogni cammino aumentante P incrementa il valore del flusso di almeno 1.

Il flusso **aumenta** perché il primo arco di P in G_f esce da s e P non ritorna in s ;
siccome non ci sono archi entranti in s in G il primo arco di P è in avanti.

Più precisamente il flusso aumenta ad ogni iterazione di $b = \text{bottleneck}(P, f)$.

Osservazione: Se le capacità non fossero interi, l'algoritmo potrebbe continuare all'infinito.

Complessità di tempo

Supponiamo che tutti i nodi abbiano almeno un arco incidente, quindi $m \geq n/2$ e $O(m+n) = O(m)$.

Corollario. Complessità tempo di Ford-Fulkerson è $O(mC)$.

Prova.

Al massimo $v(f^*) \leq C$ iterazioni

In ogni iterazione:

- troviamo un cammino aumentante in $O(m)$ mediante BFS o DFS nel grafo residuale.
- il grafo residuale ha $\leq 2m$ archi
- il grafo residuale rappresentato utilizzando le liste delle adiacenze (in e out)
- $\text{Augment}(f, c, P)$ ha complessità $O(n)$ (P ha al più $n-1$ archi)
- aggiornamento grafo residuale in $O(m)$ (per ogni arco costruiamo gli archi indietro e avanti opportuni)

Teorema Integralità. Se tutte le capacità sono numeri interi, allora vi è un flusso massimo f per cui ogni valore del flusso $f(e)$ è un intero.

Prova. L'algoritmo termina, quindi il teorema segue dall'invariante.

L'algoritmo di Ford-Fulkerson è pseudo-polinomiale

D. L' algoritmo di Ford-Fulkerson è polinomiale nella taglia dell'input?

$m, n, e \log C$

R. No: l'algoritmo può fare anche C iterazioni, a seconda della scelta del cammino aumentante e quindi un numero esponenziale ($C = 2^{\log C}$)

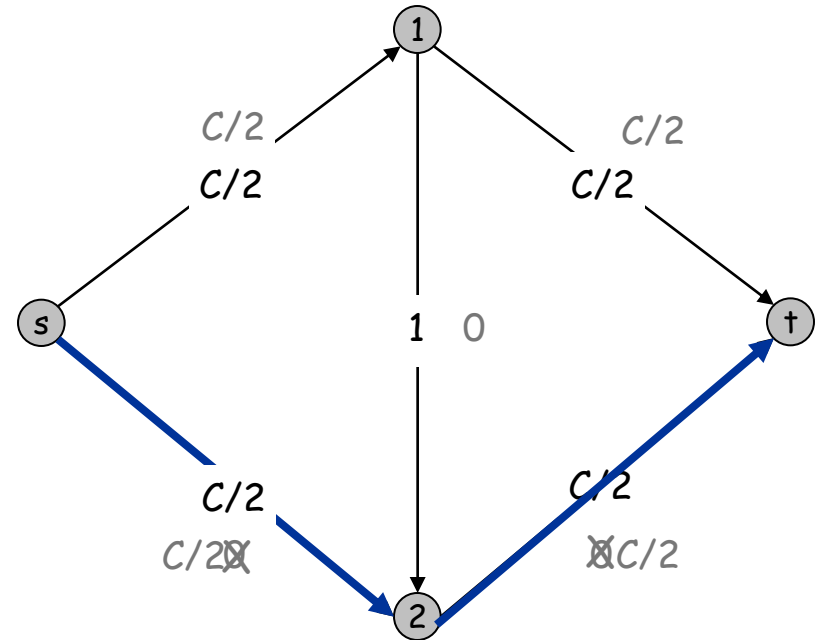
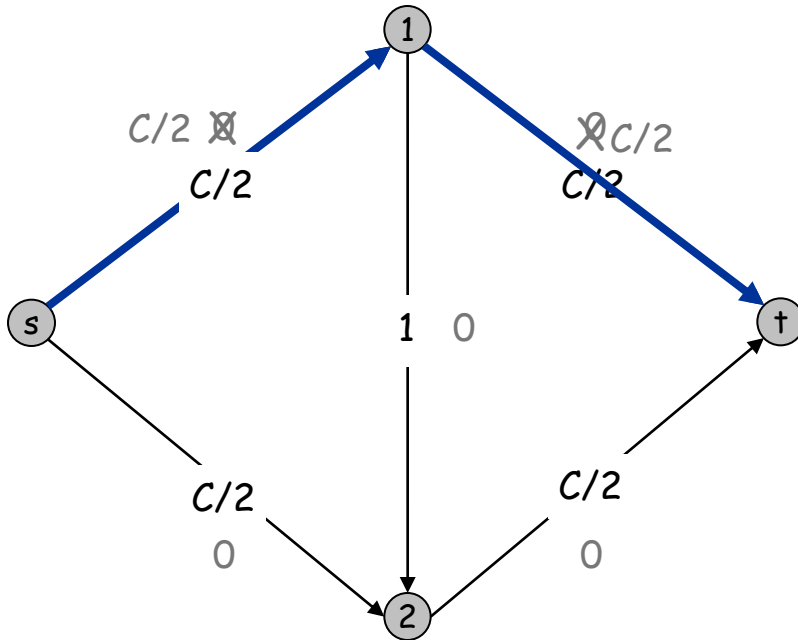
Esempio: prima scelta dei cammini

Esempio: massimo flusso = C con $f((1,2))=0$ e $f(e)=C/2$ per gli altri

Calcolabile in 2 iterazioni dell'algoritmo di Ford-Fulkerson scegliendo

$P_1 = s-1-t$

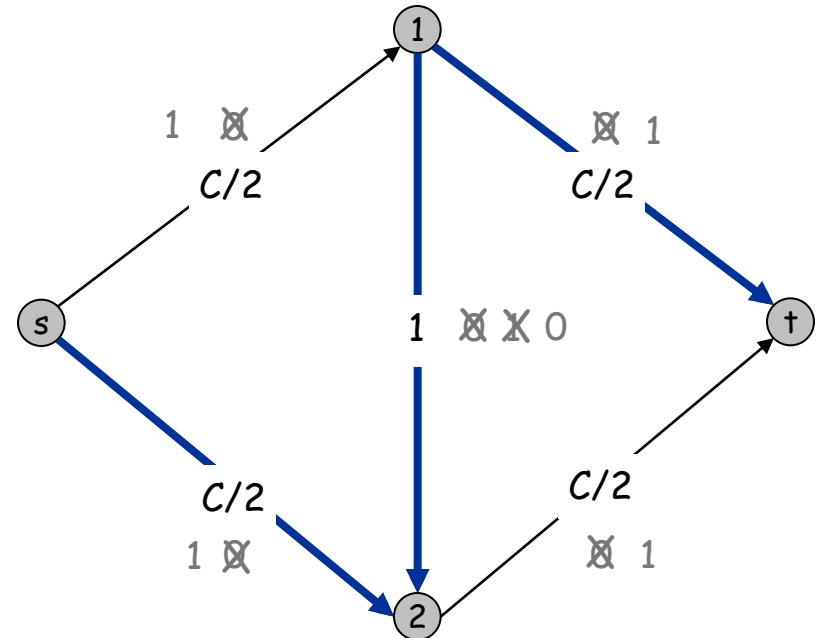
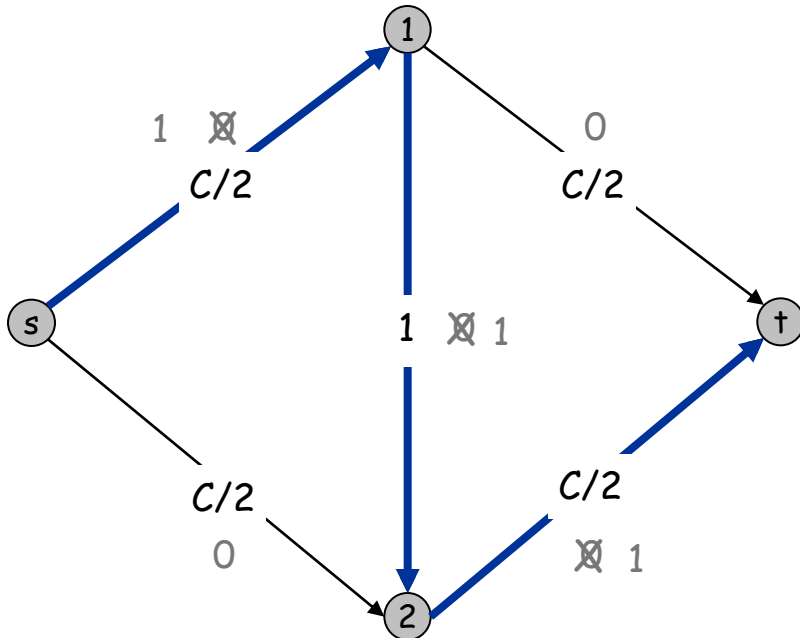
$P_2 = s-2-t$



Esempio: seconda scelta dei cammini

Esempio: massimo flusso = C con $f((1,2))=0$ e $f(e)=C/2$ per gli altri

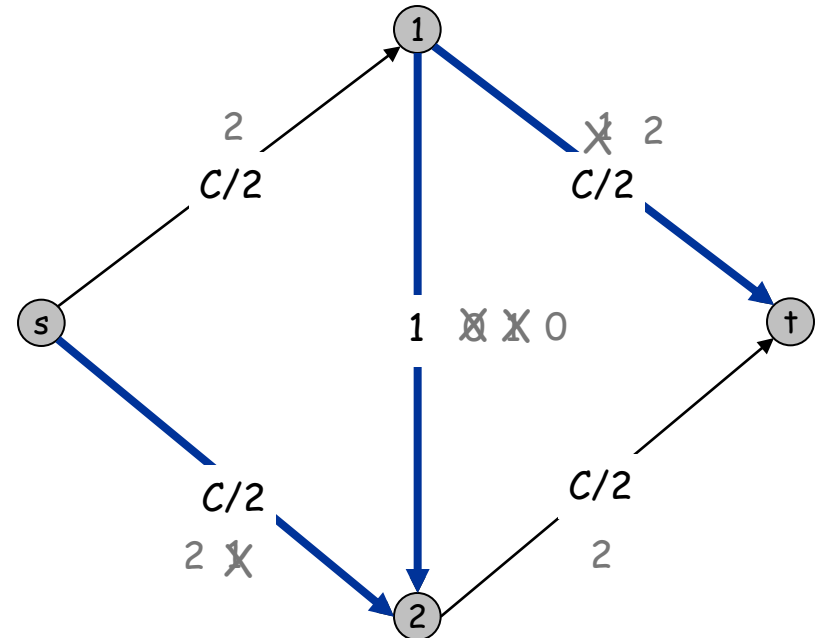
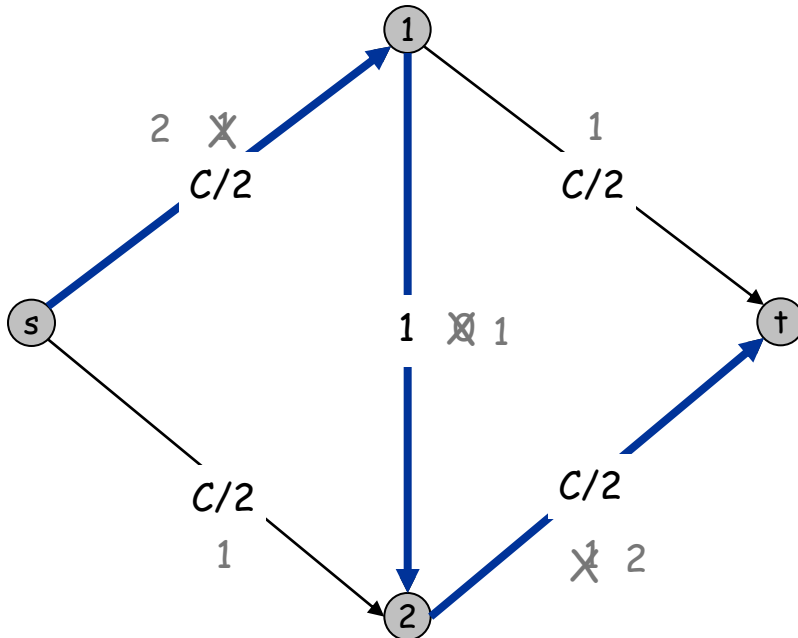
Calcolabile in C iterazioni dell'algoritmo di Ford-Fulkerson scegliendo $P_3 = s-1-2-t$ e $P_4 = s-2-1-t$ alternativamente per $C/2$ volte ognuno. In G_f compare $(1,2)$ o $(2,1)$ alternativamente.



Esempio: numero esponenziale di incrementi del flusso

Esempio: massimo flusso = C con $f((1,2))=0$ e $f(e)=C/2$ per gli altri

Calcolabile in C iterazioni dell'algoritmo di Ford-Fulkerson scegliendo $P_3 = s-1-2-t$ e $P_4 = s-2-1-t$ alternativamente per $C/2$ volte ognuno.
 In G_f compare $(1,2)$ o $(2,1)$ alternativamente.



Scegliere Buoni Cammini Aumentanti

Fare attenzione quando si scelgono i cammini aumentanti.

- Alcune scelte portano ad algoritmi esponenziali.
- Buone scelte portano ad algoritmi polinomiali.
- Se le capacità fossero irrazionali, l'algoritmo potrebbe non terminare!

Obiettivo: scegliere cammini aumentanti in modo tale che:

- Possiamo trovare cammini aumentanti efficientemente.
- Poche iterazioni.

Scegliere cammini aumentanti con: [Edmonds-Karp 1972, Dinitz 1970]

- Massima capacità bottleneck (però può richiedere molto tempo).
- Sufficientemente grande capacità bottleneck.

