

ALGORITMI MOD2

LEZ1 – 05/03/2024 (algoritmi greedy per la risoluzione di problema di interval scheduling e problema di interval partitioning)

PROBLEMA DI INTERVAL SCHEDULING

→ Descrizione:

Il job j inizia a s_j e termina a f_j .

Due job sono compatibili se non si sovrappongono.

Obiettivo: trovare il massimo sottoinsieme di job reciprocamente compatibili.

→ Formalizzazione:

- Input = un insieme di n intervalli I_1, \dots, I_n
l'intervallo I_i ha un orario di inizio s_i e l'orario di fine f_i
- Soluzione fattibile = Un sottoinsieme S degli intervalli reciprocamente compatibili, cioè per ciascuno I_i, I_j appartenenti ad S , I_i non si sovrappone a I_j ;
- Misura (massimizzare):
numero di intervalli programmati, cioè cardinalità di S

GREEDY ALGORITHMS E INTERVAL SCHEDULING

Greedy template: Considera i job in un ordine naturale.

Accetta ogni job purché compatibile con quelli già svolti.

- [Earliest start time] Considera i job in ordine crescente di s_j .
- [Earliest finish time] Considera i job in ordine crescente di f_j .
- [Shortest interval] Considera i job in ordine crescente di $f_j - s_j$.
- [Fewest conflicts] Per ogni job j , contare il numero di lavori in conflitto c_j . Schedule in ordine crescente di c_j .

→ Conveniente per problema di interval scheduling: earliest finish time

EARLIEST-FINISH-TIME-FIRST ($n, s_1, s_2, \dots, s_n, f_1, f_2, \dots, f_n$)

SORT jobs by finish times and renumber so that $f_1 \leq f_2 \leq \dots \leq f_n$.

$S \leftarrow \emptyset$. ← set of jobs selected

FOR $j = 1$ TO n

IF (job j is compatible with S)

$S \leftarrow S \cup \{j\}$.

RETURN S .

Controesempi per gli altri:

counterexample for earliest start time



counterexample for shortest interval



counterexample for fewest conflicts



→ Proposizione:

Si può implementare earliest-finish-time first in tempo $O(n \log n)$.

- Tieni traccia del job j^* che è stato aggiunto per ultimo a S .
- Il job j è compatibile con S se $S_j \geq f_{j^*}$.
- L'ordinamento in base ai tempi di fine richiede tempo $O(n \log n)$.

→ Analisi algoritmo earliest-finish-time-first:

Siano i_1, i_2, \dots, i_k l'insieme dei job selezionati da greedy (ordinati per tempi di fine).

Siano j_1, j_2, \dots, j_m l'insieme dei job in una soluzione ottima (ordinati per tempi di fine)

Lemma: per ogni $r=1,2,\dots,k$ abbiamo $f(i_r) < f(j_r)$

Pf. [by induction]

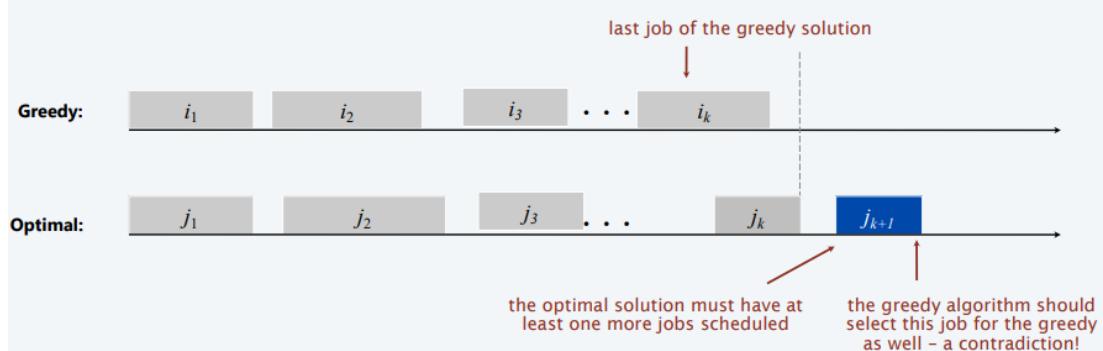
- $r=1$: Obvious.
- $r>1$:



Teorema: l'algoritmo earliest-finish-time-first è ottimale

Pf. [by contradiction]

- Let i_1, i_2, \dots, i_k be set of jobs selected by greedy (ordered by finish times).
- Let j_1, j_2, \dots, j_m be set of jobs in an optimal solution (ordered by finish times)
- Assume greedy is not optimal
- then $m > k$



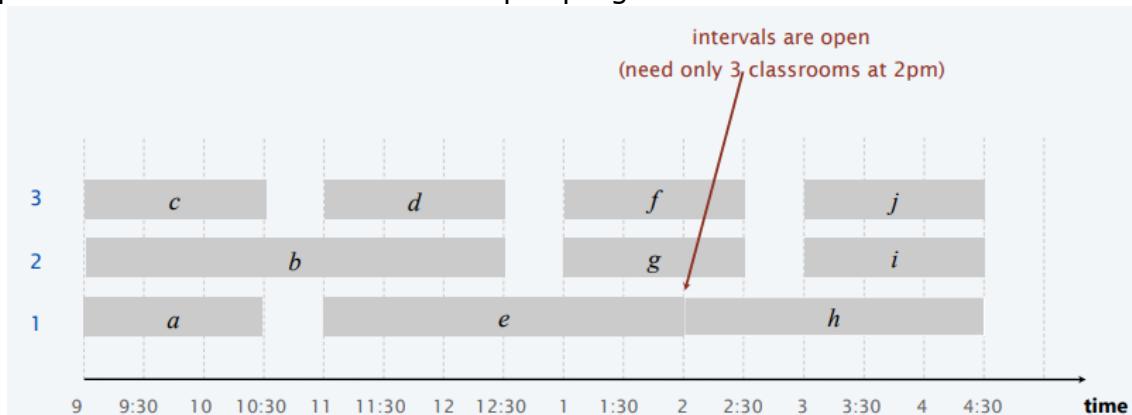
UN ALTRO PROBLEMA: IL PROBLEMA DI INTERVAL PARTITIONING

→ Descrizione:

La lezione j inizia a s_j e termina a f_j .

Obiettivo: trovare il numero minimo di aule per programmare tutte le lezioni in modo che non si svolgano due lezioni contemporaneamente nella stessa stanza.

Esempio di schedule che utilizza 3 aule per programmare 10 lezioni:



→ Formalizzazione:

- Input = Un insieme di n intervalli I_1, \dots, I_n . In intervallo I_i , ha un orario di inizio s_i e l'ora di fine f_i
- Soluzione fattibile = Una partizione degli intervalli in sottoinsiemi (chiamati aule) C_1, \dots, C_d tale che ogni C_i contenga intervalli reciprocamente compatibili
- Misura (minimizzare) = numero di aule, ovvero d

GREEDY ALGORITHMS E INTERVAL PARTITIONING

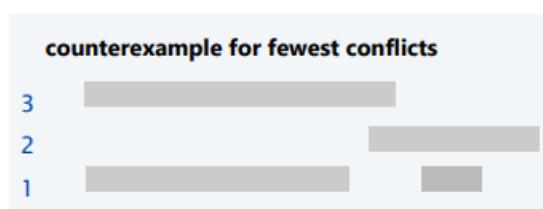
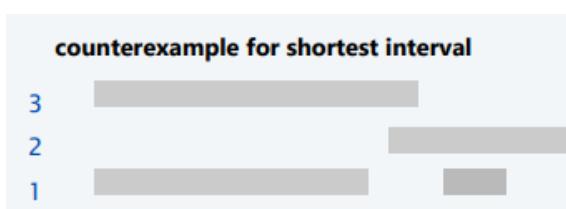
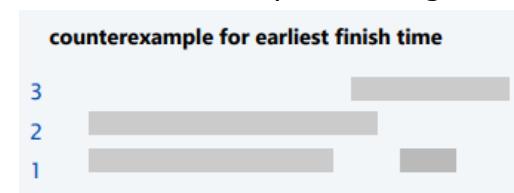
Greedy template: Considera le lezioni in un ordine naturale.

Assegna ciascuna lezione in un'aula disponibile (quale?);

Assegna una nuova aula se nessuna è disponibile.

- [Earliest start time] Considera le lezioni in ordine crescente di s_j
- [Earliest finish time] Considera le lezioni in ordine crescente di f_j
- [Shortest interval] Considera le lezioni in ordine crescente di $f_j - s_j$
- [Fewest conflicts] Per ogni lezione j , conta il numero di lezioni contrastanti c_j . Programma in ordine crescente di c_j .

→ Conveniente per problema di interval partitioning? Controesempi:



Quindi conveniente per problema di interval partitioning: Earliest start time

EARLIEST-START-TIME-FIRST ($n, s_1, s_2, \dots, s_n, f_1, f_2, \dots, f_n$)

SORT lectures by start times and renumber so that $s_1 \leq s_2 \leq \dots \leq s_n$.

$d \leftarrow 0$. \leftarrow number of allocated classrooms

FOR $j = 1$ TO n

IF (lecture j is compatible with some classroom)

Schedule lecture j in any such classroom k .

ELSE

Allocate a new classroom $d + 1$.

Schedule lecture j in classroom $d + 1$.

$d \leftarrow d + 1$.

RETURN schedule.

→ Proposizione:

L'algoritmo "earliest-start-time-first" può essere implementato in tempo $O(n \log n)$.

Dim:

L'ordinamento in base agli orari di inizio richiede tempo $O(n \log n)$.

Memorizza le aule in una coda prioritaria (chiave = ora di fine dell'ultima lezione)

- per assegnare una nuova aula, INSERT aula nella coda prioritaria.
- per programmare la lezione j nell'aula k , INCREASE-KEY dall'aula k a f_j .
- determinare se la lezione j è compatibile con qualsiasi classe, confrontare s_j con FIND-MIN

Il numero totale di operazioni in coda con priorità è $O(n)$; ciascuna richiede tempo $O(\log n)$.

→ Osservazione. Questa implementazione sceglie un'aula k la cui ora di fine della sua ultima conferenza è la prima.

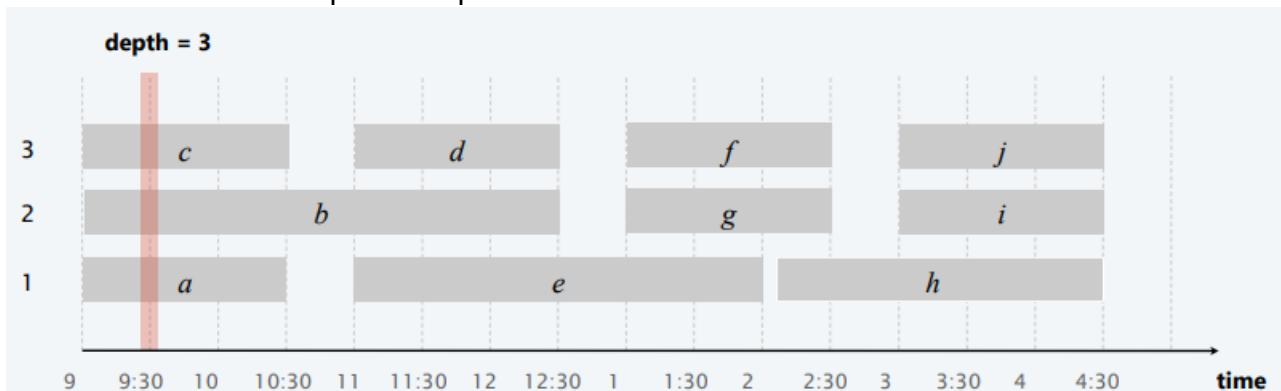
→ Lower bound sulla soluzione ottimale per interval partitioning:

Def = La profondità di una serie di intervalli aperti è il numero massimo di intervalli che contengono un dato punto.

Osservazione chiave = Numero di aule necessarie \geq profondità.

Domanda: Il numero minimo di aule necessarie corrisponde sempre alla profondità?

Risposta: Sì! Inoltre, l'algoritmo earliest-start-time-first trova una pianificazione il cui numero di aule è pari alla profondità.



→ Analisi dell'algoritmo earliest-start-time-first:

Osservazione = l'algoritmo "earliest-start-time-first" non pianifica mai due lezioni incompatibili nella stessa aula.

Teorema= L'algoritmo Earliest-start-time-first è ottimale.

Dim:

- Sia d = numero di aule assegnate dall'algoritmo.
- L'aula d è aperta perché dovevamo programmare una lezione, diciamo j, che è incompatibile con una lezione in ciascuna delle altre d - 1 aule.
- Pertanto, queste conferenze terminano ciascuna dopo sj.
- Poiché abbiamo ordinato in base all'ora di inizio, ciascuna di queste lezioni incompatibili inizia entro e non oltre sj.
- Pertanto, abbiamo d lezioni che si sovrappongono ai tempi sj + epsilon.
- Osservazione chiave => tutti gli orari utilizzano $\geq d$ aule.

LEZ2 – 07/03/2024 (problema Union-find della gestione degli insiemi disgiunti: quickFind , quickUnion ed euristiche per migliorie)

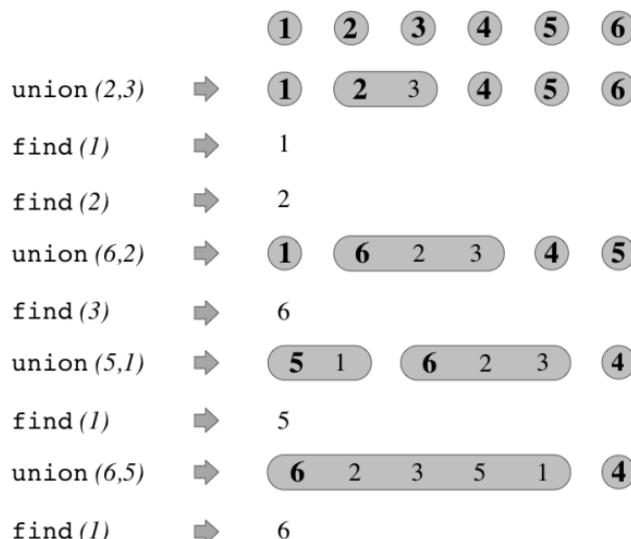
IL PROBLEMA UNION-FIND:

Mantenere una collezione di insiemi disgiunti contenenti elementi distinti (ad esempio, interi in 1 ... n) durante l'esecuzione di una sequenza di operazioni del seguente tipo:

- makeSet(x) = crea il nuovo insieme x={x} di nome x
- union(A,B) = unisce gli insiemi A e B in un unico insieme, di nome A, e distrugge i vecchi insiemi A e B (si suppone di accedere direttamente agli insiemi A,B)
- find(x) = restituisce il nome dell'insieme contenente l'elemento x (si suppone di accedere direttamente all'elemento x)

→ Applicazioni: algoritmo di Kruskal per la determinazione del minimo albero ricoprente di un grafo, calcolo dei minimi antenati comuni, ecc.

→ Esempio: n=6, se ho n elementi posso fare al più n-1 union



→ Obiettivo = progettare una struttura dati che sia efficiente su una sequenza arbitraria di operazioni

→ Idea = Rappresentare gli insiemi disgiunti con una foresta.

Ogni insieme è un albero radicato

La radice contiene il nome dell'insieme (elemento rappresentativo)

→ Approcci elementari basati su alberi (strategie): QuickFind e QuickUnion

QUICK FIND

Usiamo una foresta di alberi di altezza 1 per rappresentare gli insiemi disgiunti.

In ogni albero:

- Radice = nome dell'insieme
- Foglie = elementi (incluso l'elemento rappresentativo, il cui valore è nella radice e dà il nome all'insieme)

→ Realizzazione:

classe QuickFind implementa UnionFind:

dati: $S(n) = O(n)$

una collezione di insiemi disgiunti di elementi *elem*; ogni insieme ha un nome *name*.

operazioni:

makeSet(*elem e*) $T(n) = O(1)$

crea un nuovo albero, composto da due nodi: una radice ed un unico figlio (foglia). Memorizza *e* sia nella foglia dell'albero che come nome nella radice.

union(*name a, name b*) $T(n) = O(n)$

considera l'albero *A* corrispondente all'insieme di nome *a*, e l'albero *B* corrispondente all'insieme di nome *b*. Sostituisce tutti i puntatori dalle foglie di *B* alla radice di *B* con puntatori alla radice di *A*. Cancella la vecchia radice di *B*.

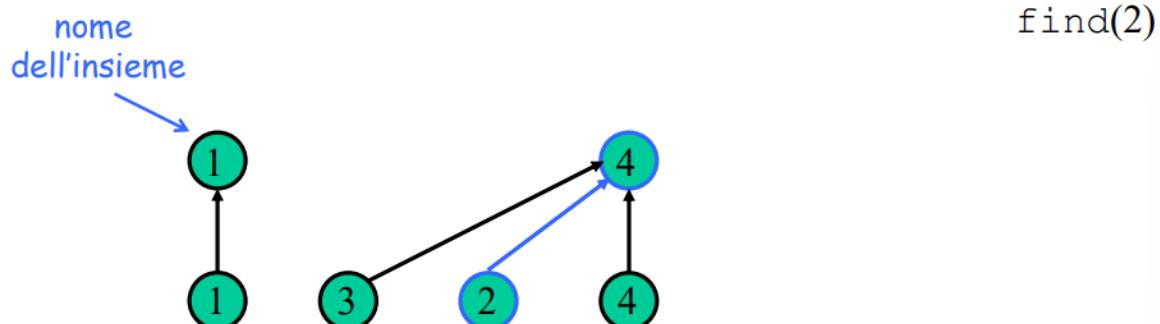
find(*elem e*) → *name* $T(n) = O(1)$

accede alla foglia *x* corrispondente all'elemento *e*. Da tale nodo segue il puntatore al padre, che è la radice dell'albero, e restituisce il nome memorizzato in tale radice.

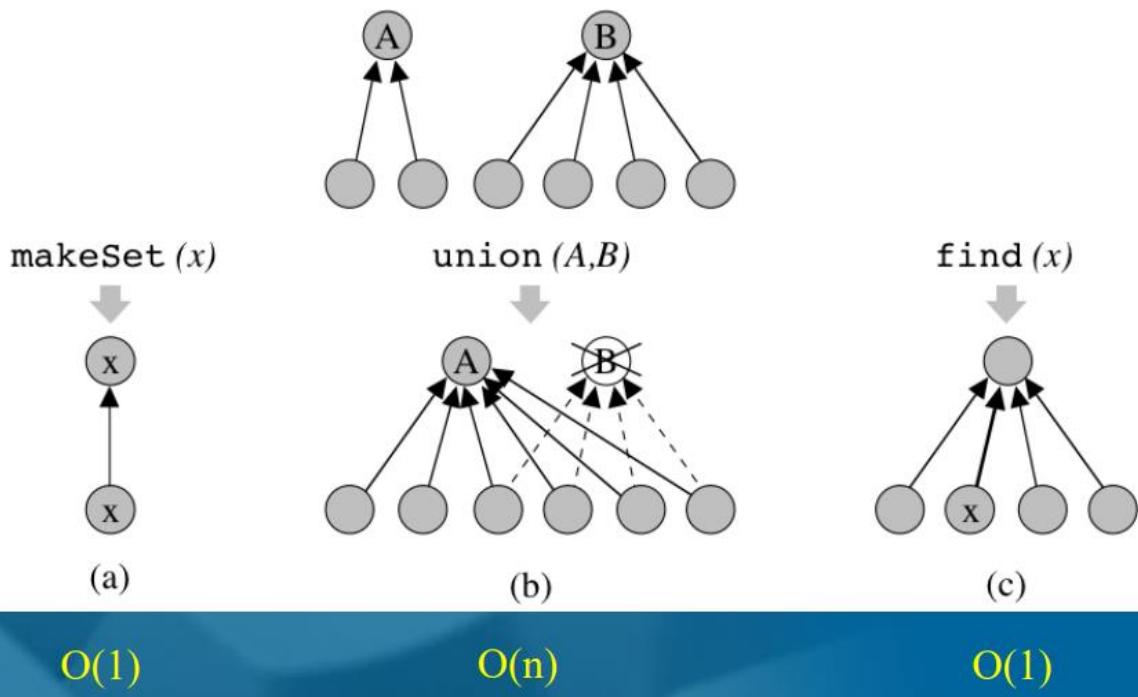
→ Esempio:

Sequenza di operazioni:

makeSet(1)makeSet(3)makeSet(2)makeSet(4)union(2,3)
union(4,2)



→ Complessità:



E se eseguo una sequenza arbitraria di operazioni? UNION DI COSTO LINEARE:

find e makeSet richiedono solo tempo $O(1)$, ma particolari sequenze di union possono essere molto inefficienti:

union ($n-1, n$)	1 cambio puntatore
union ($n-2, n-1$)	2 cambi puntatori
union ($n-3, n-2$)	3 cambi puntatori
:	\vdots
union ($2, 3$)	$n-2$ cambi puntatori
union ($1, 2$)	$n-1$ cambi puntatori

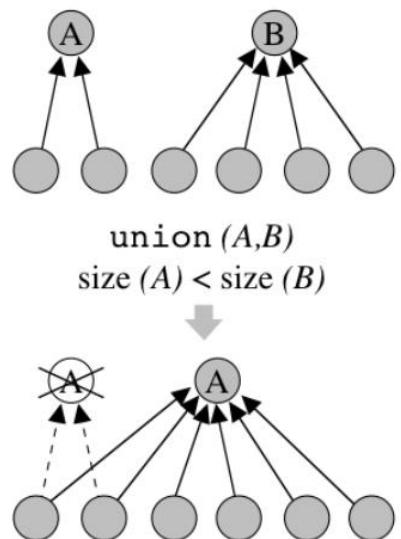
EURISTICA UNION BY SIZE

Utilizzata per migliorare la struttura QuickFind.

Idea: fare in modo che un nodo/elemento non cambi troppo spesso padre.

Nell'unione degli insiemi A e B, attacchiamo gli elementi dell'insieme di cardinalità minore a quello di cardinalità maggiore, e se necessario modifichiamo la radice dell'albero ottenuto (per aggiornare il nome).

Ogni insieme mantiene esplicitamente anche la propria size (numero di elementi).



→ Realizzazione:

classe QuickFindBilanciato implementa UnionFind:

dati: $S(n) = O(n)$

una collezione di insiemi disgiunti di elementi $elem$; ogni insieme ha un nome $name$.

operazioni:

makeSet($elem e$) $T(n) = O(1)$

crea un nuovo albero, composto da due nodi: una radice ed un unico figlio (foglia). Memorizza e sia nella radice che nella foglia dell'albero. Inizializza la cardinalità del nuovo insieme ad 1, assegnando il valore $size(x) = 1$ alla radice x .

find($elem e \rightarrow name$) $T(n) = O(1)$

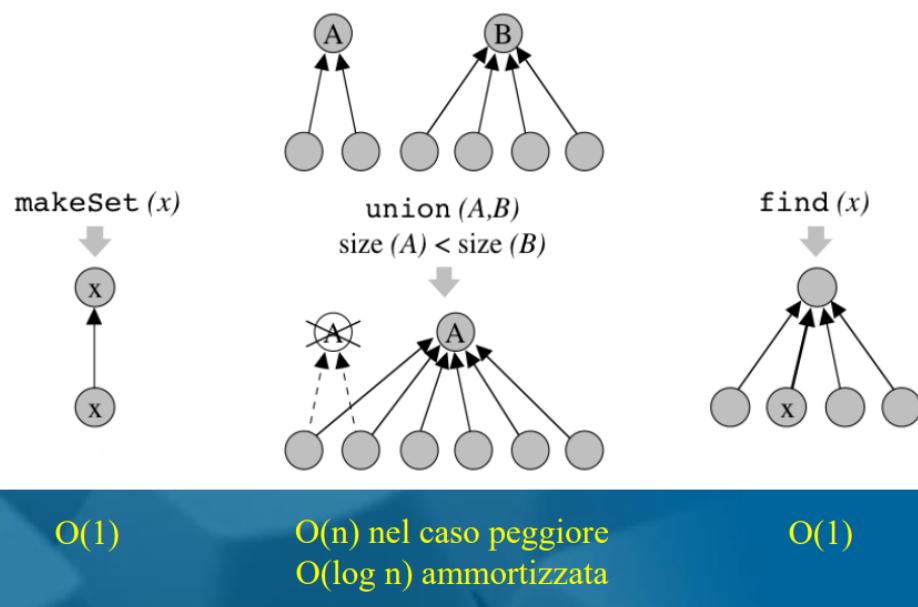
accede alla foglia x corrispondente all'elemento e . Da tale nodo segue il puntatore al padre, che è la radice dell'albero, e restituisce il nome memorizzato in tale radice.

union($name a, name b$) $T_{am} = O(\log n)$

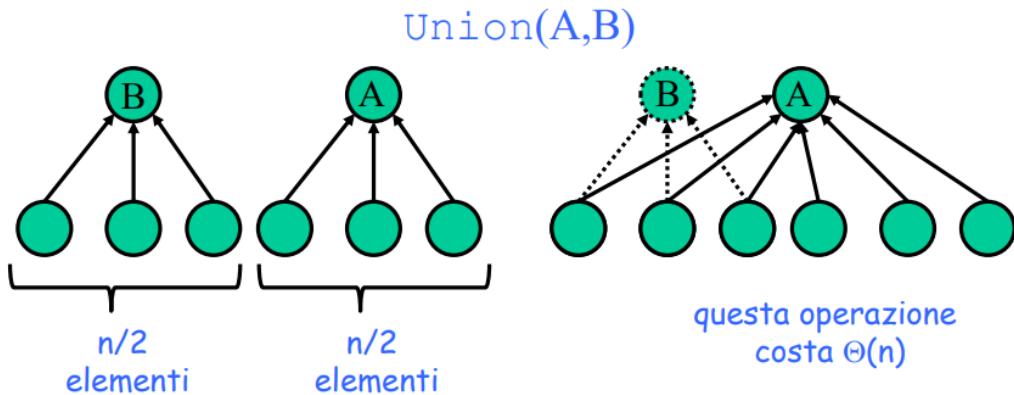
considera l'albero A corrispondente all'insieme di nome a , e l'albero B corrispondente all'insieme di nome b . Se $size(A) \geq size(B)$, muovi tutti i puntatori dalle foglie di B alla radice di A , e cancella la vecchia radice di B . Altrimenti ($size(B) > size(A)$) memorizza nella radice di B il nome A , muovi tutti i puntatori dalle foglie di A alla radice di B , e cancella la vecchia radice di A . In entrambi i casi assegna al nuovo insieme la somma delle cardinalità dei due insiemi originali ($size(A) + size(B)$).

→ Complessità:

T_{am} = tempo per operazione ammortizzato sull'intera sequenza di unioni vedremo che una singola union può costare $\Theta(n)$, ma l'intera sequenza di $n-1$ union costa $O(n \log n)$.



Complessità di una operazione di union:



domanda: quanto costa cambiare padre a un nodo?
...tempo costante!

domanda (cruciale): quante volte può cambiare padre un nodo?
...al più $\log n$!

Analisi ammortizzata:

Vogliamo dimostrare che se eseguiamo m find, n makeSet, e al più n-1 union, il tempo richiesto dall'intera sequenza di operazioni è $O(m + n \log n)$.

Idea della dimostrazione:

- È facile vedere che find e makeSet richiedono tempo $\Theta(m+n)$
- Per analizzare le operazioni di union, ci concentriamo su un singolo nodo/elemento e dimostriamo che il tempo speso per tale nodo è $O(\log n)$ ⇒ in totale, tempo speso è $O(n \log n)$
- Quando eseguiamo una union, per ogni nodo che cambia padre pagheremo tempo costante
- Osserviamo ora che ogni nodo può cambiare al più $O(\log n)$ padri, poiché ogni volta che un nodo cambia padre la cardinalità dell'insieme al quale apparterrà è almeno doppia rispetto a quella dell'insieme cui apparteneva!
 - o all'inizio un nodo è in un insieme di dimensione 1,
 - o poi se cambia padre in un insieme di dimensione almeno 2,
 - o all'i-esimo cambio è in un insieme di dimensione almeno 2^i

⇒ il tempo speso per un singolo nodo sull'intera sequenza di n union è $O(\log n)$.

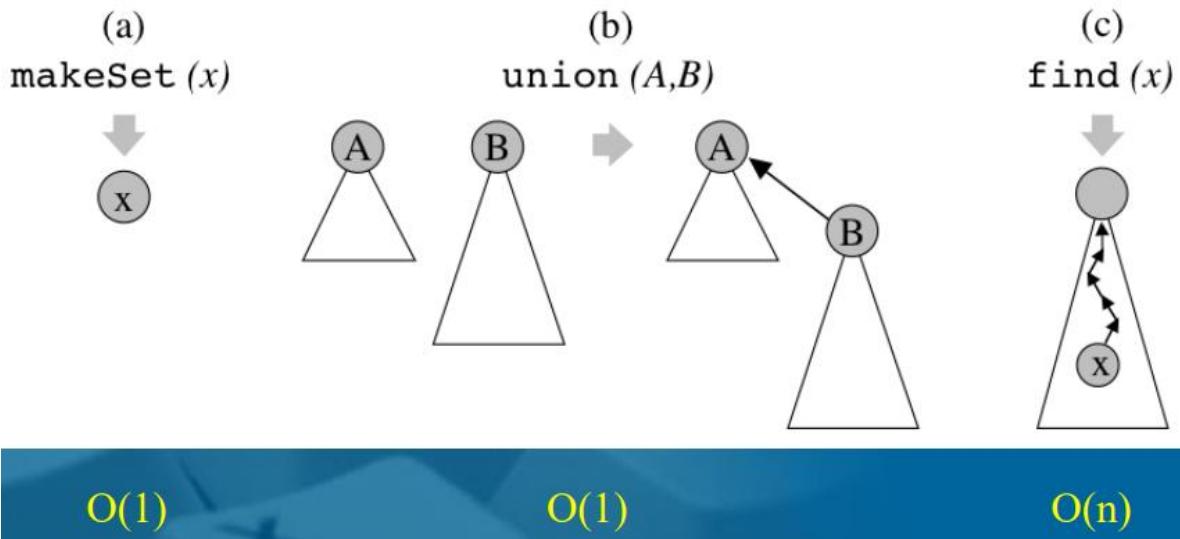
⇒ L'intera sequenza di operazioni costa $O(m+n+n \log n)=O(m+n \log n)$.

QUICK UNION

Usiamo una foresta di alberi di altezza anche maggiore di 1 per rappresentare gli insiemi disgiunti. In ogni albero:

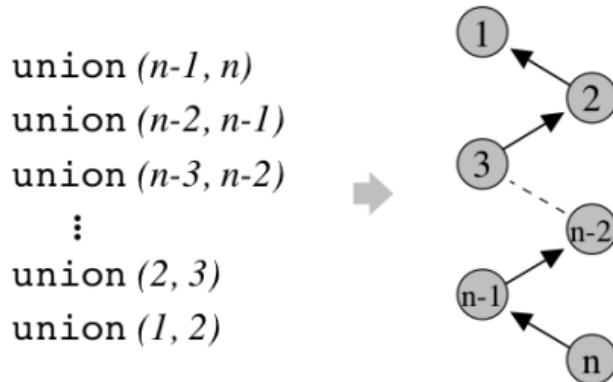
- Radice = elemento rappresentativo dell'insieme
- Rimanenti nodi = altri elementi (escluso l'elemento nella radice)

→ Implementazioni e complessità delle operazioni:



e se eseguo una sequenza arbitraria di operazioni?

Find di costo lineare: particolari sequenze di union possono generare un albero di altezza lineare, e quindi la find è molto inefficiente (costa $n-1$ nel caso peggiore)



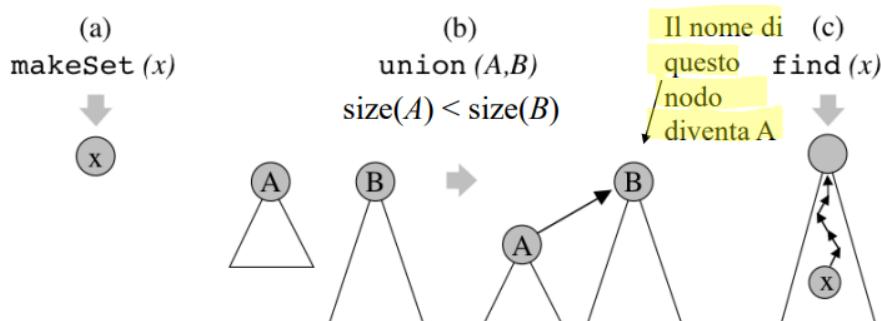
⇒ Se eseguiamo n makeSet, $n-1$ union come sopra, seguite da m find, il tempo richiesto dall'intera sequenza di operazioni è $O(n+n-1+mn)=O(mn)$

EURISTICA UNION BY SIZE

Utilizzata per migliorare la struttura QuickUnion.

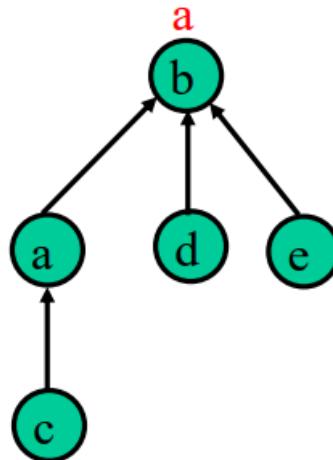
Idea: fare in modo che per ogni insieme l'albero corrispondente abbia altezza piccola.

Bilanciamento in alberi QuickUnion tramite Union by size: nell'unione degli insiemi A e B , rendiamo la radice dell'albero con meno nodi figlia della radice dell'albero con più nodi



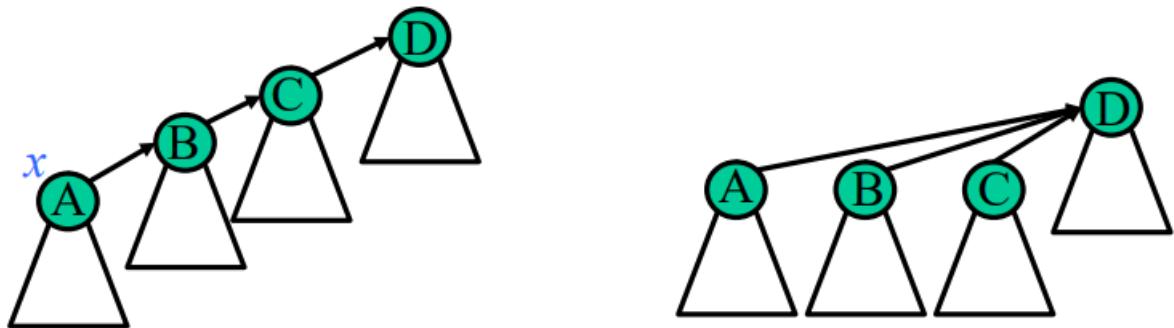
Esempio:

```
makeSet(a)  makeSet(c)  makeSet(b)  makeSet(d)  union(b,d)
union(a,c)  makeSet(e)  union(e,b)  union(a,e)
```



- Lemma: Con la union by size, dato un albero QuickUnion con size (numero di nodi) s e altezza h vale che $s \geq 2^h$
 - ⇒ L'operazione find richiede tempo $O(\log n)$
 - ⇒ L'intera sequenza di operazioni costa $O(n+m \log n)$.

UN'ULTERIORE EURISTICA: COMPRESSIONE DEI CAMMINI



Idea: quando eseguo $\text{find}(x)$ e attraverso il cammino da x alla radice, comprimo il cammino, ovvero rendo tutti i nodi del cammino figli della radice.

Intuizione: $\text{find}(x)$ ha un costo ancora lineare nella lunghezza del cammino attraversato, ma prossime find costeranno di meno.

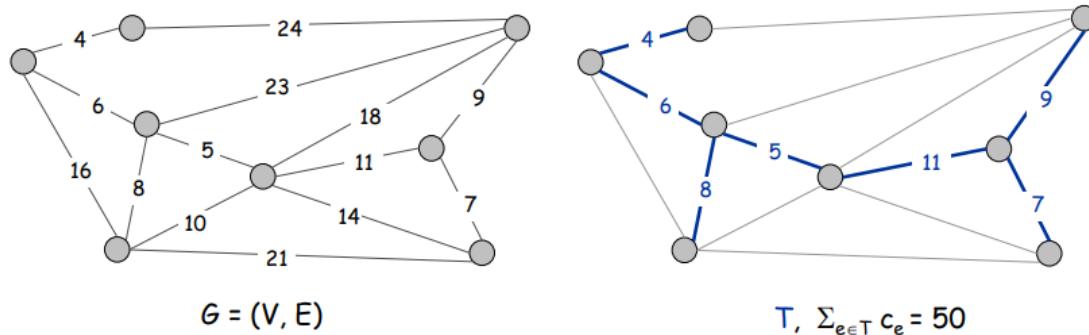
- Teorema (Tarjan&van Leeuwen)

Usando in QuickUnion le euristiche di union by rank (o by size) e compressione dei cammini, una qualsiasi sequenza di n makeSet , $n-1$ union e m find hanno un costo di $O(n+m \alpha(n+m,n))$

con $\alpha(x,y)$: funzione inversa della funzione di Ackermann (maggiori dettagli su questa funzione sulle slide, omessa perché non chiesta all'esame).

MINIMUM SPANNING TREE (MST)

Dato un grafo connesso $G = (V, E)$ con pesi degli archi a valori reali c_e , un MST è un sottoinsieme degli archi $T \subseteq E$ tali che T è un albero ricoprente la cui somma dei pesi dei bordi è minimizzata.



Thm: ci sono n^{n-2} spanning trees di K_n .

IL PROBLEMA MST

Input = un grafo connesso e non orientato $G = (V, E)$ con pesi reali degli archi c_e .
Soluzione ammissibile = uno spanning tree T di G , cioè un albero $T = (V, F)$ con $T \subseteq E$ (raggiungendo tutti i vertici di G).

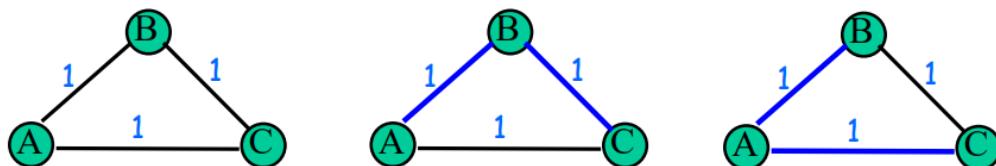
Misura (minimizzare): il peso (o costo) di T , ovvero $c(T) = \sum_{e \in T} c_e$.

→ Applicazioni = MST è un problema fondamentale in diverse applicazioni, tra cui:

- Progettazione della rete (telefono, elettrico, idraulico, cavo TV, computer, strada)
- Algoritmi di approssimazione per problemi NP-difficili (Problema del commesso viaggiatore)
- Applicazioni indirette (percorsi massimi di colli di bottiglia, apprendimento delle caratteristiche salienti per la verifica del volto in tempo reale)
- Cluster analysis (divisione in gruppi).

UNICITA' DI MST

MST non è unico in generale.

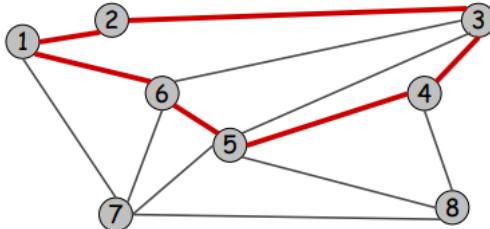


→ Proprietà: se G ha pesi distinti allora l'MST è unico

CICLI , CUTS E CUTSET

→ Definizioni:

- 1) Ciclo (cycle) = insieme di archi di forma a-b, b-c, c-d, ..., y-z, z-a.

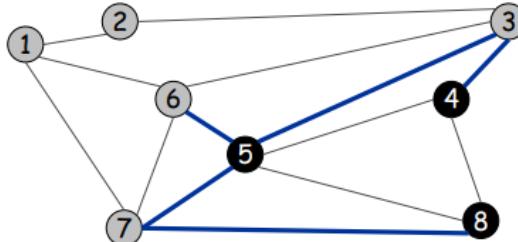


Cycle $C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1$

- 2) Cut (taglio) = è un sottoinsieme di nodi S.

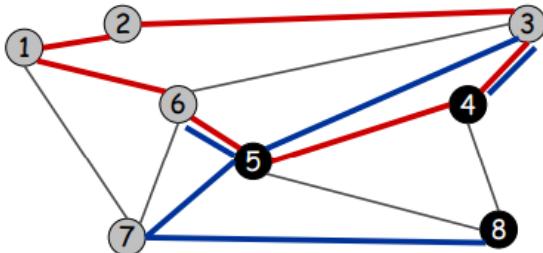
(talvolta cut definito anche come partizione di V in S e $V \setminus S$)

- 3) Cutset = il corrispondente insieme di tagli D di un taglio S è il sottoinsieme di bordi con esattamente un punto finale in S.



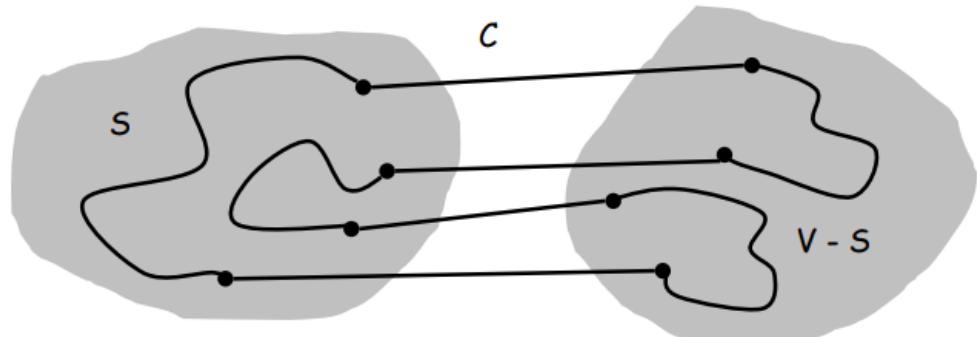
Cut $S = \{4, 5, 8\}$
Cutset $D = 5-6, 5-7, 3-4, 3-5, 7-8$

→ Osservazione(Cycle-Cut intersection): Un ciclo e un cut si intersecano in un numero pari di bordi.



Cycle $C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1$
Cutset $D = 3-4, 3-5, 5-6, 5-7, 7-8$
Intersection = 3-4, 5-6

Dim tramite immagine:



CUT PROPERTY E CICLE PROPERTY (PROPRIETA' FONDAMENTALI)

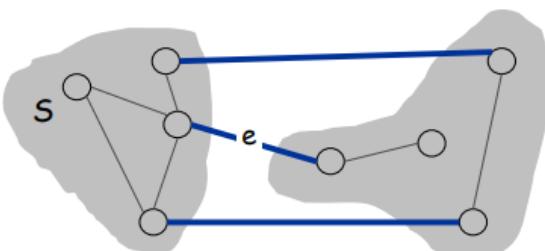
→ Cut property (proprietà di taglio)

Sia S un qualsiasi sottoinsieme di nodi e sia e l'arco di costo minimo con esattamente un punto finale in S . Allora esiste un MST contenente e .

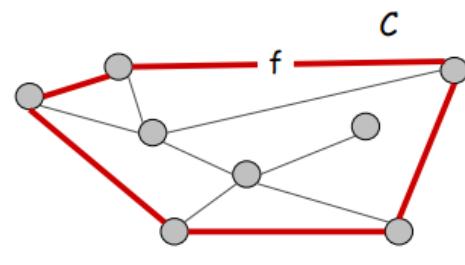
→ Cycle property (proprietà del ciclo)

Sia C un ciclo qualsiasi e f l'arco di costo massimo appartenente a C .

Allora esiste un MST che non contiene f .



e is in some MST



f is not in some MST

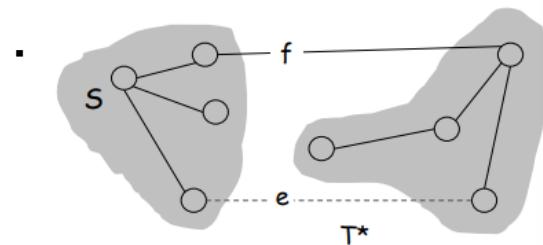
Dimostrazioni:

1) Proprietà di taglio.

Sia S un qualsiasi sottoinsieme di nodi e sia e un costo minimo bordo con esattamente un punto finale in S . Allora esiste un MST T^* contenente e .

Dim(argomentazione di scambio):

- Supponiamo che e non appartenga a T^* .
- L'aggiunta di e a T^* crea un ciclo C in T^* .
- L'arco e è sia nel ciclo C che nel cutset D corrispondente a $S \Rightarrow$ esiste un altro arco, diciamo f , che è sia in C che in D .
- Anche $T' = T^* \cup \{ e \} - \{ f \}$ è uno spanning tree.
- Poiché $c_e \leq c_f$, $\text{costo}(T') \leq \text{costo}(T^*)$.
- Allora T' è un MST contenente e

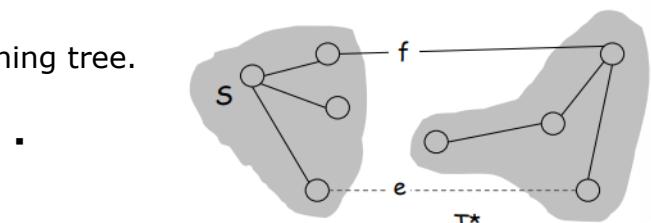


2) Proprietà del ciclo.

Sia C un ciclo qualsiasi di G e sia f il margine di costo massimo appartenente a C . Allora esiste un MST T^* che non contiene f .

Dim(exchange argument):

- Supponiamo che f appartenga a T^* .
- L'eliminazione di f da T^* crea un taglio S in T^* .
- Il bordo f è sia nel ciclo C che nel cutset D corrispondente a $S \Rightarrow$ esiste un altro arco, diciamo e , che è sia in C che in D .
- Anche $T' = T^* \cup \{ e \} - \{ f \}$ è uno spanning tree.
- Poiché $c_e \leq c_f$, $\text{costo}(T') \leq \text{costo}(T^*)$.
- Allora T' è un MST che non contiene f .



ALGORITMI:

- Algoritmo di Kruskal.
Inizia con $T = \emptyset$. Considera gli archi in ordine crescente di costo.
Inserisci l'arco e in T a meno che ciò non crei un ciclo.
- Algoritmo di eliminazione inversa.
Inizia con $T = E$. Considera gli archi interni in ordine decrescente di costo.
Elimina l'arco e da T a meno che ciò non disconnetta T .
- Algoritmo di Prim.
Inizia con alcuni nodi radice e fai crescere in modo greedy un albero T da s verso l'esterno. Ad ogni passaggio, aggiungi l'arco più economico e a T che ha esattamente un punto finale in T .

Osservazione. Tutti e tre gli algoritmi producono un MST.

ALGORITMO DI KRUSKAL

Inizia con $T = \emptyset$. Considera gli archi in ordine crescente di costo. Inserisci l'arco e in T a meno che ciò non crei un ciclo.

Osservazione: un'implementazione efficiente dell'algoritmo di Kruskal utilizza una struttura dati Union-Find per:

- mantenere le componenti connesse delle soluzioni correnti
- verificare se l'arco corrente forma un ciclo (con la soluzione corrente)

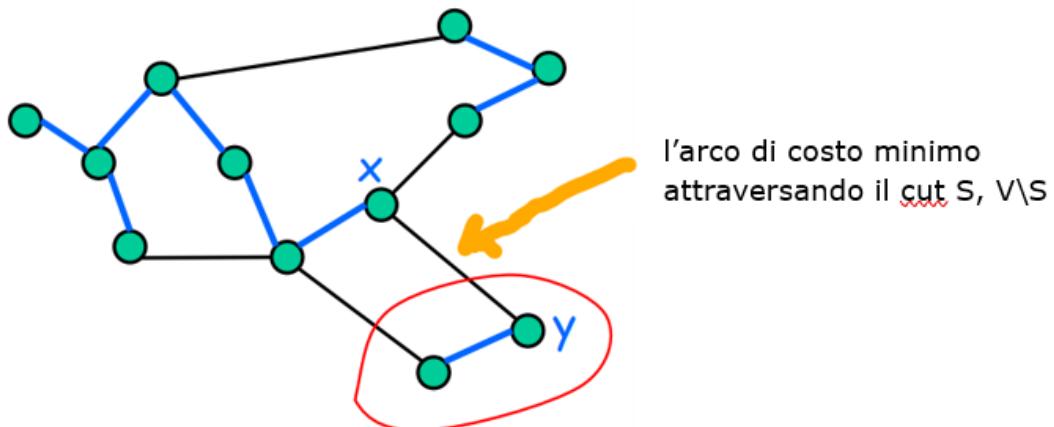
➔ Pseudocodice:

```
algorithm Kruskal (graph  $G=(V,E,c)$ )
    UnionFind UF
     $T=\emptyset$ 
    sort the edges in ascending order of costs
    for each vertex  $v$  do UF.makeset( $v$ )
    for each edge  $(x,y)$  in order do
         $T_x=UF.find(x)$ 
         $T_y=UF.find(y)$ 
        if  $T_x \neq T_y$  then
            UF.union( $T_x, T_y$ )
            add edge  $(x,y)$  to  $T$ 
    return  $T$ 
```

→ Correttezza:

quando l'algoritmo decide di aggiungere l'arco (x,y) alla soluzione

poiché l'algoritmo guarda gli archi in ordine crescente di costo...



consideriamo l'insieme S dei vertici appartenenti alla stessa componente连通的 di y

→ Tempo di esecuzione algoritmo di Kruskal:

algorithm Kruskal (graph $G=(V,E,c)$)

UnionFind UF

$T = \emptyset$

sort the edges in ascending order of costs

for each vertex v **do** $UF.makeset(v)$

for each edge (x,y) in order **do**

$T_x = UF.find(x)$

$T_y = UF.find(y)$

if $T_x \neq T_y$ **then**

$UF.union(T_x, T_y)$

add edge (x,y) to T

return T

- sorting the edges: $O(m \log m) = O(m \log n)$

- Union-Find operations:

- n makeset ops

- $n-1$ union ops

- m find ops

→ $O(m \log n + UF(m,n))$

-using QuickFind with *union by size*

$O(m \log n + m + n \log n) = O(m \log n)$

-using QuickUnion with *union by size*

$O(m \log n + m \log n + n) = O(m \log n)$

$O(m \log n)$

PROBLEMA MST (albero dei cammini minimi)

Input = un grafo non orientato, connesso e pesato $G=(V,E)$ con archi di peso c_e di valori reali

Soluzione ammissibile = uno spanning tree T di G , cioè un albero $T=(V,F)$ con $T \subseteq E$ (raggiungendo tutti i vertici di G)

Misura (minimizzare) = il peso (o costo) di T , ovvero $c(T)= \sum_{e \in T} c_e$

CUT PROPERTY E CYCLE PROPERTY

Cut property = Sia S un qualsiasi sottoinsieme di nodi e sia e un arco di costo minimo con esattamente un punto finale in S . Allora esiste un MST contenente e .

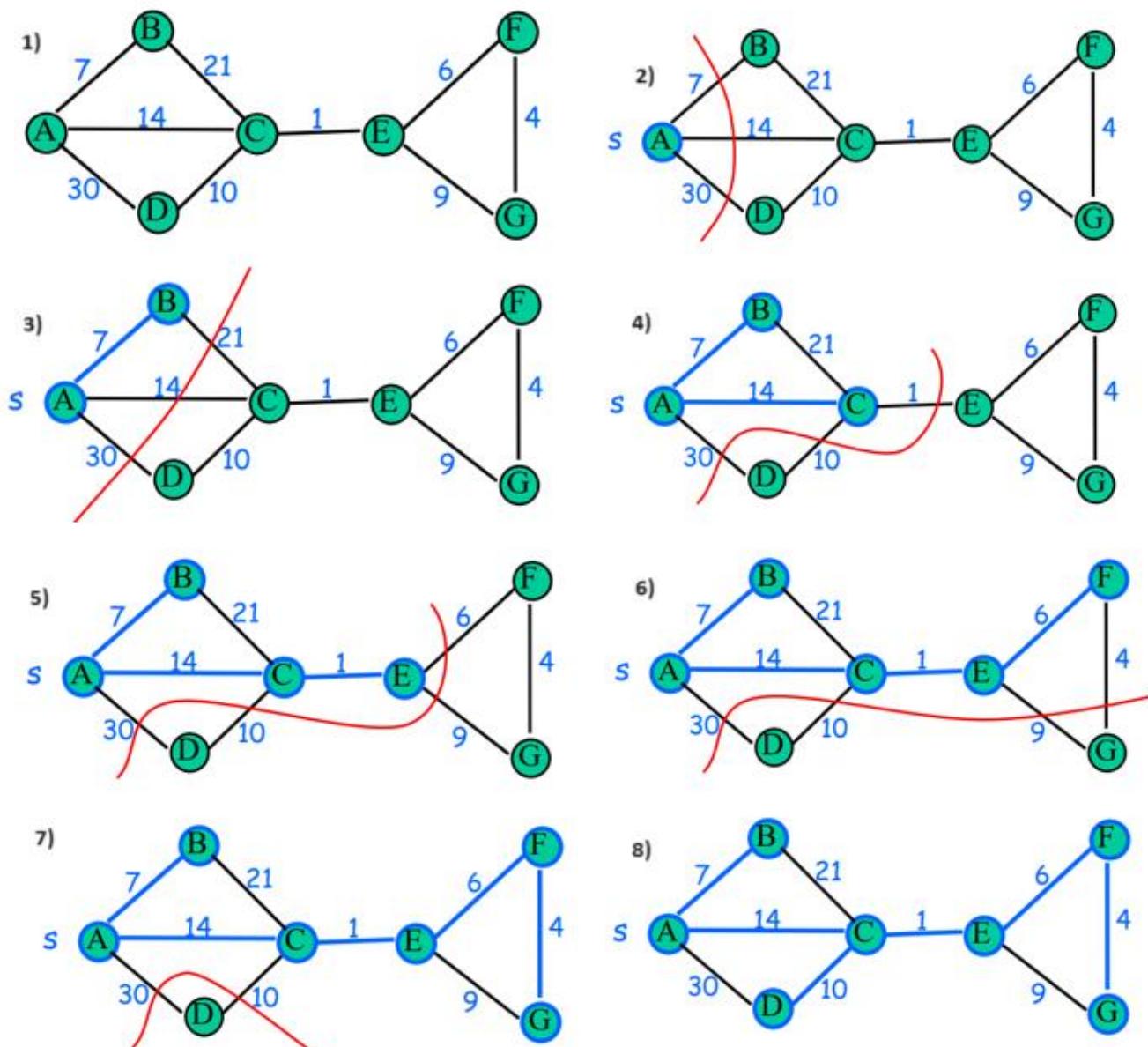
Cycle property = Sia C un ciclo qualsiasi e f l'arco di costo massimo appartenente a C . Allora esiste un MST che non contiene f .

ALGORITMO DI PRIM

Inizia con un nodo radice s e fai crescere in modo greedy un albero T da s verso l'esterno. Ad ogni passaggio, aggiungi l'arco più economico e a T che ne ha esattamente un punto finale(endpoint) in T .

Correttezza: immediata conseguenza della cut property, usata esattamente $n-1$ volte.

Esecuzione:



Tempo di esecuzione:

→ Un'implementazione semplice (e inefficiente):

Per $n-1$ volte, trovare l'arco più economico che attraversa il taglio indotto dall'albero parziale corrente in tempo $O(m)$.

Tempo di esecuzione totale: $O(mn)$.

→ Un'implementazione molto più rapida:

- Mantenere l'insieme dei nodi esplorati S .
- Utilizzare una coda con priorità (struttura dati) per mantenere i nodi inesplorati.
- Per ogni nodo inesplorato v , la priorità è il costo di collegamento $a[v] =$ costo del più economico arco incidente in v avente l'altro endpoint in S .

→ Quindi tempo di esecuzione:

- tempo $O(m+n)$ più il costo delle operazioni in coda prioritaria
- n inserts, n operazioni minime di delete, m operazioni di decrease key
- $O(n^2)$ con un array; $O(m \log n)$ con un heap binario;
- $O(m + n \log n)$ con gli heap di Fibonacci

⇒

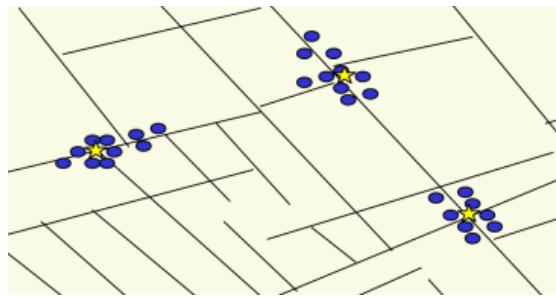
$$O(m + n \log n)$$

Pseudocodice:

```
Prim(G, s) {
    foreach (v ∈ V) a[v] ← ∞
    a[s] ← 0
    Initialize an empty priority queue Q
    foreach (v ∈ V) insert v onto Q with priority a[v]
    Initialize set of explored nodes S ← ∅
    Initialize T to the tree containing only 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]))
                make u parent of v in T
                decrease priority a[v] to ce
    }
    return T
}
```

CLUSTERING



Clustering = dato un insieme U di n oggetti etichettati p_1, \dots, p_n classificarli in gruppi coerenti
(esempi di oggetti possono essere foto, documenti,...)

Distance function = valore numerico che specifica la "vicinanza" di due oggetti
(esempio di vicinanza può essere il numero di pixel corrispondenti di cui le intensità differiscono di una certa soglia)

Problema fondamentale = dividere in gruppi in modo che i punti appartenenti a gruppi diversi siano distanti

CLUSTERING DI SPAZIO MASSIMO

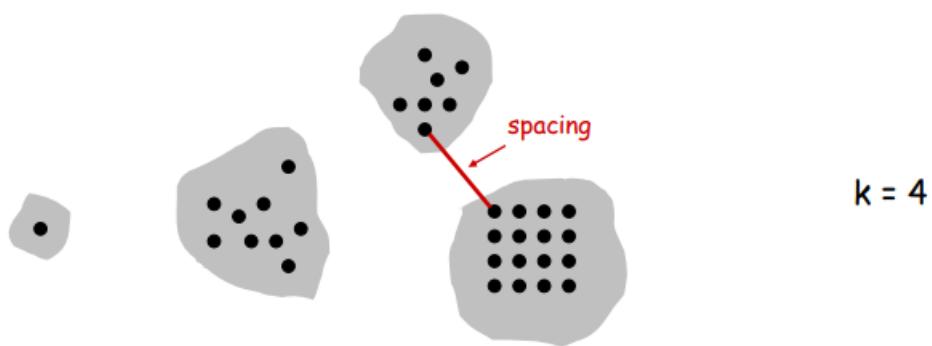
k-clustering = divisione oggetti in k gruppi non vuoti.

Distance function = Supponiamo che soddisfi diverse proprietà naturali.

- $d(p_i, p_j) = 0$ se e solo se $p_i = p_j$ (identità degli indiscernibili)
- $d(p_i, p_j) \geq 0$ (non negatività)
- $d(p_i, p_j) = d(p_j, p_i)$ (simmetria)

Spacing = distanza minima tra qualsiasi coppia di punti in cluster diversi

Clustering di spazio massimo = dato un intero k , trova un k -clustering di spaziatura massima



ALGORITMO DI CLUSTERING DI TIPO GREEDY

Algoritmo di k -clustering a collegamento singolo (Single-linkage k -clustering)

- Formare un grafo sull'insieme dei vertici U , corrispondente a n cluster.
- Trovare la coppia di oggetti più vicina in modo tale che ciascun oggetto sia in cluster diversi e aggiungere un arco tra di loro.
- Ripeti $n-k$ volte finché non ci sono esattamente k cluster.

Osservazione chiave = Questa procedura è proprio l'algoritmo di Kruskal
(tranne che ci fermiamo quando ci sono k componenti collegati)

Osservazione = Equivalente a trovare un MST ed eliminare il maggior numero di $k-1$ archi costosi

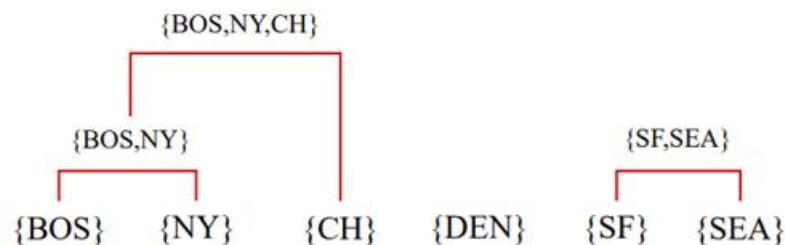
Clustering gerarchico = Eseguendo l'algoritmo di Kruskal fino alla fine si produce implicitamente un clustering gerarchico, cioè un k -clustering per ciascun valore di $k=n, n-1, \dots, 1$.



	BOS	NY	CHI	DEN	SF	SEA
BOS	0	206	963	1949	3095	2979
NY		0	802	1771	2934	2815
CHI			0	966	2142	2013
DEN				0	1235	1307
SF					0	808
SEA						0

Single linkage

$k=3$



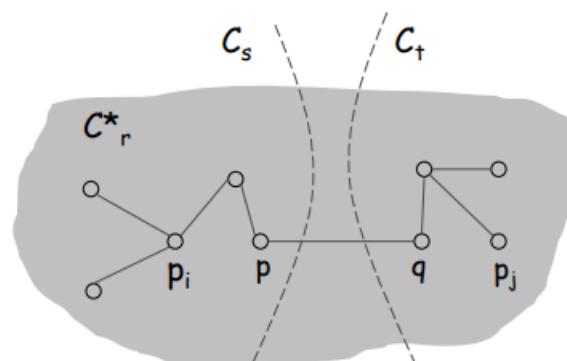
ANALISI ALGORITMO CLUSTERING DI TIPO GREEDY

Teorema: Dato C^* che denota il clustering C^*_1, \dots, C^*_k formato eliminando i $k-1$ esimi archi più costosi di un MST allora C^* è un k -cluster di spaziatura massima.

Dim:

Sia dato C che denota qualche altro clustering C_1, \dots, C_k

- Lo spacing di C^* è la lunghezza d^* del $(k-1)$ esimo arco più costoso di MST.
- Siano p_i, p_j nello stesso cluster in C^* , diciamo C^*_r , ma cluster diversi in C , diciamo C_s e C_t
- Qualche arco (p, q) sul percorso p_i-p_j in C^*_r si estende su due diversi cluster in C .
- Tutti i bordi sul percorso p_i-p_j ha lunghezza $\leq d^*$ da quando Kruskal li ha scelti.
- Lo spacing di C è $\leq d^*$ poiché p_i e p_j sono in cluster diversi •



ESERCIZIO1

Sia $G = (V, E)$ un grafo con pesi positivi distinti sugli archi ed $e \in E$ un arco di G .
 Progettare un algoritmo lineare in grado di determinare se esiste un MST di G che contiene l'arco e .

Input =grafo non diretto pesato

Goal =capire se $e \in \text{MST}(G)$

Tempo soluzione semplice = $O(m+n\log n)$ oppure $O(m\log n)$

Tempo goal = $O(m+n)$

Idea: per dire $e \notin \text{MST}$ si deve cercare un ciclo per cui e è l'arco di peso massimo

Soluzione: immaginare e ragionare sulla struttura che cerco per arrivare alla soluzione

Alg(G, e)

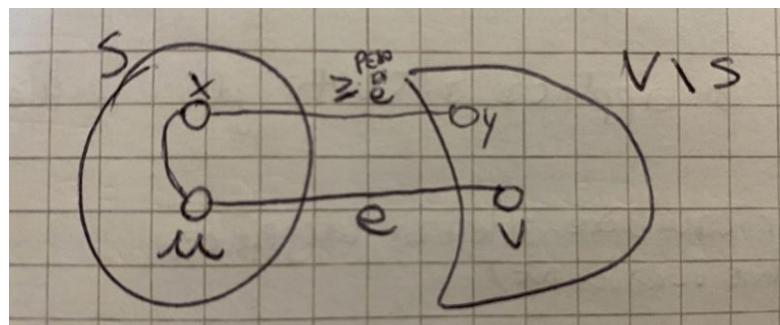
- costruisci il grafo G' da G ma solo con archi più leggeri di e
- fare visita di G' per capire se v è raggiungibile da u
- se si rispondere $e \notin \text{MST}(G)$, altrimenti rispondere $e \in \text{MST}(G)$

Prop correttezza: $e \notin \text{MST}(G)$ se e solo se v è raggiungibile da u in G'

- Dim: dimostro " \leq " del se e solo se prendendo un cammino P da u a v in G'
 $\rightarrow P \cup \{e\}$ è un ciclo in cui e è MAX \rightarrow cycle property $e \notin \text{MST}$.

Dimostro ora " \geq " come segue:

$S = \{w \in V \text{ t.c. } w \text{ è raggiungibile da } u \text{ in } G'\}$



(u, v) attraversa $(S, V \setminus S)$ quindi e è MIN che attraversa $(S, V \setminus S)$

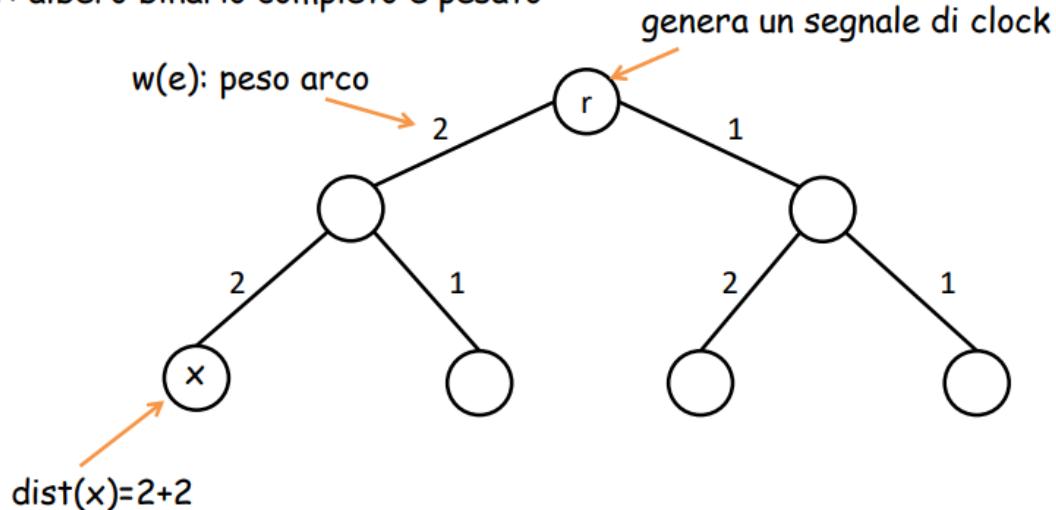
$\Rightarrow e \in \text{MST}(G)$

OSSERVAZIONE DA ESERCIZIO: Cycle e cut properties sono legate!!

ESERCIZIO2

SINCRONIZZAZIONE DI CIRCUITI

T: albero binario completo e pesato



Goal: sincronizzare le foglie (metterle tutte alla stessa distanza)

Come: posso incrementare i pesi degli archi

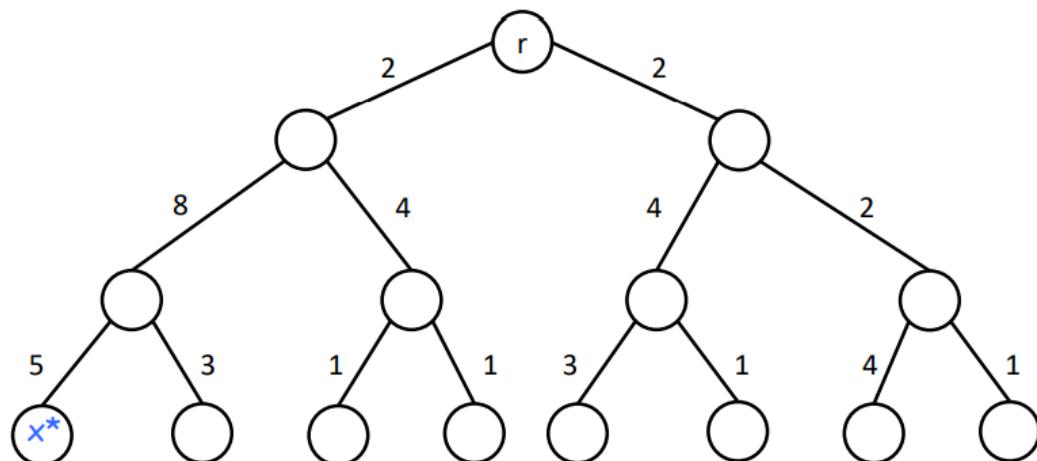
Misura (da minimizzare): peso totale dell'albero risultante (min somma incrementi)

Input: albero binario completo e pesato T

Soluzione ammissibile: nuova pesatura w' per T con $w'(e) \geq w(e)$ tale che tutte le foglie rispetto a w' hanno la stessa distanza dalla radice

Misura (da minimizzare): $w'(T) = \sum_e w'(e)$

Soluzione:



x^* : foglia più lontana a distanza $L=15$

idea: mettere tutte le foglie a distanza $L=15$

intuizione: conviene aumentare gli archi "alti"

definizione: arco e copre una foglia x se il cammino dalla radice verso x passa per e

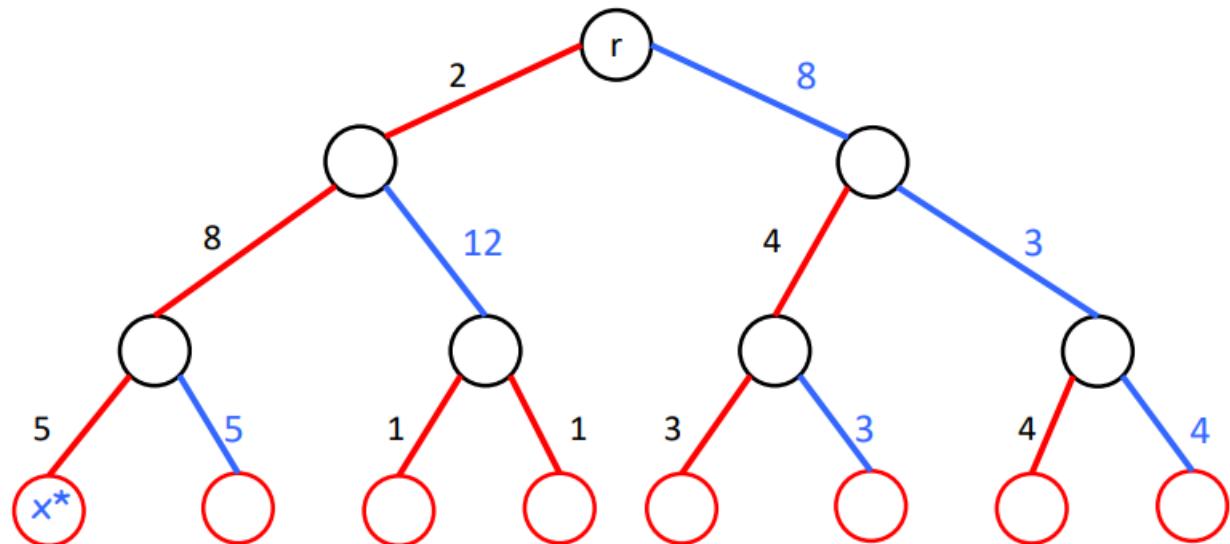
F: insieme delle foglie

F_e : insieme delle foglie coperte da e

per ogni sottoinsieme di archi X
 $\{F_e\}_{e \in X}$ famiglia laminare di insiemi:
 per ogni e, e' in X
 - $F_e \subseteq F_{e'}$ oppure $F_e \cap F_{e'} = \emptyset$

algoritmo greedy:

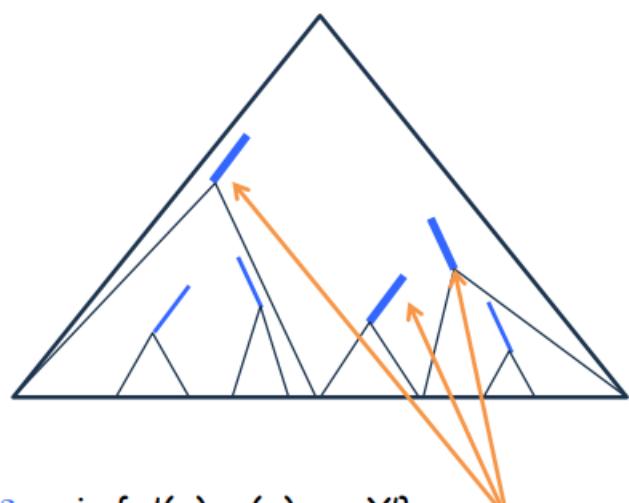
- marca tutti gli archi del cammino $r-x^*$
- considera gli archi di T top-down (in ordine ascendente di profondità)
 - se l'arco e non è marcato
 - alza il peso di e finché una foglia $x \in F_e$ non diventa a distanza L
 - marca tutti gli archi lungo il cammino verso x
- restituisci la nuova pesatura



Dimostrazione di ottimalità:

claim: una soluzione ottima mette tutte le foglie a distanza L

supponi Opt mette tutte le foglie a distanza $L' > L$



X : archi incrementati da Opt

poiché tutte le foglie hanno aumentato la loro distanza:

$$\cup_{e \in X} F_e = F$$

$X' \subseteq X$: tale che ogni foglia è coperta da un solo $e \in X'$

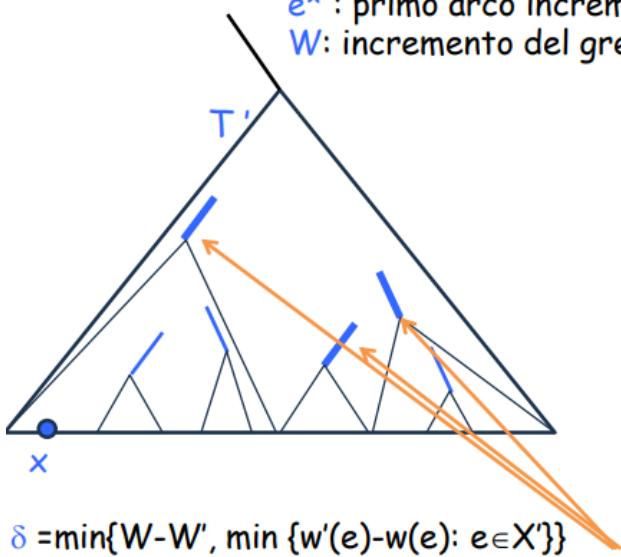
$$\delta = \min \{w'(e) - w(e) : e \in X'\}$$

decremento di δ e ottengo una soluzione ammissibile strettamente migliore

allora Opt non era ottima: assurdo!

idea: faccio vedere che il greedy non sbaglia mai

ottimalità



e^* : primo arco incrementato dal greedy

W : incremento del greedy che porta x a distanza L

assumi Opt incrementa e^* di $W' < W$

X : archi incrementati da Opt in T'

poiché tutte le foglie di T' devono aumentare la loro distanza:

$$\cup_{e \in X} F_e = \text{foglie di } T'$$

$X' \subseteq X$: tale che ogni foglia di T' è coperta da un solo $e \in X'$

nota: $|X'| \geq 2$

$$\delta = \min\{W - W', \min \{w'(e) - w(e) : e \in X'\}\}$$

allora Opt non era ottima: assurdo!

decremento di δ e incremento e^* di δ

ottengo una soluzione ammissibile strettamente migliore

Quindi l'ottimo deve incrementare e^* come il greedy: stesse argomentazioni per i prossimi archi.

SOMMARIO LEZIONE:

- La tecnica della programmazione dinamica all'opera
- Un problema interessante: insieme indipendente di peso massimo (per un grafo a cammino)
 - perché le altre tecniche non funzionano
 - ragionare sulla struttura/proprietà della soluzione
- Un algoritmo di programmazione dinamica con complessità lineare
- Principi generali della programmazione dinamica – sottoproblemi, relazioni fra sottoproblemi, tabelle

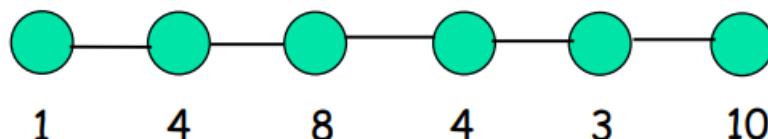
INSIEME INDIPENDENTE DI PESO MASSIMO (SU GRAFI A CAMMINO)

Input: Un cammino G di n nodi. Ogni nodo vi ha un peso w_i .

Goal: trovare un insieme indipendente di peso massimo, ovvero un insieme S di nodi tale che:

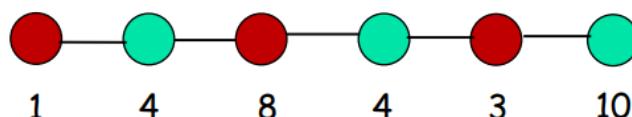
i) S è un II (Insieme Indipendente)

ii) $w(S) = \sum_{v_i \in S} w_i$ è più grande possibile

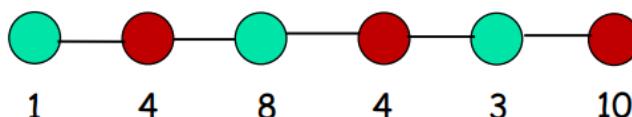


un insieme indipendente (II) di G è un sottoinsieme di nodi che non contiene due nodi adiacenti, ovvero per ogni coppia di nodi dell'insieme i due nodi non sono collegati da un arco.

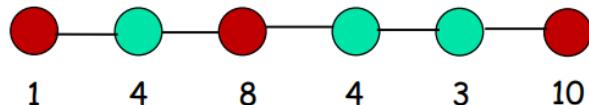
Esempio:



$S = \{v_1, v_3, v_5\}$ un insieme
 $w(S) = 12$ indipendente



$S = \{v_2, v_4, v_6\}$ un insieme
 $w(S) = 18$ indipendente migliore



$$S = \{v_1, v_3, v_6\}$$

$$w(S) = 19$$

un insieme
indipendente
ancora
migliore
(è un II di peso
massimo!)

Progettiamo un algoritmo: che approccio utilizzare

FORZA BRUTA: ENUMERAZIONE

Idea: enumeriamo tutti i sottoinsiemi degli n nodi, per ognuno verifichiamo che è un insieme indipendente, ne calcoliamo il peso e teniamo quello di peso massimo.

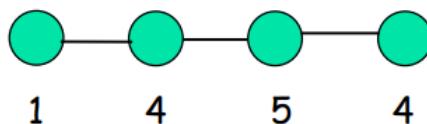
Domanda: quanti sottoinsiemi guardiamo? risposta: tanti! (troppi) ... sono 2^n !!!

APPROCCIO GOLOSO (GREEDY)

Idea: costruisco la soluzione in modo incrementale scegliendo ogni volta il nodo indipendente di valore massimo.

Domanda: funziona?

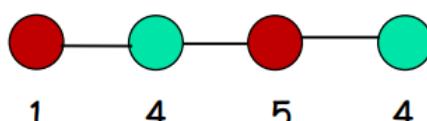
Risposta: no!



istanza



soluzione
ottima



soluzione
algoritmo
greedy

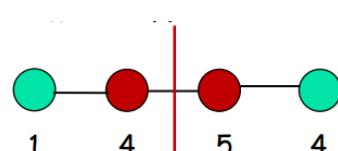
DIVIDE ET IMPERA

Idea: divido il cammino a metà, calcolo ricorsivamente l'II di peso massimo sulle due metà e poi ricombino le soluzioni

Domanda: è corretto?