Complessità: $O(m^2 \log_2 C)$

Altro algoritmo che sceglie cammino con minor numero di archi

7.5 Matching Bipartito

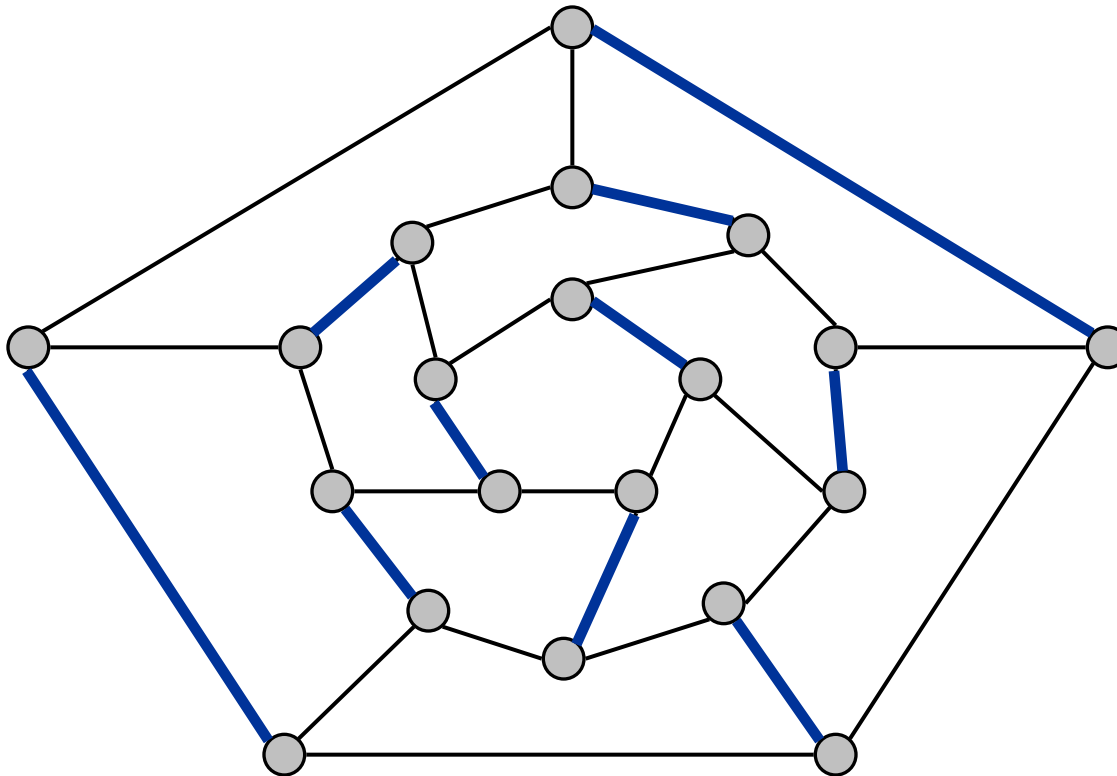
Un'applicazione del calcolo del flusso massimo

Matching

Matching.

- Input: grafo non-orientato $G = (V, E)$.
- $M \subseteq E$ è un **matching** se ogni nodo appare in al più un arco in M .

Problema del max matching: trovare un matching di cardinalità massima.



Matching bipartito

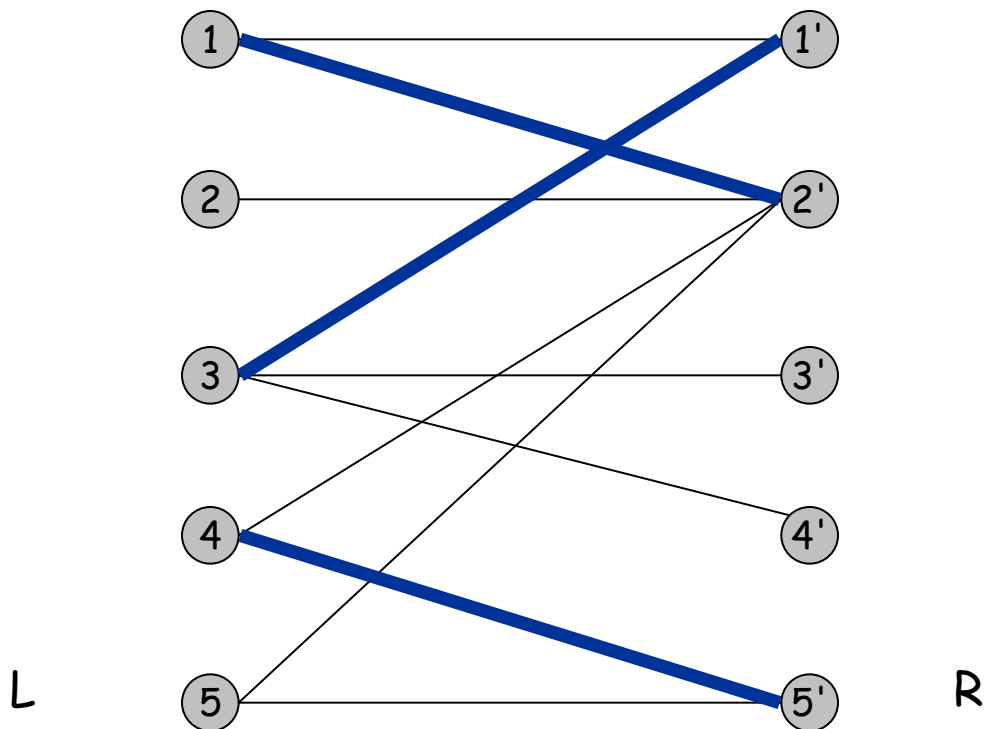
Ogni arco ha un estremo in L e l'altro in R



Matching bipartito

- Input: grafo **bipartito** non orientato $G = (L \cup R, E)$.
- $M \subseteq E$ è un **matching** se ogni nodo appare in al più un arco in M .

Problema del max matching: trovare un matching di cardinalità massima.



matching
1-2', 3-1', 4-5'

Matching bipartito

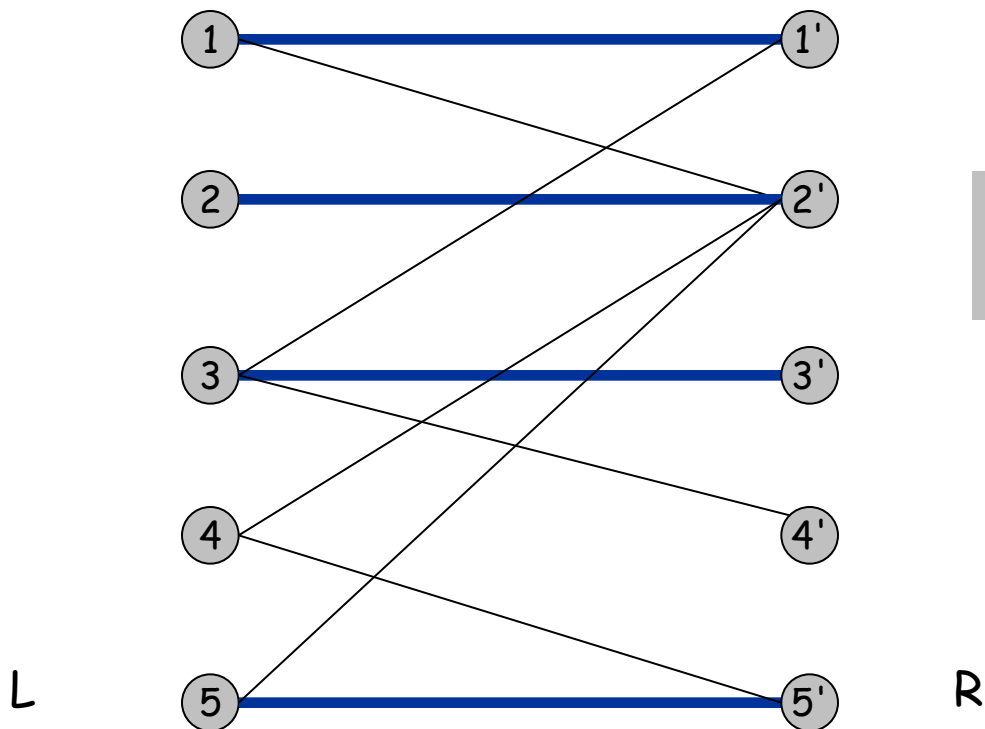
Ogni arco ha un estremo in L e l'altro in R



Matching bipartito

- Input: grafo **bipartito** non orientato $G = (L \cup R, E)$.
- $M \subseteq E$ è un **matching** se ogni nodo appare in al più un arco in M .

Problema del max matching: trovare un matching di cardinalità massima.



max matching

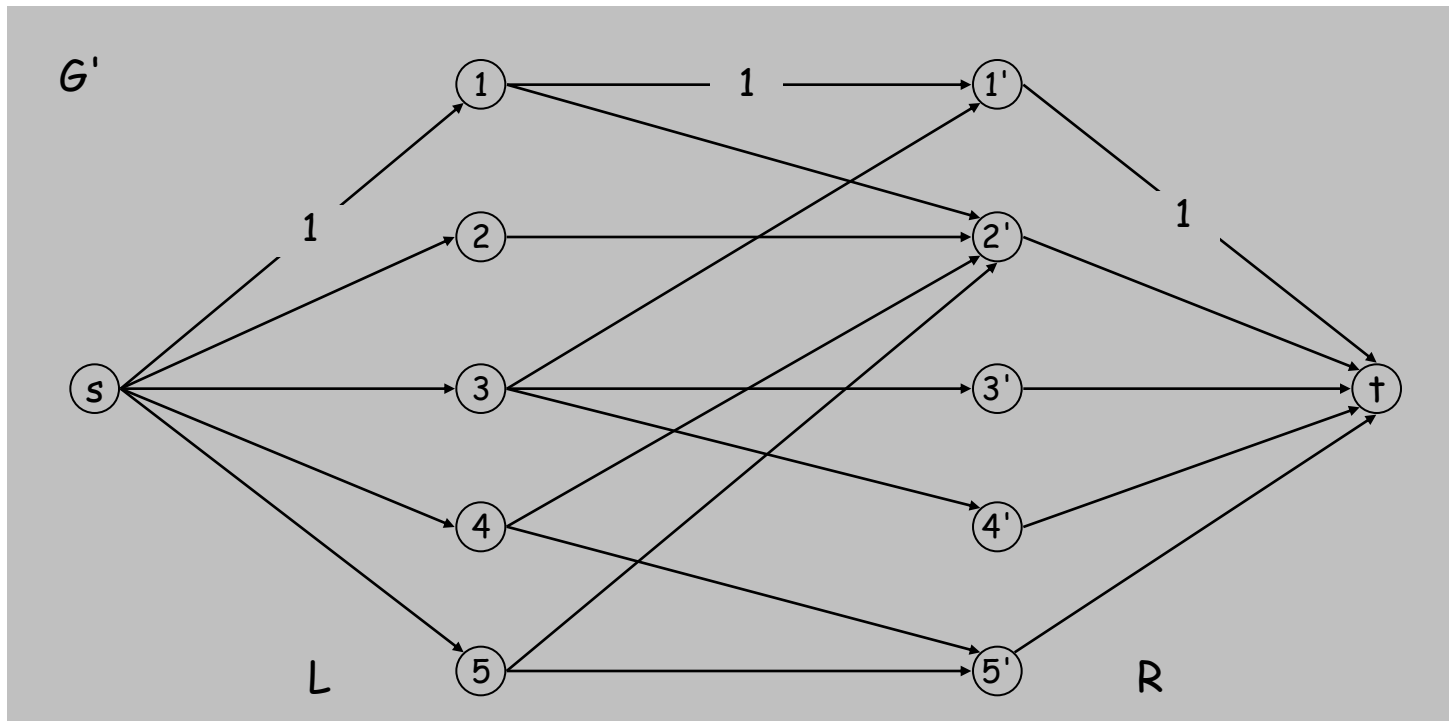
1-1', 2-2', 3-3' 4-4'

Matching bipartito e flusso

Formulazione in termini di flusso massimo.

- Creare un grafo $G' = (L \cup R \cup \{s, t\}, E')$.
- Orientare tutti gli archi da L a R , e assegnare capacità 1.
- Aggiungere sorgente s , e archi di capacità 1 da s ad ogni nodo in L .
- Aggiungere pozzo t , e archi di capacità 1 da ogni nodo in R a t .

La cardinalità massima di un matching in G = valore di massimo flusso in G' .

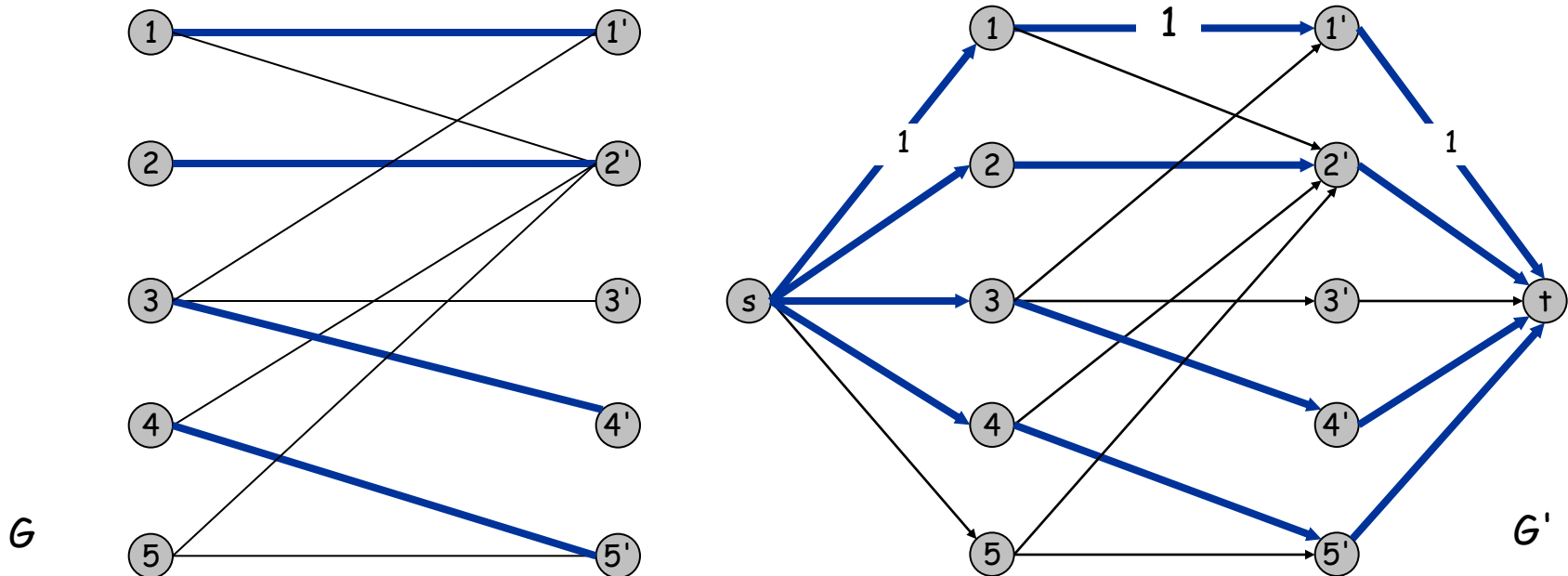


Correttezza

Teorema. La cardinalità massima di un matching in G = valore di massimo flusso in G' .

Dim. \leq

- Dato un matching massimo M di cardinalità k .
- Si consideri il flusso f che invia 1 unità lungo ognuno dei k cammini da s a t che contengono gli archi del matching.
- f è un flusso e ha valore k .