Domanda: posso risolvere (efficientemente) i conflitti che ho quando ricombino?

Sembra difficile...



difficile ricombinare le
soluzioni!!!!

Cosa non sta funzionando?

...non stiamo capendo davvero la struttura del problema.

...la comprensione della struttura del problema ci porterà a sviluppare un nuovo approccio.

CERCANDO UN NUOVO APPROCCIO

Passaggio critico: ragionare sulla struttura/proprietà della soluzione (ottima) del problema

- in termini di soluzioni (ottime) di sottoproblemi più "piccoli"
- non davvero diverso da come si ragiona implicitamente quando si usa la tecnica del divide-et-impera

Obiettivo: esprimere la soluzione del problema come combinazione di soluzioni di opportuni sottoproblemi. Se le combinazioni sono "poche" possiamo cercare la combinazione giusta per forza bruta.

Ragionando sulla struttura della soluzione:

sia S^* la soluzione ottima, ovvero l'II di peso massimo di G .

Considera l'ultimo nodo v_n di G .

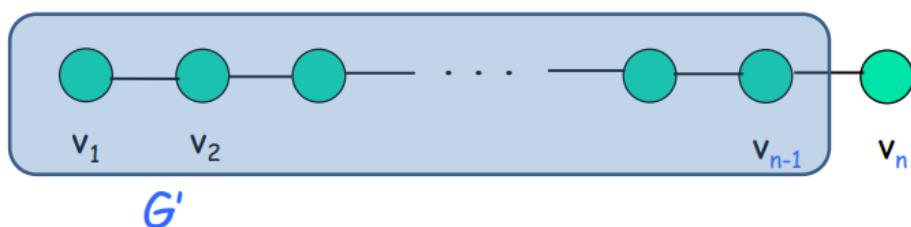
osservazione: $v_n \notin S^*$ o $v_n \in S^*$

caso 1: $v_n \notin S^*$

considera $G' = G - \{v_n\}$.

allora S^* è una soluzione ottima per G' .

se esistesse una soluzione S migliore per G' , S sarebbe migliore anche per G : assurdo!

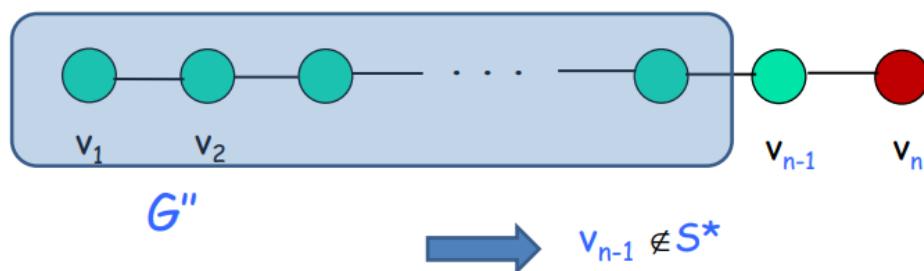


caso 2: $v_n \in S^*$

considera $G'' = G - \{v_{n-1}, v_n\}$.

allora $S^* \setminus \{v_n\}$ è una soluzione ottima per G'' .

se esistesse una soluzione S migliore per G'' , $S \cup \{v_n\}$ sarebbe migliore di S^* per G : assurdo!



verso un algoritmo

proprietà: l'II di peso massimo per G deve essere o:

- (i) l'II di peso massimo per G' ,
- (ii) v_n unito all'II di peso massimo per G'' .

Idea (forse folle): calcolare tutte e due le soluzioni e ritornare la migliore delle due.

quale è il tempo dell'algoritmo se calcolo le due soluzioni ricorsivamente?

$$T(n) = T(n-1) + T(n-2) + O(1)$$

(è quella di Fibonacci2)

$$T(n) = \Theta(\phi^n)$$

esponenziale!!!



...però forse non tutto è perduto

domanda fondamentale: quanti problemi distinti sono risolti dall'algoritmo ricorsivo?

$$\Theta(n)$$

c'è un sottoproblema per ogni prefisso di G



Idea: procediamo iterativamente considerando prefissi di G dai più piccoli verso i più grandi.

esempio

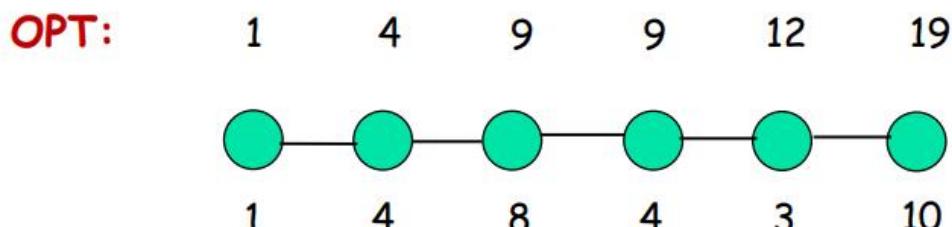
G_j : sottocammino composto dai primi j vertici di G

Sottoproblema j : calcolare il peso del miglior II per G_j

OPT[j]: valore soluzione sottoproblema j , ovvero peso dell'II di peso massimo di G_j

$$\text{OPT}[1]=w_1; \text{OPT}[2]=\max\{w_1, w_2\}$$

$$\text{OPT}[j]=\max\{\text{OPT}[j-1], w_j+\text{OPT}[j-2]\}$$



l'algoritmo

G_j : sottocammino composto dai primi j vertici di G

OPT[]: vettore di n elementi;

dentro OPT[j] voglio mettere il peso dell'II di peso massimo di G_j

1. $\text{OPT}[1]=w_1; \text{OPT}[2]=\max\{w_1, w_2\}$
2. **for** $j=3$ **to** n **do**
3. $\text{OPT}[j]=\max\{\text{OPT}[j-1], w_j+\text{OPT}[j-2]\}$
4. **return** $\text{OPT}[n]$

$$T(n)=\Theta(n)$$

Oss: l'algoritmo calcola il valore della soluzione ottima, ma non la soluzione.

possiamo trovare in tempo lineare
anche l'II di peso massimo?

Ricostruire la soluzione (in tempo lineare)

ricostruire la soluzione

Idea semplice: mentre calcoliamo i valori $\text{OPT}[j]$ possiamo mantenere esplicitamente anche la soluzione.

corretta ma non ideale: spreco di tempo e spazio

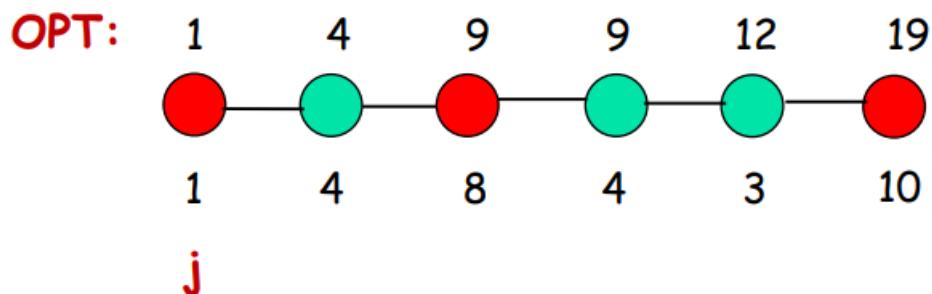
un'idea migliore: ricostruire la soluzione solo alla fine sfruttando il vettore $\text{OPT}[]$.

proprietà chiave:

$$v_j \in II \text{ di peso massimo di } G_j \quad \leftrightarrow \quad w_j + \text{OPT}[j-2] \geq \text{OPT}[j-1]$$

un algoritmo per ricostruire la soluzione

1. $S^* = \emptyset; j=n;$
 2. **while** $j \geq 3$ **do**
 3. **if** $\text{OPT}[j-1] \geq w_j + \text{OPT}[j-2]$
 then $j=j-1;$
 else $S^* = S^* \cup \{v_j\}; j=j-2;$
 4. **if** $j=2$ e $w_2 > w_1$ **then** $S^* = S^* \cup \{v_2\}$
 else $S^* = S^* \cup \{v_1\};$
 5. **return** S^*
- complessità temporale?
- $T(n) = \Theta(n)$



PROGRAMMAZIONE DINAMICA : PRINCIPI GENERALI

1) identificare un numero piccolo di sottoproblemi

es: calcolare l'ITI di peso massimo di Gj, j=1, ... ,n

2) descrivere la soluzione di un generico sottoproblema in funzione delle soluzioni di sottoproblemi più "piccoli"

es: $\text{OPT}[j] = \max \{\text{OPT}[j-1], w_j + \text{OPT}[j-2]\}$

3) le soluzioni dei sottoproblemi sono memorizzate in una tabella

4) avanzare opportunamente sulla tabella, calcolando la soluzione del sottoproblema corrente in funzione delle soluzioni di sottoproblemi già risolti.

Proprietà che devono avere i sottoproblemi:

- 1) essere pochi
- 2) risolti tutti i sottoproblemi si può calcolare velocemente la soluzione al problema originale spesso la soluzione cercata è semplicemente quella del sottoproblema più grande
- 3) ci devono essere sottoproblemi "piccoli" casi base
- 4) ci deve essere un ordine in cui risolvere i sottoproblemi e quindi un modo di avanzare nella tabella e riempirla

SOTTOPROBLEMI

- ✓ La chiave di tutto è la definizione dei "giusti" sottoproblemi.
- ✓ La definizione dei "giusti" sottoproblemi è un punto di arrivo.
- ✓ Solo una volta definiti i sottoproblemi si può verificare che l'algoritmo è corretto.
- ✓ Se la definizione dei sottoproblemi è un punto di arrivo, come ci arrivo?
Ci arrivo ragionando sulla struttura della soluzione (ottima) cercata.
- ✓ La struttura della soluzione può suggerire i sottoproblemi e l'ordine in cui considerarli.

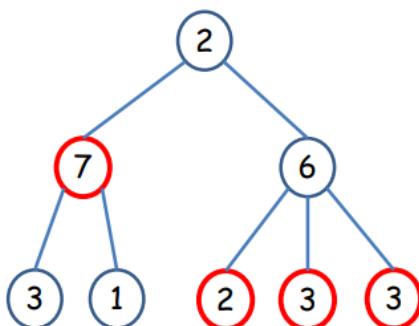
ESERCIZIO II DI PESO MASSIMO SU ALBERI (PROBLEMA FESTA AZIENDALE)

problema: invita i dipendenti alla festa aziendale

massimizza: il divertimento totale degli invitati

vincolo: tutti devono divertirsi

 non invitare un dipendente e il suo boss diretto!



input: un albero con pesi sui nodi

goal: un II di peso totale massimo

OPT= 15

PARADIGMI ALGORITMICI

1. GREED.

Elaborare l'input in un certo ordine, in modo miope prende decisioni irrevocabili.

2. DIVIDE-AND-CONQUER.

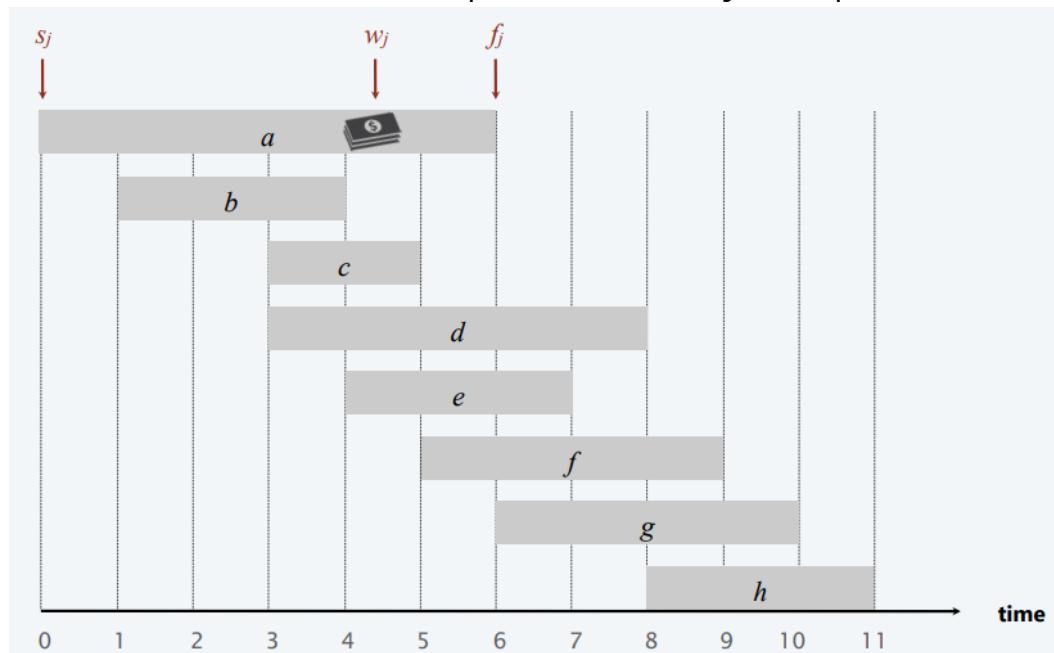
Suddividere un problema in sottoproblemi indipendenti; risolvere ogni sottoproblema; combinare soluzioni ai sottoproblemi per formare una soluzione al problema originale.

3. Programmazione dinamica (nome di fantasia per memorizzazione nella cache dei risultati intermedi in una tabella per un riutilizzo successivo).

Suddividere un problema in una serie di sottoproblemi sovrapposti, combinare le soluzioni a sottoproblemi più piccoli per formare una soluzione a un sottoproblema di grandi dimensioni.

WEIGHTED INTERVAL SCHEDULING (scheduling di intervalli pesati)

- Il lavoro j inizia in s_j , termina in f_j , e ha peso $w_j > 0$.
- Due lavori sono compatibili se non si sovrappongono.
- Obiettivo: trovare un sottoinsieme di peso massimo di job reciprocamente compatibili.



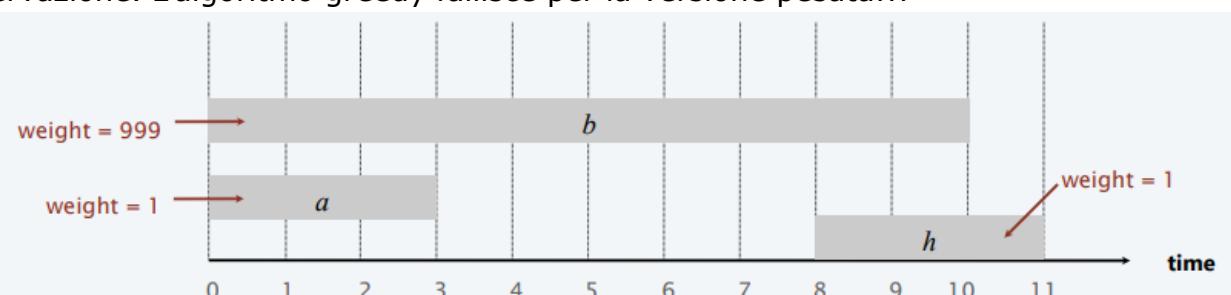
→ EARLIEST-FINISH-TIME FIRST ALGORITHM

Tradotto: "Prima l'orario di arrivo più breve".

- Considera i job in ordine crescente di ora di fine.
- Aggiungi job al sottoinsieme se è compatibile con i lavori scelti in precedenza.

Recall: L'algoritmo greedy è corretto se tutti i pesi sono 1.

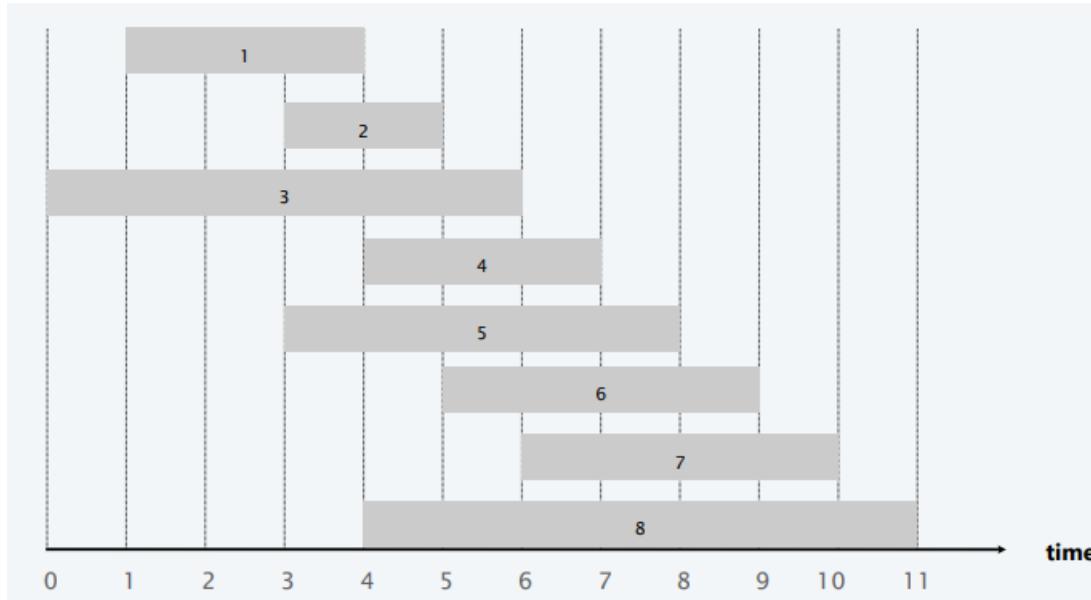
Osservazione. L'algoritmo greedy fallisce per la versione pesata!!!



Convenzione: i job sono in ordine ascendente di finish time: $f_1 \leq f_2 \leq \dots \leq f_n$

Def: $p(j) =$ l'indice $i < j$ più grande tale che il job i è compatibile con j .

Esempio: $p(8)=1$, $p(7)=3$, $p(2)=0$



DYNAMIC PROGRAMMING: BINARY CHOICE

Def: $\text{OPT}(j)$ = peso massimo di qualsiasi sottoinsieme di job reciprocamente compatibili per sottoproblema costituito solo dai job $1, 2, \dots, j$.

Obiettivo: $\text{OPT}(n)$ = peso massimo di qualsiasi sottoinsieme di job reciprocamente compatibili.

Caso 1: $\text{OPT}(j)$ non seleziona il job j .

- Soluzione ottimale al problema tra i job $1, 2, \dots, j-1$.

Caso 2: $\text{OPT}(j)$ seleziona il job j .

- Collezione il profitto w_j .
- Impossibile utilizzare job incompatibili $\{p(j)+1, p(j)+2, \dots, j-1\}$.
- Deve includere la soluzione ottimale nei rimanenti job compatibili $1, 2, \dots, p(j)$.

Equazione di Bellman:

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

PROG.DINAMICA DI TIPO BOTTOM-UP PER IL PROBLEMA WEIGHTED INTERVSCHEDULING

BOTTOM-UP($n, s_1, \dots, s_n, f_1, \dots, f_n, w_1, \dots, w_n$)

Sort jobs by finish time and renumber so that $f_1 \leq f_2 \leq \dots \leq f_n$.

Compute $p[1], p[2], \dots, p[n]$ via binary search.

$M[0] \leftarrow 0$.

FOR $j = 1$ **TO** n

previously computed values

$M[j] \leftarrow \max \{ M[j-1], w_j + M[p[j]] \}$.

RETURN $M[n]$.

Tempo di esecuzione. La versione bottom-up richiede tempo $O(n \log n)$.

- Ordinare per finish time: $O(n \log n)$ tramite Mergesort.
- Calcolare $p[j]$ per ogni j : $O(n \log n)$ tramite ricerca binaria.
- Il ciclo FOR richiede tempo $O(n)$.

RITORNO A UTILIZZO RICORSIONE PER IL PROBLEMA WEIGHTED INTERVSCHEDULING

BRUTE-FORCE ($n, s_1, \dots, s_n, f_1, \dots, f_n, w_1, \dots, w_n$)

Sort jobs by finish time and renumber so that $f_1 \leq f_2 \leq \dots \leq f_n$.

Compute $p[1], p[2], \dots, p[n]$ via binary search.

RETURN COMPUTE-OPT(n).

COMPUTE-OPT(j)

IF ($j = 0$)

RETURN 0.

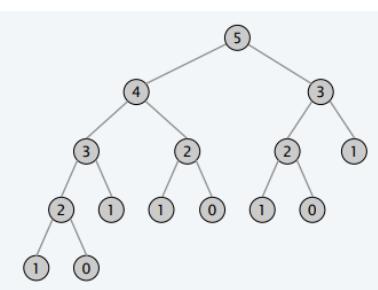
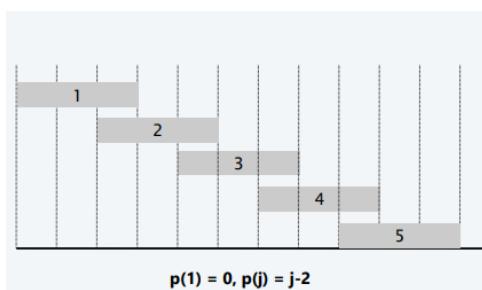
ELSE

RETURN $\max \{ \text{COMPUTE-OPT}(j-1), w_j + \text{COMPUTE-OPT}(p[j]) \}$.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n-1) + \Theta(1) & \text{if } n > 1 \end{cases} \quad T(n)=\Theta(2^n)$$

Osservazione: l'algoritmo ricorsivo è straordinariamente lento a causa di sottoproblemi sovrapposti = algoritmo in tempo esponenziale.

Ex. Il numero di chiamate ricorsive per la famiglia di istanze "a strati" cresce come con la sequenza di Fibonacci.



MEMOIZATION (MEMORIZZAZIONE)

La programmazione dinamica top-down (memoizzazione)

- Memorizzare nella cache il risultato del sottoproblema j in $M[j]$.
- Utilizzare $M[j]$ per evitare di risolvere il sottoproblema j più di una volta.

TOP-DOWN($n, s_1, \dots, s_n, f_1, \dots, f_n, w_1, \dots, w_n$)

Sort jobs by finish time and renumber so that $f_1 \leq f_2 \leq \dots \leq f_n$.

Compute $p[1], p[2], \dots, p[n]$ via binary search.

$M[0] \leftarrow 0$. ————— global array

RETURN M-COMPUTE-OPT(n).

M-COMPUTE-OPT(j)

IF ($M[j]$ is uninitialized)

$M[j] \leftarrow \max \{ M\text{-COMPUTE-OPT}(j-1), w_j + M\text{-COMPUTE-OPT}(p[j]) \}$.

RETURN $M[j]$.

TEMPO DI ESECUZIONE DEL PROBLEMA WEIGHTED INTERVAL SCHEDULING

La versione memorizzata dell'algoritmo richiede tempo $O(n \log n)$.

Dimostrazione:

- Ordinare per finish time: $O(n \log n)$ tramite Mergesort.
- Calcolare $p[j]$ per ogni j : $O(n \log n)$ tramite ricerca binaria.
- M-COMPUTE-OPT(j): ogni invocazione richiede tempo $O(1)$ e ognuna
 - (1) restituisce un valore inizializzato $M[j]$
 - (2) inizializza $M[j]$ ed effettua due chiamate ricorsive
- Misura di avanzamento $\Phi = \#$ voci inizializzate tra $M[1 \dots n]$.
 - inizialmente $\Phi=0$; in tutto $\Phi \leq n$.
 - (2) aumenta Φ di 1 $\rightarrow \leq 2n$ chiamate ricorsive.
- Il tempo di esecuzione complessivo di M-COMPUTE-OPT(n) è $O(n)$.

TROVARE UNA SOLUZIONE DI WEIGHTED INTERVAL SCHEDULING

Domanda: L'algoritmo DP calcola il valore ottimale. Come trovare la soluzione ottimale?

Risposta: Effettuando un secondo passaggio chiamando FIND-SOLUTION(n).

FIND-SOLUTION(j)

IF ($j = 0$)

RETURN \emptyset .

ELSE IF ($w_j + M[p[j]] > M[j-1]$)

RETURN $\{j\} \cup$ FIND-SOLUTION($p[j]$).

ELSE

RETURN FIND-SOLUTION($j-1$).

Analisi: # di chiamate ricorsive $\leq n \rightarrow O(n)$.

MEMOIZATION (TOP-DOWN) vs TABLE-BASED (BOTTOM-UP)

Vantaggi memoization(top-down):

1. Più intuitivo
2. Più facile indicizzare i sottoproblemi da altri oggetti (ad esempio insiemi)
3. Calcola solo i sottoproblemi necessari

Svantaggi memoization(top-down):

1. Overhead chiamate di funzione
2. Complessità temporale più difficile da analizzare

Vantaggi table-based(bottom up):

1. Nessuna ricorsione, più efficiente in cache
2. Complessità temporale facile da analizzare
3. Codice breve e pulito

Svantaggi table-based(bottom up):

1. Più difficile da afferrare (meno intuitiva)
2. Necessità di indicizzare i sottoproblemi con numeri interi
3. Calcolo sempre tutti i sottoproblemi

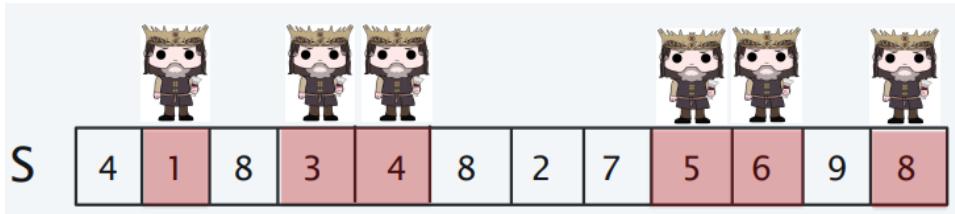
ESERCIZIO VISTO A LEZIONE (SOTTOSEQUENZA CRESCENTE PIU' LUNGA)

Il seguente problema è conosciuto come il problema di Longest Increasing Subsequence.

Re Robert vuole bere il più possibile.

- Robert cammina per le strade di approdo del Re e incontri nelle taverne t_1, t_2, \dots, t_n , in ordine
- Quando Robert incontra una taverna t_i , può fermarsi a bere qualcosa o continuare a camminare
- Il vino servito nella taverna t_i ha forza s_i (più alto questo valore più ciò che ha bevuto è forte)
- La forza di ciò che Robert beve deve essere in ordine crescente nel tempo
- Obiettivo: calcolare il numero massimo di soste per bere di Robert

Esempio:



Soluzione ottima: 6

RISOLUZIONE:

- definizione sottoproblema = $\text{OPT}[i]$: lunghezza della LIS di $S[1], \dots, S[i]$ che termina con $S[i]$
- caso base = $\text{OPT}[1]=1$
- soluzione = il massimo $i=1,2,\dots,n$ $\text{OPT}[i]$
- ordine sottoproblemi = $\text{OPT}[1], \text{OPT}[2], \dots, \text{OPT}[n]$
- formula ricorsiva:

$$\text{OPT}[i] = 1 + \max \left\{ 0, \max_{\substack{j=1,2,\dots,i-1 \\ \text{tc } S[j] < S[i]}} \text{OPT}[j] \right\}$$

- quindi:

LIS($S[1:n]$)

$\text{OPT}[1]=1$

FOR $i = 2$ TO n

$$\text{OPT}[i] = 1 + \max \left\{ 0, \max_{\substack{j=1,2,\dots,i-1 \\ \text{tc } S[j] < S[i]}} \text{OPT}[j] \right\}$$

RETURN $\max_i \text{OPT}[i]$.

- Tempo esecuzione: ogni $\text{OPT}[i]$ è computato in tempo $O(i)=O(n)$, perciò tempo esecuzione pari a $O(n^2)$

PROBLEMA DELLA COLORAZIONE DELLE CASE

Obiettivo. Dipingere una fila di n case in rosso, verde o blu in questo modo:

- Non esistono due case adiacenti dello stesso colore.
- Ridurre al minimo il costo totale, dove cost(i, color) è il costo per dipingere il colore i.



	A	B	C	D	E	F
Red	7	6	7	8	9	20
Green	3	8	9	22	12	8
Blue	16	10	4	2	5	7

Sottoproblemi:

- $R[i]$ = costo minimo per dipingere le case 1, ..., i con i rossi.
- $G[i]$ = costo minimo per dipingere le case 1, ..., i con i verdi.
- $B[i]$ = costo minimo per dipingere le case 1, ..., i con i di blu.
- Costo ottimale = $\min \{ R[n], G[n], B[n] \}$.

Equazione dynamic programming:

- $R[i] = \text{costo}(i, \text{rosso}) + \min \{ B[i-1], G[i-1] \}$
- $G[i] = \text{costo}(i, \text{verde}) + \min \{ R[i-1], B[i-1] \}$
- $B[i] = \text{costo}(i, \text{blu}) + \min \{ R[i-1], G[i-1] \}$

Tempo d'esecuzione: $O(n)$

LEZ8 – 28/03/2024 (continuo pdf precedente, segmented least squares)

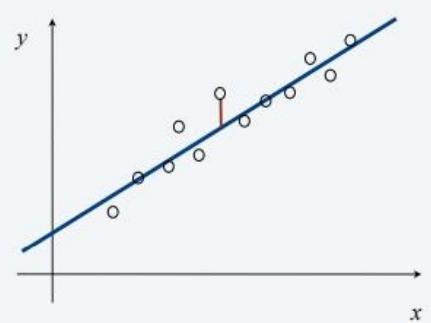
LEAST SQUARES (QUADRATI MINIMI)

Il problema least square è un problema fondamentale in statistica, secondo cui

- Dati n punti nel piano: $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$.
- Trovare una linea $y = ax + b$ che minimizzi la somma dell'errore quadrato

$$SSE = \sum_{i=1}^n (y_i - ax_i - b)^2$$

SSE = scarto quadratico medio



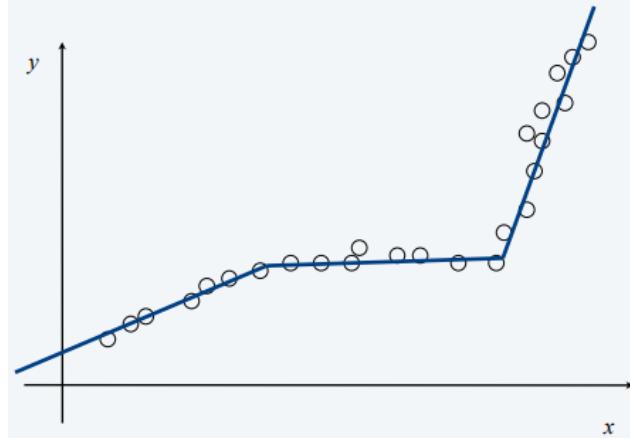
Soluzione: calcolo → errore minimo si ottiene quando

$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2}, \quad b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

SEGMENTED LEAST SQUARES

- I punti si trovano all'incirca su una sequenza di più segmenti di linea.
- Dati n punti nel piano: $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ con $x_1 < x_2 < \dots < x_n$, trovare una sequenza di righe che minimizzi $f(x)$.

Domanda: Qual' è una scelta ragionevole per $f(x)$ per bilanciare accuratezza e parsimonia?
(parsimonia intesa come numero di linee)



Goal: minimizzare $f(x) = E + cL$ per qualche costante $c > 0$, con E =somma delle somme degli scarti quadratici in ogni segmento, ed L =numero di linee.

DYNAMIC PROGRAMMING: SCELTA MULTIPLA PER LA SUA RISOLUZIONE

Notazioni:

- $OPT(j)$ = costo minimo per i punti p_1, p_2, \dots, p_j
- e_{ij} = SSE per i punti p_i, p_{i+1}, \dots, p_j per qualche $i \leq j$

Calcolare $OPT(j)$:

- L'ultimo segmento utilizza i punti p_i, p_{i+1}, \dots, p_j per alcuni $i \leq j$.
- Costo = $e_{ij} + c + OPT(i-1)$.

Equazione di Bellman:

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \min_{1 \leq i \leq j} \{ e_{ij} + c + OPT(i-1) \} & \text{if } j > 0 \end{cases}$$

Algoritmo:

SEGMENTED-LEAST-SQUARES(n, p_1, \dots, p_n, c)

FOR $j = 1$ **TO** n

FOR $i = 1$ **TO** j

 Compute the SSE e_{ij} for the points p_i, p_{i+1}, \dots, p_j .

$M[0] \leftarrow 0.$

FOR $j = 1$ **TO** n

$M[j] \leftarrow \min_{1 \leq i \leq j} \{ e_{ij} + c + M[i-1] \}.$

previously computed value

RETURN $M[n]$.

ANALISI PROBLEMA SEGMENTED LEAST SQUARES

Teorema(Bellman): L'algoritmo DP risolve il problema in tempo $O(n^3)$ e in spazio $O(n^2)$.

Dim:

- Collo di bottiglia = calcolo SSE e_{ij} per ogni i e j .

$$a_{ij} = \frac{n \sum_k x_k y_k - (\sum_k x_k)(\sum_k y_k)}{n \sum_k x_k^2 - (\sum_k x_k)^2}, \quad b_{ij} = \frac{\sum_k y_k - a_{ij} \sum_k x_k}{n}$$

- $O(n)$ per calcolare e_{ij}

Oss: Può essere migliorato in tempo $O(n^2)$.

- Per ogni i : precalcolo delle somme cumulative
- Utilizzando somme cumulative, è possibile calcolare e_{ij} in tempo $O(1)$.

KNAPSACK PROBLEM

Obiettivo. Prepara lo zaino in modo da massimizzare il valore totale degli oggetti presi.

- Ci sono n oggetti: l'oggetto i fornisce valore $v_i > 0$ e pesa $w_i > 0$.
- Valore di un sottoinsieme di elementi = somma dei valori dei singoli elementi.
- Lo zaino ha un limite di peso di W .

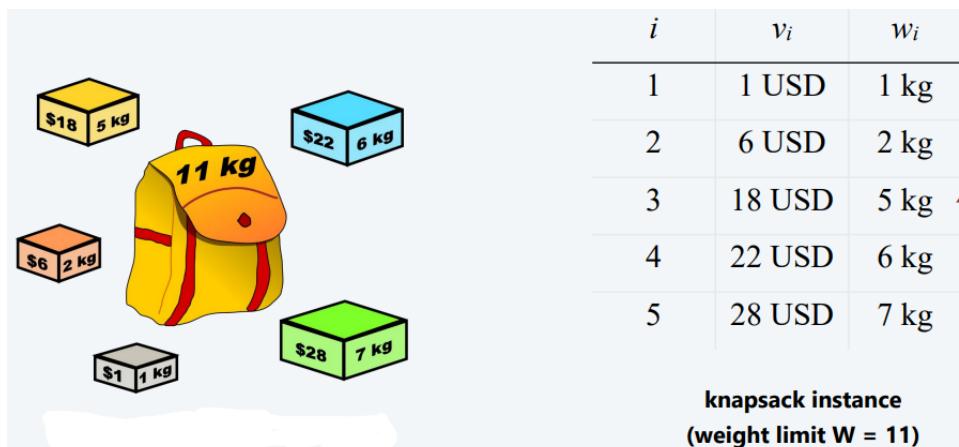
Esempio1: Il

sottoinsieme $\{1, 2, 5\}$
ha valore \$ 35 (e peso
10).

Esempio2: Il

sottoinsieme $\{3, 4\}$ ha
valore \$40 (e peso 11).

Assunzione: Tutti i
valori e i pesi sono
valori interi.



PROG. DINAMICA : TWO VARIABLES (RISOLUZIONE KNAPSACK PROBLEM)

Def: $OPT(i, w) =$ valore ottimo knapsack problem con oggetti $1, \dots, i$ soggetti a peso limite w .

Goal: $OPT(n, W)$.

Caso1: $OPT(i, w)$ non seleziona l'elemento i (probabilmente perché $w_i > w$)

- $OPT(i, w)$ seleziona il migliore tra $\{1, 2, \dots, i-1\}$ soggetto a limite di peso w

Caso2: $OPT(i, w)$ seleziona l'elemento i

- Colleziona il valore v_i
- Nuovo peso limite = $w - w_i$
- $OPT(i, w)$ seleziona il migliore tra $\{1, 2, \dots, i-1\}$ soggetto a limite di peso $w - w_i$

Equazione di Bellman:

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max \{ OPT(i-1, w), v_i + OPT(i-1, w - w_i) \} & \text{otherwise} \end{cases}$$

Algoritmo risoluzione knapsack problem tramite prog.dinamica di tipo bottom-up :

KNAPSACK($n, W, w_1, \dots, w_n, v_1, \dots, v_n$)

FOR $w = 0$ TO W

$M[0, w] \leftarrow 0.$

FOR $i = 1$ TO n

FOR $w = 0$ TO W

IF ($w_i > w$) $M[i, w] \leftarrow M[i-1, w].$

ELSE $M[i, w] \leftarrow \max \{ M[i-1, w], v_i + M[i-1, w - w_i] \}.$

previously computed values

RETURN $M[n, W].$

Demo:

i	v_i	w_i	$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max \{ OPT(i-1, w), v_i + OPT(i-1, w - w_i) \} & \text{otherwise} \end{cases}$
1	1 USD	1 kg	
2	6 USD	2 kg	
3	18 USD	5 kg	
4	22 USD	6 kg	
5	28 USD	7 kg	

		weight limit w											
		0	1	2	3	4	5	6	7	8	9	10	11
subset of items $1, \dots, i$	{ }	0	0	0	0	0	0	0	0	0	0	0	0
	{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
	{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
	{ 1, 2, 3 }	0	1	6	7	7	18	24	25	25	25	25	25
	{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
	{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	35	40

$OPT(i, w)$ = optimal value of knapsack problem with items $1, \dots, i$, subject to weight limit w

46

Tempo di esecuzione (teorema):

L'algoritmo DP risolve il knapsack problem con n oggetti e peso massimo W in tempo $\Theta(nW)$ e spazio $\Theta(nW)$ (in quanto i pesi sono interi tra 1 e W).

Dim:

- Richiede $O(1)$ tempo per table entry.
- Ci sono $\Theta(nW)$ entries nella tabella.
- Dopo aver calcolato i valori ottimali, è possibile risalire per trovare la soluzione: $OPT(i, w)$ prende l'elemento i se e solo se $M[i, w] > M[i-1, w]$.

Osservazioni:

- L'algoritmo dipende in modo critico dal presupposto che i pesi siano interi.
- Non è stato utilizzato il presupposto che i valori siano interi.

Il tempo di esecuzione dell'algoritmo DP per il knapsack problem è polinomiale?

- No, perché $\Theta(nW)$ non è una funzione polinomiale della dimensione dell'input.
- È pseudo-polinomiale.

Algoritmo pseudo-polinomiale: un algoritmo il cui tempo di esecuzione è polinomiale nei valori dell'input (ad esempio il più grande intero presente in ingresso).

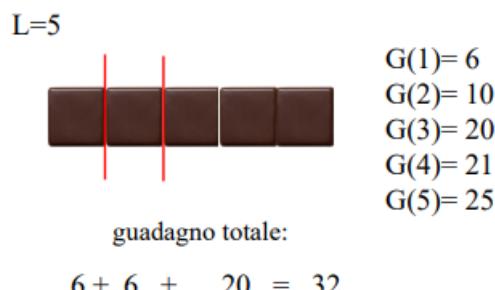
Efficiente quando i numeri coinvolti nell'input sono ragionevolmente piccoli (ad esempio, nel problema dello zaino quando w_i sono piccoli).

Polinomio non necessario nella dimensione dell'input (numero di bit richiesti per rappresentare l'input).

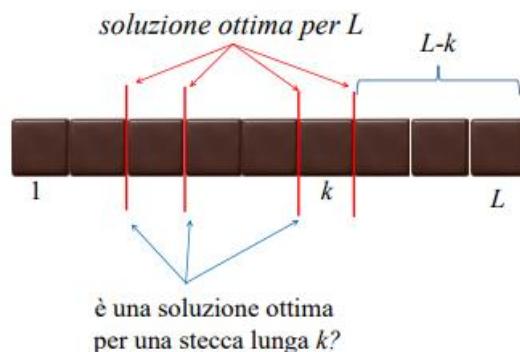
ESERCIZIO1

Esercizio Il signor Valter Bianchi, dopo aver fatto un bel gruzzoletto vendendo cristalli non proprio legali, ha deciso di diversificare la sua attività e si è messo a vendere della roba - anch'essa non proprio legale - che per comodità chiameremo stecca di cioccolata. La stecca di cioccolata può essere venduta tutta intera o può essere spezzata in segmenti più piccoli da vendere separatamente. La lunghezza della stecca di cioccolata è di L centimetri, con L intero. Si assume che nello spezzare la stecca la lunghezza dei pezzi ottenuti (in centimetri) debba essere ancora un numero intero. Per esempio un pezzo lungo 3 centimetri può essere venduto così, spezzato in tre pezzi da 1 centimetro o in due pezzi: uno da 2 centimetri e l'altro da 1 centimetro (mentre non si possono fare due pezzi da mezzo centimetro e due centimetri e mezzo). Il guadagno che il signor Valter Bianchi riesce a fare se vende un pezzo lungo t centimetri è $G(t)$, $t = 1, 2, \dots, L$. Progettare un algoritmo di programmazione dinamica che aiuti il signor Valter Bianchi a guadagnare il più possibile. La complessità temporale dell'algoritmo deve essere polinomiale in L .

Un esempio di istanza e soluzione è:



Un esempio di istanza con $L = 5$. La soluzione ottima, ovvero il modo di spezzare la cioccolata per ottenere guadagno massimo `e mostrato in rosso.



Ragionare sulla struttura della soluzione cercata: La chiave per progettare un algoritmo di programmazione dinamica è essenzialmente individuare il giusto insieme di sottoproblemi. Fare questo è un punto di arrivo che passa per la comprensione della struttura combinatorica del problema in esame. In particolar modo, quello che bisogna davvero capire è la struttura della soluzione ottima cercata, in modo da poterla esprimere in funzione di soluzioni ottime per sottoproblemi opportuni (e più "piccoli"). Proviamo a farlo in questo caso. Immaginiamo di avere la soluzione ottima, il modo migliore di spezzare la stecca di cioccolata di lunghezza L . Sia k l'ultima spezzata (se le spezzate le pensiamo ordinate da sinistra a destra) della soluzione ottima. Dato k , è chiaro che il guadagno ottenuto dalla soluzione ottima è uguale al guadagno ottenuto dall'ultimo pezzo di cioccolata (che ha lunghezza $L-k$) più il guadagno relativo ai pezzi in cui è poi ridiviso il pezzo di lunghezza k . Ora, la domanda cruciale è: il modo in cui questo pezzo di lunghezza

k è ridiviso è la soluzione ottima per una stecca di cioccolata lunga k ? Se ci si pensa un po' è facile convincersi che la risposta a questa domanda è sì. Infatti, se le spezzate della soluzione ottima per L a sinistra di k non fossero quelle ottime per una stecca lunga k , allora esisterebbe un modo migliore di spezzare il pezzo da k e guadagnare di più dal quel pezzo, il che implicherebbe che potrei guadagnare di più anche dal pezzo lungo L semplicemente usando questa soluzione alternativa per il pezzo lungo k e vendendo il pezzo lungo $L - k$ tutto intero. Ma questo non è possibile perché la soluzione che stiamo considerando è una soluzione ottima per un pezzo lungo L . Insomma: la solita argomentazione di tipo cut&paste, taglia e cuci. Abbiamo quindi espresso la soluzione ottima in questo modo:

$$\text{guadagno ottimo per } L = G(L - k) + \text{guadagno ottimo per } k.$$

Abbiamo fatto un passo avanti nella comprensione della struttura della soluzione ottima. Restano però due punti da chiarire. Il primo è: ma come possiamo conoscere k ? La risposta a questa domanda è semplice: non possiamo. Però possiamo "indovinarlo". L'ultima spezzata dell'ottimo infatti può trovarsi in $O(L)$ posizioni; ovvero k può essere uguale a $L-1$, o $L - 2, \dots$, o 1 , o 0 (nel senso che la soluzione ottima consiste nel vendere tutta intera la stecca da L). L'idea è quindi quella di provare tutte queste possibilità e prendere la migliore che, quindi, sarà la soluzione ottima. Questo è l'aspetto enumerativo della programmazione dinamica. Fare questo vuol dire di fatto considerare la seguente relazione:

$$\text{guadagno ottimo per } L = \max_{k=0,1,\dots,L-1} (G(L - k) + \text{guadagno ottimo per } k).$$

Il secondo punto da chiarire è: come definire i sottoproblemi? Una volta intuita la struttura della soluzione, ovvero la relazione che lega la soluzione ottima all'ottimo di altri problemi, tutto diventa più semplice. Se si analizza la relazione di sopra, è chiaro che la soluzione ottima per L è definita in funzione della soluzione ottima per k , con $k < L$. Inoltre, se si reitera il ragionamento, la soluzione ottima per k sarà definita in funzione della soluzione ottima per k' , con $k' < k$. Abbiamo quindi un sottoproblema distinto per ogni valore di $k=0,1, \dots, L$.

Una soluzione. Siamo ora pronti per descrivere una algoritmo di programmazione dinamica per il problema. Partiamo, chiaramente, dalla definizione dei sottoproblemi. Per ogni $j = 0, 1, \dots, L$, definiamo $\text{Opt}(j)$ =il guadagno massimo ottenibile da una stecca di cioccolata lunga j .

A questo punto, dobbiamo capire se i sottoproblemi che abbiamo definito vanno bene, ovvero se tutto si incarta in modo da ottenere un algoritmo di programmazione dinamica. Essenzialmente, è necessario rispondere alle seguenti domande:

1. I sottoproblemi sono pochi? Visto che dovremo risolverli tutti, e per ogni sottoproblema dovremo spendere almeno tempo contante, il numero di sottoproblemi fornisce una delimitazione inferiore al tempo speso dall'algoritmo. Se questo numero è troppo grande (per esempio il numero di problemi è confrontabile con la complessità dell'algoritmo banale che risolve il problema per enumerazione) i sottoproblemi individuati non vanno bene. In questo caso, comunque, il numero di sottoproblemi è $O(L)$, quindi lineari nella dimensione dell'istanza.
2. Una volta risolti tutti i sottoproblemi ho risolto il mio problema? La soluzione del problema deve essere infatti esplicitamente o implicitamente derivabile dalle soluzioni dei sottoproblemi. In questo caso, chiaramente, questo è vero: la soluzione cercata è $\text{Opt}(L)$.
3. E' possibile definire la soluzione del sottoproblema generico in funzione di sottoproblemi più piccoli? Questo è di fatto il banco di prova, il momento in cui si capisce se si sono individuati i giusti sottoproblemi. Se si è capita la struttura della soluzione ottima,

comunque, la maggior parte del lavoro per rispondere a questa domanda è stato già fatto. Infatti, tutto quello che abbiamo detto per la soluzione ottima del nostro problema vale per ogni sottoproblema j -esimo. Quindi possiamo dire che:

$$\text{Opt}(j) = \max_{k=0,1,\dots,j-1} (G(j-k) + \text{Opt}(k)).$$

4. In che ordine è possibile guardare/risolvere i sottoproblemi? Quali sono i casi base? Generalmente questo punto è facile da risolvere. La domanda soggiacente è: come definire i casi base e in che ordine risolvere i sottoproblemi in modo che la formula generica del punto precedente risulti corretta e applicabile? Nel nostro caso è semplice. L'unico caso base è $\text{Opt}(0) = 0$.

E i sottoproblemi vanno quindi considerati nell'ordine $j = 0, 1, \dots, L$.

A questo punto, l'algoritmo di programmazione dinamica è semplicissimo. La complessità dell'algoritmo è $O(L^2)$: ogni iterazione del ciclo for corrisponde ad uno degli $O(L)$ sottoproblemi, ed ogni sottoproblema può essere risolto in tempo $O(L)$.

Algorithm 1: Cioccolata(G, L)

```

Opt(0)=0;
for j = 1 to L do
    Opt(j) = maxk=0,1,...,j-1(G(j-k) + Opt(k))
return Opt(L);

```

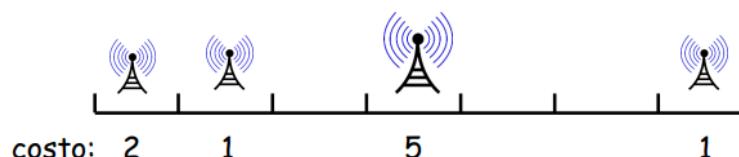
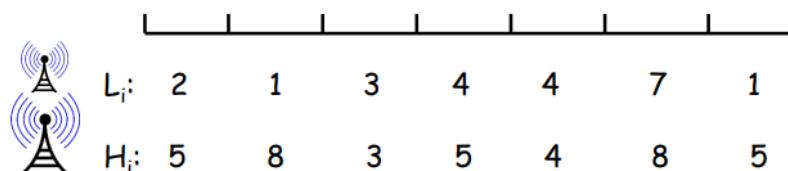
ESERCIZIO2

Si vuole dotare una pista ciclabile di un buon servizio wifi. Per fare questo si devono installare dei ripetitori wireless. La pista ciclabile è lunga n tratte. Il costo di installazione di un ripetitore non è uniforme e dipende dalla tratta in cui si installa il ripetitore e dal tipo del ripetitore. In particolare, ci sono due tipi di ripetitori, uno di tipo high (H) e uno di tipo low (L). Se nella tratta i si installa:

- un ripetitore di tipo L, il ripetitore è in grado fornire il servizio alle tratte i e $i+1$. Il costo di installazione è L_i
- un ripetitore di tipo H, il ripetitore è in grado fornire il servizio alle tratte $i, i+1$ e $i+2$. Il costo di installazione è $H_i \geq L_i$

Progettare un algoritmo di programmazione dinamica che calcoli una soluzione di costo totale minimo che fornisce il servizio a tutte le tratte.

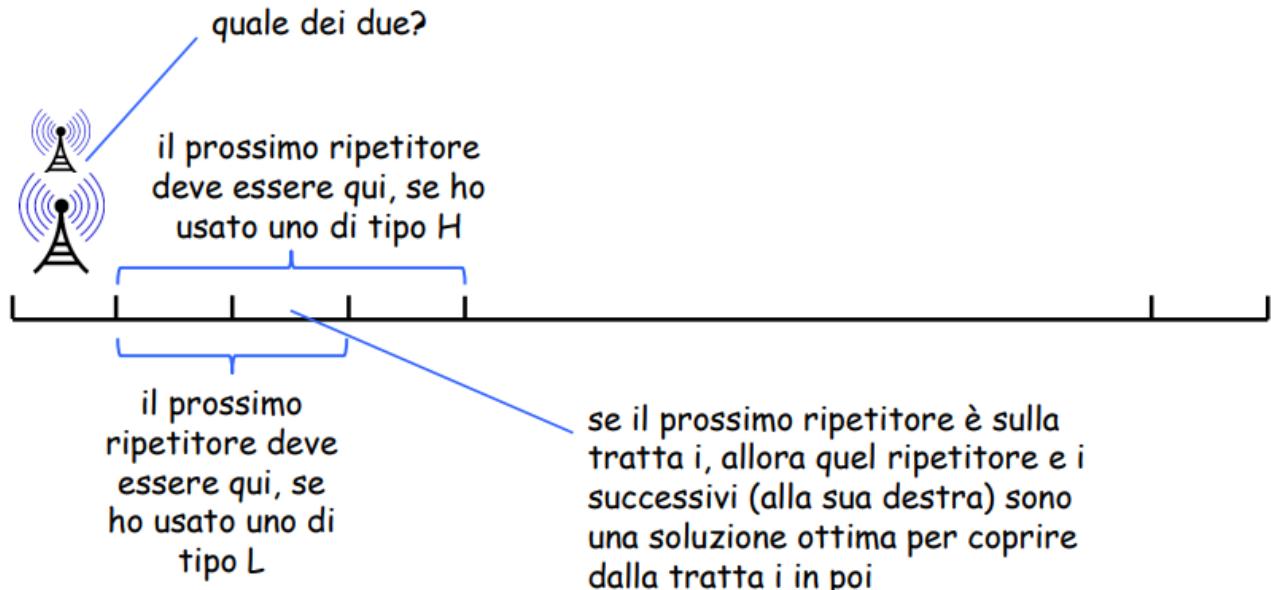
Esempio:



una soluzione di costo totale 9

Idea:

- "indovinare" il tipo di ripetitore installato nella soluzione ottima nella tratta 1
- questo copre alcune tratte che ora non è più necessario coprire
- coprire le tratte "restanti" all'ottimo
- attenzione: non è detto che nell'ottimo non ci sia un ripetitore anche nella tratta 2!



SOLUZIONE

Sottoproblemi: $\text{Opt}[i]$ costo minimo per coprire le tratte $i, i+1, \dots, n$

Soluzione cercata: $\text{Opt}[1]$

Equazione di Bellman: $\text{Opt}[i] = \min\{\text{L}_i + \text{Opt}[i+1], \text{L}_i + \text{Opt}[i+2], \text{H}_i + \text{Opt}[i+3]\}$

Casi base: $\text{Opt}[i] = 0$ se $i > n$

Ordine dei sottoproblemi: $\text{Opt}[n], \text{Opt}[n-1], \dots, \text{Opt}[2], \text{Opt}[1]$

Pseudocodice:

CalcoloCostoInstallazione:

1. $\text{Opt}[n+1] = \text{Opt}[n+2] = 0$
2. for $i=n$ down to 1 do
 1. $\text{Opt}[i] = \min \{\text{L}_i + \text{Opt}[i+1], \text{L}_i + \text{Opt}[i+2], \text{H}_i + \text{Opt}[i+3]\}$
 3. return $\text{Opt}[1]$

Tempo esecuzione: $O(n)$

- come al solito l'algoritmo calcola solo il valore della soluzione ottima
- dato il vettore $\text{Opt}[]$ sapete ricostruire in tempo $O(n)$ anche la soluzione? (si, lasciato per esercizio)

SEQUENCE ALIGNMENT PROBLEM

Quanto simili sono due stringhe?

Esempio: "occurranc" e "occurrence"

 6 mismatches, 1 gap	 0 mismatches, 3 gaps
-------------------------	--------------------------

EDIT DISTANCE

Gap penalty δ ; mismatch penalty α_{pq}

Costo = somma delle penalty derivanti da gap e mismatch

 $\text{cost} = \delta + \alpha_{CG} + \alpha_{TA}$	 assuming $\alpha_{AA} = \alpha_{CC} = \alpha_{GG} = \alpha_{TT} = 0$
--	--

Esempi di utilizzo: bioinformatica, spell correction digitale, ...

Un altro esempio: quanta distanza c'è tra le due stringhe "P A L E T T E" e "P A L A T E"?

Assumiamo gap penalty=2 e mismatch penalty=1

 1 gap, 1 mismatch

FORMALIZZAZIONE SEQUENCE ALIGNMENT

Goal: Date due stringhe $x_1, x_2 \dots x_m$ e $y_1, y_2 \dots y_n$ trovare costo minimo di allineamento.

Def: un allineamento (alignment) M è un insieme di coppie ordinate x_i-y_i tale che ogni carattere appare in al massimo una coppia e non si incrocia.

Def: il costo di allineamento M è:

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

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

an alignment of CTACCG and TACATG

$$M = \{ x_2-y_1, x_3-y_2, x_4-y_3, x_5-y_4, x_6-y_6 \}$$

SEQUENCE ALIGNMENT: STRUTTURA DEL PROBLEMA

Def: $\text{OPT}(i, j)$ =costo minimo per allineare i prefissi delle stringhe $x_1 x_2 \dots x_{i-1}$ e $y_1 y_2 \dots y_j$.

Goal: $\text{OPT}(m, n)$.

Caso1: $\text{OPT}(i, j)$ matches x_i-y_j

Pago il mismatch per x_i-y_j + costo minimo per allineare $x_1 x_2 \dots x_{i-1}$ e $y_1 y_2 \dots y_{j-1}$

Caso 2a: $\text{OPT}(i, j)$ lascia x_i unmatched

Pago il gap per x_i + il costo minimo per allineare $x_1 x_2 \dots x_{i-1}$ e $y_1 y_2 \dots y_j$.

Caso 2b: $\text{OPT}(i, j)$ lascia y_j unmatched

Pago il gap per y_j + il costo minimo per allineare $x_1 x_2 \dots x_i$ e $y_1 y_2 \dots y_{j-1}$.

Equazione di Bellman:

equation.

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

Algoritmo:

SEQUENCE-ALIGNMENT($m, n, x_1, \dots, x_m, y_1, \dots, y_n, \delta, \alpha$)

FOR $i = 0$ TO m

$M[i, 0] \leftarrow i \delta.$

FOR $j = 0$ TO n

$M[0, j] \leftarrow j \delta.$

FOR $i = 1$ TO m

FOR $j = 1$ TO n

$M[i, j] \leftarrow \min \{ \alpha_{x_i y_j} + M[i-1, j-1],$

$\delta + M[i-1, j],$

$\delta + M[i, j-1] \}.$

already
computed

RETURN $M[m, n].$

Traceback:

		P	A	L	A	T	E			
		0	2	4	6	8	10	12		
		P	2	0	2	4	6	8	10	
		A	4	2	0	2	4	6	8	
		L	6	4	2	0	2	4	6	
		E	8	6	4	2	1	3	4	
		T	10	8	6	4	3	1	3	
		T	12	10	8	6	5	3	2	
		E	14	12	10	8	7	5	3	

P	A	L	E	T	T	E
---	---	---	---	---	---	---

P	A	L	-	A	T	E
---	---	---	---	---	---	---

1 gap, 1 mismatch
(gap penalty = 2, mismatch penalty = 1)

Analisi: l'algoritmo DP calcola l'edit distance (e un alignment ottimo) di due stringhe di lunghezza m e n in tempo $\Theta(mn)$ e in spazio $\Theta(mn)$.

Possiamo fare meglio?? Sì, miglioriamo lo spazio con l'algoritmo di Hirschberg.

SEQUENCE ALIGNMENT IN SPAZIO LINEARE: TEOREMA DI HIRSCHBERG

Esiste un algoritmo per trovare un alignment ottimo in tempo $O(mn)$ e in spazio $O(m+n)$.
 (intelligente combinazione di divide-and-conquer e dynamic programming)

Primo trick per migliorare lo spazio: per calcolare la prossima colonna/riga della matrice abbiamo bisogno soltanto della colonna/riga precedente. Possiamo allora mantenere soltanto 2 colonne/righe alla volta. Allora lo spazio necessario è $O(m+n)$
 Nota: questo permette di calcolare la edit distance ma non l'alignment.

Grafo dell'edit distance:

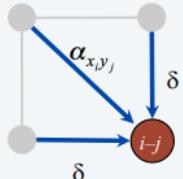
- sia $f(i,j)$ la lunghezza del percorso più breve da $(0,0)$ a (i,j)
- Lemma: $f(i,j) = OPT(i,j)$ per tutti gli i ed j .

Dim del lemma:

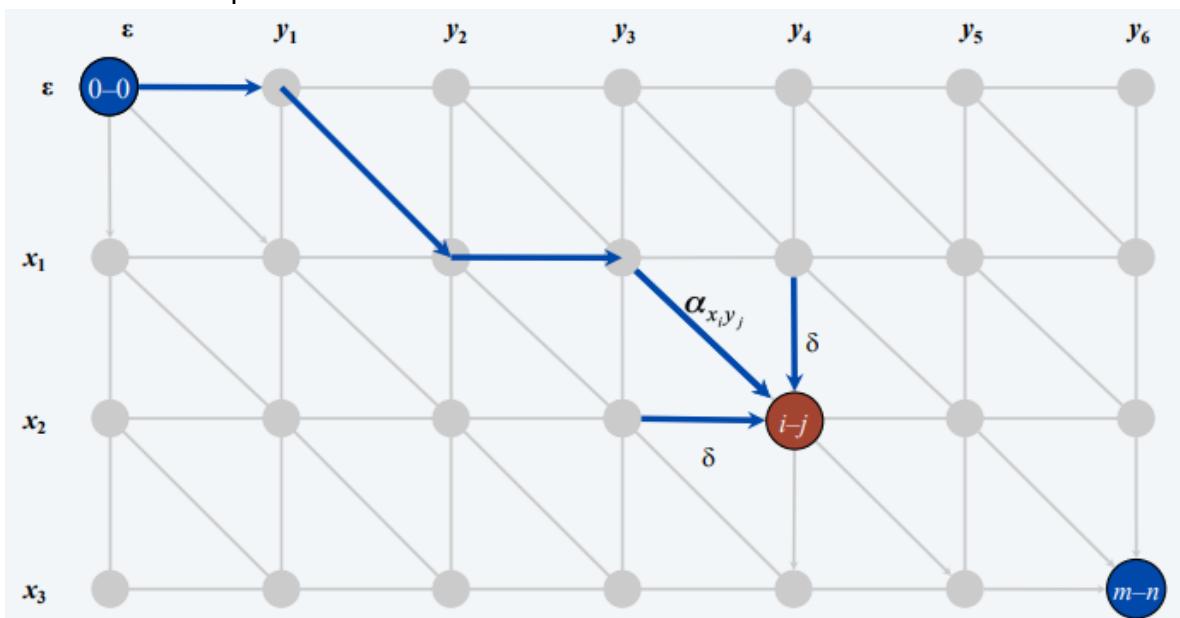
caso base $f(0,0) = OPT(0,0) = 0$

ipotesi induttiva assume vero per tutti gli (i', j') con $i'+j' < i+j$

l'ultimo arco sul percorso più breve fino a (i,j) è da $(i-1,j-1), (i-1,j)$ oppure $(i,j-1)$
 dunque

$$\begin{aligned}
 f(i,j) &= \min\{\alpha_{x_i y_j} + f(i-1, j-1), \delta + f(i-1, j), \delta + f(i, j-1)\} \\
 &= \min\{\alpha_{x_i y_j} + OPT(i-1, j-1), \delta + OPT(i-1, j), \delta + OPT(i, j-1)\} \\
 &= OPT(i,j) .
 \end{aligned}$$


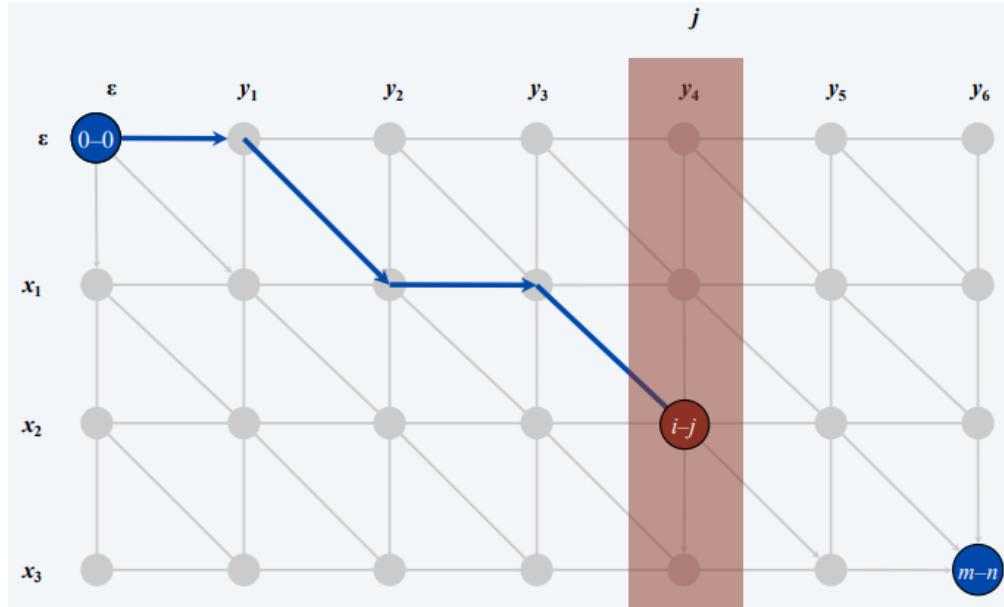
Grafo edit distance quindi:



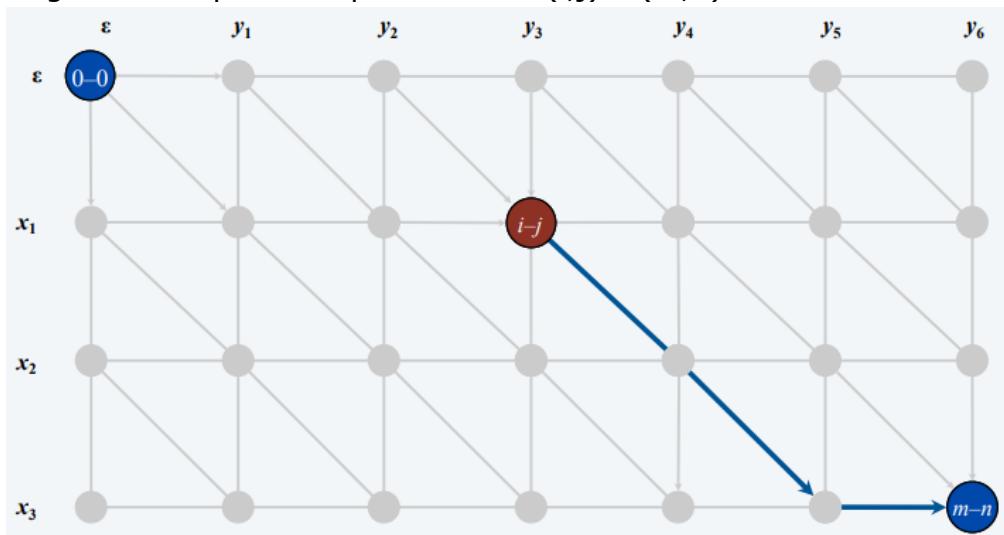
ALGORITMO DI HIRSCHBERG

Edit distance graph:

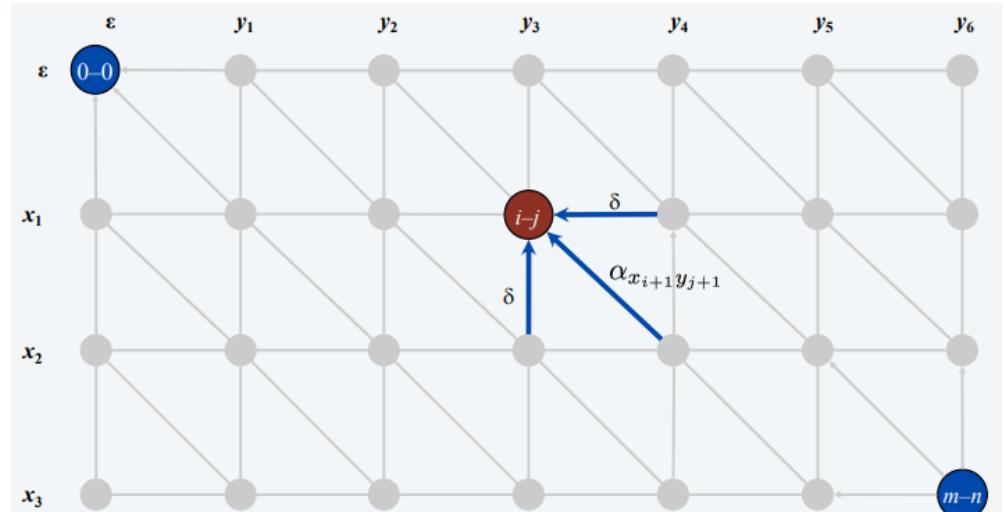
- sia $f(i,j)$ la lunghezza del percorso più breve da $(0,0)$ a (i,j)
- Lemma: $f(i,j) = \text{OPT}(i,j)$ per tutti gli i ed j
- Si può calcolare $f(,j)$ per qualunque j in tempo $O(mn)$ e in spazio $O(m+n)$



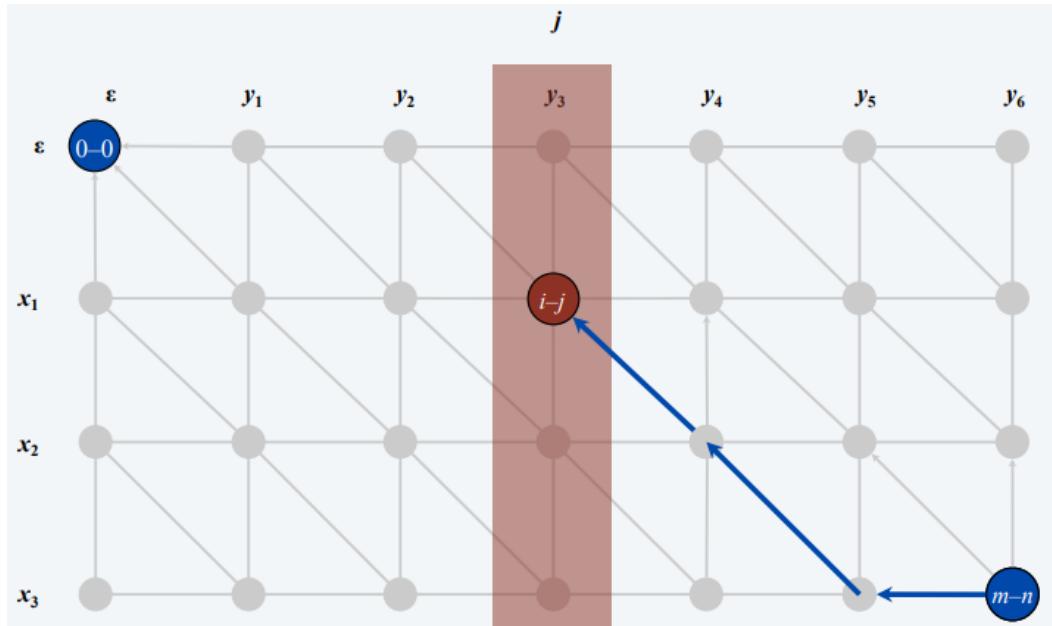
Sia $g(i,j)$ la lunghezza del percorso più breve da (i,j) a (m,n)



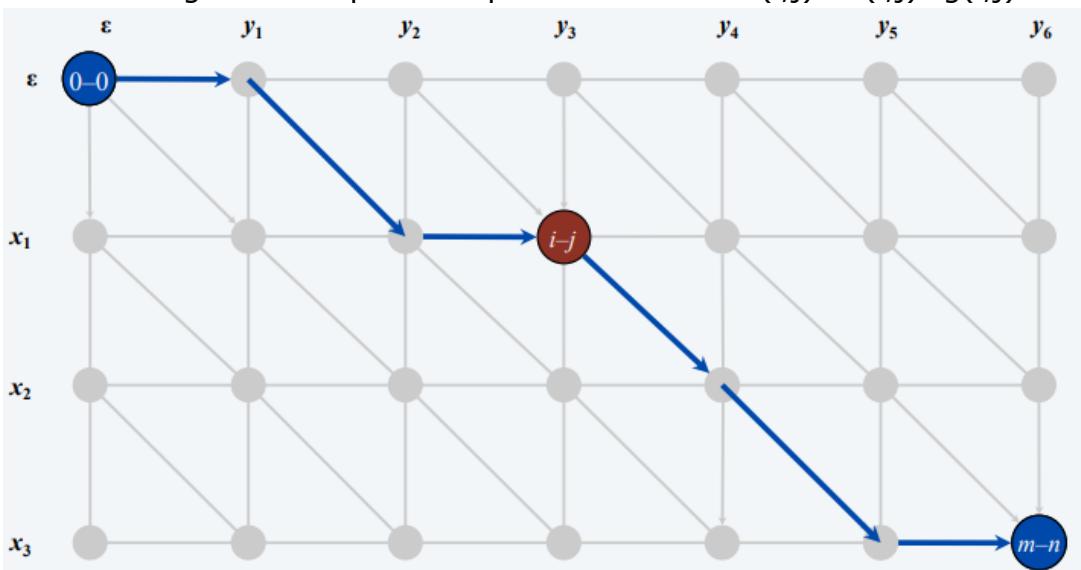
Si può calcolare $g(i,j)$ invertendo gli archi e invertendo i ruoli di $(0,0)$ e (m,n) .



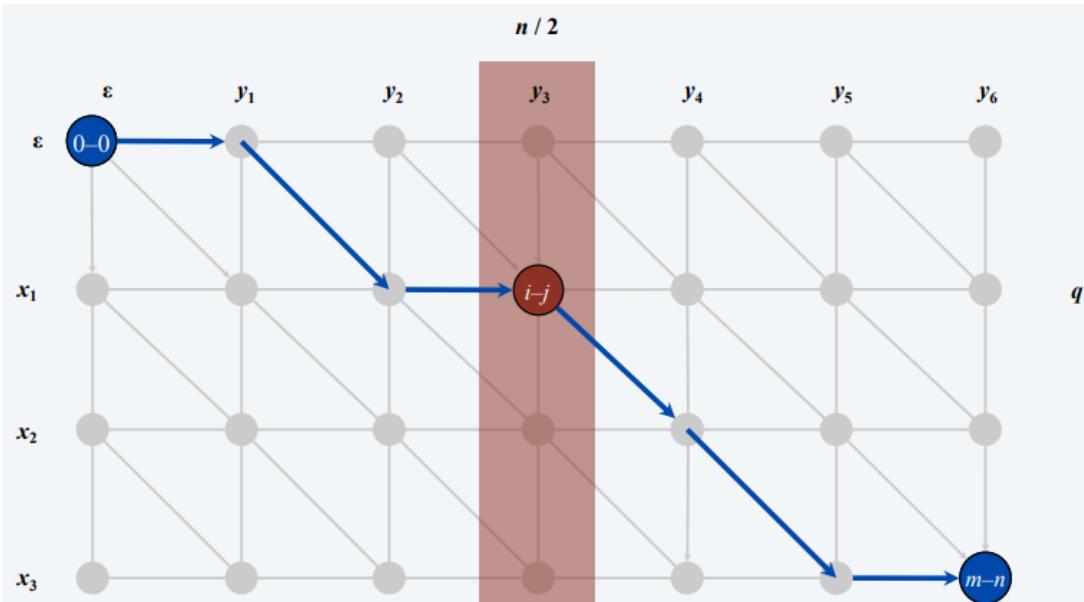
Si può calcolare $g(, j)$ per ogni j in tempo $O(mn)$ e in spazio $O(m+n)$



Osservazione1: la lunghezza del percorso più breve che usa (i,j) è $f(i,j)+g(i,j)$

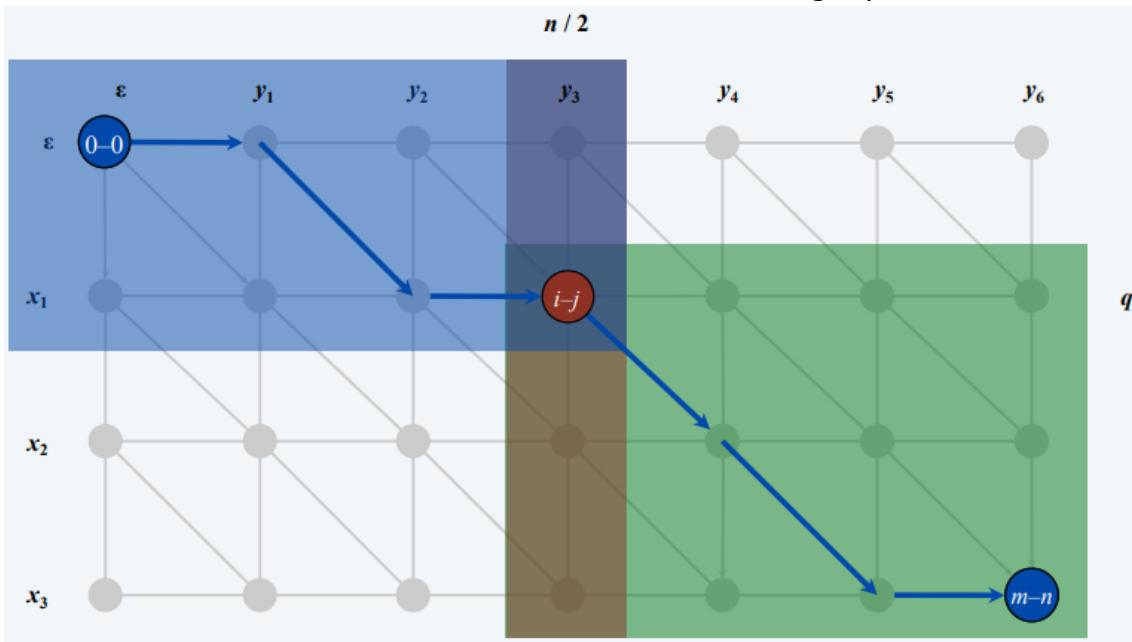


Osservazione2: sia q l'indice che minimizza $f(q, n/2)+g(q, n/2)$. Allora esiste un percorso più breve da $(0,0)$ a (m,n) che usa $(q, n/2)$.



Divide: Trovare un indice q che minimizza $f(q, n/2) + g(q, n/2)$; salvare il nodo $i-j$ come parte della soluzione.

Conquer. Calcolare ricorsivamente l'allineamento ottimale in ogni pezzo.



Analisi spaziale dell'algoritmo: l'algoritmo di Hirschberg usa spazio $\Theta(m+n)$

Dim:

- ogni chiamata ricorsiva usa spazio $\Theta(m)$ per calcolare $f(\cdot, n/2)$ e $g(\cdot, n/2)$
- solo spazio $\Theta(1)$ necessita di essere mantenuto per le chiamate ricorsive
- numero di chiamate ricorsive $\leq n$

Analisi tempo di esecuzione dell'algoritmo:

sia $T(m,n)=\max$ running time dell'algoritmo di Hirschberg's su stringhe di lunghezza al massimo m e n . Allora $T(m,n)=O(mn)$

Dim(induzione forte su $m+n$):

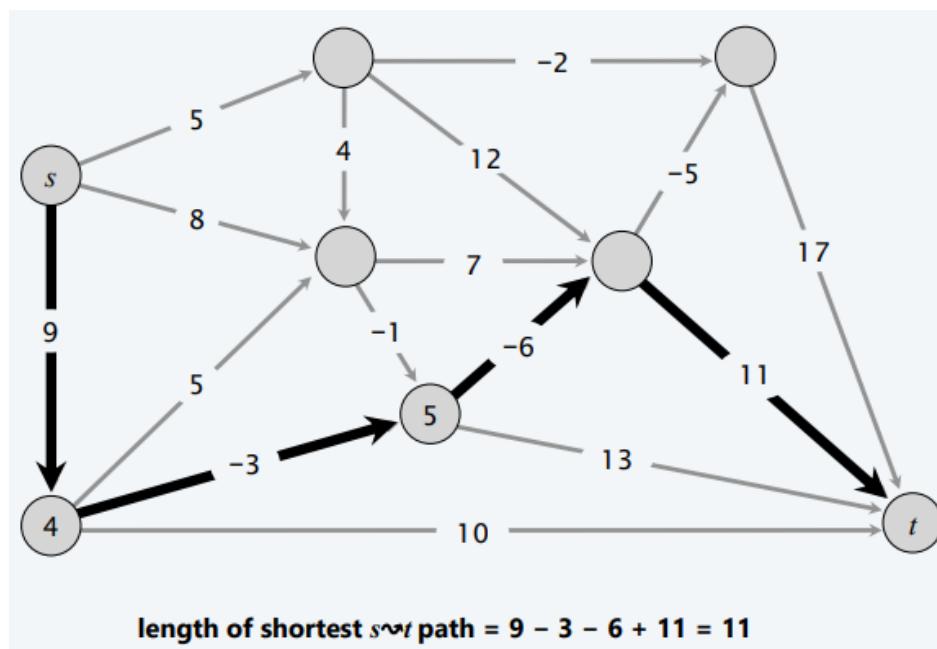
- $O(mn)$ time to compute $f(\cdot, n/2)$ and $g(\cdot, n/2)$ and find index q .
- $T(q, n/2) + T(m - q, n/2)$ time for two recursive calls.
- Choose constant c so that: $T(m, 2) \leq cm$
 $T(2, n) \leq cn$
 $T(m, n) \leq cmn + T(q, n/2) + T(m - q, n/2)$
- Claim. $T(m, n) \leq 2cmn$.
- Base cases: $m = 2$ and $n = 2$.
- Inductive hypothesis: $T(m', n') \leq 2cm'n'$ for all (m', n') with $m' + n' < m + n$.

$$\begin{aligned}
 T(m, n) &\leq T(q, n/2) + T(m - q, n/2) + cmn \\
 &\leq 2cq n/2 + 2c(m - q)n/2 + cmn \\
 &\stackrel{\text{inductive hypothesis}}{=} cq n + cmn - cq n + cmn \\
 &= 2cmn
 \end{aligned}$$

LEZ11 – 11/04/2024 (Bellman-Ford-Moore algorithm)

SHORTEST-PATH PROBLEM (WITH NEGATIVE WEIGHTS)

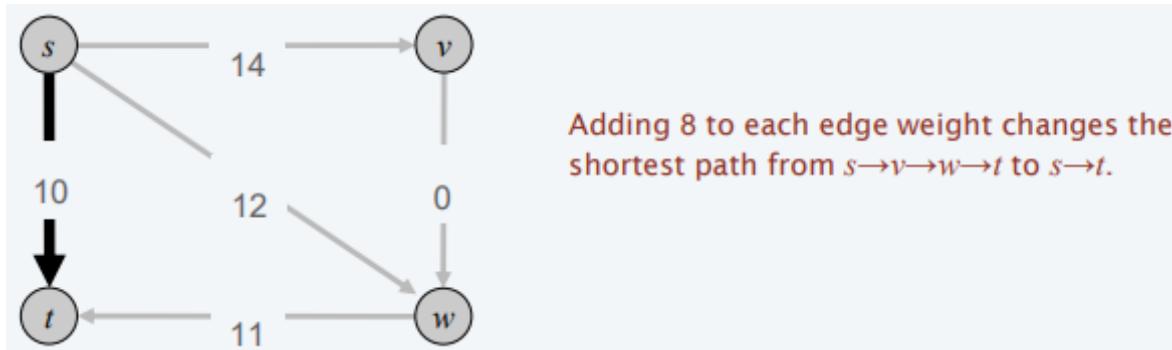
Dato un grafo $G=(V,E)$ con lunghezza arbitraria degli archi l_{vw} , trovare il percorso più corto dal nodo radice s al nodo destinazione t .