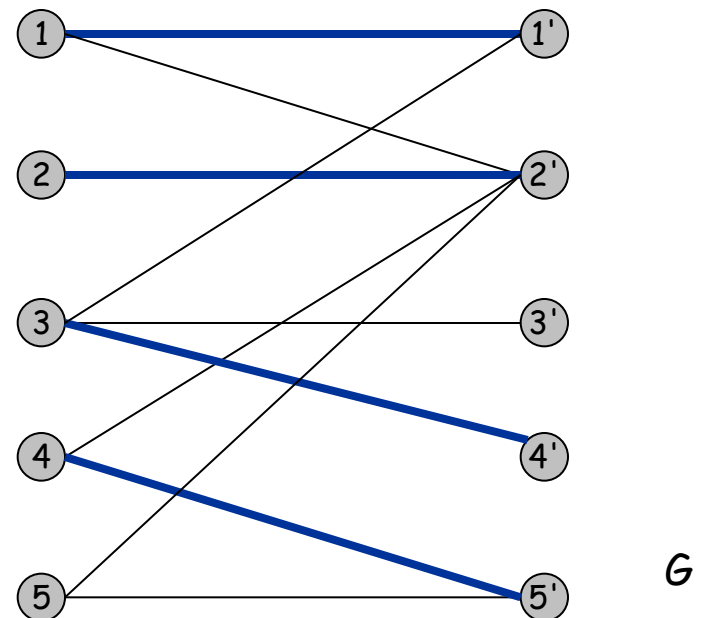
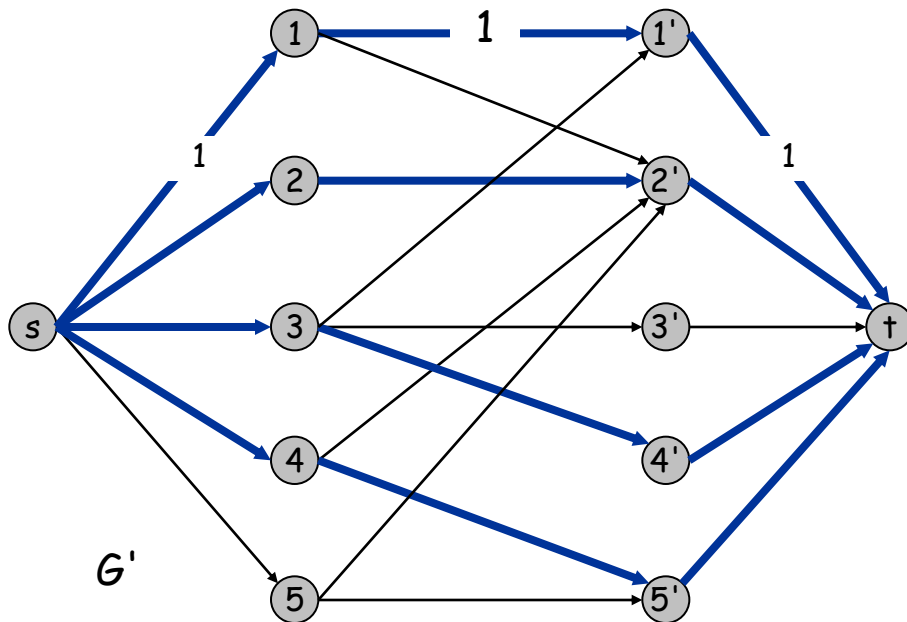


Correttezza

Teorema. La cardinalità massima di un matching in G = valore massimo flusso in G' .

Dim. \geq

- Sia f un flusso massimo in G' di valore k .
- Per il teorema di integralità $\Rightarrow k$ è intero e quindi f è 0-1.
- Si consideri M = insieme di archi da L a R con $f(e) = 1$.
 - Ogni nodo in L e R partecipa in al più 1 arco in M (**conservazione**)
 - $|M| = k$: considera taglio $(L \cup s, R \cup t)$ (**ricorda:** $\sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) = v(f).$)



Matching bipartito: complessità di tempo

Quale algoritmo per il flusso massimo usare per il matching bipartito?

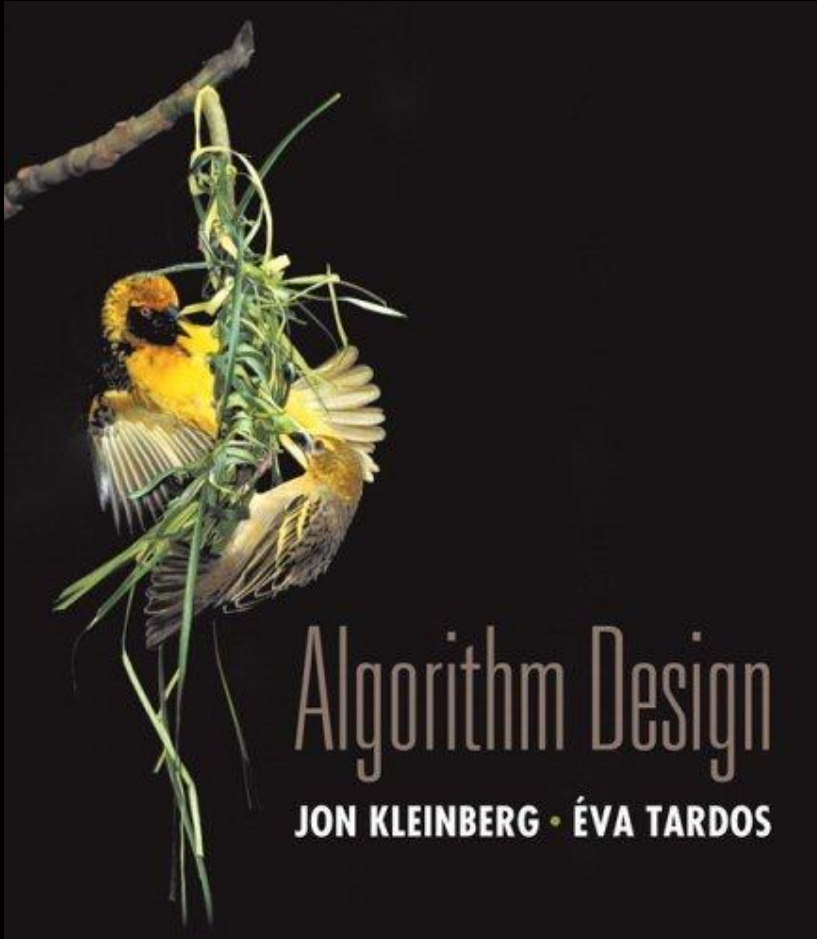
- Ford-Fulkerson generico: $O(m \text{ val}(f^*)) = O(mn)$.
- Edmonds-Karp: $O(m^2 \log C) = O(m^2)$.
- Algoritmo col minor numero di archi : $O(m n^{1/2})$.

Matching su grafi non-bipartiti.

- La struttura dei grafi non-bipartiti è più complicata, ma ben nota. [Tutte-Berge, Edmonds-Galai]
- Algoritmo di Blossom : $O(n^4)$. [Edmonds 1965]
- Migliore al momento: $O(m n^{1/2})$. [Micali-Vazirani 1980]

Chapter 7

Network flow



Slides by Kevin Wayne.
Copyright © 2005 Pearson-Addison Wesley.
All rights reserved.

Riprendiamo l'algoritmo di Ford-Fulkerson che risolve il **problema del flusso** per

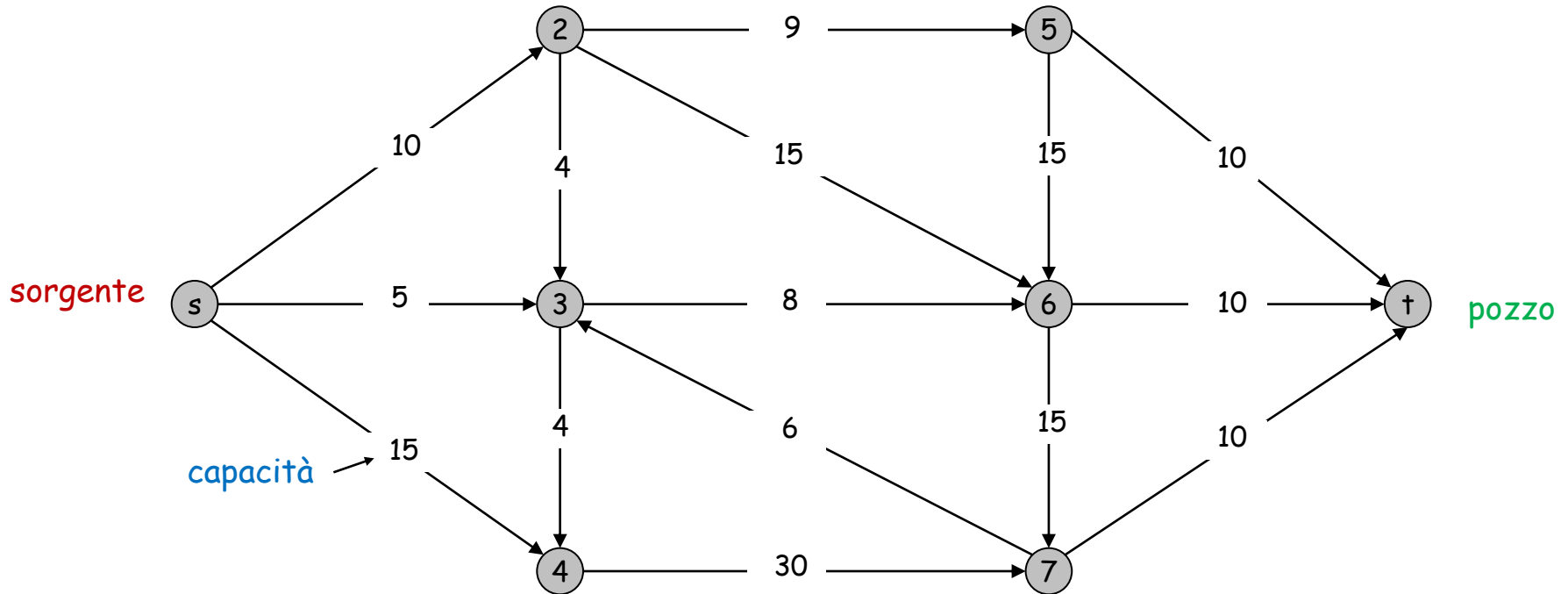
- Analizzarne la complessità
- Vederne un'applicazione

Reti di flusso

Astrazione per materiale che "scorre" attraverso gli archi (*come liquidi nei tubi*).

Una rete di flusso è $G = (V, E)$ = grafo orientato con

- due nodi particolari: **s = sorgente** (senza archi entranti)
t = pozzo (senza archi uscenti).
- $c(e)$ = capacità dell'arco e .

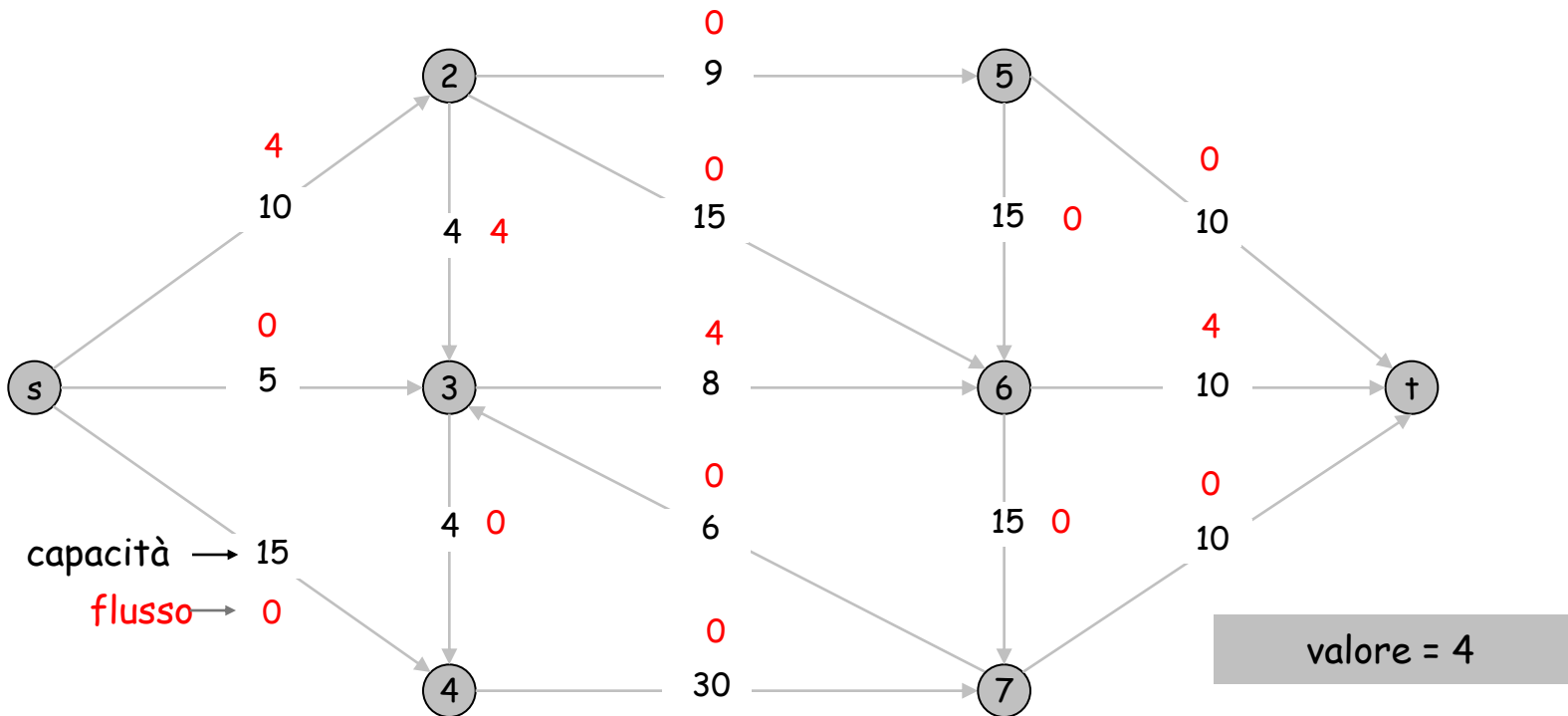


Flusso

Def. Un **flusso s-t** è una funzione $f: E \rightarrow \mathbb{R}^+$ che soddisfa:

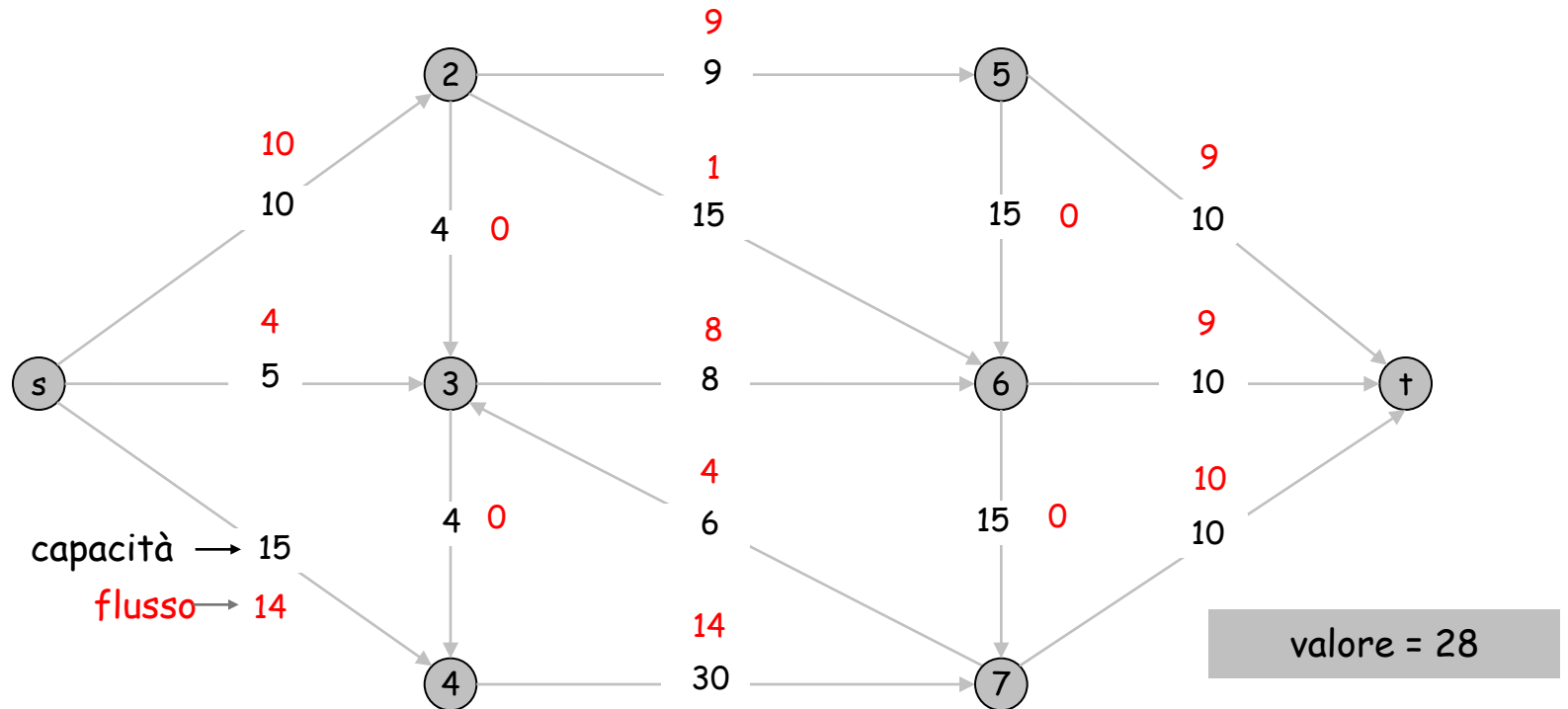
- per ogni $e \in E$: $0 \leq f(e) \leq c(e)$ (capacità)
- per ogni $v \in V - \{s, t\}$: $\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e)$ (conservazione)

Def. Il **valore** del flusso f è: $v(f) = \sum_{e \text{ out of } s} f(e)$.



Problema del massimo flusso

Problema del massimo flusso. Trovare il flusso s-t di massimo valore.



Algoritmo del cammino aumentante

Sia P un cammino semplice s - t in G_f

```
Augment(f, c, P) {  
  b ← bottleneck(P, f)  
  foreach e ∈ P {  
    if (e ∈ E) in G: f(e) ← f(e) + b  
    else          in G: f(eR) ← f(e) - b  
  }  
  return f  
}
```

Minima capacità residuale
di un arco di P

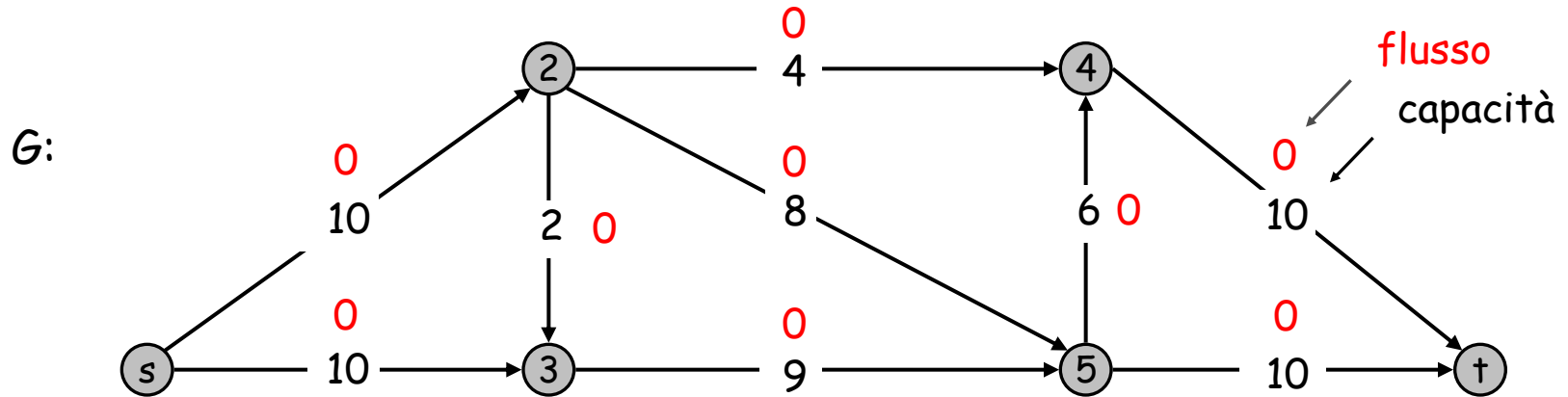


Arco in avanti

Arco inverso

```
Ford-Fulkerson(G, s, t, c) {  
  foreach e ∈ E f(e) ← 0  
  Gf ← grafo residuale  
  
  while (esiste un cammino aumentante P in Gf) {  
    f ← Augment(f, c, P)  
    aggiorna Gf  
  }  
  return f  
}
```

Esempio esecuzione algoritmo di Ford-Fulkerson

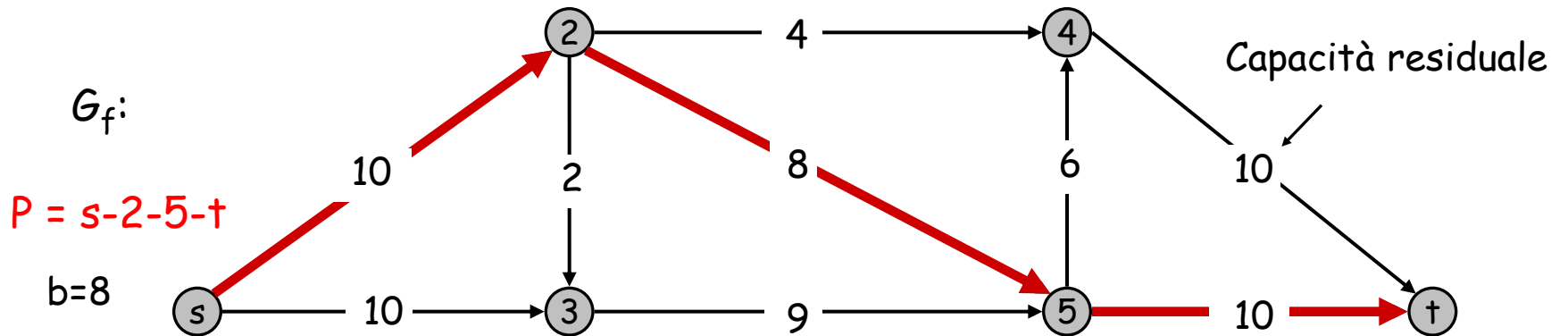
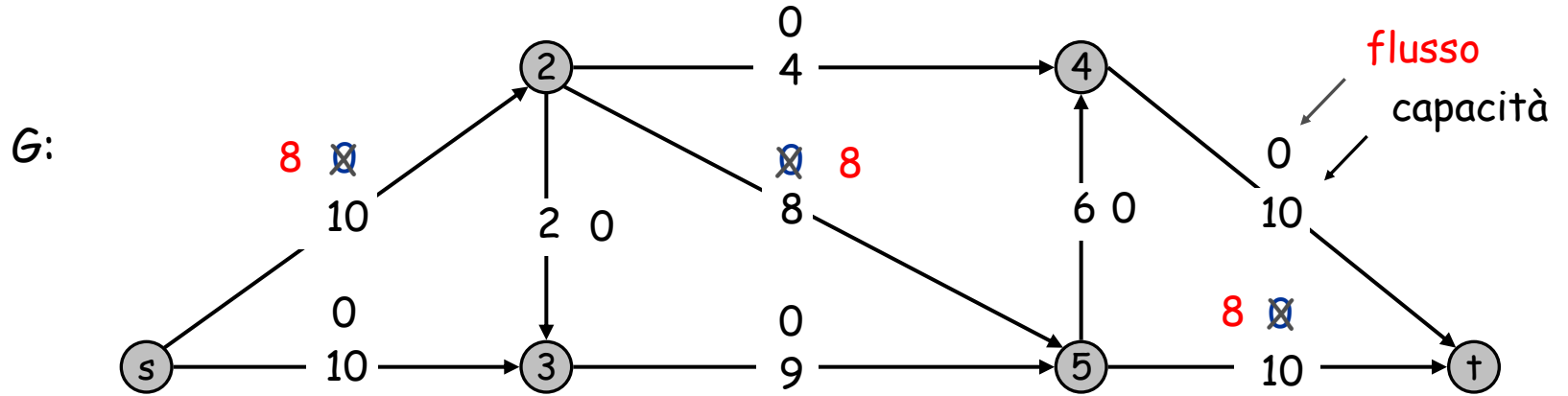


valore flusso = 0

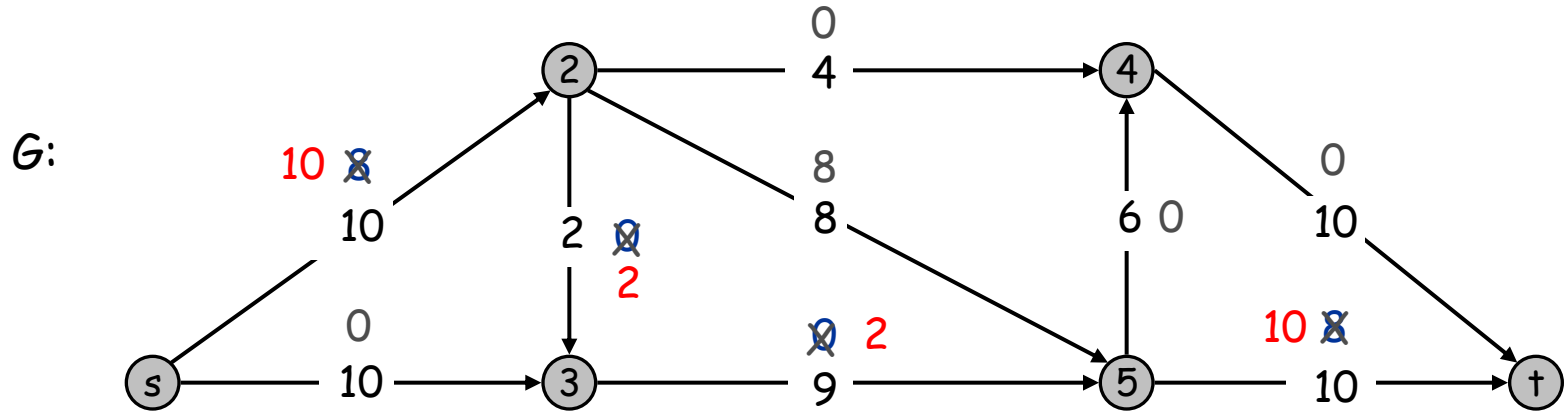
Rivediamo l'esempio per capire quale può essere la complessità di tempo;
in particolare:

Come varia il valore del flusso ad ogni iterazione?
Quante iterazioni ci sono?