Nota: Dijkstra non produce lo shortest path quando gli archi sono negativi

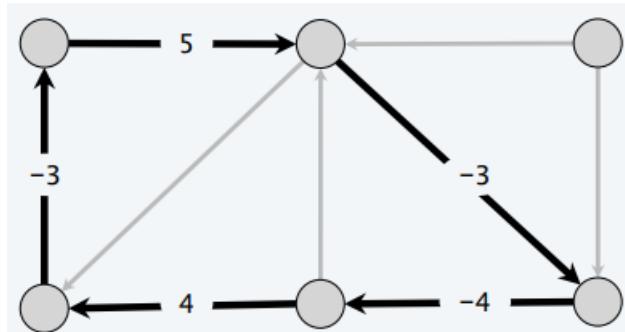


Nota: Aggiungere una costante al peso di ogni arco non necessariamente fa in modo che l'algoritmo di Dijkstra produce lo shortest path



CICLI NEGATIVI

Definizione: Un ciclo negativo è un ciclo diretto per il quale la somma delle lunghezze dei suoi archi è negativa.

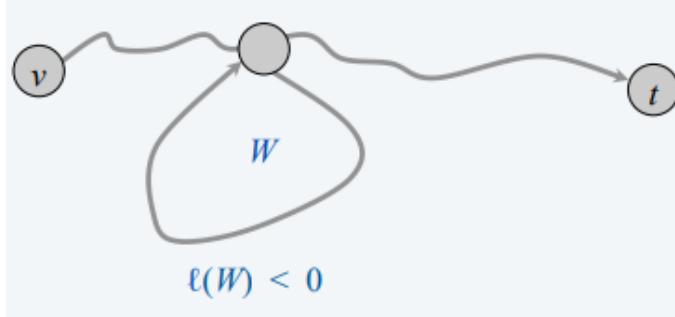


$$\text{a negative cycle } W : \quad \ell(W) = \sum_{e \in W} \ell_e < 0$$

LEMMA1

Se qualche percorso $v \rightsquigarrow t$ contiene cicli negativi, allora li non esiste uno shortest path.

Dim: se esiste un tale ciclo W , allora è possibile costruire un percorso $v \rightsquigarrow t$ arbitrariamente di lunghezza negativa deviando intorno a W quante volte si desidera.

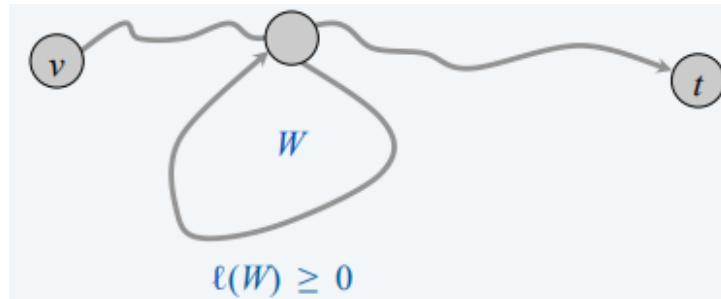


LEMMA2

Se G non ha cicli negativi, allora esiste uno shortest path da v a t che è semplice (ed ha numero di archi $\leq n-1$)

Dim:

- tra tutti gli shortest path, considerarne uno che usa il minor numero di archi.
- Se quel path P contiene un ciclo diretto W , è possibile rimuovere la porzione di P corrispondente a W senza aumentare la sua lunghezza

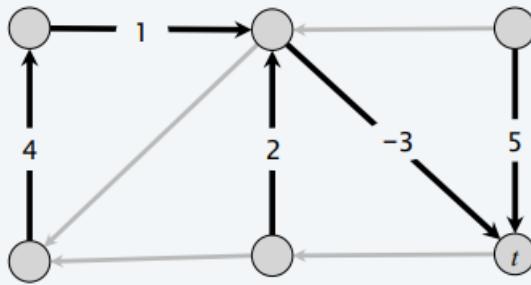


SINGLE-DESTINATION SHORTEST-PATHS PROBLEM

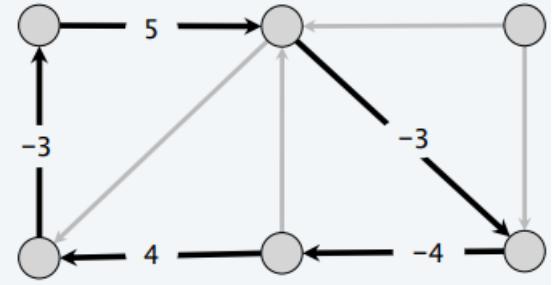
Dato un grafo $G=(V,E)$ con lunghezza degli archi ℓ_{vw} (ma senza cicli negativi) e con nodo t distinto, trovare uno shortest path da v a t per ogni nodo v .
(equivalente al single-source shortest-paths problem)

NEGATIVE-CYCLE PROBLEM

Dato un grafo $G=(V,E)$ con lunghezza degli archi ℓ_{vw} , trovare un ciclo negativo (se ne esiste uno)



(reverse) shortest-paths tree



negative cycle

DYNAMIC PROGRAMMING PER RISOLUZIONE "SHORTEST PATHS WITH NEGATIVE WEIGHTS"

Definizione: $OPT(i, v)$ =lunghezza del path più corto da v a t che usa al più i archi.

Goal: $OPT(n-1, v)$ per ogni v (dal lemma 2, se non ci sono cicli negativi, esiste uno shortest path che è semplice).

Caso1: shortest path da v a t usa numero di archi $\leq i-1$

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

Caso2: shortest path da v a t usa esattamente i archi

- Se (v, w) è il primo arco nel percorso più breve da v a t , occorre un costo di ℓ_{vw}
- Allora, selezionare il miglior path da w a t che usa un numero di archi $\leq i-1$

Equazione di Bellman:

$$OPT(i, v) = \begin{cases} 0 & \text{if } i = 0 \text{ and } v = t \\ \infty & \text{if } i = 0 \text{ and } v \neq t \\ \min \left\{ OPT(i-1, v), \min_{(v,w) \in E} \{ OPT(i-1, w) + \ell_{vw} \} \right\} & \text{if } i > 0 \end{cases}$$

Implementazione
(non efficiente):

SHORTEST-PATHS(V, E, ℓ, t)

FOREACH node $v \in V$:

$M[0, v] \leftarrow \infty$.

$M[0, t] \leftarrow 0$.

FOR $i = 1$ **TO** $n - 1$

FOREACH node $v \in V$:

$M[i, v] \leftarrow M[i-1, v]$.

FOREACH edge $(v, w) \in E$:

$M[i, v] \leftarrow \min \{ M[i, v], M[i-1, w] + \ell_{vw} \}$.

Teorema: Dato un grafo $G=(V,E)$ che non contiene cicli negativi, l'algoritmo DP calcola la lunghezza dello shortest path da v a t per tutti i nodi v in tempo $\Theta(mn)$ e spazio $\Theta(n^2)$

Dim:

- la tabella richiede spazio $\Theta(n^2)$
- ogni iterazione i impiega tempo spazio $\Theta(m)$ poiché esaminiamo ogni arco una volta

Trovare lo shortest path:

- Approccio1 = mantenere $\text{successor}[i,v]$ che punta al prossimo nodo su uno shortest path da v a t usando numero di archi $\leq i$
- Approccio2 = calcolare lunghezze ottime $M[i,v]$ e considerare solo gli archi con $M[i,v] = M[i-1,w] + \ell_{vw}$. Qualsiasi percorso diretto in questo sottografo è un percorso più breve.

MIGLIORALMENTI PRATICI PER SHORTEST PATHS WITH NEGATIVE WEIGHTS

Ottimizzazione spazio: mantenere due array a una dimensione (invece di due dimensioni)

- $d[v]$ = lunghezza dello shortest path da v a t che abbiamo trovato fino ad ora
- $\text{successor}[v]$ = prossimo nodo su un path da v a t

Ottimizzazione performance: se $d[w]$ non è stato aggiornato nell'iterazione $i - 1$, allora non c'è motivo di considerare gli archi che entrano in w nell'iterazione i .

IMPLEMENTAZIONE PIU' EFFICIENTE (BELLMAN-FORD-MOORE)

BELLMAN-FORD-MOORE(V, E, ℓ, t)

FOREACH node $v \in V$:

$d[v] \leftarrow \infty.$

$\text{successor}[v] \leftarrow \text{null}.$

FOR $i = 1$ TO $n - 1$

FOREACH node $w \in V$:

IF ($d[w]$ was updated in previous pass)

FOREACH edge $(v, w) \in E$:

IF ($d[v] > d[w] + \ell_{vw}$)

$d[v] \leftarrow d[w] + \ell_{vw}.$

$\text{successor}[v] \leftarrow w.$

IF (no $d[\cdot]$ value changed in pass i) **STOP.**

↑
pass i
 $O(m)$ time

LEMMA3

Per ogni nodo v : $d[v]$ è la lunghezza di qualche percorso da v a t

LEMMA4

Per ogni nodo v : $d[v]$ è monotona non crescente

LEMMA5

Dopo aver superato i, $d[v] \leq$ lunghezza dello shortest path da v a t che usa numero di archi $\leq i$.

Pf. [by induction on i]

- Base case: $i = 0$.
- Assume true after pass i .
- Let P be any $v \rightsquigarrow t$ path with $\leq i + 1$ edges.
- Let (v, w) be first edge in P and let P' be subpath from w to t .
- By inductive hypothesis, at the end of pass i , $d[w] \leq \ell(P')$ because P' is a $w \rightsquigarrow t$ path with $\leq i$ edges.
- After considering edge (v, w) in pass $i + 1$:

and by Lemma 4,
 $d[w]$ does not increase

$$\begin{aligned} d[v] &\leq \ell_{vw} + d[w] \\ &\leq \ell_{vw} + \ell(P') \\ &= \ell(P) \quad \blacksquare \end{aligned}$$

and by Lemma 4,
 $d[v]$ does not increase

TEOREMA2 (BELLMAN-FORD-MOORE: ANALISI)

Assumendo che non ci siano cicli negativi, Bellman-ford-Moore calcola le lunghezze dello shortest path da v a t in tempo $O(mn)$ e in spazio $\Theta(n)$.

Dim: lemma2(shortest path esiste ed ha al massimo $n-1$ archi) + lemma5(dopo i passi, $d[v] \leq$ lunghezza dello shortest path che usa numero di archi $\leq i$).

OSSERVAZIONE

Bellman-Ford-Moore è tipicamente veloce nella pratica.

- Arco (v, w) considerato al passo $i+1$ solo se $d[w]$ modificato al passo i
- Se shortest path ha k archi, allora l'algoritmo lo trova dopo numero di passi $\leq k$

NOTA

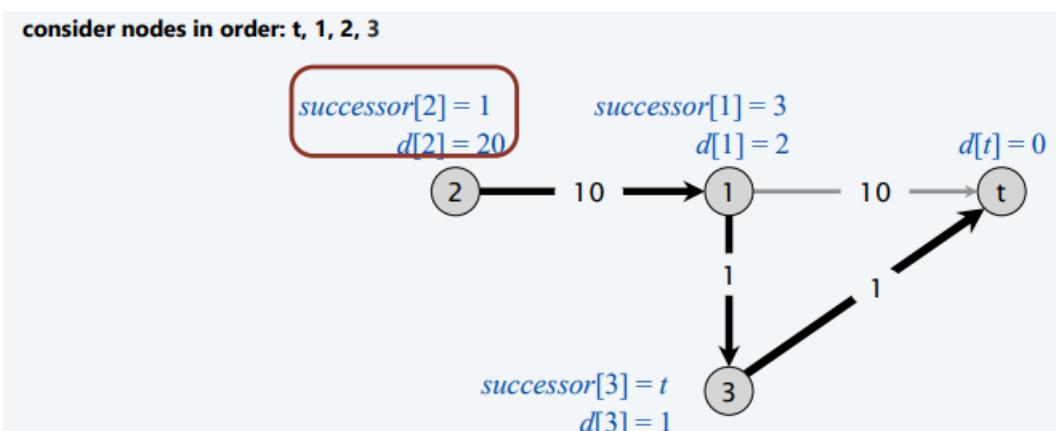
È vera l'affermazione seguente?

Tramite Bellman-Ford-Moore seguire il puntatore $\text{successor}[v]$ da un percorso diretto da v a t di lunghezza $d[v]$.

Risposta: NO!

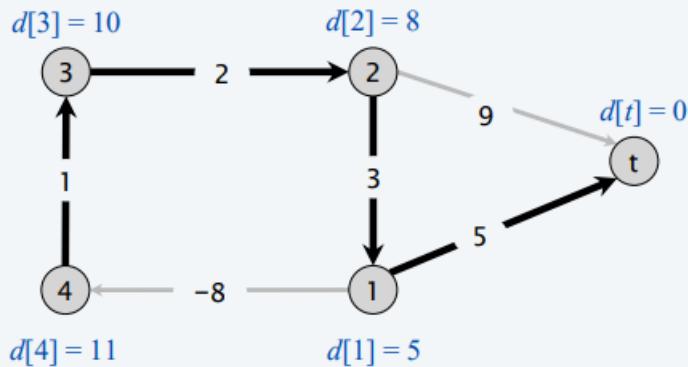
Controesempio:

1. la lunghezza del successor path da v a t può essere strettamente più corta rispetto a $d[v]$.

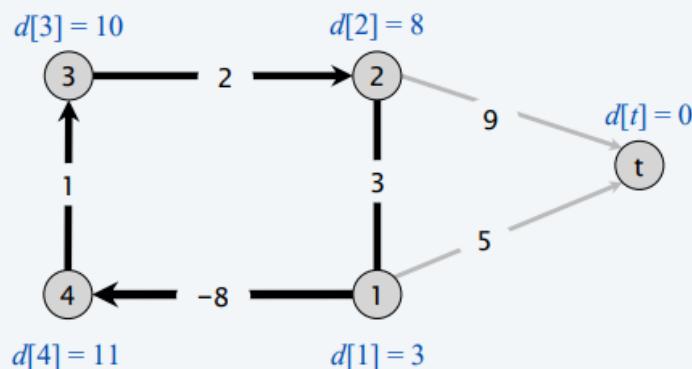


2. Se c'è un ciclo negativo, successor graph potrebbe avere cicli diretti

consider nodes in order: t, 1, 2, 3, 4



consider nodes in order: t, 1, 2, 3, 4



LEMMA6 (BELLMAN-FORD-MOORE, TROVARE LO SHORTEST PATH)

Qualsiasi ciclo diretto W nel successor graph è un ciclo negativo.

Dim:

- If $\text{successor}[v] = w$, we must have $d[v] \geq d[w] + \ell_{vw}$.
(LHS and RHS are equal when $\text{successor}[v]$ is set; $d[w]$ can only decrease; $d[v]$ decreases only when $\text{successor}[v]$ is reset)
- Let $v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k \rightarrow v_1$ be the sequence of nodes in a directed cycle W .
- Assume that (v_k, v_1) is the last edge in W added to the successor graph.
- Just prior to that:

$$\begin{aligned} d[v_1] &\geq d[v_2] + \ell(v_1, v_2) \\ d[v_2] &\geq d[v_3] + \ell(v_2, v_3) \\ &\vdots && \vdots \\ d[v_{k-1}] &\geq d[v_k] + \ell(v_{k-1}, v_k) \\ d[v_k] &> d[v_1] + \ell(v_k, v_1) \end{aligned}$$

← holds with strict inequality since we are updating $d[v_k]$
- Adding inequalities yields $\ell(v_1, v_2) + \ell(v_2, v_3) + \dots + \ell(v_{k-1}, v_k) + \ell(v_k, v_1) < 0$.

$\overbrace{\hspace{300pt}}$
W is a negative cycle

TEOREMA3 (BELLMAN-FORD-MOORE: TROVARE IL PERCORSO PIU' BREVE)

Assumendo non ci siano cicli negativi, Bellman-Ford-Moore trova gli shortest paths da v a t per ogni nodo in tempo $O(mn)$ e in spazio $\Theta(n)$.

Dim:

- The successor graph cannot have a directed cycle. [Lemma 6]
 - Thus, following the successor pointers from v yields a directed path to t .
 - Let $v = v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k = t$ be the nodes along this path P .
 - Upon termination, if $\text{successor}[v] = w$, we must have $d[v] = d[w] + \ell_{vw}$.
(LHS and RHS are equal when $\text{successor}[v]$ is set; $d[\cdot]$ did not change)
 - Thus,

$$\begin{aligned} d[v_1] &= d[v_2] + \ell(v_1, v_2) \\ d[v_2] &= d[v_3] + \ell(v_2, v_3) \\ \vdots &\quad \vdots \quad \vdots \\ d[v_{k-1}] &= d[v_k] + \ell(v_{k-1}, v_k) \end{aligned}$$
since algorithm terminated
 - Adding equations yields $d[v] = d[t] + \ell(v_1, v_2) + \ell(v_2, v_3) + \dots + \ell(v_{k-1}, v_k)$. ▪
-

BELLMAN-FORD-MOORE: CONTROLLO PER CICLI NEGATIVI

BELLMAN-FORD-MOORE(V, E, ℓ, t)

FOREACH node $v \in V$:

$d[v] \leftarrow \infty$.
 $\text{successor}[v] \leftarrow \text{null}$.

$d[t] \leftarrow 0$.

FOR $i = 1$ TO $n - 1$

FOREACH node $w \in V$:

IF ($d[w]$ was updated in previous pass)

FOREACH edge $(v, w) \in E$:

IF ($d[v] > d[w] + \ell_{vw}$)
 $d[v] \leftarrow d[w] + \ell_{vw}$.
 $\text{successor}[v] \leftarrow w$.

IF (no $d[\cdot]$ value changed in pass i) STOP.

FOREACH edge $(v, w) \in E$

IF ($d[v] > d[w] + \ell_{vw}$) THEN return “there is a negative cycle”

Se c'è un ciclo negativo che si può raggiungere questo algoritmo lo segnala. Dim:

- If there is no negative cycle the pass #n do nothing
- Let $v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k \rightarrow v_1$ a directed negative cycle W .
- assume by contradiction that the algorithm does not return it
- Then: condition of the last IF is always false.
- Hence:
$$\begin{aligned} d[v_1] &\leq d[v_2] + \ell(v_1, v_2) \\ d[v_2] &\leq d[v_3] + \ell(v_2, v_3) \\ &\vdots & \vdots \\ d[v_{k-1}] &\leq d[v_k] + \ell(v_{k-1}, v_k) \\ d[v_k] &\leq d[v_1] + \ell(v_k, v_1) \end{aligned}$$
- Adding inequalities yields $\ell(v_1, v_2) + \ell(v_2, v_3) + \dots + \ell(v_{k-1}, v_k) + \ell(v_k, v_1) \geq 0$. W is cannot be a negative cycle: a contradiction

LEZ12 – 16/04/2024 (Ultima esercitazione programmazione dinamica)

ESERCIZIO1

Problema: massima sottostringa palindroma

Input: una stringa S di n caratteri

Goal: eliminare da S dei caratteri (anche nessuno) in modo tale che la sottostringa risultante S^* sia palindroma e di lunghezza massima.

OPT: massima lunghezza sottostringa palindroma.

$S = \boxed{\text{A L G O R I T M O}}$

$S^* = \boxed{\text{A L G O R I T M O}}$

$$OPT = 3$$

SVOLGIMENTO

Sottoproblema: $OPT[i, j]$ = massima lunghezza sottostringa palindroma di $S[i \dots j]$

Casi base: se $i > j$ allora $OPT[i, j] = 0$
 $OPT[i, i] = 1$

Soluzione: $OPT[1, n]$

Formula ricorsiva:

$$OPT[i, j] = \begin{cases} 2 + OPT[i+1, j-1] & \text{se } S[i] = S[j] \\ \max \{ OPT[i+1, j], OPT[i, j-1] \} & \text{altrimenti} \end{cases}$$

Aggiungo $S[i]$ ed $S[j]$ a S^* , e ricorsivamente osservo $S[i+1 \dots j-1]$

Rimuovo $S[i]$
Rimuovo $S[j]$

Tabella:

	A	L	G	O	R	I	T	M	O
A	1	0	0	0	0	0	0	0	0
L	0	1	0	0	0	0	0	0	0
G	0	0	1	0	0	0	0	0	0
O	0	0	0	1	0	0	0	0	0
R	0	0	0	0	1	0	0	0	0
I	0	0	0	0	0	1	0	0	0
T	0	0	0	0	0	0	1	0	0
M	0	0	0	0	0	0	0	1	0
O	0	0	0	0	0	0	0	0	1

Casi base

	A	L	G	O	R	I	T	M	O
A	1	1	0	0	0	0	0	0	0
L	0	1	1	0	0	0	0	0	0
G	0	0	1	1	0	0	0	0	0
O	0	0	0	1	1	0	0	0	0
R	0	0	0	0	1	1	0	0	0
I	0	0	0	0	0	1	1	0	0
T	0	0	0	0	0	0	1	1	0
M	0	0	0	0	0	0	0	1	1
O	0	0	0	0	0	0	0	0	1

Caso generale

Attenzione: da riempire
in diagonale! Perché?

	A	L	G	O	R	I	T	M	O
A	1	1	1	1	1	1	1	1	3
L	0	1	1	1	1	1	1	1	3
G	0	0	1	1	1	1	1	1	3
O	0	0	0	1	1	1	1	1	3
R	0	0	0	0	1	1	1	1	1
I	0	0	0	0	0	1	1	1	1
T	0	0	0	0	0	0	1	1	1
M	0	0	0	0	0	0	0	1	1
O	0	0	0	0	0	0	0	0	1

Valore ottimo

Time $O(n^2)$

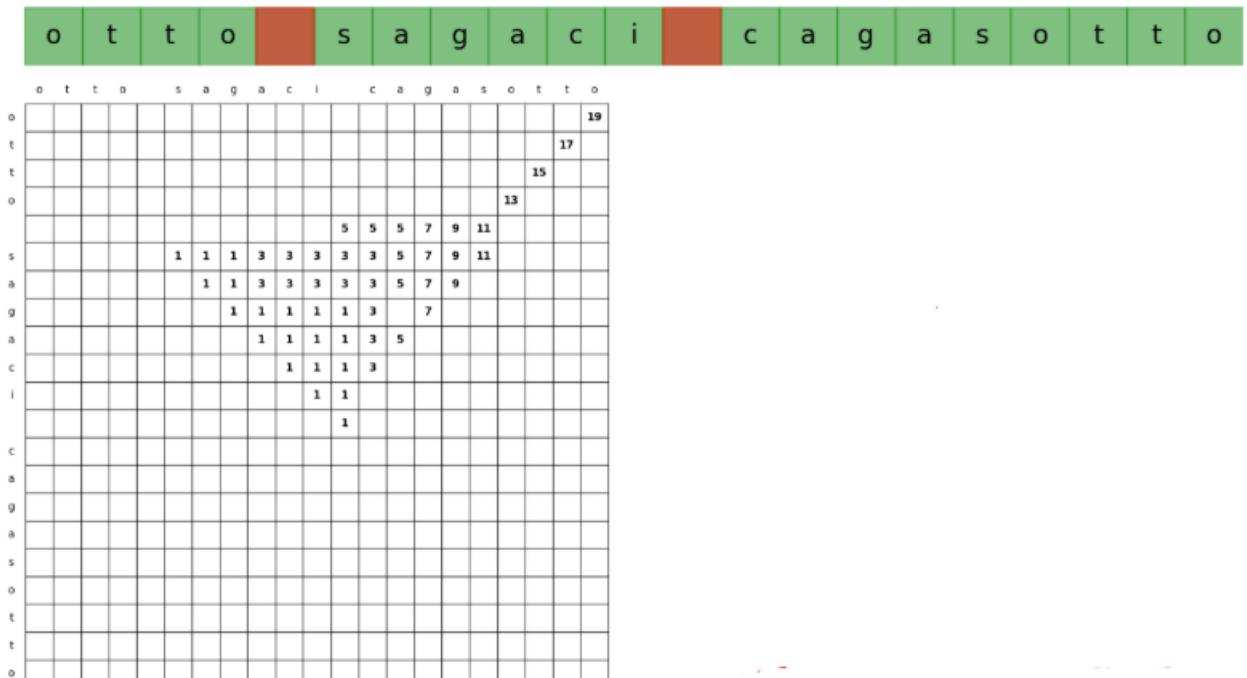
Codice implementazione in py con approccio tabulation (bottom-up):

```
def opt(s: str) -> int:
    n = len(s)
    M = [[0] * n for _ in range(n)]
    for i in range(n):
        M[i][i] = 1

    for d in range(1, n):
        for i in range(n - d):
            j = i + d
            if s[i] == s[j]:
                M[i][j] = 2 + M[i + 1][j - 1]
            else:
                M[i][j] = max(M[i + 1][j], M[i][j - 1])

    return M[0][n - 1]
```

NOTA: Tante celle sono vuote o inutili, possiamo ottenere miglioramenti riempendo solo le celle necessarie!



Codice:

```
def opt(s: str, i: int, j: int, cache: dict = dict()) -> int:
    if (i, j) in cache:
        return cache[i, j]

    if i > j:
        cache[i, j] = 0
    elif i == j:
        cache[i, j] = 1
    else:
        if s[i] == s[j]:
            cache[i, j] = 2 + opt(s, i + 1, j - 1, cache)
        else:
            cache[i, j] = max(opt(s, i + 1, j, cache), opt(s, i, j - 1, cache))

    return cache[i, j]
```

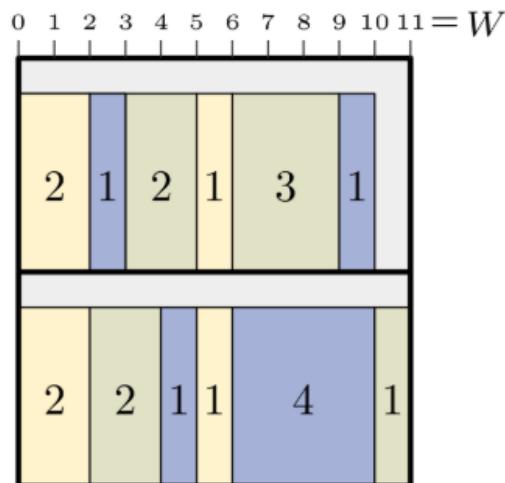
ESERCIZIO2

Input: una collezione di n libri di spessore $t_1, \dots, t_n \in \mathbb{N}$.

Goal: vuoi comprare due scaffali S_1 ed S_2 , entrambi di lunghezza $W \in \mathbb{N}$, per disporre tutti i tuoi libri.

OPT: lunghezza minima necessaria per gli scaffali S_1 ed S_2

Inptu: { 2, 1, 2, 2, 1, 1, 2, 1, 4, 1, 3, 1 }



OPT: 11

SVOLGIMENTO

Sottoproblema: $\text{OPT}[i, w] = \text{minimo spazio usato sullo scaffale } S_1 \text{ se sullo scaffale } S_2 \text{ ho a disposizione spazio } w \text{ e voglio piazzare i libri di spessore } t_1, \dots, t_i$.

Caso base: $\text{OPT}[0, w] = 0$

Soluzione: $\min W$ tale che $\text{OPT}[n, W] \leq W$

$$W \leq T = \sum t_i$$

Spazio: $O(n T)$ dove $T = \sum t_i$

Tempo: $O(n T)$, ogni volta devo combinare un numero costante di sottoproblemi.

PSEUDO POLINOMIALE!

Formula ricorsiva:

$$\text{OPT}[i, w] = \begin{cases} t_i + \text{OPT}[i - 1, w] & \text{se } t_i > w \\ \min \{t_i + \text{OPT}[i - 1, w], \text{OPT}[i - 1, w - t_i]\} & \text{altrimenti} \end{cases}$$

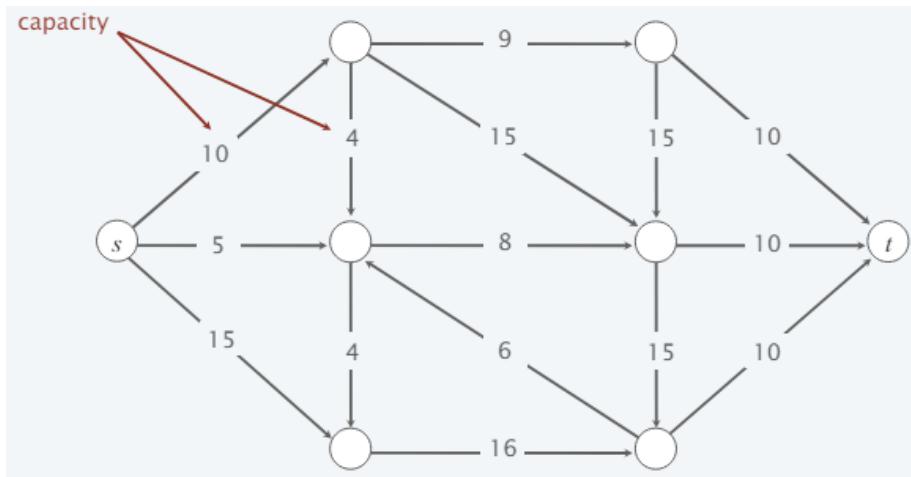
t_i non entra nello scaffale S₂, quindi lo devo mettere per forza in S₁
 Inserisco t_i nello scaffale S₁ Inserisco t_i nello scaffale S₂

FLOW NETWORK (FLUSSO DI RETE)

Un flow network è una tupla $G=(V,E,s,t,c)$

- Digraph (V,E) con sorgente $s \in V$ e nodo finale $t \in V$ (assumendo che tutti i nodi siano raggiungibili da s)
- Capacità $c(e) > 0$ per ogni $e \in E$

Intuitivamente: Materiale che scorre attraverso una rete di trasporto; il materiale proviene dalla fonte s e viene inviato al nodo t .

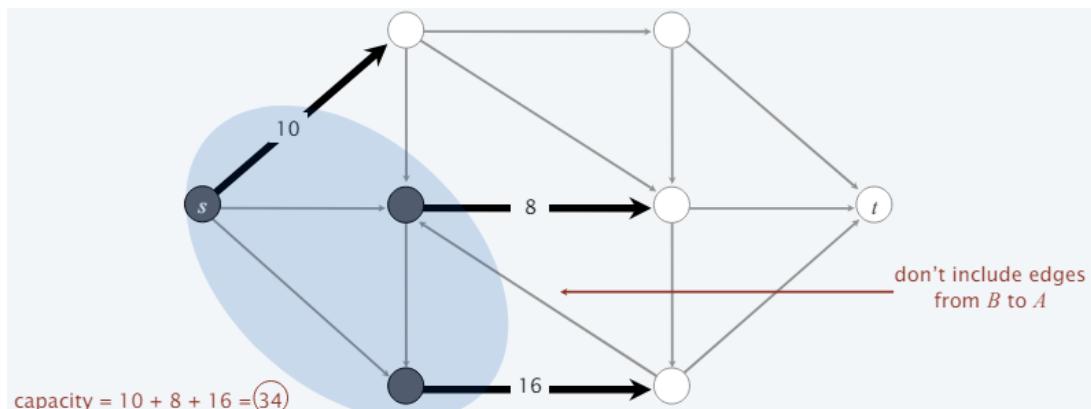
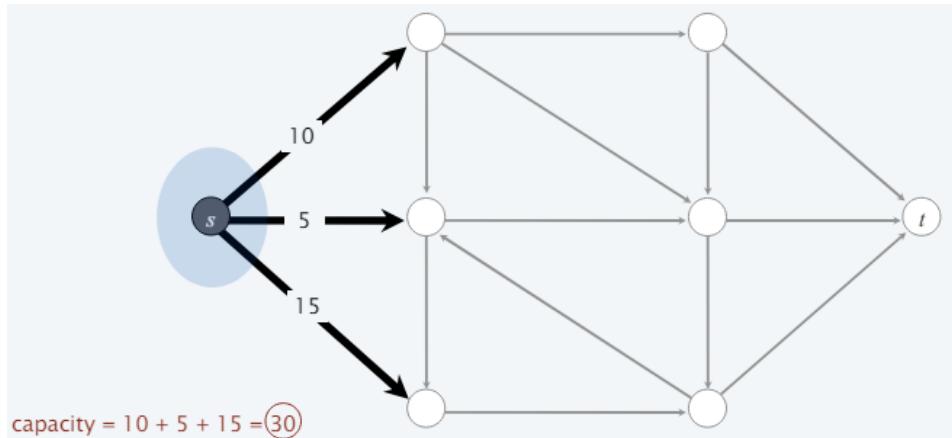


MINIMUM-CUT PROBLEM (PROBLEMA DI MINIMO TAGLIO)

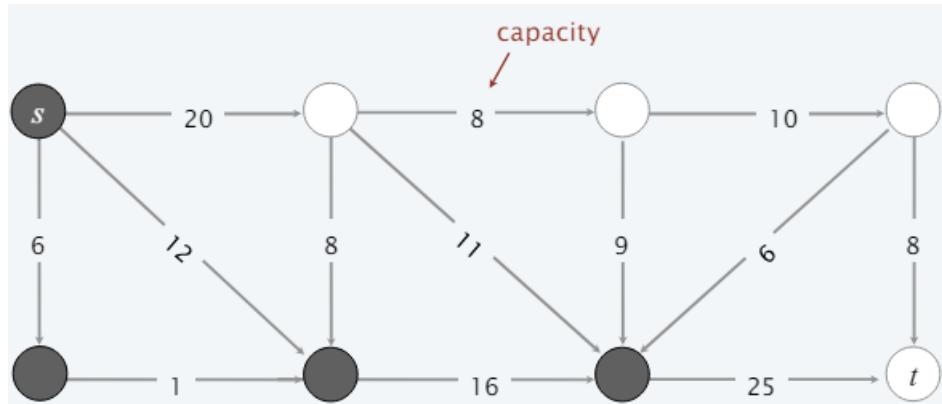
DEF: un st-cut (cut) è una partizione (A, B) dei nodi con $s \in A$ e $t \in B$.

DEF: la sua capacità è la somma delle capacità degli archi da A a B

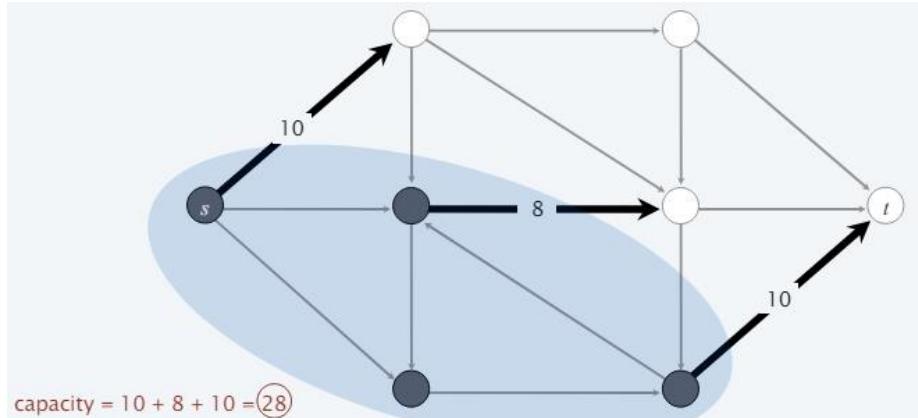
$$cap(A, B) = \sum_{e \text{ out of } A} c(e)$$



Esempio ulteriore: la capacità del seguente st-cut è $20+25=45$



Quindi, Min-cut problem: trovare un taglio di capacità minima.

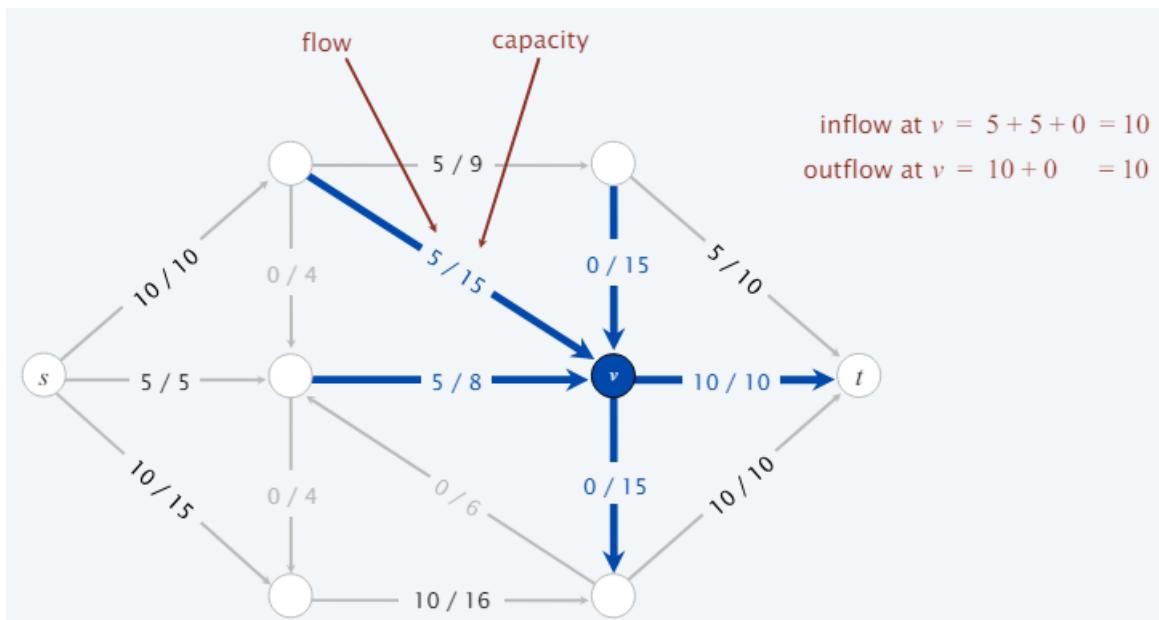


MAXIMUM-FLOW PROBLEM (PROBLEMA DEL FLUSSO MASSIMO)

DEF: un st-flow (flow) f è una funzione che soddisfa le proprietà seguenti:

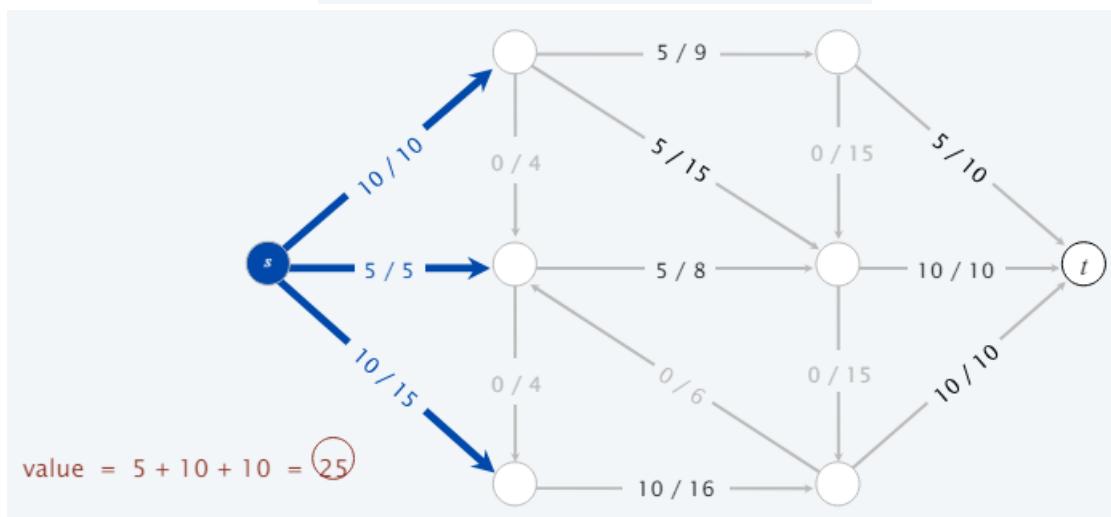
1. Per ogni $e \in E$ vale che $0 \leq f(e) \leq c(e)$ [capacity]

2. Per ogni $v \in V - \{s, t\}$ vale che $\sum_{e \text{ in to } v} f(e) = \sum_{e \text{ out of } v} f(e)$ [flow conservation]

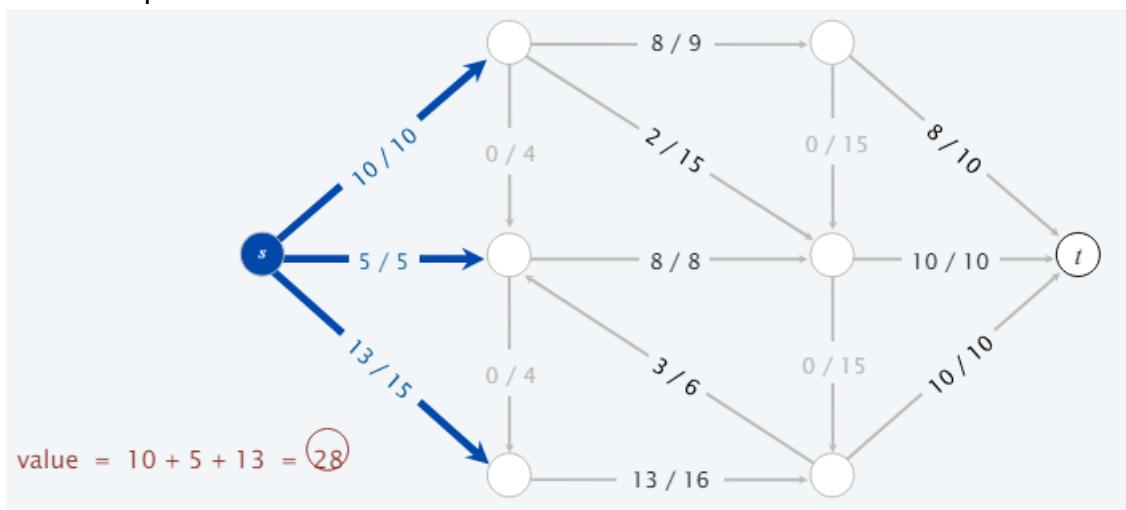


DEF: Il valore di un flow f è

$$val(f) = \sum_{e \text{ out of } s} f(e) - \sum_{e \text{ in to } s} f(e)$$



Quindi, Max-flow problem: trovare un flusso di valore massimo

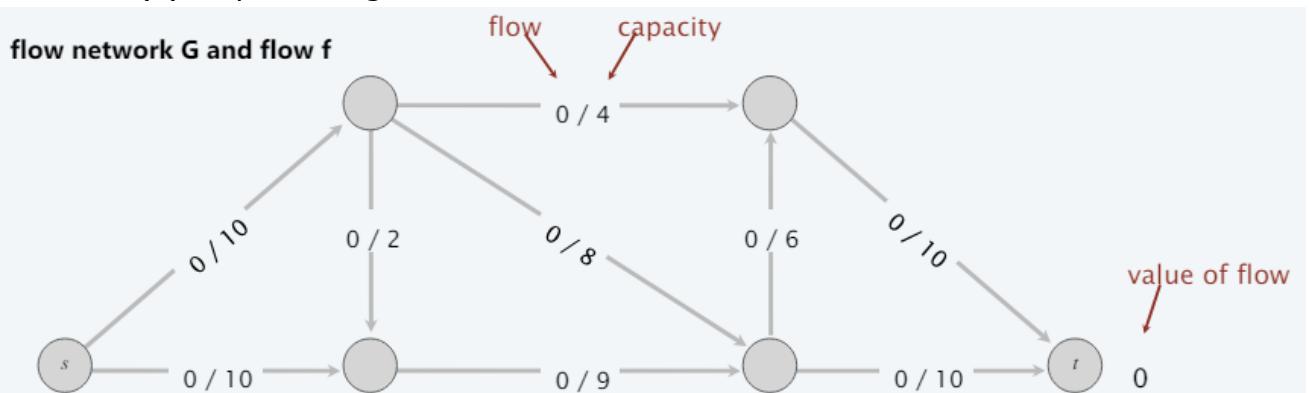


ALGORITMO PER MAX-FLOW

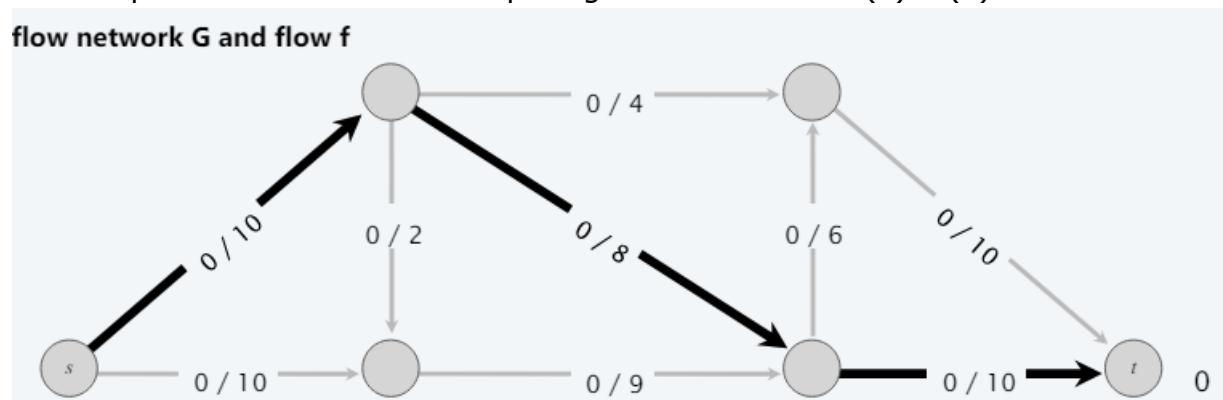
Prima di presentare l'algoritmo di Ford-Fulkerson iniziamo col ragionare con un algoritmo di tipo greedy e vediamo perché non funziona.

Algoritmo Greedy:

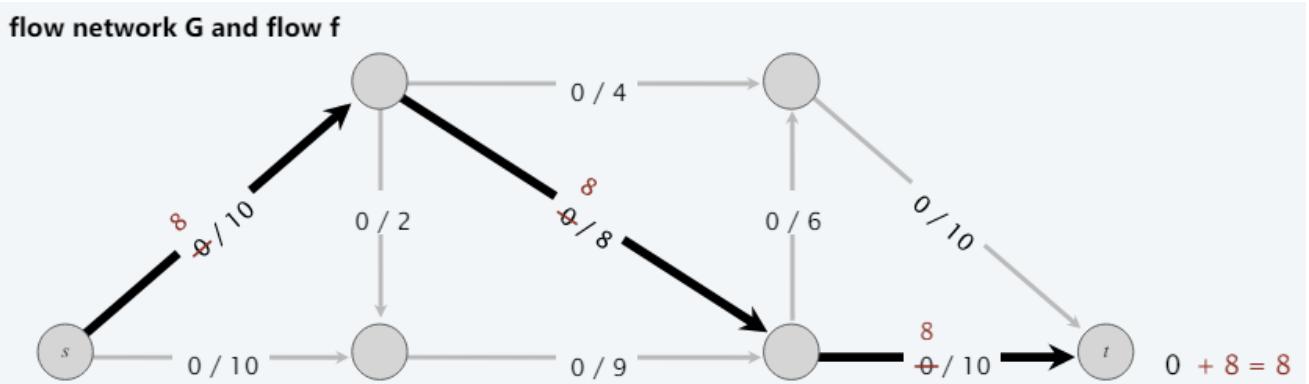
- 1) All'inizio $f(e)=0$ per tutti gli archi $e \in E$



- 2) Trovare un percorso P da s a t in cui per ogni arco si ha che $f(e) < c(e)$

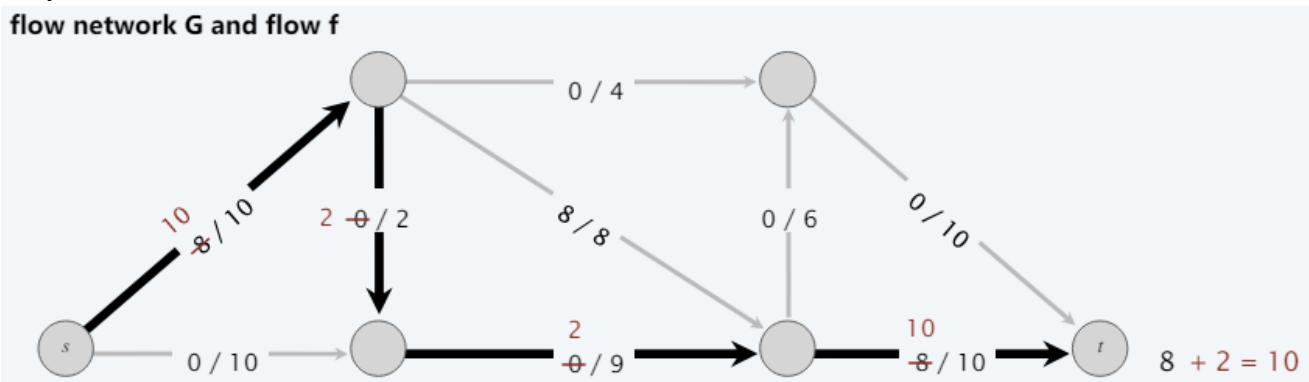


- 3) Aumentare il flusso lungo il percorso p



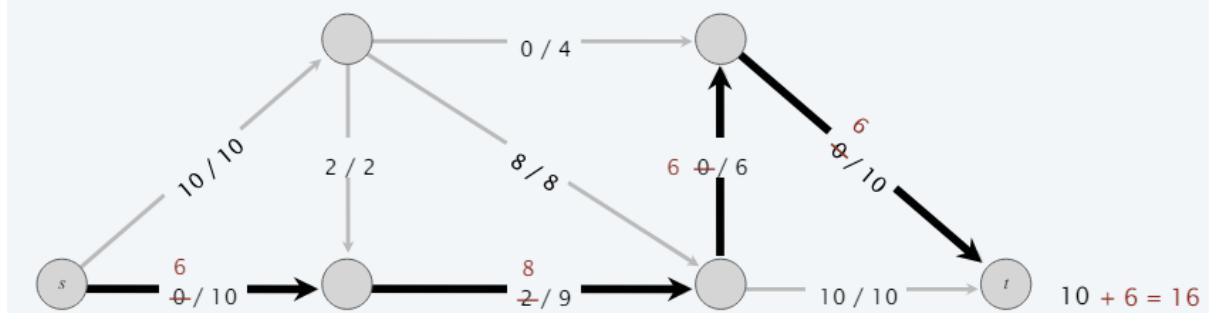
- 4) Ripetere finché non ci si blocca

4.1)



4.2)

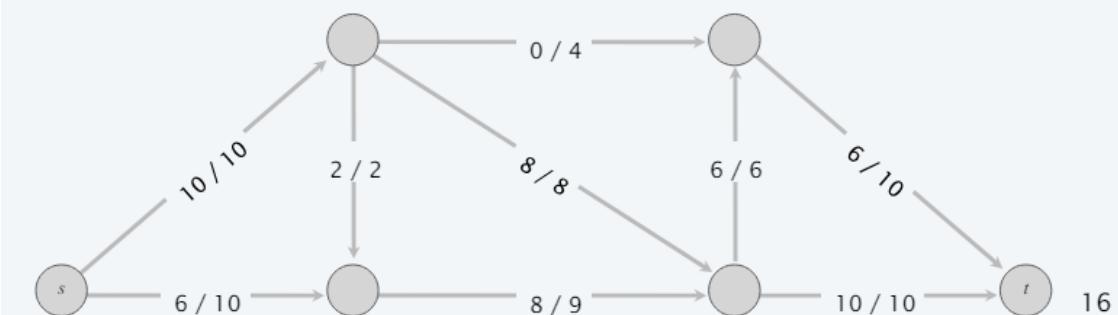
flow network G and flow f



4.3)

ending flow value = 16

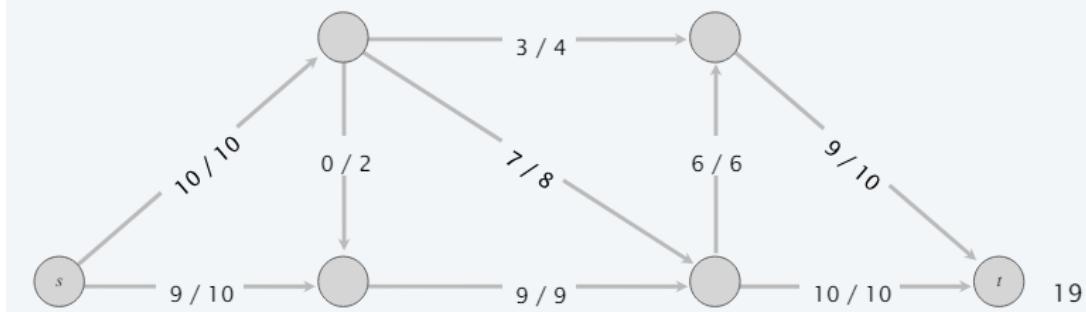
flow network G and flow f



NOTA: Il valore corretto di max flow è 19 non 16!

but max-flow value = 19

flow network G and flow f

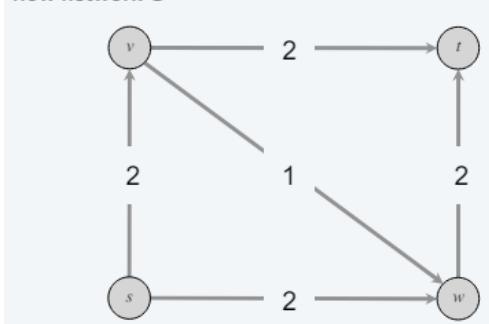


Perché l'algoritmo greedy fallisce? Fallisce perché una volta che l'algoritmo greedy aumenta il flusso su un arco, non lo fa diminuire mai più.

Esempio: considerare il seguente flow network G

- L'unico max flow f^* ha $f^*(v,w)=0$
- L'algoritmo greedy poteva scegliere $s \rightarrow v \rightarrow w \rightarrow t$ come primo percorso

flow network G

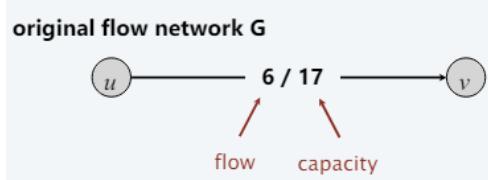


Quindi abbiamo capito che c'è la necessità di introdurre un meccanismo per "annullare" una cattiva decisione.

RESIDUAL NETWORK

- Original edge. $E = (u, v) \in E$.

- Flow $f(e)$
- Capacity $c(e)$



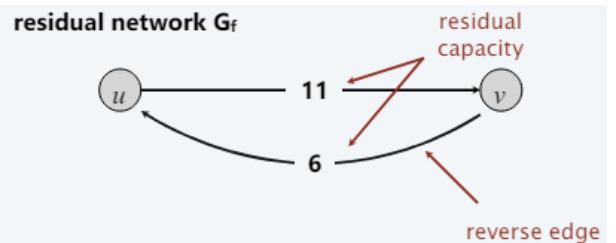
- Reverse edge. $e^{\text{reverse}} = (v, u)$.

- "Undo" del flusso inviato

- Residual capacity.

Residual capacity.

$$c_f(e) = \begin{cases} c(e) - f(e) & \text{if } e \in E \\ f(e^{\text{reverse}}) & \text{if } e^{\text{reverse}} \in E \end{cases}$$



- Residual network.

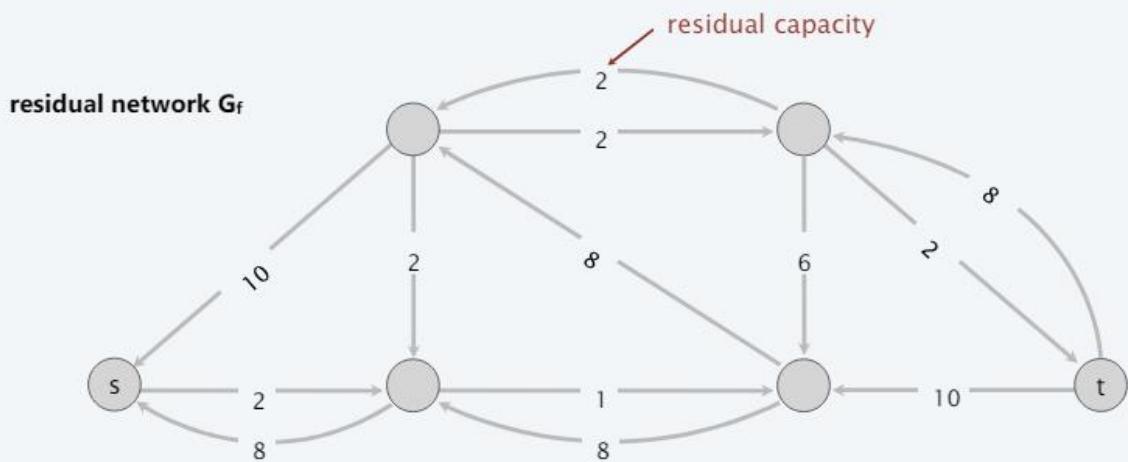
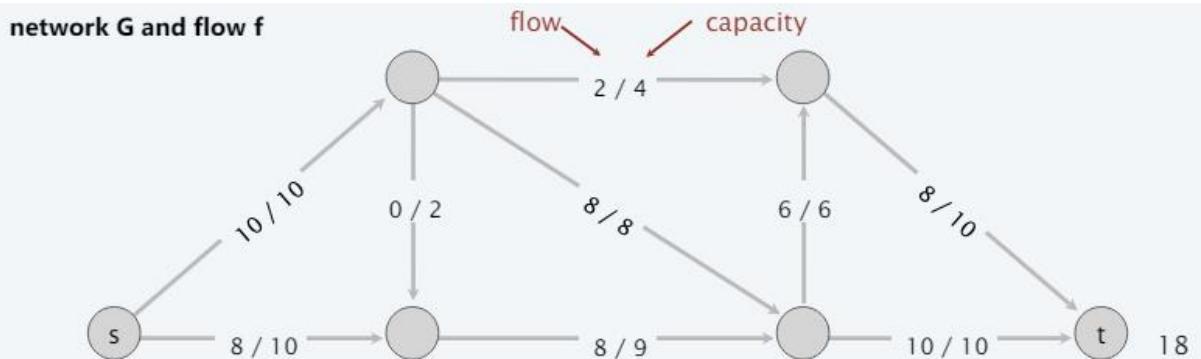
Residual network. $G_f = (V, E_f, s, t, c_f)$.

- $E_f = \{e : f(e) < c(e)\} \cup \{e : f(e^{\text{reverse}}) > 0\}$.
- Key property: f' is a flow in G_f iff $f + f'$ is a flow in G .

edges with positive residual capacity

where flow on a reverse edge negates flow on corresponding forward edge

Un esempio di Residual network è il seguente:



AUGMENTING PATH (PERCORSO AUMENTANTE)

DEF: Un augmenting path è un semplice path che va da s a t all'interno di un residual network G_f .

DEF: La bottleneck capacity(capacità del collo di bottiglia) di un augmenting path P è la minima capacità residua tra tutti gli archi in P .

PROP.CHIAVE: Sia f un flusso e P un augmenting path in G_f .

Allora dopo aver chiamato $f' \leftarrow \text{AUGMENT}(f, c, P)$ la risultante f' è un flusso e $\text{val}(f') = \text{val}(f) + \text{bottleneck}(G_f, P)$.

AUGMENT(f, c, P)

$\delta \leftarrow$ bottleneck capacity of augmenting path P .

FOREACH edge $e \in P$:

IF ($e \in E$) $f(e) \leftarrow f(e) + \delta$.

ELSE $f(e^{\text{reverse}}) \leftarrow f(e^{\text{reverse}}) - \delta$.

RETURN f .

ALGORITMO FORD-FULKERSON

- 1) All'inizio $f(e)=0$ per tutti gli archi $e \in E$
- 2) Trovare un percorso P da s a t nel residual network G_f
- 3) Aumentare il flusso lungo il percorso P
- 4) Ripetere finché non ci si blocca

FORD-FULKERSON(G)

FOREACH edge $e \in E : f(e) \leftarrow 0$.

$G_f \leftarrow$ residual network of G with respect to flow f .

WHILE (there exists an $s \rightsquigarrow t$ path P in G_f)

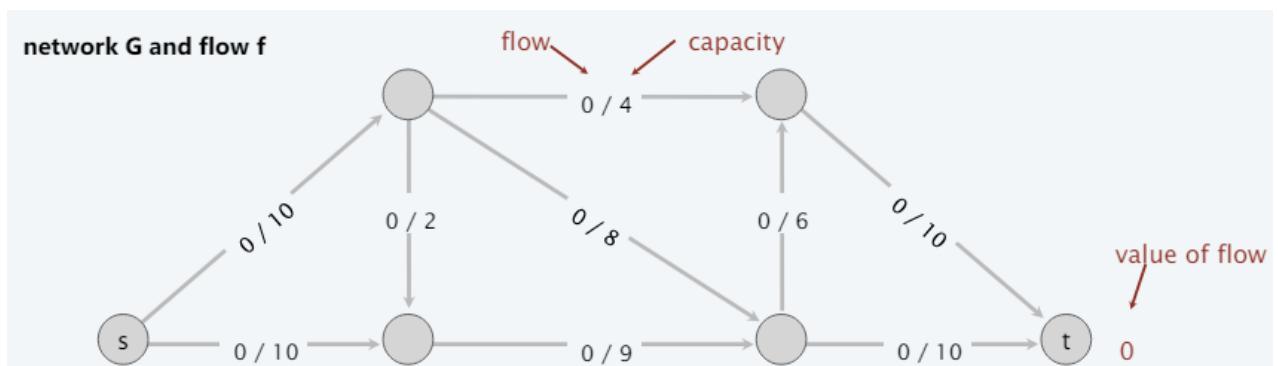
$f \leftarrow \text{AUGMENT}(f, c, P)$.

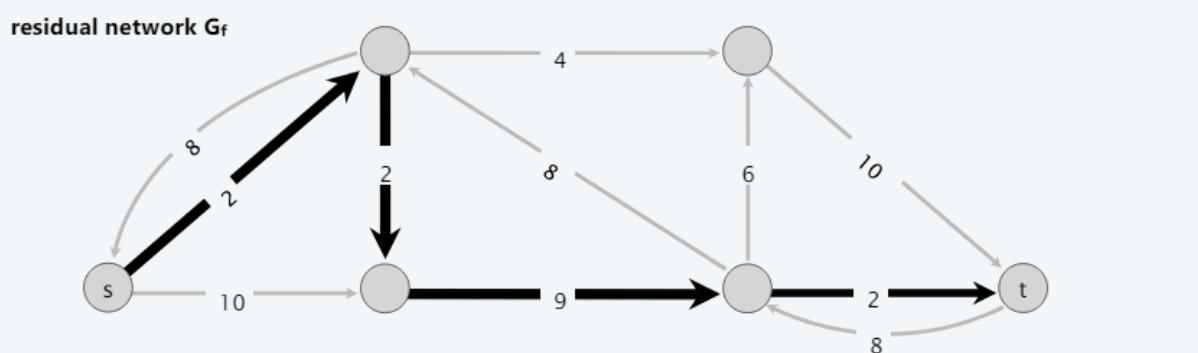
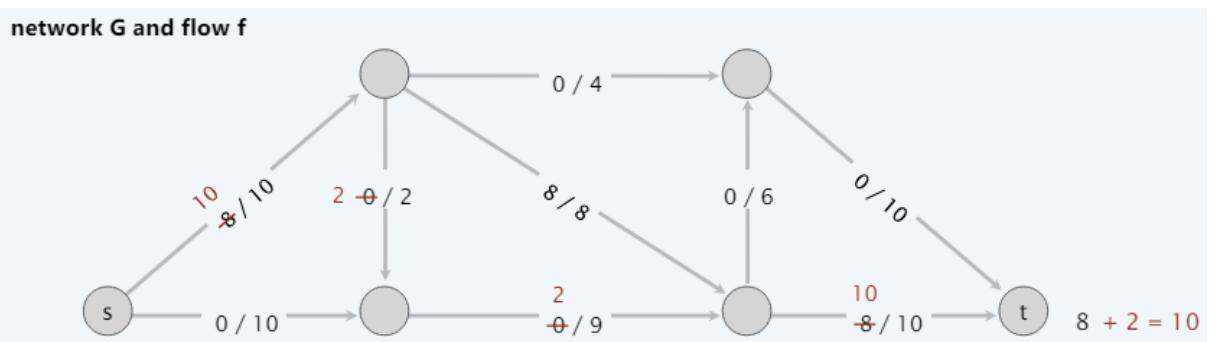
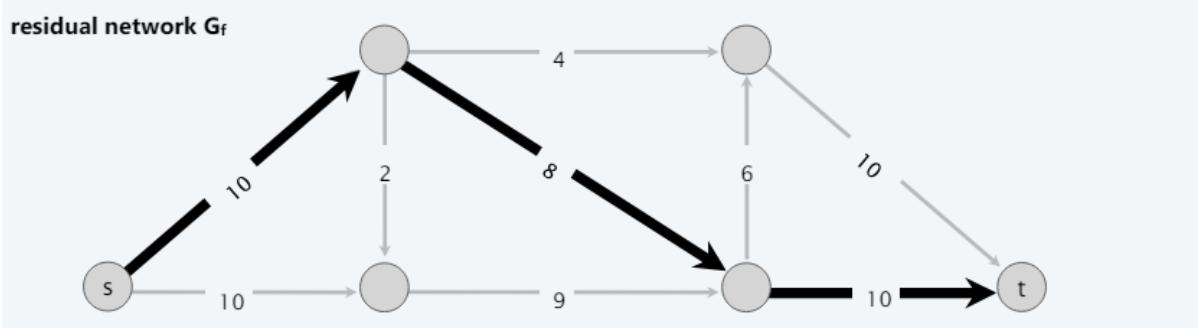
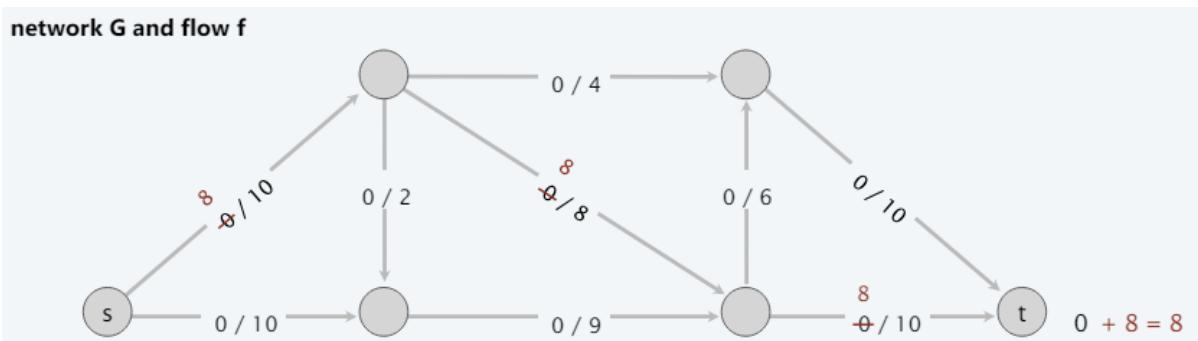
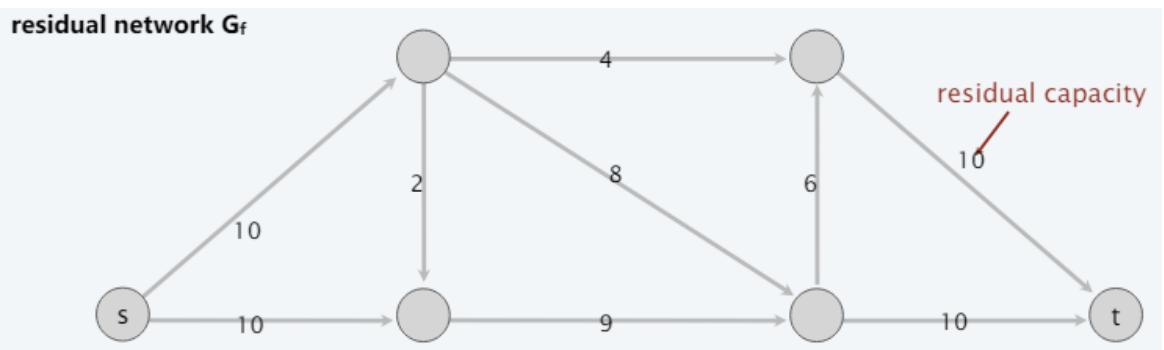
Update G_f .

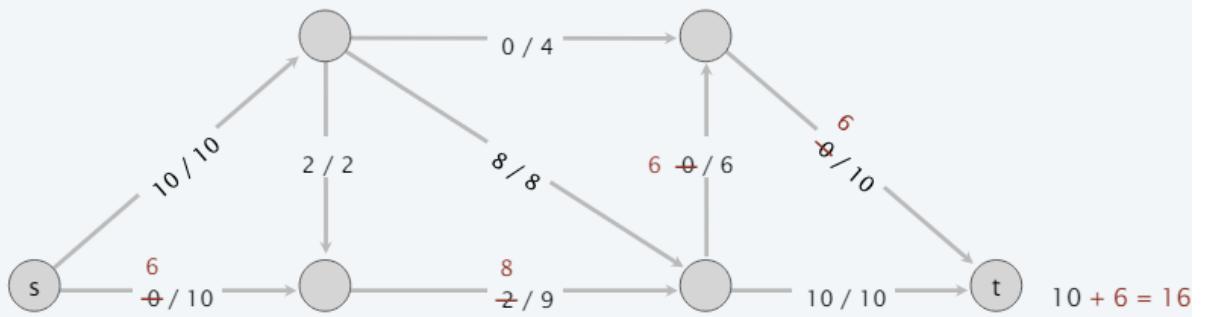
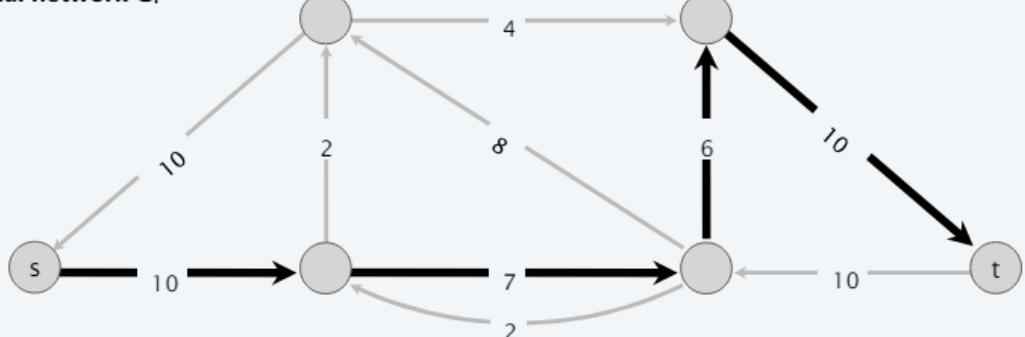
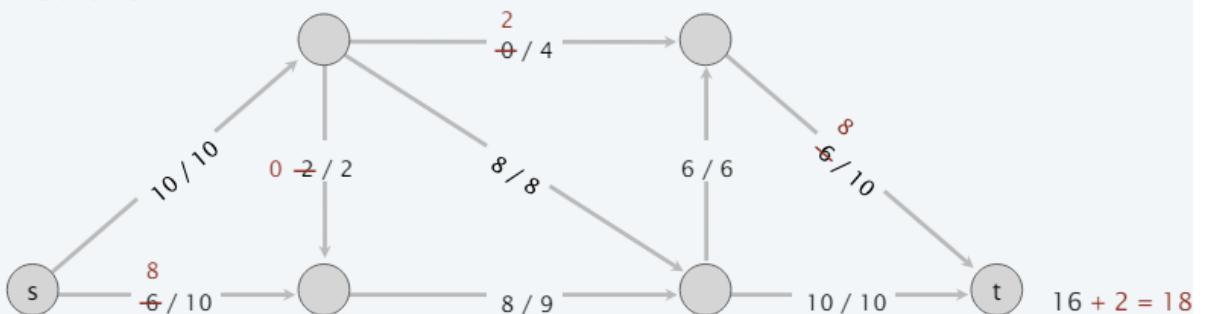
augmenting path

RETURN f .