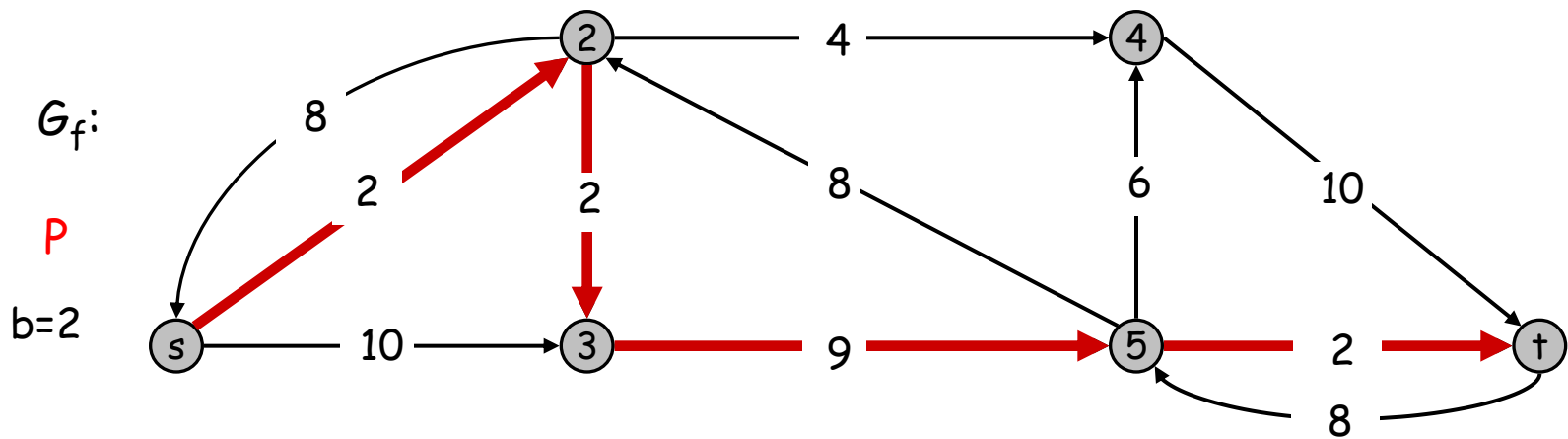
Algoritmo di Ford-Fulkerson



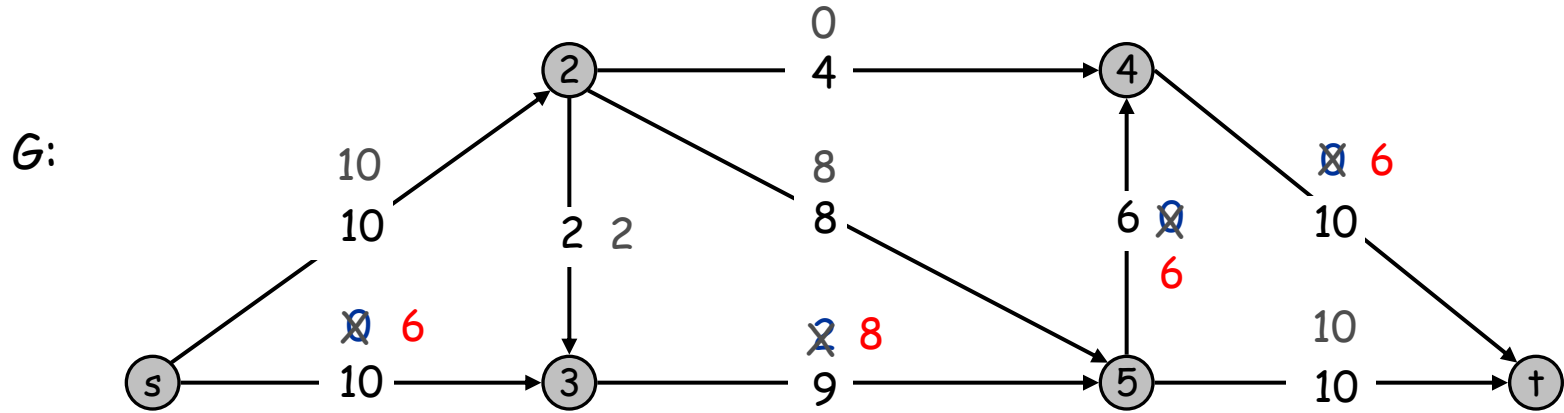
Algoritmo di Ford-Fulkerson



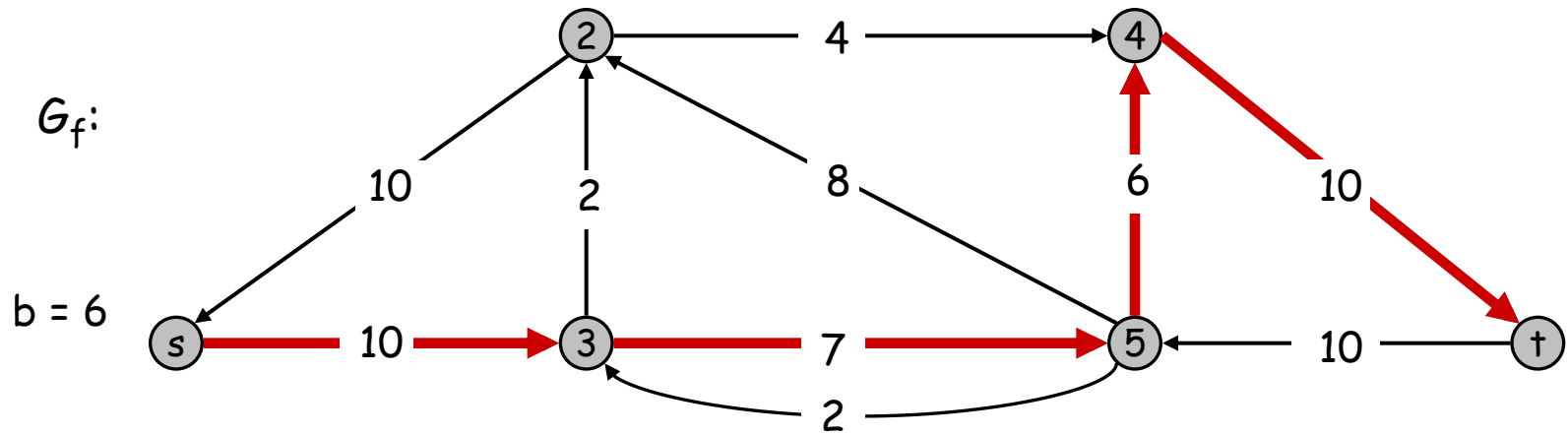
valore flusso = \otimes 10



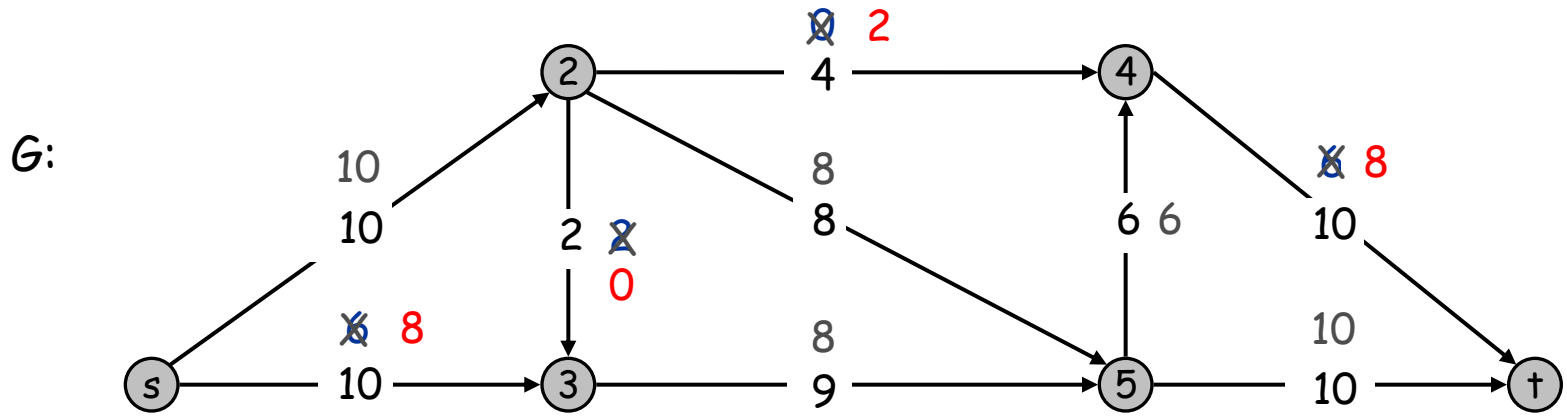
Algoritmo di Ford-Fulkerson



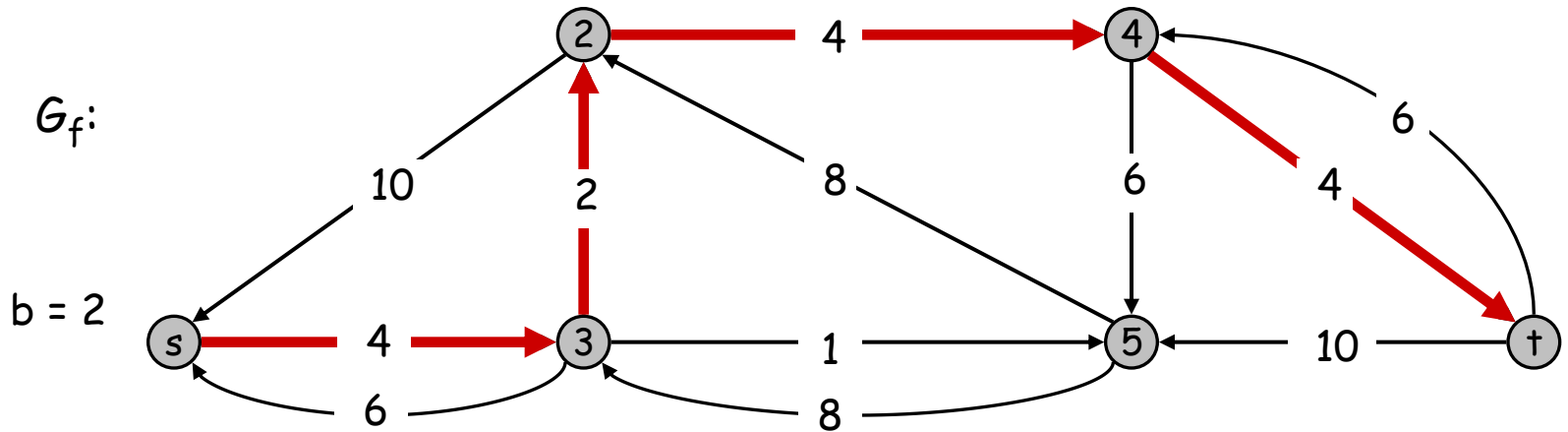
valore flusso = ~~10~~ 16



Algoritmo di Ford-Fulkerson

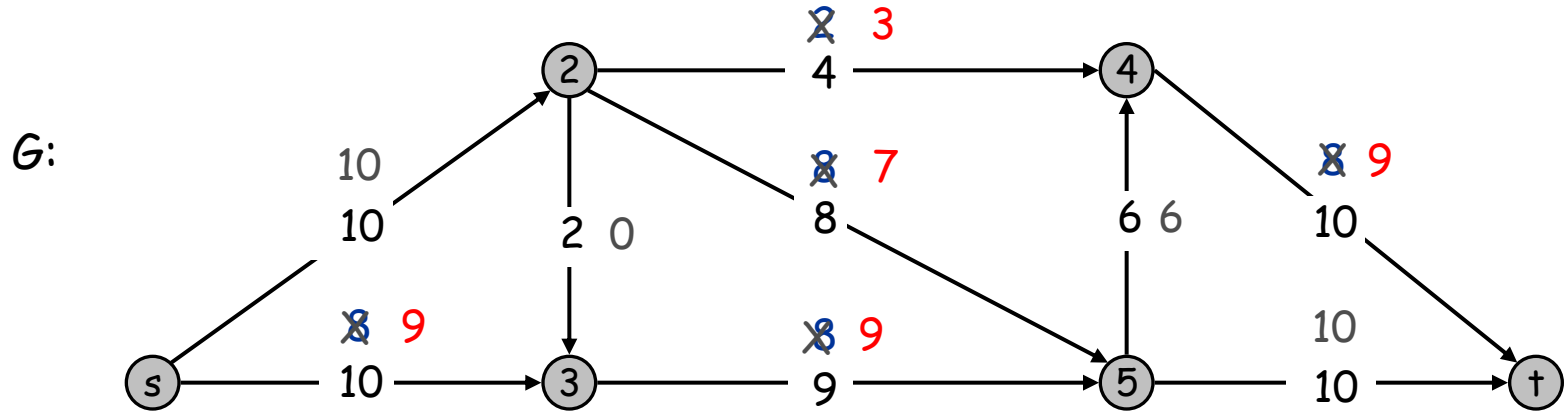


valore flusso = ~~16~~ 18

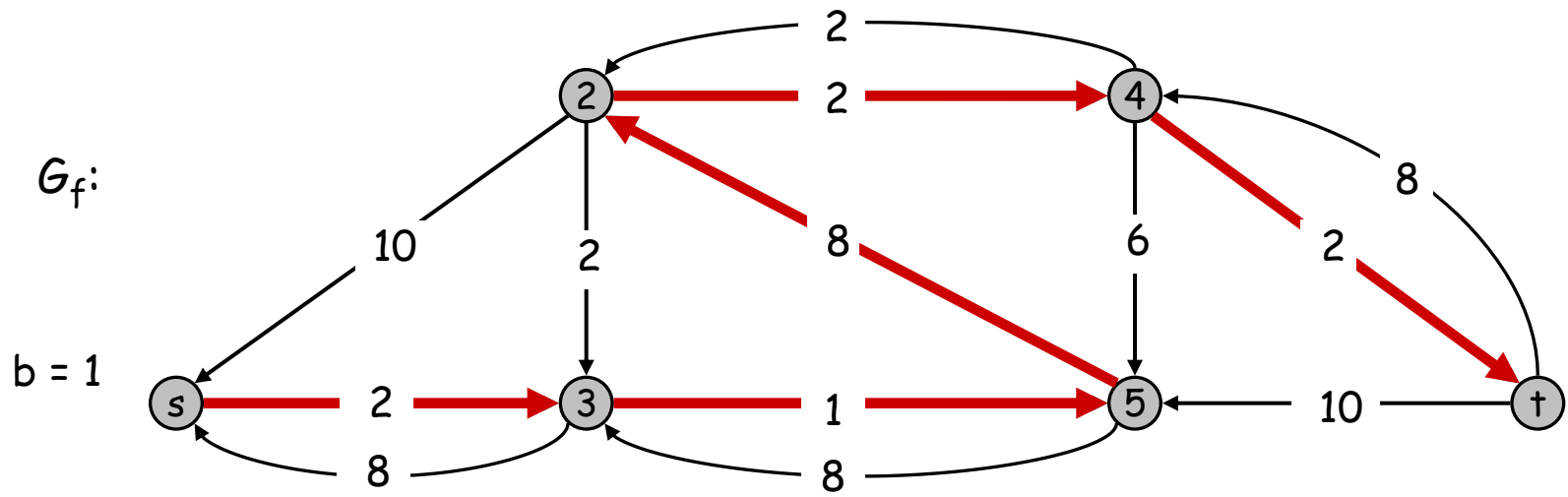


Nota: l'arco $(3,2)$ in G_f è un arco indietro: $(2,3)$ è in G , quindi $f((2,3)) \leftarrow f((2,3)) - b$