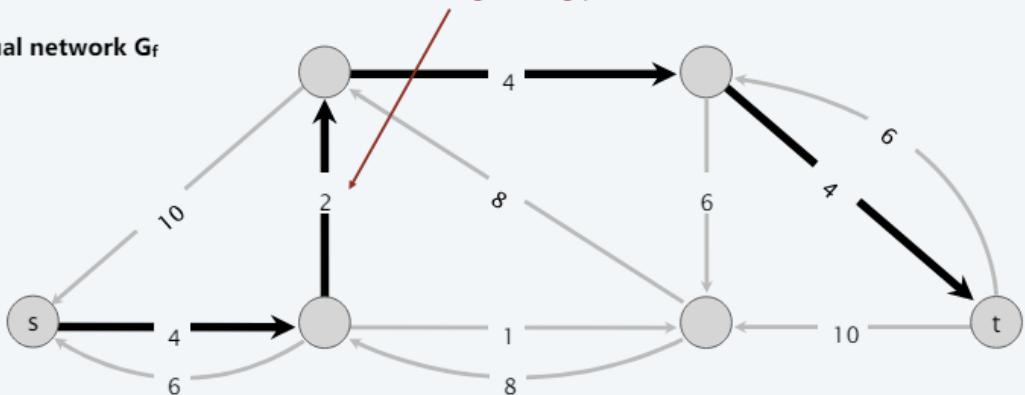
DEMO:

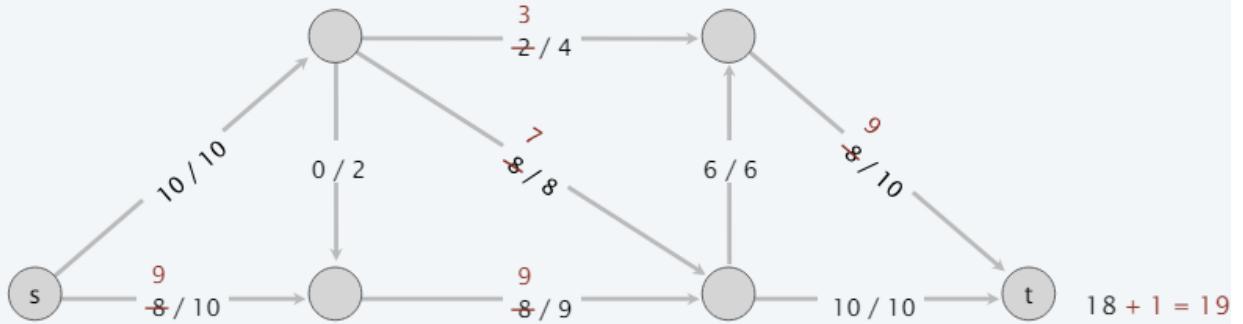
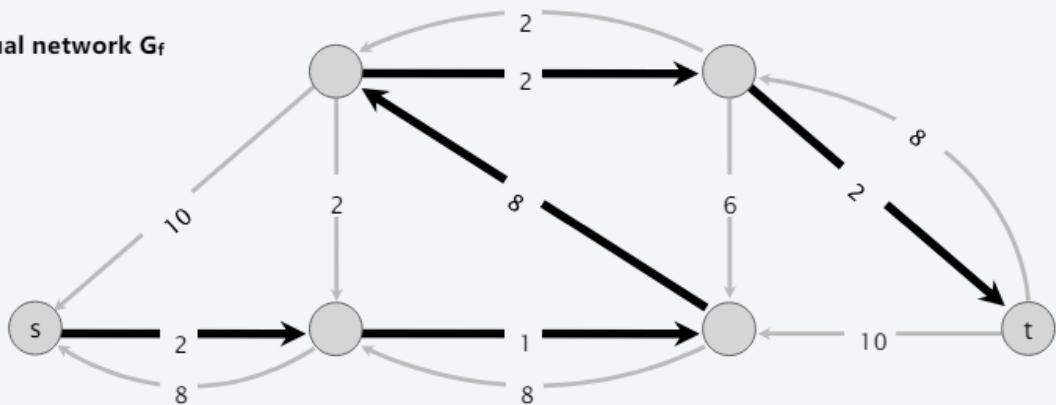
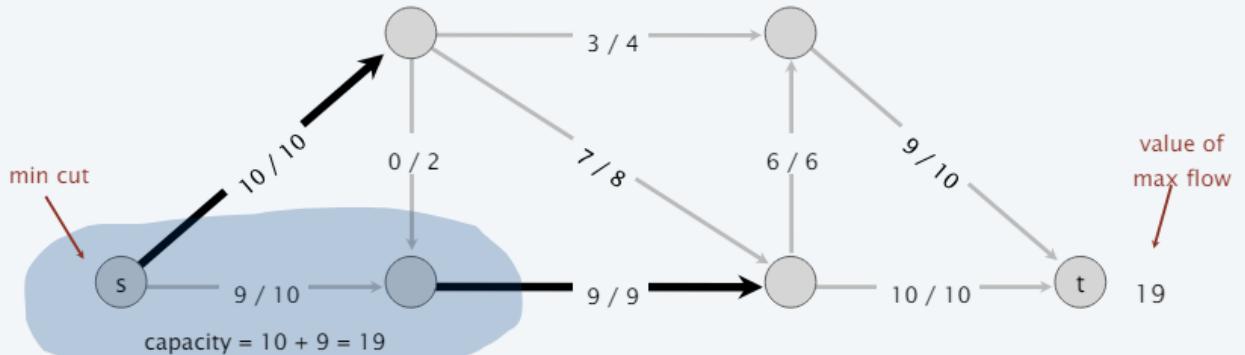
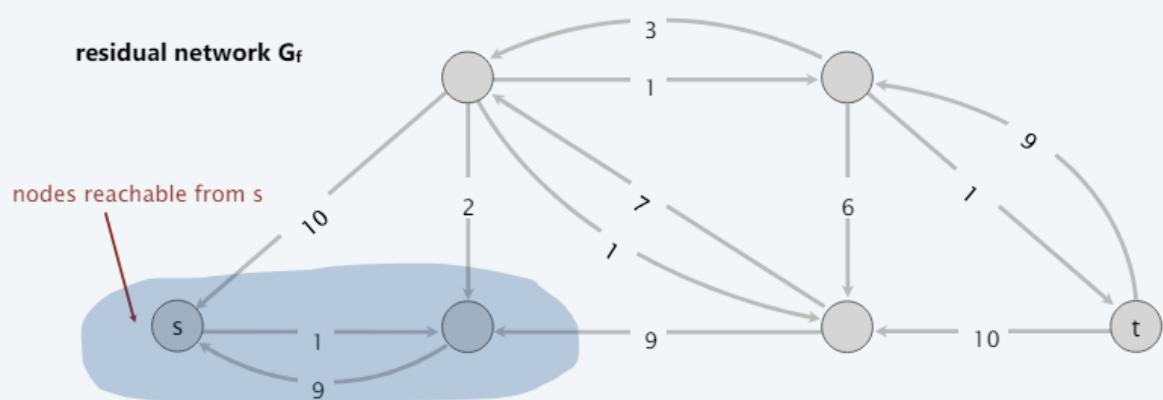




network G and flow f**residual network G_f** **network G and flow f**

fixes mistake from
second augmenting path

residual network G_f 

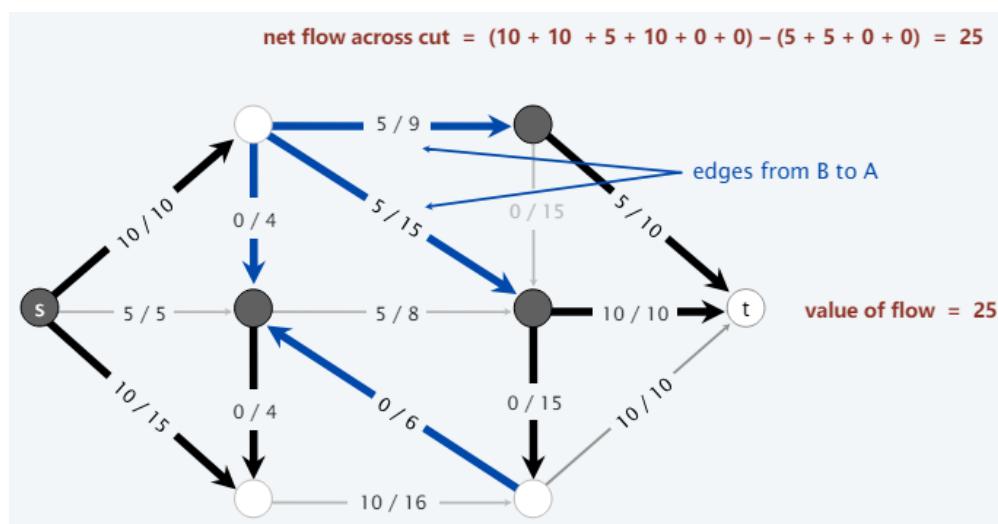
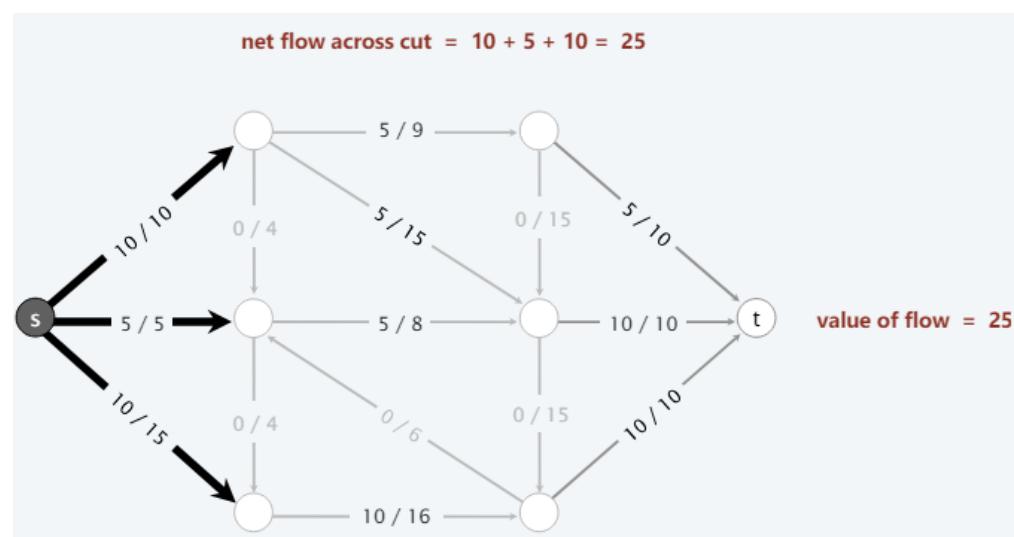
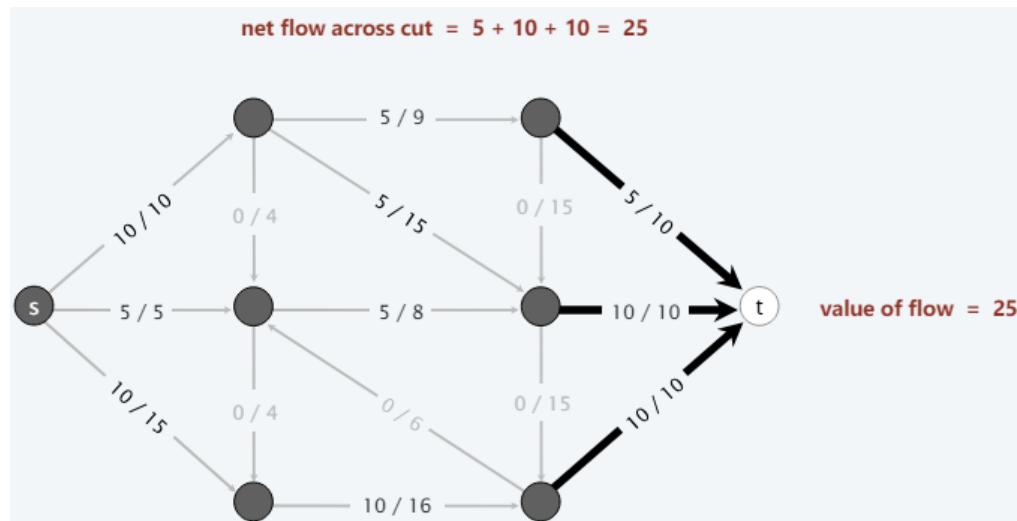
network G and flow f**residual network G_f** **network G and flow f****residual network G_f** 

LEZ14 – 23/04/2024 (max-flow min-cut theorem, choosing good augmenting paths)

RELAZIONE TRA FLOW(FLUSSO) E CUT(TAGLIO)

Sia f un qualsiasi flusso e sia (A, B) un qualsiasi taglio. Allora, il valore del flusso f è uguale al flusso netto che attraversa il taglio (A, B) .

$$val(f) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e)$$



Dim:

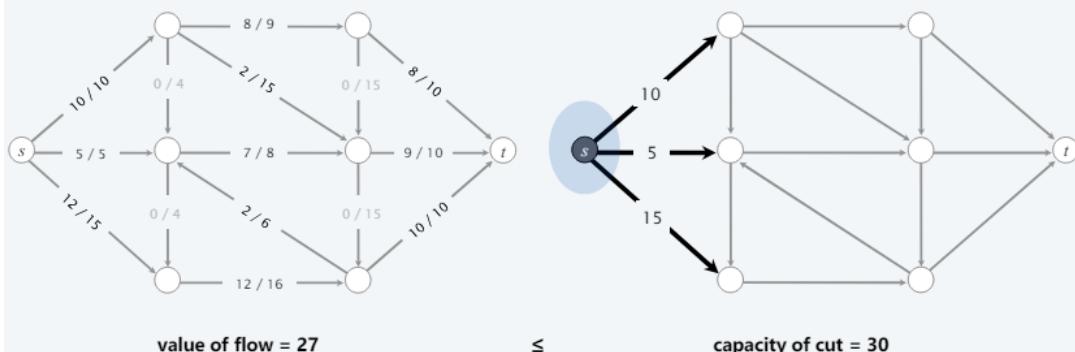
$$\begin{aligned}
 val(f) &= \sum_{e \text{ out of } s} f(e) - \sum_{e \text{ in to } s} f(e) \\
 \text{by flow conservation, all terms except for } v = s \text{ are 0} \quad \longrightarrow \quad &= \sum_{v \in A} \left(\sum_{e \text{ out of } v} f(e) - \sum_{e \text{ in to } v} f(e) \right) \\
 &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) .
 \end{aligned}$$

DUALITA' DEBOLE

Sia f un qualsiasi flusso e sia (A, B) un qualsiasi taglio. Allora $val(f) \leq cap(A, B)$.

Dim:

$$\begin{aligned}
 val(f) &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) \\
 &\stackrel{\text{flow value lemma}}{\leq} \sum_{e \text{ out of } A} f(e) \\
 &\leq \sum_{e \text{ out of } A} c(e) \\
 &= cap(A, B) \quad \blacksquare
 \end{aligned}$$



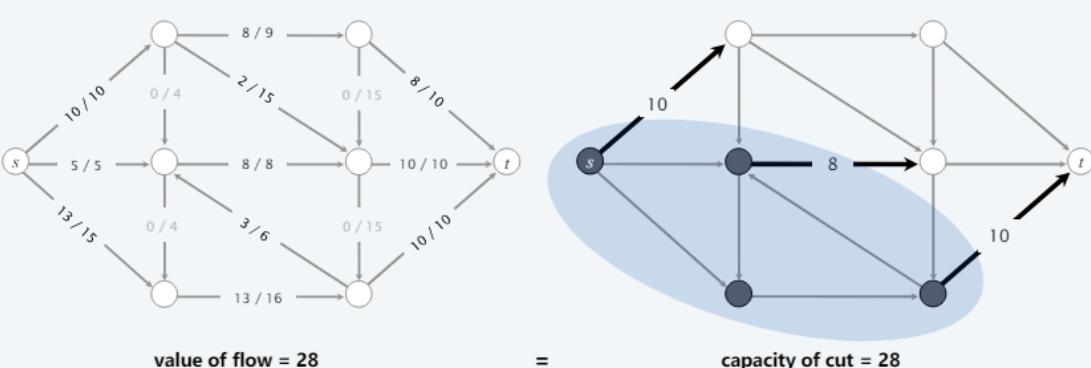
COROLLARIO

Sia f un flusso e sia (A, B) un qualsiasi taglio.

Se $val(f) = cap(A, B)$ allora f è un max flow e (A, B) è un min cut.

Pf.

- For any flow f' : $val(f') \leq cap(A, B) = val(f)$. weak duality
- For any cut (A', B') : $cap(A', B') \geq val(f) = cap(A, B)$. weak duality weak duality



MAX-FLOW MIN-CUT THEOREM

Teorema: valore di max flow = capacità di min cut

(dualità forte)

Augmenting path theorem = un flusso f è un max flow se e solo se non ci sono augmenting path (percorsi aumentanti).

Pf. The following three conditions are equivalent for any flow f :

- There exists a cut (A, B) such that $\text{cap}(A, B) = \text{val}(f)$.
- f is a max flow.
- There is no augmenting path with respect to f . if Ford-Fulkerson terminates, then f is max flow

[i \Rightarrow ii]

- This is the weak duality corollary.

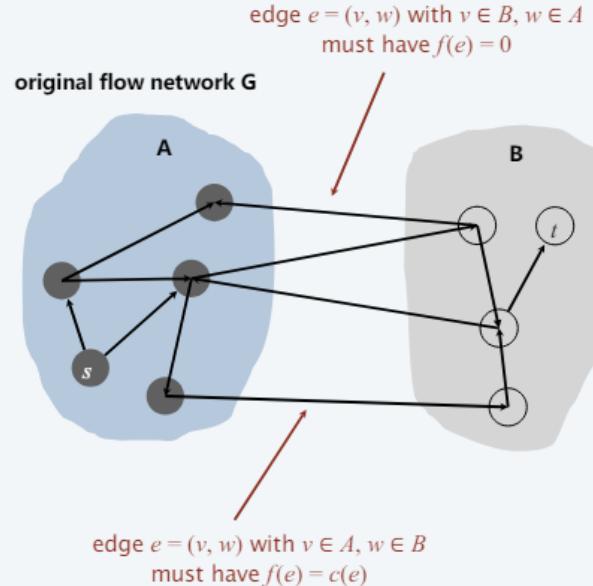
[ii \Rightarrow iii] We prove contrapositive: \neg iii \Rightarrow \neg ii.

- Suppose that there is an augmenting path with respect to f .
- Can improve flow f by sending flow along this path.
- Thus, f is not a max flow.

[iii \Rightarrow i]

- Let f be a flow with no augmenting paths.
- Let A = set of nodes reachable from s in residual network G_f .
- By definition of A : $s \in A$.
- By definition of flow f : $t \notin A$.

$$\begin{aligned} \text{val}(f) &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) \\ \text{flow value lemma} &= \sum_{e \text{ out of } A} c(e) - 0 \\ &= \text{cap}(A, B) \quad \blacksquare \end{aligned}$$

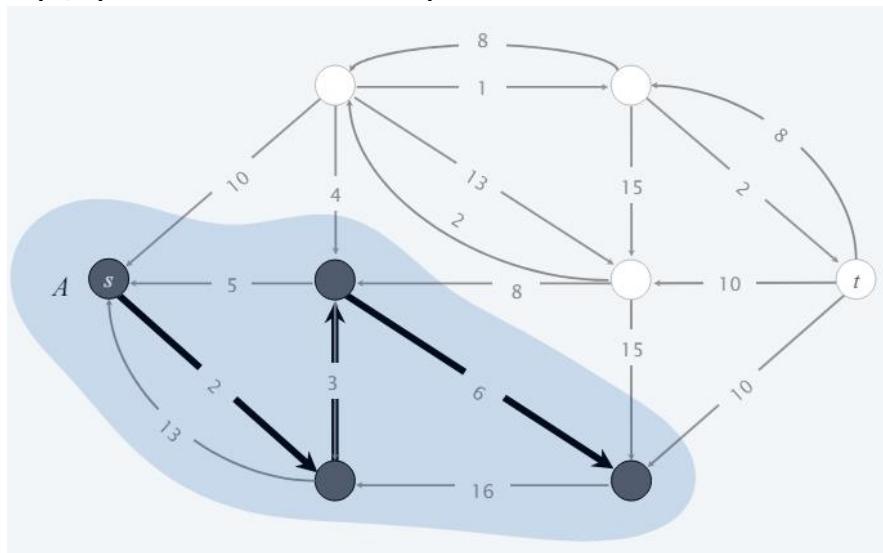


CALCOLARE UN MIN CUT A PARTIRE DA UN MAX FLOW

Teorema: Dato un qualsiasi max flow f , si può calcolare un min cut (A, B) in tempo $O(m)$.

Dim. A =insieme di nodi raggiungibili da s nella rete residua G_f

(capacità di (A,B) = valore del flusso f)



ANALISI DELL'ALGORITMO FORD-FULKERSON (PER CAPACITA' INTERE)

Assunzione: Le capacità degli archi $c(e)$ sono un intero con valore tra 1 e C .

Integrality invariant: durante Ford-Fulkerson ogni flusso di arco $f(e)$ e ogni capacità residua $c_f(e)$ sono valori interi.

Teorema: Ford-Fulkerson termina dopo al più $\text{val}(f^*) \leq nC$ aumenti dei percorsi, dove f^* è un max flow.

Dim: Ogni aumento aumenta il valore del flusso di almeno 1.

Corollario: Il tempo d'esecuzione di Ford-Fulkerson è $O(m \text{ val}(f^*)) = O(mnC)$.

Dim: Si può utilizzare BFS o DFS per trovare un percorso aumentante in tempo $O(m)$.

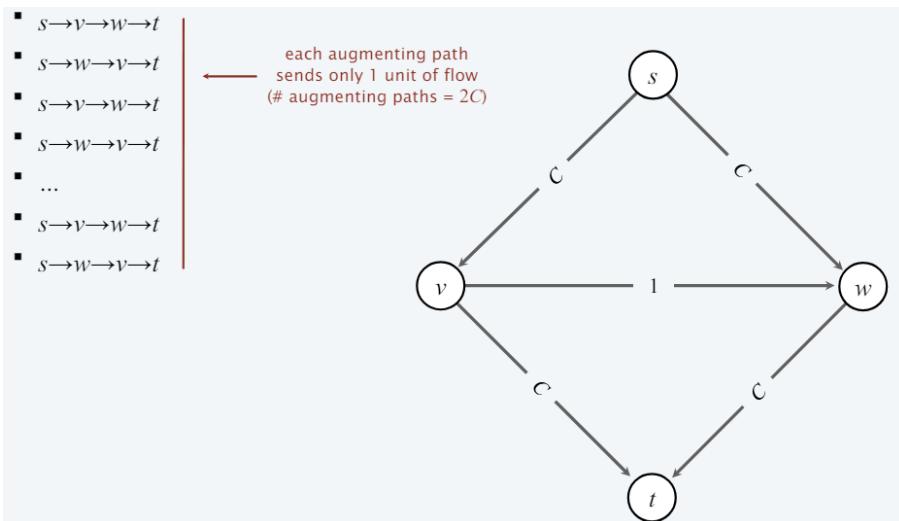
Integrality theorem: esiste un max flow integrale f^* .

Dim: Poiché Ford-Fulkerson termina, il teorema segue dall' Integrality invariant (e il teorema dell'augmenting path).

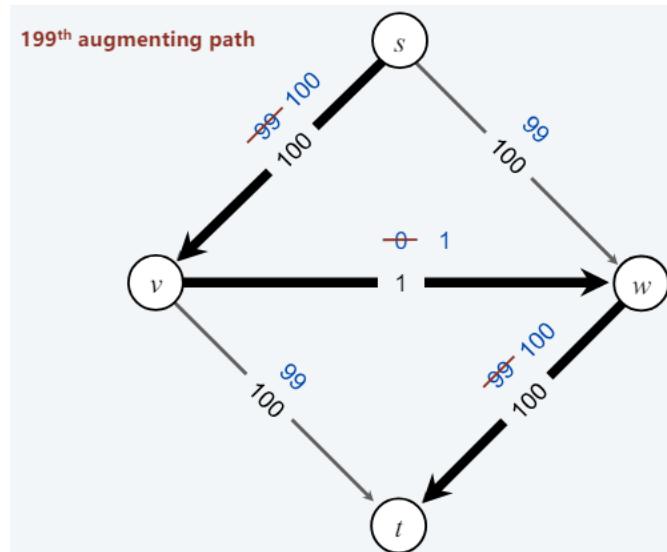
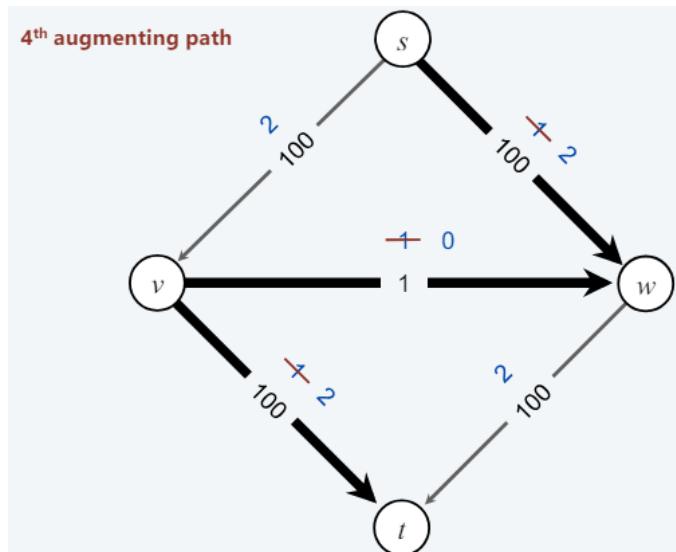
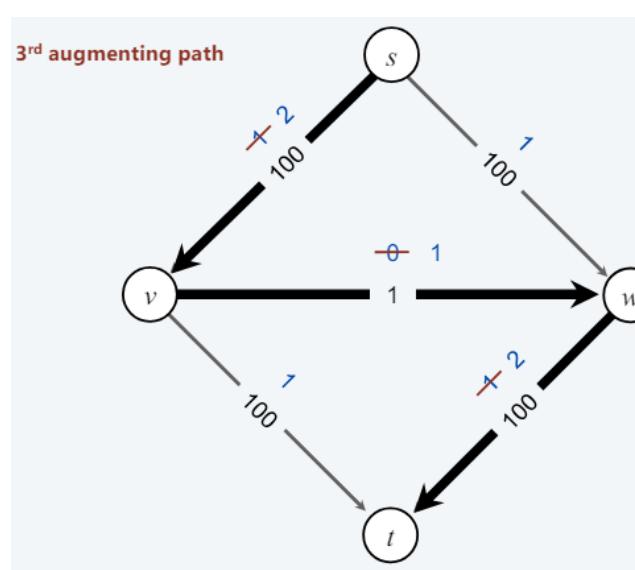
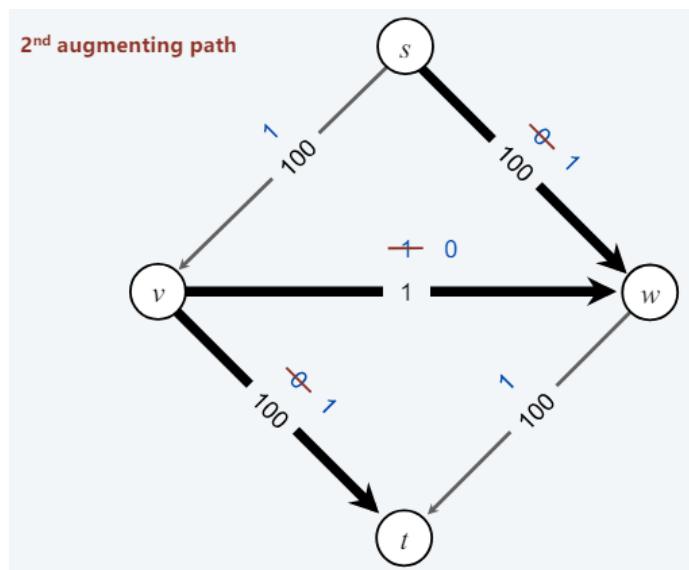
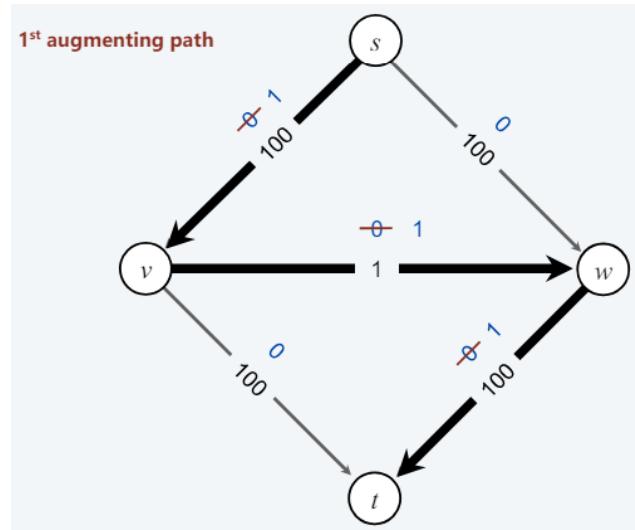
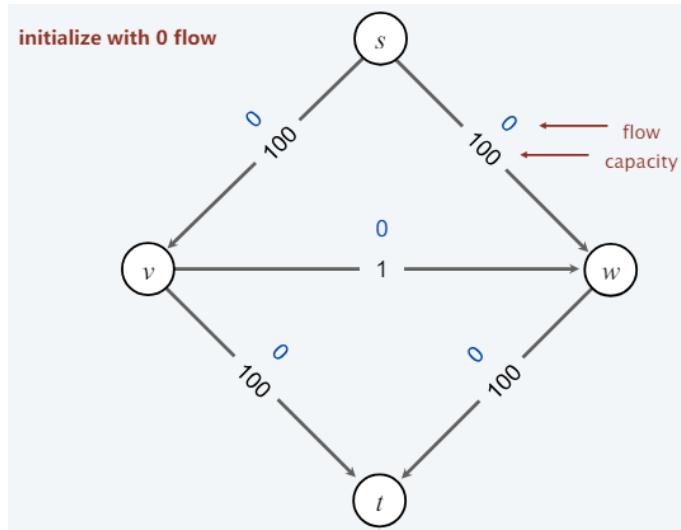
ESEMPIO ESPONENZIALE ALGORITMO FORD-FULKERSON

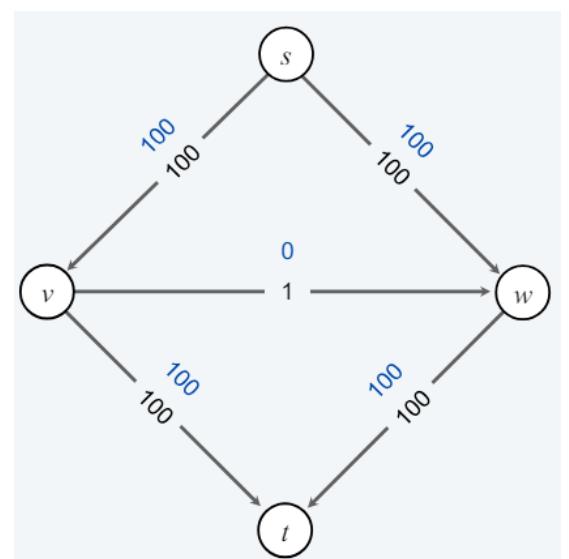
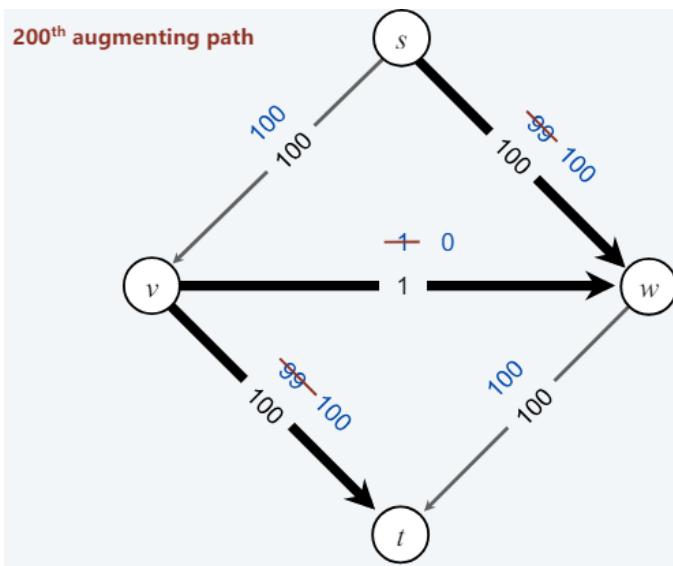
L'algoritmo è polinomiale nella dimensione dell'input(m, n e $\log C$)?

No, è pseudo-polinomiale. Se la massima capacità è C , allora l'algoritmo può impiegare un numero di iterazioni $\geq C$.



NOTA: il numero di augmenting paths può essere esponenziale nella dimensione dell'input.





SCELTA DI BUONI PERCORSI AUMENTANTI (AUGMENTING PATHS)

Usare cautela nella selezione dei percorsi aumentanti

- Alcune scelte portano ad algoritmi esponenziali.
- Scelte intelligenti portano ad algoritmi polinomiali.

Patologia. Quando le capacità degli archi possono essere irrazionali, nessuna garanzia che Ford-Fulkerson termina (o converge a un flusso massimo)!

Obiettivo. Scegliere percorsi aumentanti in modo che:

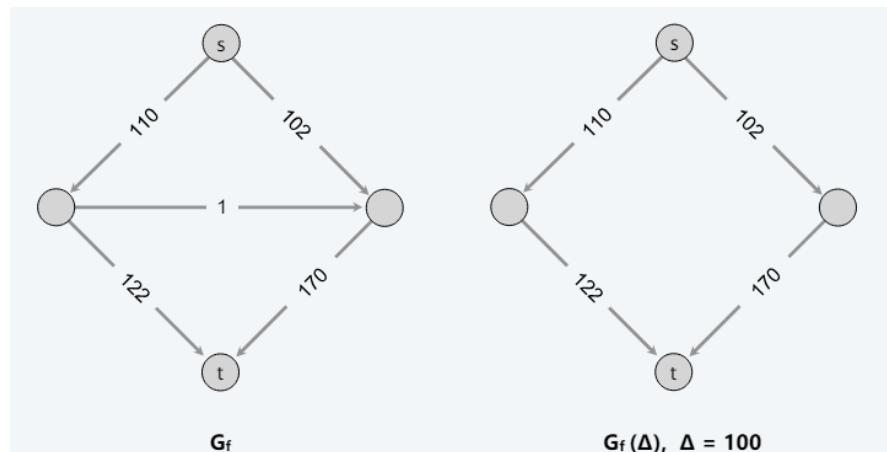
- Si possano trovare percorsi aumentanti in modo efficiente.
- Ci siano poche iterazioni.

Scegliere percorsi aumentanti con bottleneck capacity sufficientemente larga e con minor numero di archi.

CAPACITY-SCALING ALGORITHM(ALGORITMO DI RIDIMENTIONAMENTO DELLE CAPACITA')

Panoramica: Scelta di percorsi aumentanti con bottleneck capacity "grande".

- Mantenere il parametro di ridimensionamento Δ .
- Sia $G_f(\Delta)$ la parte della rete residua contenente solo gli archi con capacità $\geq \Delta$.
- Qualsiasi percorso aumentante (augmenting path) in $G_f(\Delta)$ ha bottleneck capacity $\geq \Delta$



Algoritmo capacity-scaling:

CAPACITY-SCALING(G)

FOREACH edge $e \in E : f(e) \leftarrow 0.$

$\Delta \leftarrow$ largest power of $2 \leq C.$

WHILE ($\Delta \geq 1$)

$G_f(\Delta) \leftarrow \Delta$ -residual network of G with respect to flow $f.$

WHILE (there exists an $s \rightsquigarrow t$ path P in $G_f(\Delta)$)

$f \leftarrow \text{AUGMENT}(f, c, P).$

Update $G_f(\Delta).$

$\Delta \leftarrow \Delta / 2.$

Δ -scaling phase

RETURN $f.$



Si può dimostrare quanto segue:

- Lemma 1 : ci sono $1 + \lceil \log_2 C \rceil$ scaling phases
- Lemma 2 : ci sono $\leq 2m$ aumenti per scaling phase, allora il numero totale di aumenti è $O(m \log C)$
- Teorema : l'algoritmo capacity-scaling impiega tempo $O(m^2 \log C).$

SHORTEST AUGMENTING PATH (PERCORSO AUMENTANTE PIU' BREVE)

Come scegliere il prossimo augmenting path in Ford-Fulkerson? Scegliendo quello che usa il minor numero di archi (si può fare tramite BFS).

SHORTEST-AUGMENTING-PATH(G)

FOREACH $e \in E : f(e) \leftarrow 0.$

$G_f \leftarrow$ residual network of G with respect to flow $f.$

WHILE (there exists an $s \rightsquigarrow t$ path in G_f)

$P \leftarrow \text{BREADTH-FIRST-SEARCH}(\mathcal{G}_f).$

$f \leftarrow \text{AUGMENT}(f, c, P).$

Update $G_f.$

RETURN $f.$

Si può dimostrare quanto segue:

- Lemma 1 : Il numero totale di aumenti è al più m per n
- Teorema: l'algoritmo shortest-augmenting-path impiega tempo $O(m^2 n).$

SOMMARIO ALGORITMI AUGMENTING-PATH

year	method	# augmentations	running time
1955	augmenting path	$n C$	$O(m n C)$
1972	fattest path	$m \log(mC)$	$O(m^2 \log n \log(mC))$
1972	capacity scaling	$m \log C$	$O(m^2 \log C)$
1985	improved capacity scaling	$m \log C$	$O(m n \log C)$
1970	shortest augmenting path	$m n$	$O(m^2 n)$
1970	level graph	$m n$	$O(m n^2)$
1983	dynamic trees	$m n$	$O(m n \log n)$

augmenting-path algorithms with m edges, n nodes, and integer capacities between 1 and C

ALGORITMI MAXIMUM-FLOW HIGHLIGHTS

year	method	worst case	discovered by
1951	simplex	$O(m n^2 C)$	Dantzig
1955	augmenting paths	$O(m n C)$	Ford–Fulkerson
1970	shortest augmenting paths	$O(m n^2)$	Edmonds–Karp, Dinitz
1974	blocking flows	$O(n^3)$	Karzanov
1983	dynamic trees	$O(m n \log n)$	Sleator–Tarjan
1985	improved capacity scaling	$O(m n \log C)$	Gabow
1988	push–relabel	$O(m n \log(n^2/m))$	Goldberg–Tarjan
1998	binary blocking flows	$O(m^{3/2} \log(n^2/m) \log C)$	Goldberg–Rao
2013	compact networks	$O(m n)$	Orlin
2014	interior-point methods	$\tilde{O}(m n^{1/2} \log C)$	Lee–Sidford
2016	electrical flows	$\tilde{O}(m^{10/7} C^{1/7})$	Mądry
20xx		???	

LEZ15 – 30/04/2024 (Network Flow: 4 esempi)

Sono numerosissime le applicazioni di max-flow e min-cut. Vediamo in questa lezione i 4 esempi seguenti:

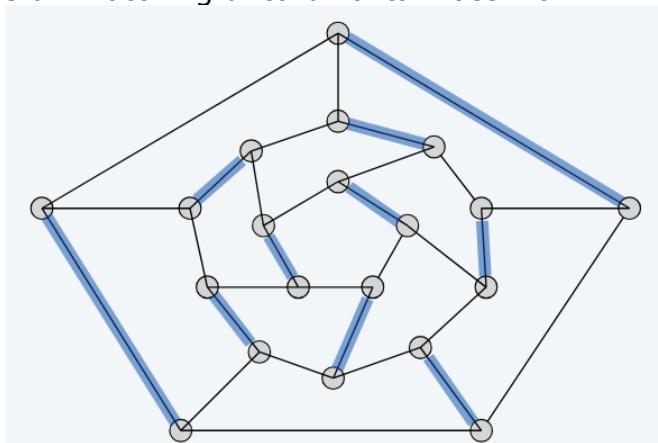
1. matching su grafi bipartiti,
2. cammini arco-disgiunti,
3. segmentazione d'immagini,
4. baseball elimination.

1. BIPARTIRE MATCHING (MATCHING SU GRAFI BIPARTITI)

Def(matching): dato un grafo non diretto $G=(V,E)$, un sottoinsieme di archi $M \subseteq E$ è un matching se ogni nodo appare in al più un arco in M .

MAX MATCHING

Dato un grafo G , trovare un matching di cardinalità massima.

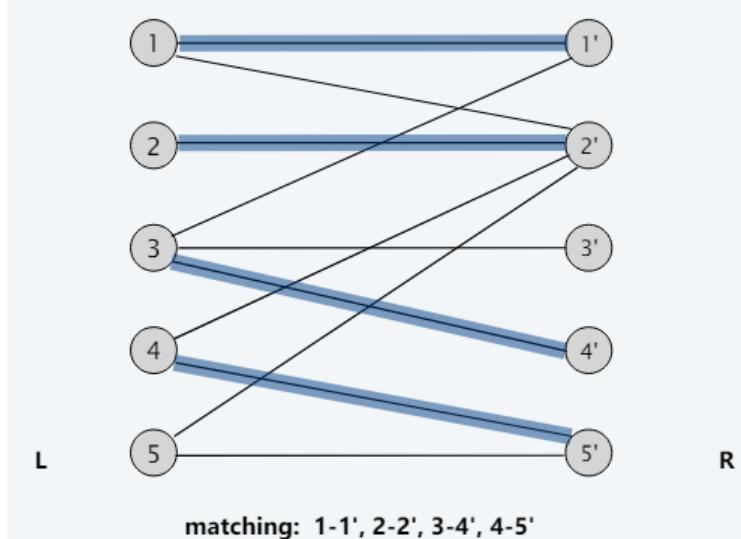


GRAFO BIPARTITO

Un grafo G è bipartito se i nodi possono essere partizionati in due sottoinsiemi L e R in cui tutti gli archi connettono un nodo in L con un nodo in R .

MATCHING BIPARTITO

Dato un grafo bipartito $G=(L \cup R, E)$ trovare un matching di cardinalità massima.

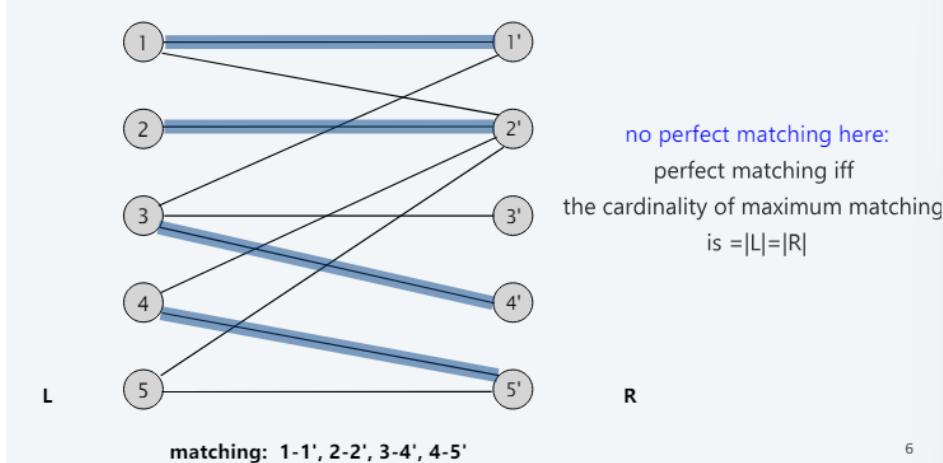


PERFECT MATCHING (IN GRAFI BIPARTITI)

Dato un grafo $G=(V,E)$, un sottoinsieme di archi $M \subseteq E$ è un perfect matching se ogni nodo appare in esattamente un arco in M .

PERFECT MATCHING PROBLEM

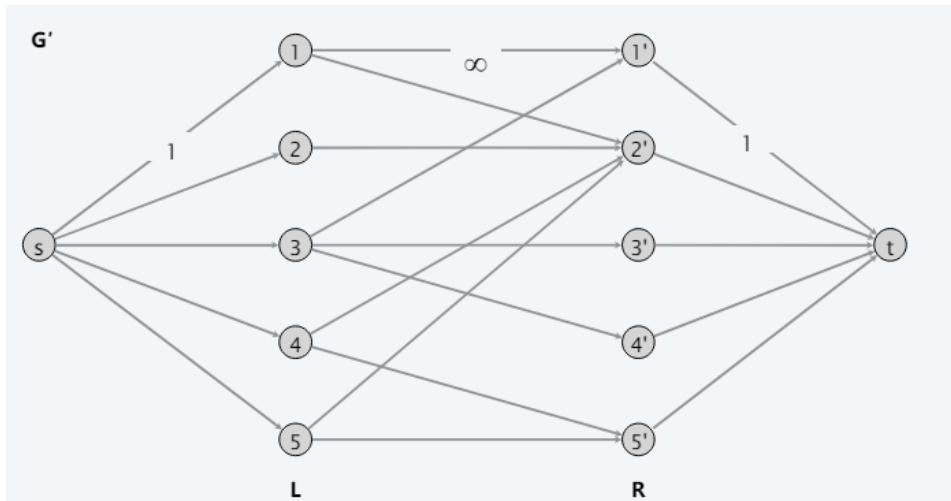
Dato un grafo bipartito $G=(L \cup R, E)$ trovare un perfect matching o riportare che non esiste.