Algoritmo di Ford-Fulkerson

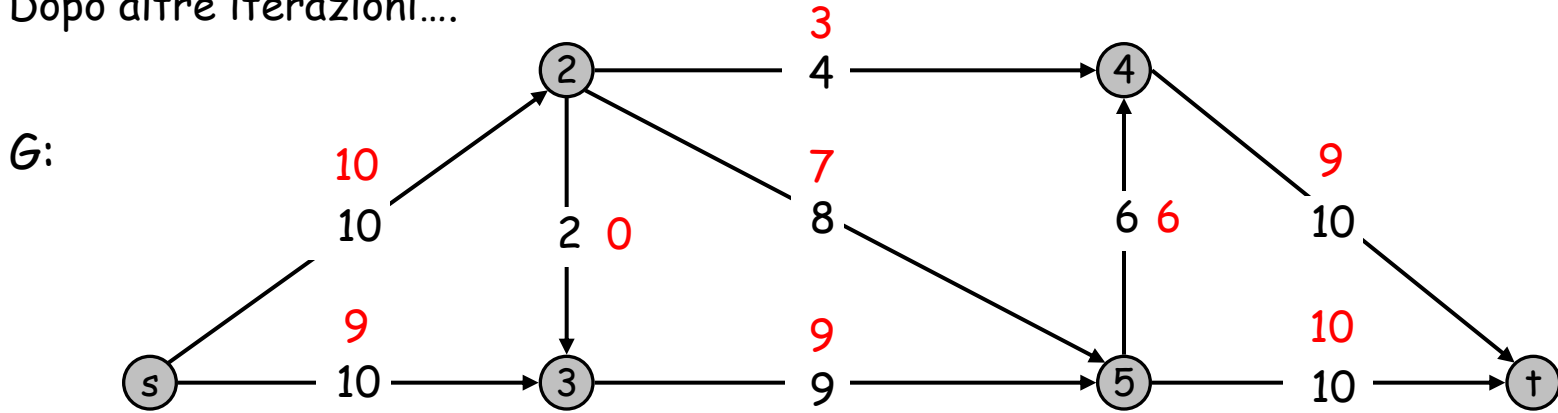


valore flusso = ~~18~~ 19

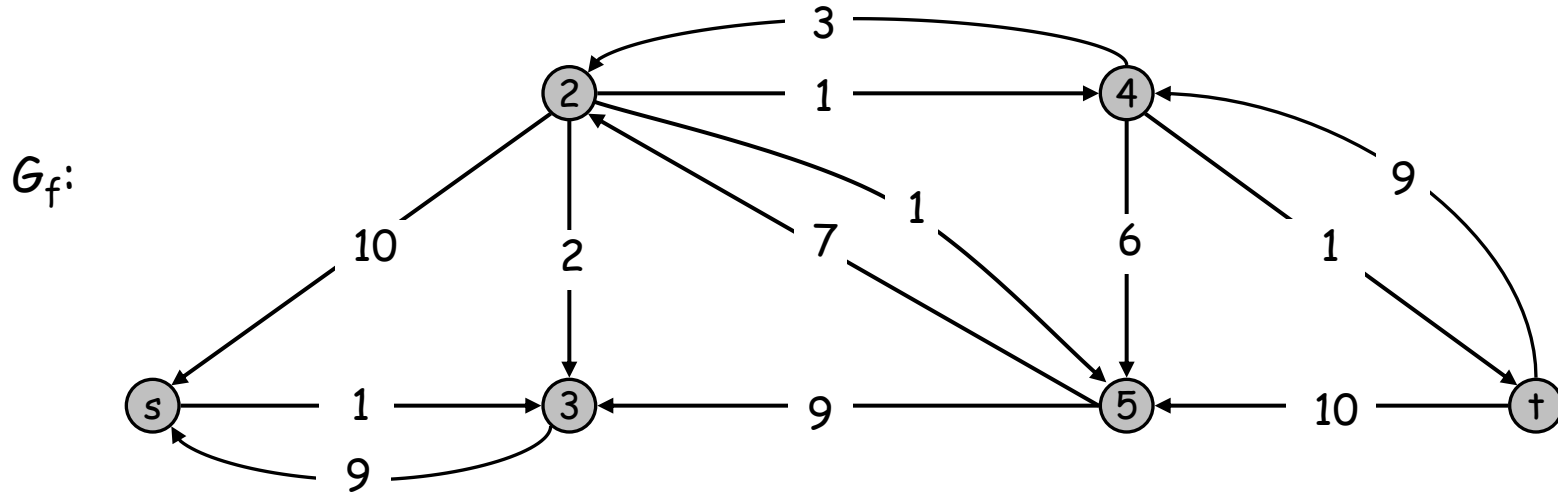


Algoritmo di Ford-Fulkerson

Dopo altre iterazioni....



valore flusso = 19



Non ci sono più cammini aumentanti P in G_f : l'algoritmo termina.

Teorema Max-flusso Min-taglio

Teorema del cammino aumentante. Un flusso f è un flusso massimo sse non ci sono cammini aumentanti.

Teorema max-flusso min-taglio. [Ford-Fulkerson 1956] Il valore del flusso massimo è uguale al valore del minimo taglio.

Dim. Proviamo entrambi mostrando che sono equivalenti:

- (i) Esiste un taglio (A, B) tale che $v(f) = \text{cap}(A, B)$.
- (ii) f è un flusso massimo.
- (iii) Non ci sono cammini aumentanti nel grafo residuale G_f .

(i) \Rightarrow (ii) Questo è il corollario al lemma di dualità debole.

(ii) \Rightarrow (iii) Mostriamo per contrapposizione: non (iii) \Rightarrow non (ii).

Sia f un flusso. Se esiste un cammino aumentante, allora possiamo migliorare f mandando delle unità lungo quel cammino. Quello che ne risulta è ancora un flusso: valgono le proprietà di capacità e conservazione.

Analisi dell'algoritmo di Ford-Fulkerson

Limitazione superiore al valore del flusso: $C = \sum_{e \text{ esce da } s} c(e)$

Infatti $v(f) = \sum_{e \text{ esce da } s} f(e) \leq \sum_{e \text{ esce da } s} c(e) = C$

Assunzione. Tutte le capacità sono **interi** fra 1 e C .

Invariante. Ogni valore del flusso $f(e)$ e ogni capacità residuale $c_f(e)$ restano interi durante l'esecuzione dell'algoritmo.

Teorema. L'algoritmo termina in al più $v(f^*) \leq C$ iterazioni
($f^* = \text{max flusso}$).

Prova.

Ogni cammino aumentante P incrementa il valore del flusso di almeno 1.

Infatti il flusso **aumenta** perché il primo arco di P in G_f esce da s e P non ritorna in s ; siccome non ci sono archi entranti in s in G il primo arco di P è in avanti.

Più precisamente il flusso aumenta ad ogni iterazione di $b = \text{bottleneck}(P, f)$.

Osservazione: Se le capacità non fossero interi, l'algoritmo potrebbe continuare all'infinito.

Complessità di tempo

Supponiamo che tutti i nodi abbiano almeno un arco incidente, quindi $m \geq n/2$ e $O(m+n) = O(m)$.

Corollario. Complessità tempo di Ford-Fulkerson è $O(mC)$.

Prova.

Al massimo $v(f^*) \leq C$ iterazioni

In ogni iterazione:

- troviamo un cammino aumentante in $O(m)$ mediante BFS o DFS nel grafo residuale.
- il grafo residuale ha $\leq 2m$ archi
- il grafo residuale rappresentato utilizzando le liste delle adiacenze (in e out)
- $\text{Augment}(f, c, P)$ ha complessità $O(n)$ (P ha al più $n-1$ archi)
- aggiornamento grafo residuale in $O(m)$ (per ogni arco costruiamo gli archi indietro e avanti opportuni)

Teorema Integralità. Se tutte le capacità sono numeri interi, allora vi è un flusso massimo f per cui ogni valore del flusso $f(e)$ è un intero.

Prova. L'algoritmo termina, quindi il teorema segue dall'invariante.

L'algoritmo di Ford-Fulkerson è pseudo-polinomiale

Complessità di tempo di Ford-Fulkerson è $O(mC)$.

Quando C non è molto grande, questa è una limitazione accettabile.
Ma non lo è più se C è grande e la taglia diventa: m , n , e $\log C$.

D. L' algoritmo di Ford-Fulkerson è polinomiale nella taglia dell'input?

m , n , e $\log C$

R. No: l'algoritmo può fare anche C iterazioni, a seconda della scelta del cammino aumentante e quindi un numero esponenziale ($C = 2^{\log C}$).

Vediamone un esempio.

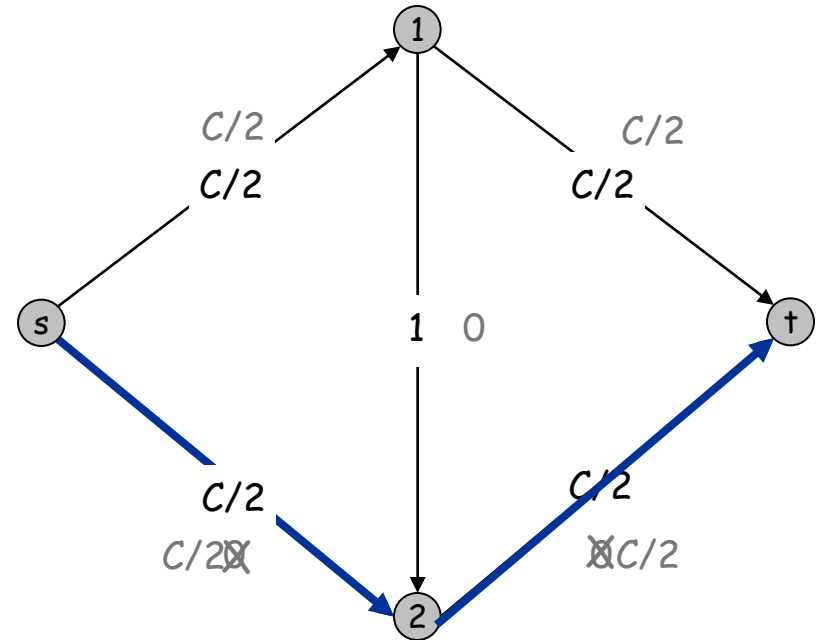
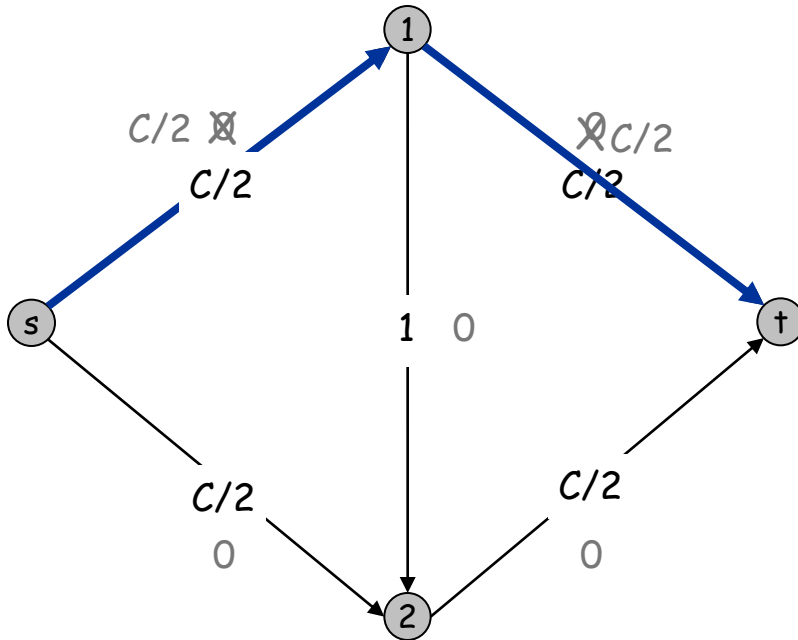
Esempio: prima scelta dei cammini

Esempio: massimo flusso = C con $f((1,2))=0$ e $f(e)=C/2$ per gli altri

Calcolabile in 2 iterazioni dell'algoritmo di Ford-Fulkerson scegliendo

$P_1 = s-1-t$

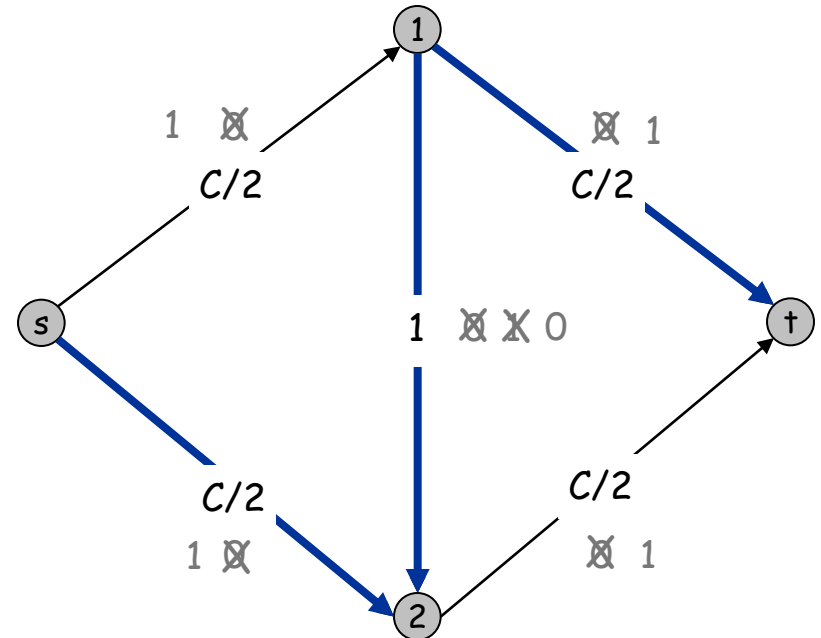
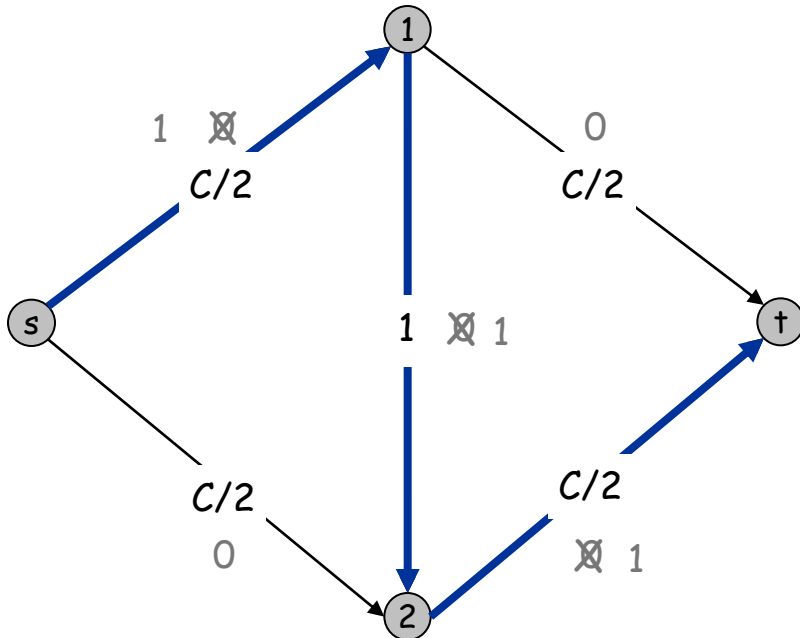
$P_2 = s-2-t$



Esempio: seconda scelta dei cammini

Esempio: massimo flusso = C con $f((1,2))=0$ e $f(e)=C/2$ per gli altri

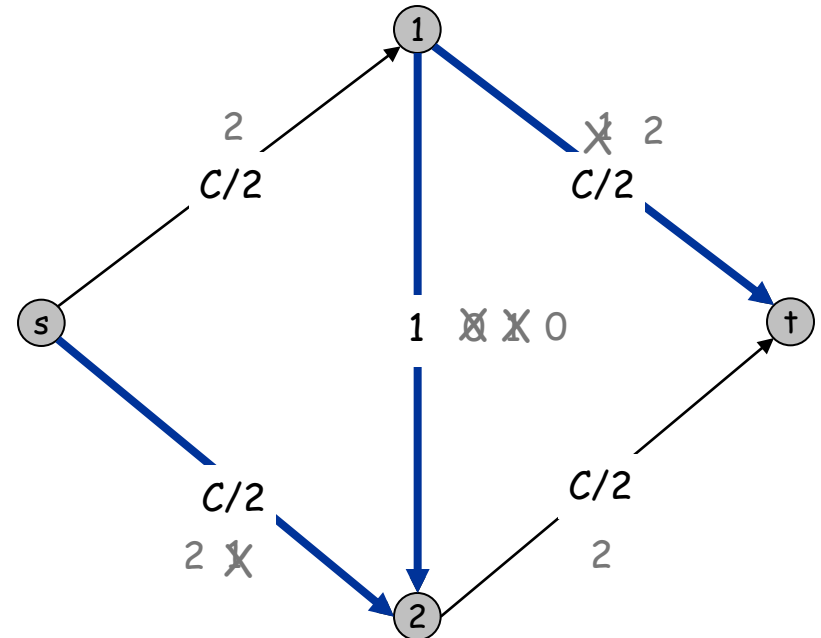
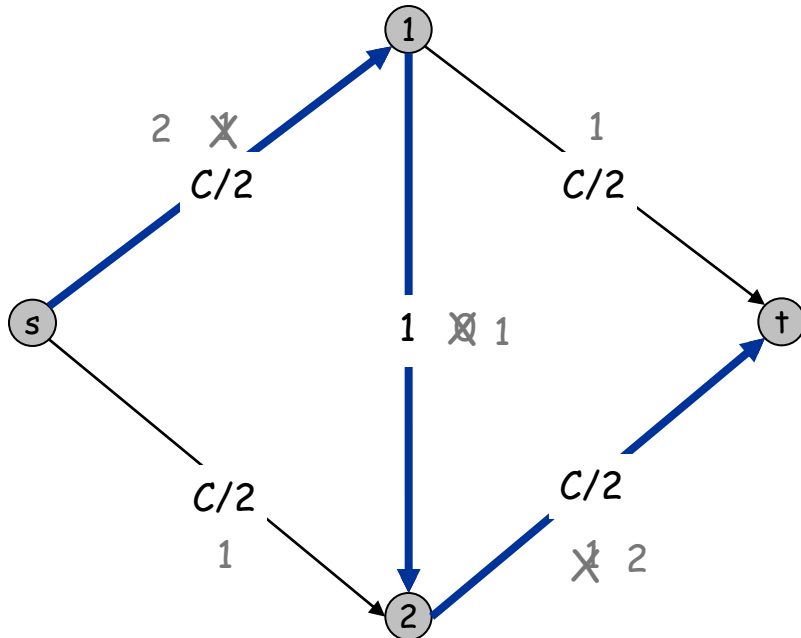
Calcolabile in C iterazioni dell'algoritmo di Ford-Fulkerson scegliendo $P_3 = s-1-2-t$ e $P_4 = s-2-1-t$ alternativamente per $C/2$ volte ognuno. In G_f compare $(1,2)$ o $(2,1)$ alternativamente.



Esempio: numero esponenziale di incrementi del flusso

Esempio: massimo flusso = C con $f((1,2))=0$ e $f(e)=C/2$ per gli altri

Calcolabile in C iterazioni dell'algoritmo di Ford-Fulkerson scegliendo $P_3 = s-1-2-t$ e $P_4 = s-2-1-t$ alternativamente per $C/2$ volte ognuno. In G_f compare $(1,2)$ o $(2,1)$ alternativamente.



Scegliere Buoni Cammini Aumentanti

Fare attenzione quando si scelgono i cammini aumentanti.

- Alcune scelte portano ad algoritmi esponenziali.
- Buone scelte portano ad algoritmi polinomiali.
- Se le capacità fossero irrazionali, l'algoritmo potrebbe non terminare!

Obiettivo: scegliere cammini aumentanti in modo tale che:

- Possiamo trovare cammini aumentanti efficientemente.
- Poche iterazioni.

Scegliere cammini aumentanti: [Edmonds-Karp 1972, Dinitz 1970]

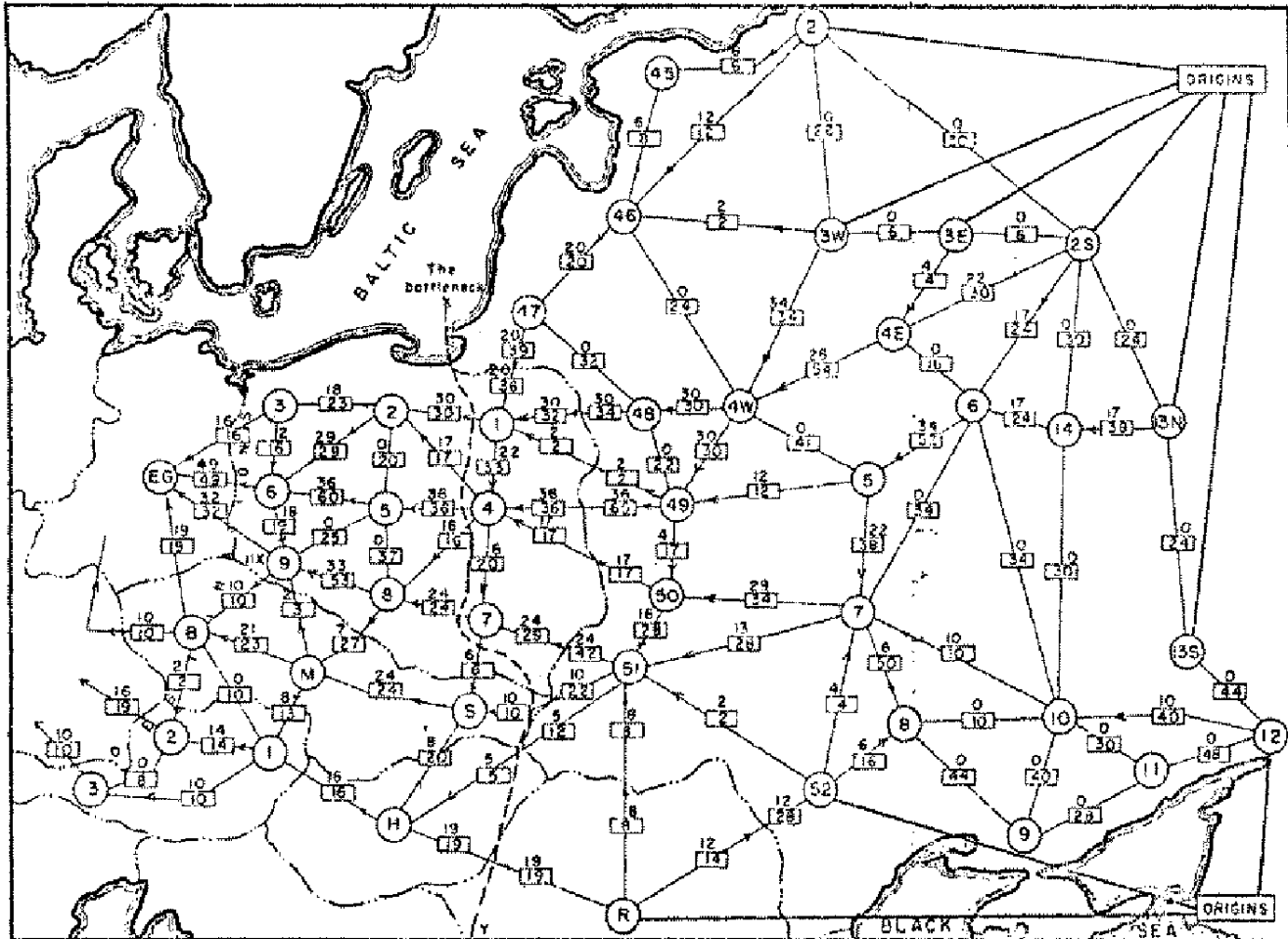
- Massima capacità bottleneck (però può richiedere molto tempo).
- Sufficientemente grande capacità bottleneck.

Complessità: $O(m^2 \log_2 C)$ [Edmonds-Karp 1972, Dinitz 1970]

Altro algoritmo che sceglie cammino con minor numero di archi

Soviet Rail Network, 1955

È interessante notare che storicamente il problema del flusso massimo fu introdotto durante la Guerra Fredda per risolvere quello del minimo taglio. Più precisamente, si intendeva determinare il minimo numero di "tagli da effettuare" (mediante bombardamenti...) alla rete ferroviaria sovietica per sconnettere Mosca dal resto dell'URSS.



Reference: *On the history of the transportation and maximum flow problems.*
Alexander Schrijver in *Math Programming*, 91: 3, 2002.

7.5 Matching Bipartito

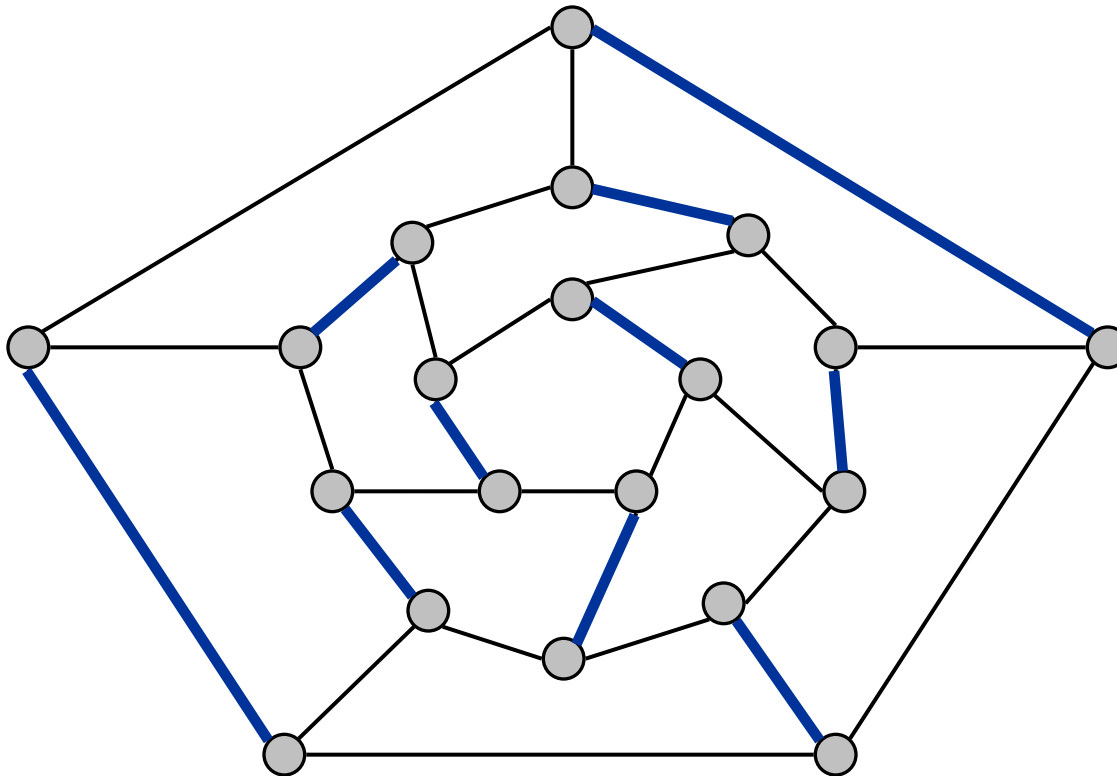
Un'applicazione del calcolo del flusso massimo

Matching

Matching.

- Input: grafo non-orientato $G = (V, E)$.
- $M \subseteq E$ è un **matching** se ogni nodo appare in al più un arco in M .

Problema del max matching: trovare un matching di cardinalità massima.



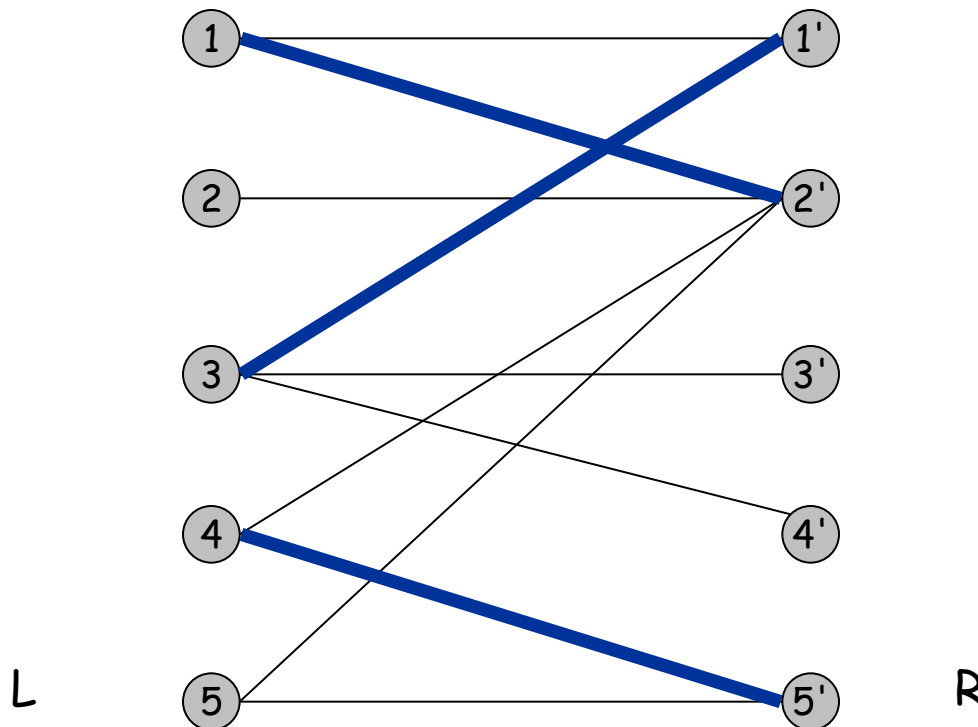
Matching bipartito

Ogni arco ha un estremo in L e l'altro in R

Matching bipartito

- Input: grafo **bipartito** non orientato $G = (L \cup R, E)$.
- $M \subseteq E$ è un **matching** se ogni nodo appare in al più un arco in M .

Problema del max matching: trovare un matching di cardinalità massima.



matching
1-2', 3-1', 4-5'

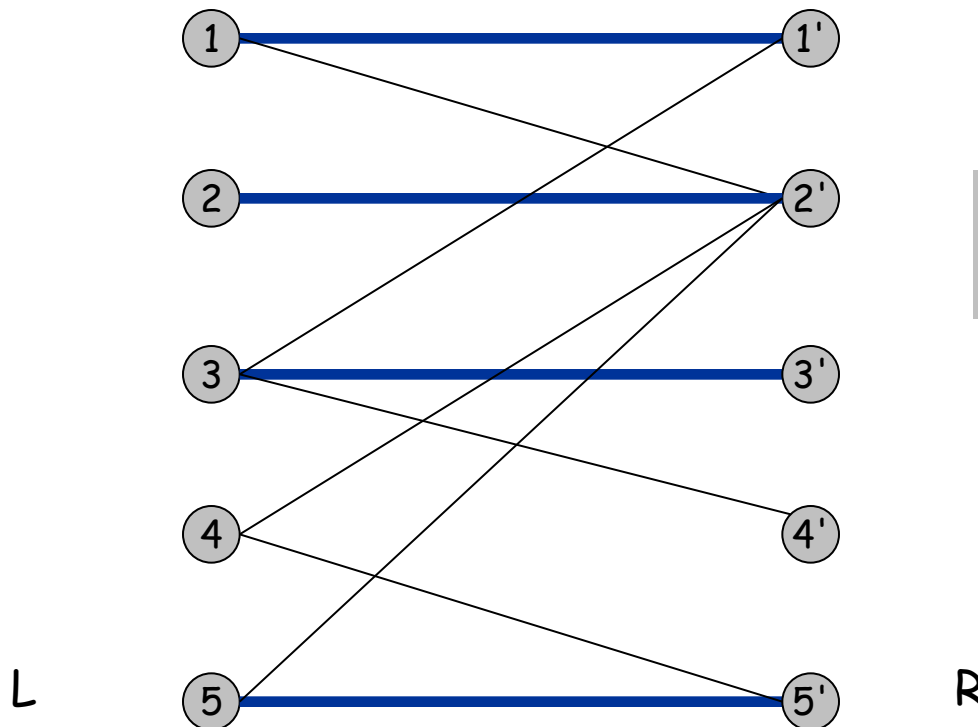
Matching bipartito

Ogni arco ha un estremo in L e l'altro in R

Matching bipartito

- Input: grafo **bipartito** non orientato $G = (L \cup R, E)$.
- $M \subseteq E$ è un **matching** se ogni nodo appare in al più un arco in M .

Problema del max matching: trovare un matching di cardinalità massima.



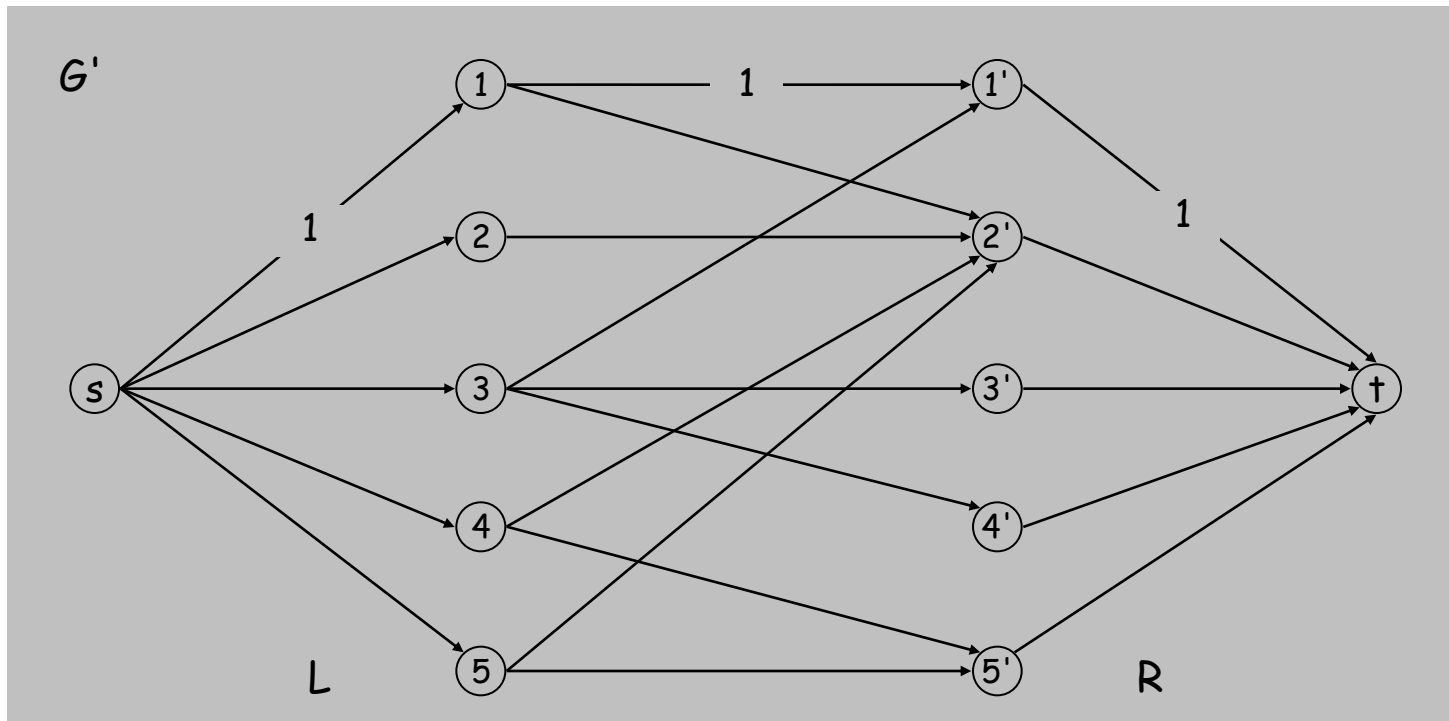
max matching

1-1', 2-2', 3-3' 4-4'

Matching bipartito e flusso

Formulazione in termini di flusso massimo.

- Creare un grafo $G' = (L \cup R \cup \{s, t\}, E')$.
- Orientare tutti gli archi da L a R , e assegnare capacità 1.
- Aggiungere sorgente s , e archi di capacità 1 da s ad ogni nodo in L .
- Aggiungere pozzo t , e archi di capacità 1 da ogni nodo in R a t .
- La cardinalità massima di un matching in $G =$ valore di massimo flusso in G' .

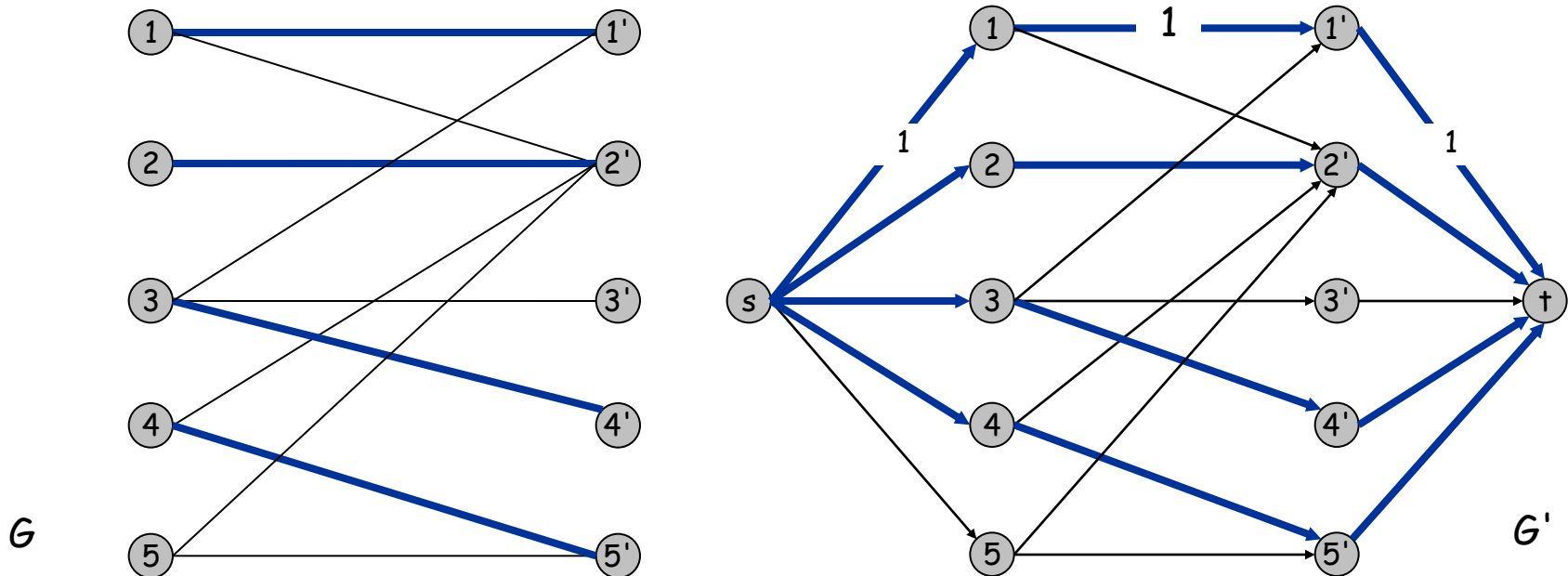


Correttezza

Teorema. La cardinalità massima di un matching in G = valore di massimo flusso in G' .

Dim. \leq

- Dato un matching massimo M di cardinalità k .
- Si consideri il flusso f che invia 1 unità lungo ognuno dei k cammini da s a t che contengono gli archi del matching.
- f è un flusso e ha valore k .

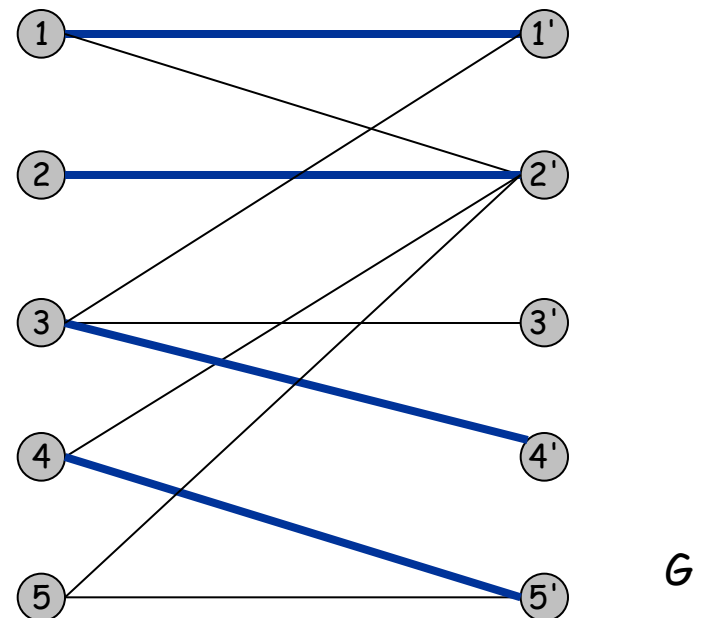
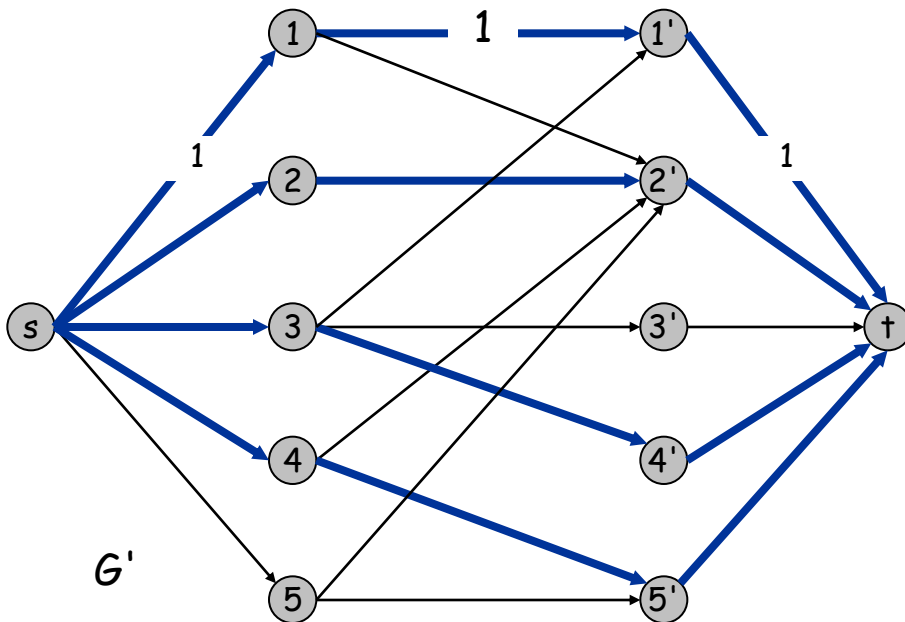


Correttezza

Teorema. La cardinalità massima di un matching in G = valore massimo flusso in G' .

Dim. \geq

- Sia f un flusso massimo in G' di valore k .
- Per il teorema di integralità $\Rightarrow k$ è intero e quindi f è 0-1.
- Si consideri M = insieme di archi da L a R con $f(e) = 1$.
 - Ogni nodo in L e R partecipa in al più 1 arco in M (**conservazione**)
 - $|M| = k$: considera taglio $(L \cup s, R \cup t)$ (**ricorda:** $\sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) = v(f).$)



Matching bipartito: complessità di tempo

Il tempo è dominato dalla ricerca del massimo flusso.

In questo caso $C = n$ (ogni arco uscente da s ha capacità 1).

Quale algoritmo per il flusso massimo usare per il matching bipartito?

- Ford-Fulkerson generico: $O(m \text{ val}(f^*)) = O(mC) = O(mn)$.
- Edmonds-Karp: $O(m^2 \log C) = O(m^2 \log n)$.
- Algoritmo col minor numero di archi : $O(m n^{1/2})$.

Matching su grafi non-bipartiti.

- La struttura dei grafi non-bipartiti è più complicata, ma ben nota. [Tutte-Berge, Edmonds-Galai]
- Algoritmo di Blossom : $O(n^4)$. [Edmonds 1965]
- Migliore al momento: $O(m n^{1/2})$. [Micali-Vazirani 1980]

FINE