MATCHING BIPARTITO: FORMULAZIONE MAX-FLOW

Formulazione:

- Creazione digrafo $G'=(L \cup R \cup \{s, t\}, E')$.
- Direzionare tutti gli archi da L a R e assegnare capacità (o unità) infinita
- Aggiungere capacità (o unità) agli archi da s a ogni nodo in L
- Aggiungere capacità (o unità) da ogni nodo in R a t



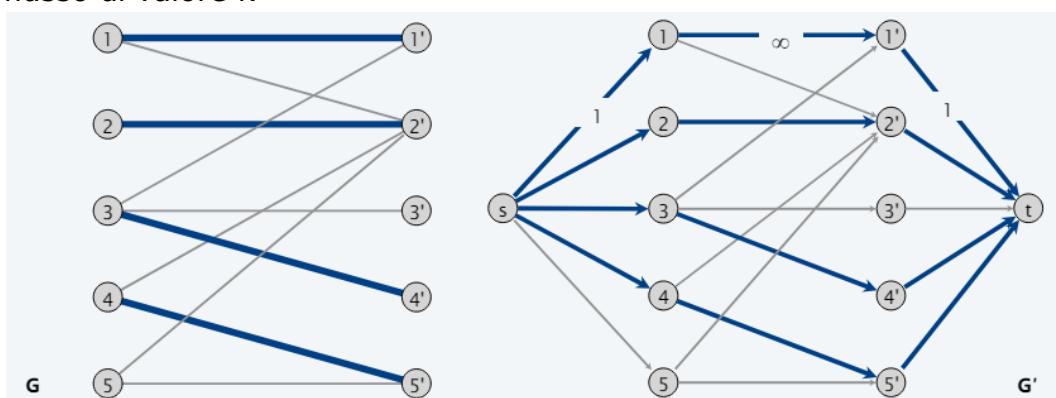
MAX-FLOW FORMULATION: PROVA DI CORRETTEZZA

Teorema: C'è una corrispondenza di 1-1 tra matchings di cardinalità k in G e flussi interi di valore k in G' .

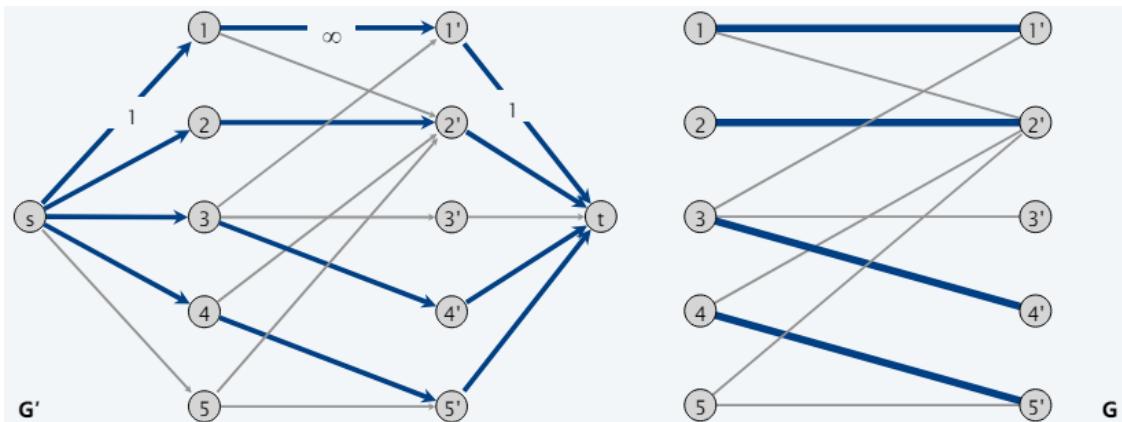
for each edge $e: f(e) \in \{0, 1\}$

Dim teorema:

⇒ sia M un matching in G di cardinalità k
Considerare il flusso f che invia 1 unità in ognuno dei k path corrispondenti
 f è un flusso di valore k



- ⇐ Sia f un flusso intero in G' di valore k
 Considerare M =insieme di archi da L a R con $f(e)=1$
 - Ogni nodo in L e R partecipa in al più un arco in M
 - $|M|=k$: applicare il lemma flow-value per tagliare($L \cup \{s\}$, $R \cup \{t\}$)



COROLLARIO

Si può risolvere il problema del matching bipartito tramite una opportuna formulazione di max-flow.

Dim: integrality theorem \Rightarrow esiste un max-flow f^* in G' che è intero
 corrispondenza 1-1 $\Rightarrow f^*$ corrisponde al matching di cardinalità massima

RUNNING TIME MAX-FLOW FORMULATION

Usando Ford-Fulkerson $\leq n$ augmentations \Rightarrow tempo $O(m n)$.

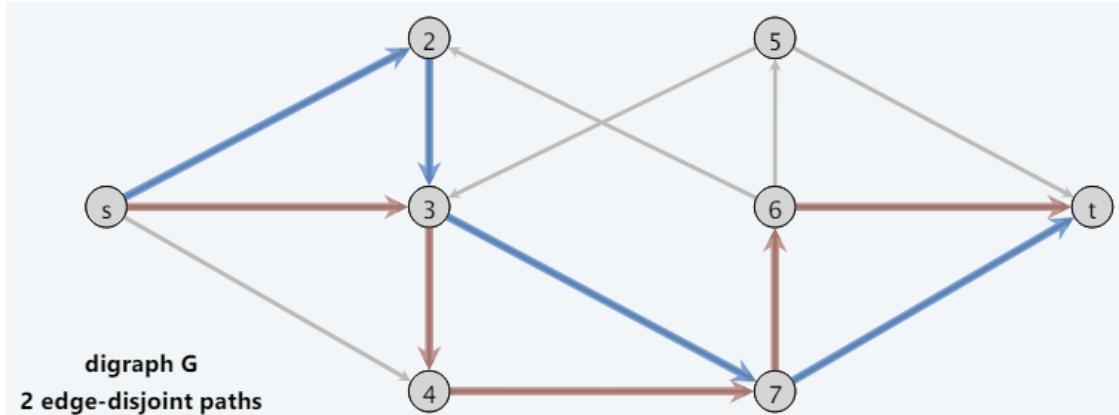
2. DISJOINT PATHS (CAMMINI ARCO-DISGIUNTI)

Def: Due cammini sono edge-disjoint se non hanno archi in comune.

EDGE DISJOINT PATHS PROBLEM

Dato un digrafo $G=(V,E)$ e due nodi s e t , trovare il massimo numero di archi disgiunti (edge-disjoint) nel cammino da s a t .

Esempio: Communication networks



MAX-FLOW FORMULAZIONE

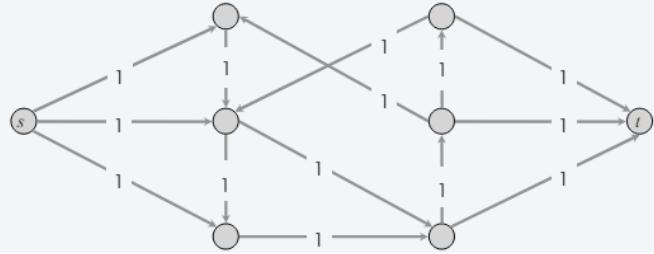
Assegnare unit capacity a tutti gli archi.

Teorema: corrispondenza 1-1 tra i k archi disgiunti nel cammino da s a t in G e flussi interi di valore k in G' .

Dim thm:

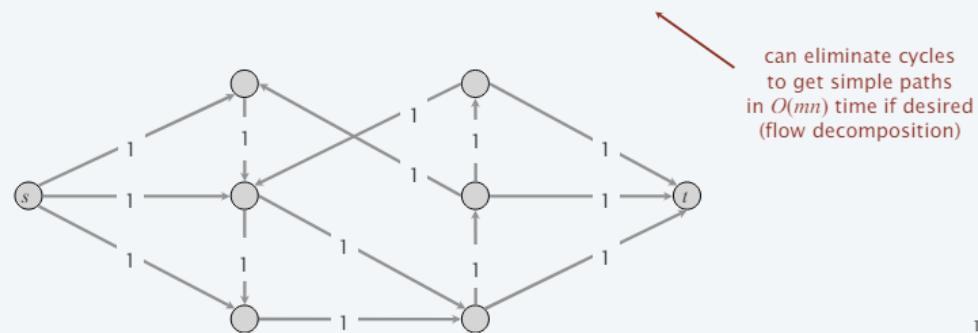
Pf. \Rightarrow

- Let P_1, \dots, P_k be k edge-disjoint $s \rightsquigarrow t$ paths in G .
- Set $f(e) = \begin{cases} 1 & \text{edge } e \text{ participates in some path } P_j \\ 0 & \text{otherwise} \end{cases}$
- Since paths are edge-disjoint, f is a flow of value k .



Pf. \Leftarrow

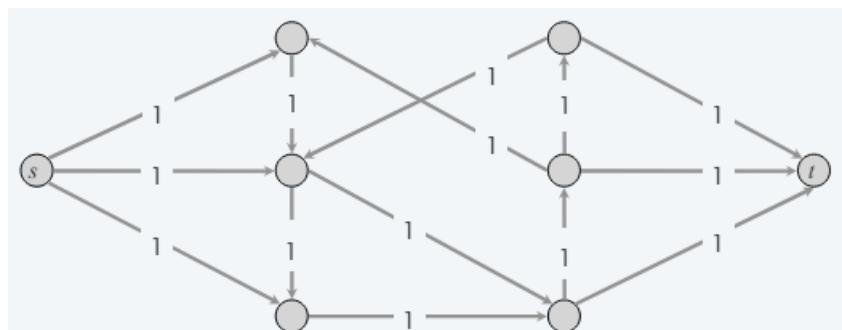
- Let f be an integral flow in G' of value k .
- Consider edge (s, u) with $f(s, u) = 1$.
 - by flow conservation, there exists an edge (u, v) with $f(u, v) = 1$
 - continue until reach t , always choosing a new edge
- Produces k (not necessarily simple) edge-disjoint paths.



COROLLARIO

Si può risolvere l' edge-disjoint paths problem tramite formulazione del max-flow.

Dim: integrality theorem \Rightarrow esiste un max flow f^* in G' che è intero
corrispondenza 1-1 $\Rightarrow f^*$ corrisponde al numero massimo di archi disgiunti nel cammino da s a t in G .



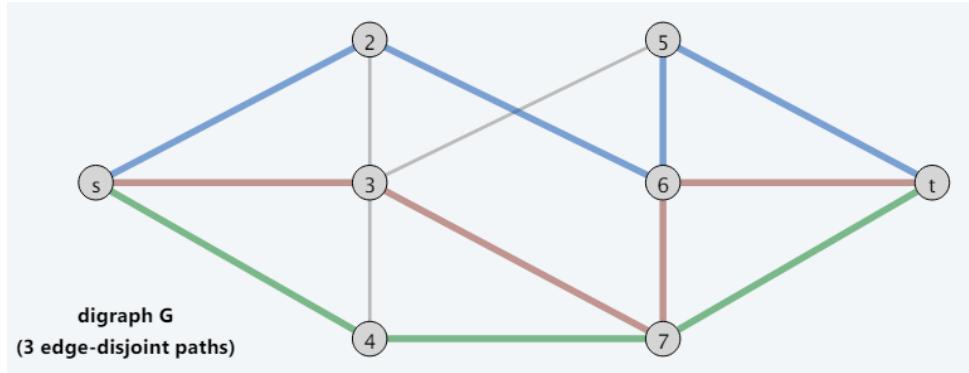
RUNNING TIME

Usando Ford-Fulkerson con $\leq n$ augmentations \Rightarrow tempo $O(m n)$

EDGE-DISJOINT PATHS IN UNIDIRECTED GRAPHS

Def: due percorsi sono archi disgiunti se non hanno archi in comune.

Edge-disjoint paths problem in unidirected graphs: dato un grafo $G=(V,E)$ e due nodi s e t , trovare il numero massimo di archi disgiunti nel cammino da s a t .



3. IMAGE SEGMENTATION

Image segmentation: dividere immagine in regioni coerenti, è un problema centrale nell'immagine processing,

Esempio: separare soggetto dal background in una foto.

FOREGROUND/BACKGROUND SEGMENTATION

- Etichetta ogni pixel nell'immagine come appartenente a primo piano o sfondo.
- V = set di pixel, E = coppie di pixel vicini.
- $a_i \geq 0$ è la probabilità che il pixel i sia in primo piano.
- $b_i \geq 0$ è la probabilità che il pixel i sia in background.
- $p_{ij} \geq 0$ è un valore di penalità in caso di separazione per l'etichettatura di uno tra i e j come primo piano e l'altro come sfondo.

GOALS

- Precisione: se $a_i > b_i$ in isolamento, preferisco etichettare i in primo piano.
- Scorrivolezza: se molti vicini di i sono etichettati in primo piano, dovremmo essere inclini ad etichettare i come primo piano.
- Trovare una partizione (A, B) che massimizza:

$$\sum_{i \in A} a_i + \sum_{j \in B} b_j - \sum_{\substack{(i,j) \in E \\ |A \cap \{i,j\}|=1}} p_{ij}$$

foreground background

FORMULAZIONE IMAGE SEGMENTATION COME MIN-CUT PROBLEM

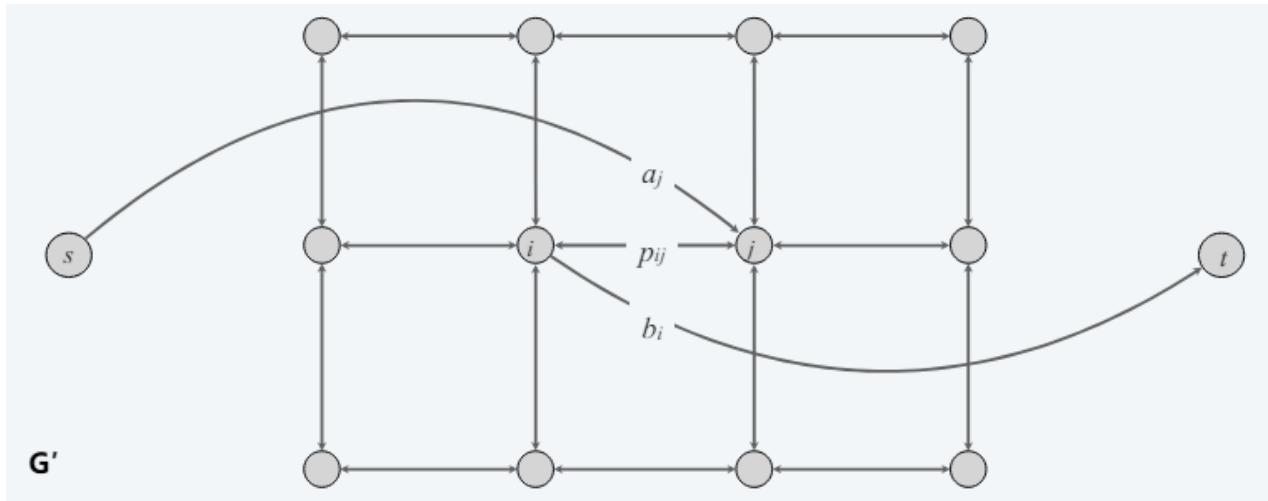
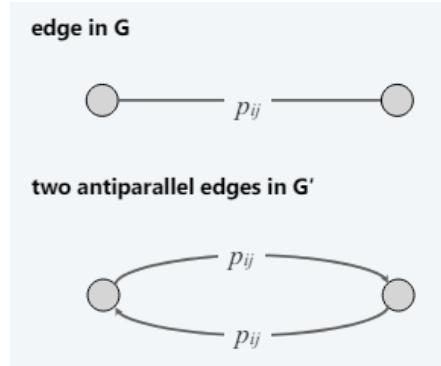
Formulare come min-cut problem: massimizzazione, nessuna fonte o punto di arrivo, grafo non diretto.

Lo facciamo diventare un problema di minimizzazione come segue:

- Maximizing $\sum_{i \in A} a_i + \sum_{j \in B} b_j - \sum_{\substack{(i,j) \in E \\ |A \cap \{i,j\}|=1}} p_{ij}$
- is equivalent to minimizing $\left(\sum_{i \in V} a_i + \sum_{j \in V} b_j \right) - \sum_{i \in A} a_i - \sum_{j \in B} b_j + \sum_{\substack{(i,j) \in E \\ |A \cap \{i,j\}|=1}} p_{ij}$
a constant
- or alternatively $\sum_{j \in B} a_j + \sum_{i \in A} b_i + \sum_{\substack{(i,j) \in E \\ |A \cap \{i,j\}|=1}} p_{ij}$

Formulazione come min-cut problem $G'=(V',E')$.

- Includere un nodo per ogni pixel
- Usare due archi antiparalleli invece di archi un arco non diretto
- Aggiunta sorgente s che corrisponde al primo piano
- Aggiunta sink t che corrisponde al background

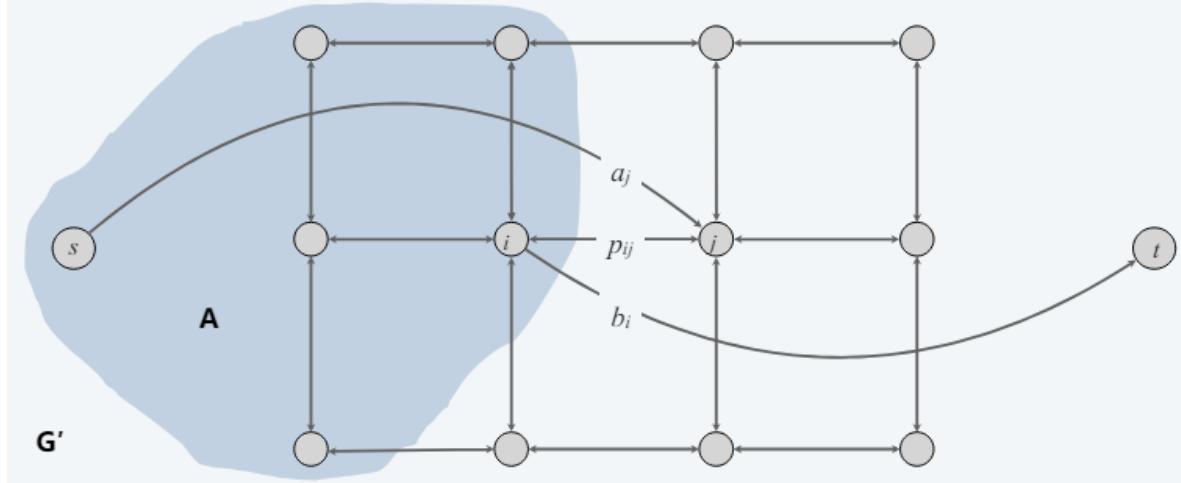


Considera min cut (A,B) in G'

- $A = \text{foreground}$.

$$cap(A, B) = \sum_{j \in B} a_j + \sum_{i \in A} b_i + \sum_{\substack{(i,j) \in E \\ i \in A, j \in B}} p_{ij} \quad \begin{matrix} \leftarrow & \text{if } i \text{ and } j \text{ on different sides,} \\ & p_{ij} \text{ counted exactly once} \end{matrix}$$

- Precisely the quantity we want to minimize.



4. BASEBALL ELIMINATION PROBLEM

Domanda: quale team ha la chance di finire la season con il maggior numero di vittorie?

i	team	wins	losses	to play	ATL	PHI	NYM	MON
0	Atlanta	83	71	8	-	1	6	1
1	Philly	80	79	3	1	-	0	2
2	New York	78	78	6	6	0	-	0
3	Montreal	77	82	3	1	2	0	-

Montreal è matematicamente eliminata, non può vincere il titolo.

Philadelphia è matematicamente eliminata, non può vincere il titolo.

- Può finire con un numero di vittorie minore o uguale a 83
- Sia New York che Atlanta possono finire con un numero di vittorie maggiore o uguale a 84

Osservazione: la risposta dipende non solo da quante partite si possono ancora vincere e mancano da giocare, ma anche da quelle delle avversarie.

Classifica attuale.

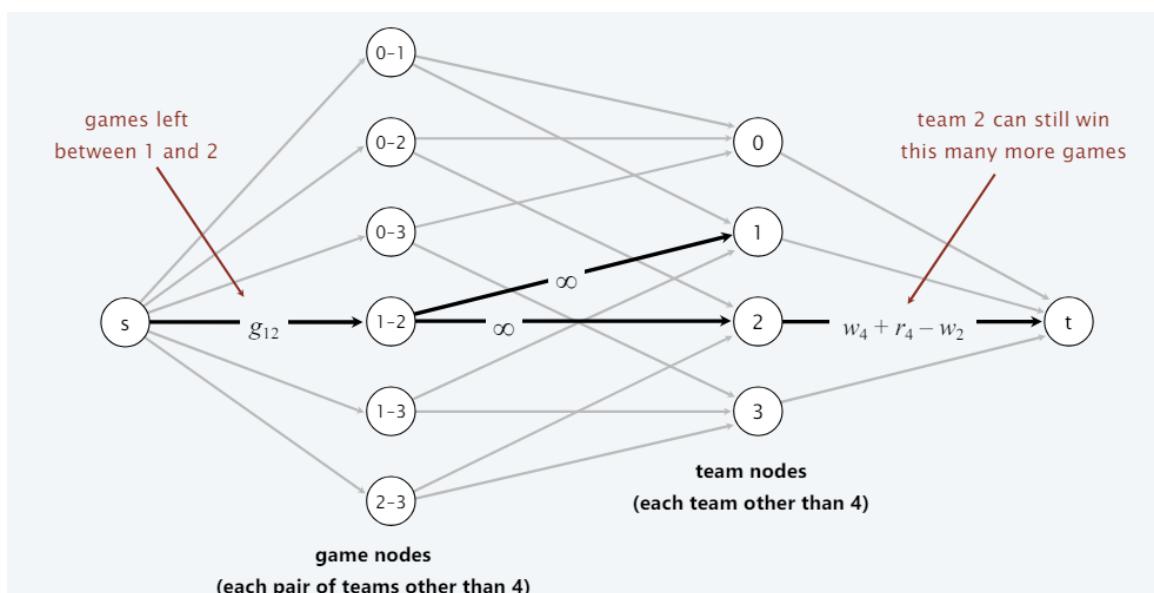
- Set di squadre S .
- Squadra distinta $z \in S$.
- Il Team x ha già vinto W_x partite.
- Le squadre x e y devono giocare tra loro ancora r_{xy} volte aggiuntive.

Baseball problema di eliminazione: Data la classifica attuale, esiste qualche risultato delle partite rimanenti per cui la squadra z finisce con il maggior numero di vittorie (o in parità)?

MAX-FLOW FORMULAZIONE

Può il team 4 finire con il maggior numero di vittorie rispetto agli altri team?

- assumere che il team 4 vinca tutte le partite restanti $\Rightarrow w_4 + R_4$ vittorie.
- combinare esiti partite restanti in modo che tutti i team abbiano un numero di vittorie che è $\leq w_4 + R_4$.



Teorema: team 4 non eliminato se e solo se max flow satura tutti gli archi tralasciando s.

Dim: Integrality theorem \Rightarrow ogni partita rimanente tra x e y aggiunta al numero di vittorie per il team x o il team y.

La capacità sugli archi (x,t) assicura che nessun team vinca troppe partite.

LEZ16 – 07/05/2024 (Esercitazione sui problemi di flusso)

NOTE ESERCITAZIONE (ELEMENTI RICORRENTI PER RISOLUZIONE PROBLEMI FLUSSO):

1. Della risorsa i-esima ho quantità b_i
2. Il concetto di risorsa inteso come unità di flusso
3. Costruisco grafo ausiliario e su di esso calcolo il max-flow
4. Quando non ci sono vincoli imposto la capacità ad infinito

PROBLEMA 1

Scenario:

- un insieme R di n risorse
- un insieme A di m attività da svolgere
- risorsa i
 - o è disponibile per b_i unità
 - o può essere assegnata solo al sottoinsieme $A_i \subseteq A$ delle attività (vincoli di compatibilità)
 - o attività j richiede almeno r_j risorse in totale per essere svolta

Goal: capire se si possono assegnare le risorse in modo da svolgere tutte le attività

Esempi:

- risorse= dipendenti
- attività= progetti dell'azienda
- compatibilità= specifica a che tipo di progetti può lavorare un certo dipendente
- risorse= sangue disponibile (di diverso tipo)
- attività= richieste di trasfusione/operazioni
- compatibilità= specifica quale tipo di sangue può essere usato per una specifica trasfusione/operazione

Soluzione:

grafo ausiliario G'

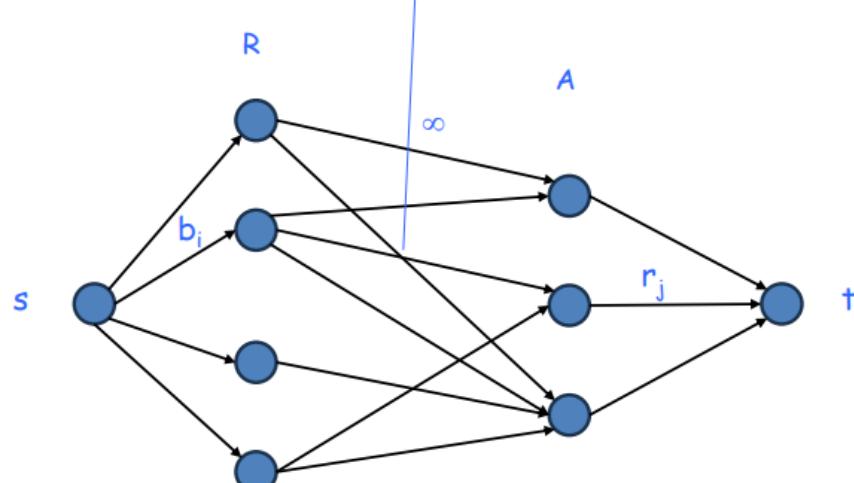
c'è arco se risorsa i può essere assegnata all'attività j

R =Risorsa
 A =attività

b_i =quantità della risorsa i-esima

r_j =risorse richieste

s =sorgente
 t =pozzo



Claim:

è possibile trovare un assegnamento e svolgere tutte le attività se e soltanto se il flusso massimo in G' è uguale a $\sum_j r_j$

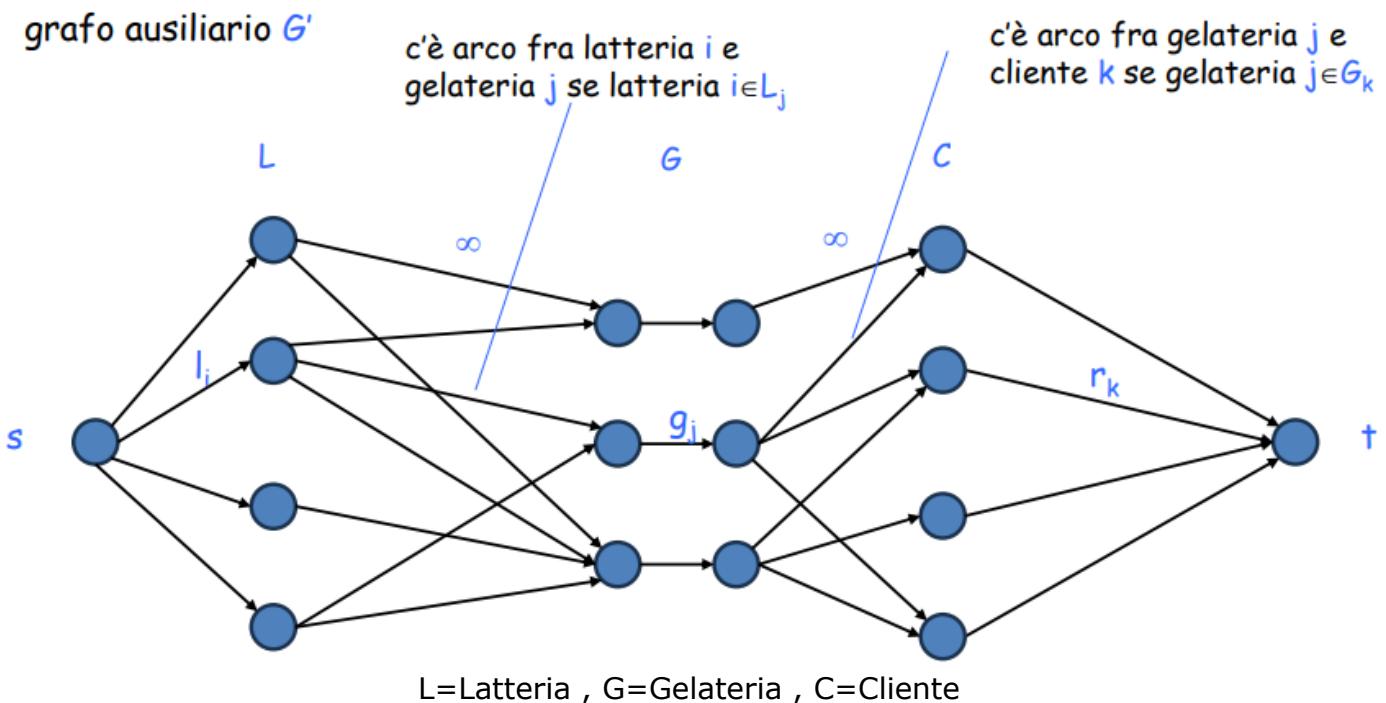
PROBLEMA 2

Scenario:

- un insieme L di n latterie
- un insieme G di m gelaterie
- un insieme C di t clienti
- lattoria i produce l_i litri di latte
- gelateria j
 - o si rifornisce solo nelle latterie $L_j \subseteq L$
 - o può lavorare al più g_j litri di latte
 - o ogni litro di latte è sufficiente per una vaschetta di gelato
- cliente k
 - o mangia solo nelle gelaterie $G_k \subseteq G$
 - o vuole r_k vaschette di gelato

Goal: capire se c'è modo di soddisfare tutti i clienti

Soluzione:



Claim:

è possibile soddisfare tutti i clienti se e soltanto se il flusso massimo in G' è uguale a $\sum_k r_k$

PROBLEMA 3

Scenario:

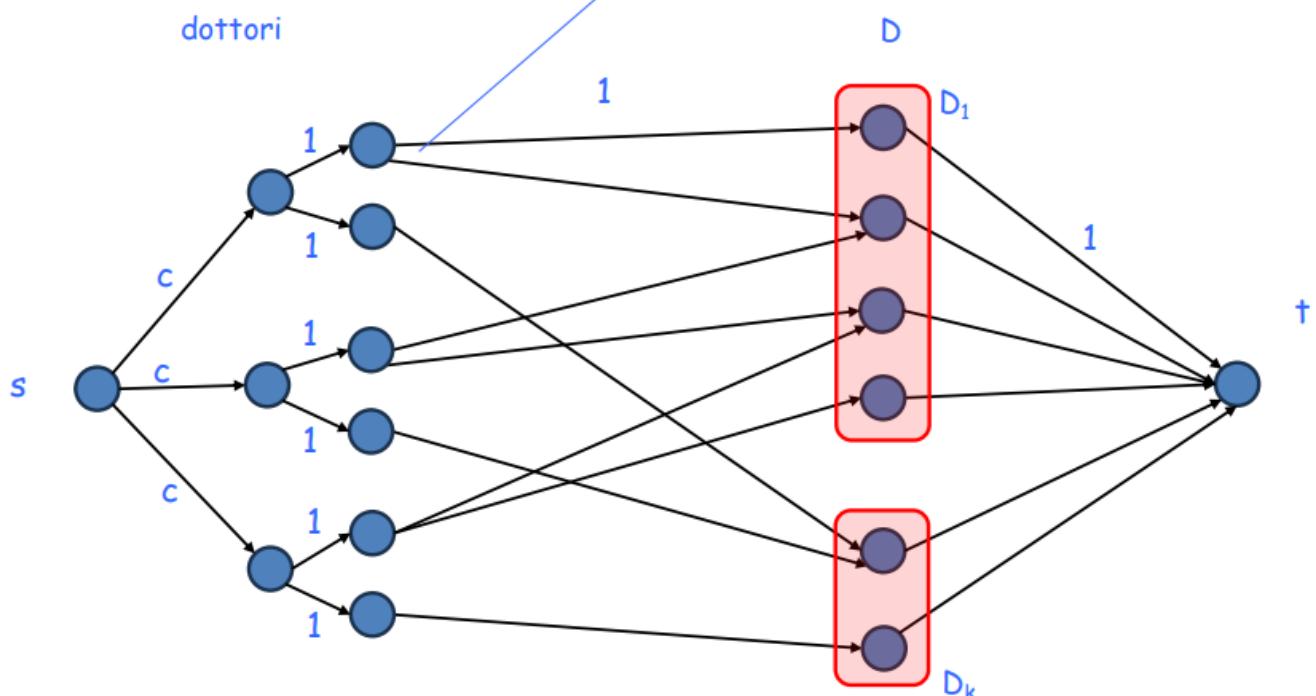
- n dottori
- k periodi di festività (es. settimana di natale, week end di pasqua,...)
- periodo j costituito da D_j giorni
- dottore i
 - o disponibile a lavorare nei giorni $S_i \subseteq D$, dove $D = \cup_j D_j$
 - o può lavorare al più c giorni festivi in totale
 - o ma mai più di un giorno per ogni periodo di festività

Goal: capire se è possibile assegnare un dottore ad ogni giorno di festività (rispettando i vincoli)

Soluzione:

grafo ausiliario G'

disponibilità dottore i per il periodo j



Claim:

esiste assegnamento che rispetta tutti i vincoli se e soltanto se il flusso massimo in G' è uguale a $|D|$

LEZ17 – 09/05/2024 (Introduzione all'NP-completezza. Concetto di riduzione polinomiale. Alcune riduzioni polinomiali)

DESIGN PATTERNS E ANTIPATTERNS

Schemi di progettazione di algoritmi (design patterns):

- greedy
- divide and conquer
- dynamic programming
- duality
- reductions
- local search
- randomization

Design antipatterns:

- | | |
|-----------------------|--|
| - NP-completeness | O(n^k) algoritmo improbabile |
| - PSPACE-completeness | O(n^k) algoritmo di certificazione improbabile |
| - Undecidability | nessun algoritmo possibile |

CLASSIFICAZIONE PROBLEMI A SECONDA DEI REQUISITI COMPUTAZIONALI

Quali problemi saremo in grado di risolvere in pratica?

- Quelli con algoritmi polinomiali (poly-time)

yes	probably no
shortest path	longest path
min cut	max cut
2-satisfiability	3-satisfiability
planar 4-colorability	planar 3-colorability
bipartite vertex cover	vertex cover
matching	3d-matching
primality testing	factoring
linear programming	integer linear programming

Teoria: La definizione è ampia e robusta (macchine di Turing, modello RAM, ...)

Nella pratica: Gli algoritmi di tempo polinomiale scalano (le costanti tendono a essere piccole, ad esempio $3n^2$) a problemi enormi.

CLASSIFICAZIONE PROBLEMI

Desiderata: Classificare i problemi in base a quelli che possono essere risolti in tempo polinomiale e quelli che non possono.

Richiede un tempo esponenziale:

- In un problema di decisione, dato un programma di dimensioni costanti, si ferma in al massimo k passi? (usando input size=c+logk)
- Nel gioco della dama, data la posizione della tavola in una generalizzazione n-by-n delle pedine, il nero può garantire una vittoria?

Notizie frustranti: Un numero enorme di problemi fondamentali hanno sfidato la classificazione per decenni.

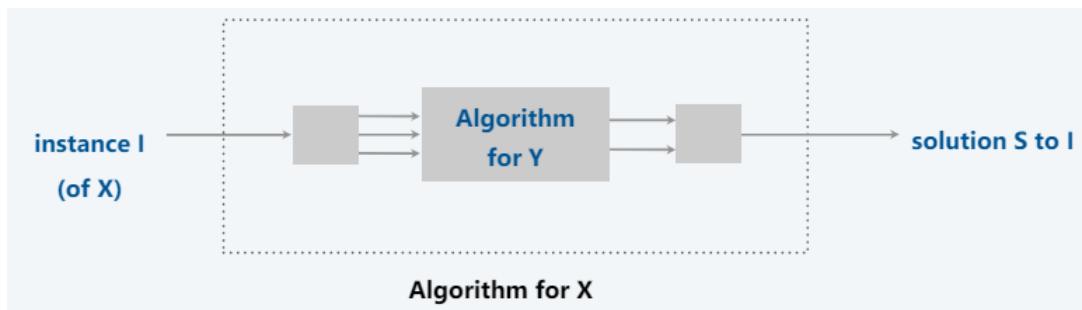
RIDUZIONI TEMPO POLINOMIALE

Desiderata: Supponiamo di poter risolvere il problema Y in tempo polinomiale.

Cos'altro potremmo risolvere in tempo polinomiale?

Riduzione: Problema X tempo polinomiale (Cook) si riduce al problema Y se istanze arbitrarie del problema X possono essere risolte utilizzando:

- Numero polinomiale di passi computazionali standard, più
- Numero polinomiale di chiamate ad oracolo che risolve il problema Y.
(oracolo = modello computazionale integrato da pezzo speciale di hardware che risolve le istanze di Y in un unico passaggio)



Notazione: $X \leq_P Y$.

Nota: paghiamo per il tempo di scrittura delle istanze di Y inviate all'oracolo => istanze di Y devono essere di dimensione polinomiale.

Errore: confondere $X \leq_P Y$ con $Y \leq_P X$

Algoritmi di progettazione: Se $X \leq_P Y$ e Y possono essere risolti in tempo polinomiale, allora X può essere risolto in tempo polinomiale.

Stabilire l'intrattabilità: Se $X \leq_P Y$ e X non possono essere risolti in tempo polinomiale, allora Y non può essere risolto in tempo polinomiale.

Stabilire l'equivalenza: Se sia $X \leq_P Y$ che $Y \leq_P X$, usiamo la notazione $X \equiv_P Y$.

In questo caso, X può essere risolto in tempo polinomiale se e solo se può anche Y.

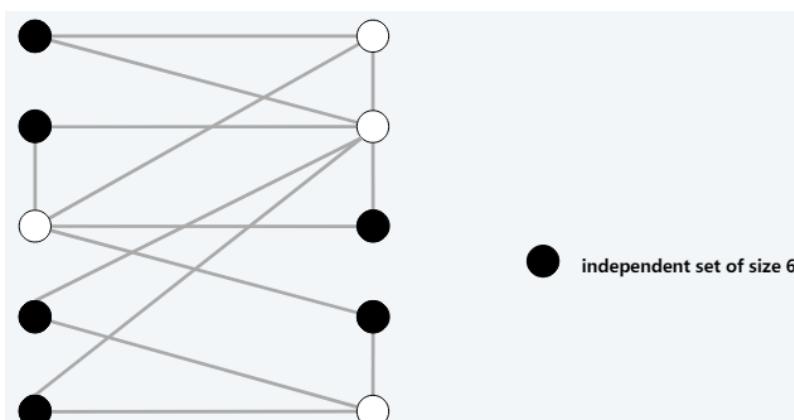
Bottom line: le riduzioni classificano i problemi in base alla difficoltà relativa.

INDIPENDENT SET (INSIEMI INDIPENDENTI)

Indipendent-Set: dato un grafo $G=(V,E)$ e dato un intero k , c'è un sottoinsieme di k (o più) vertici tale che non ce ne sono due adiacenti?

Esempio: esiste un indipendent set di dimensione ≥ 6 ?

Esempio: esiste un indipendent set di dimensione ≥ 7 ?

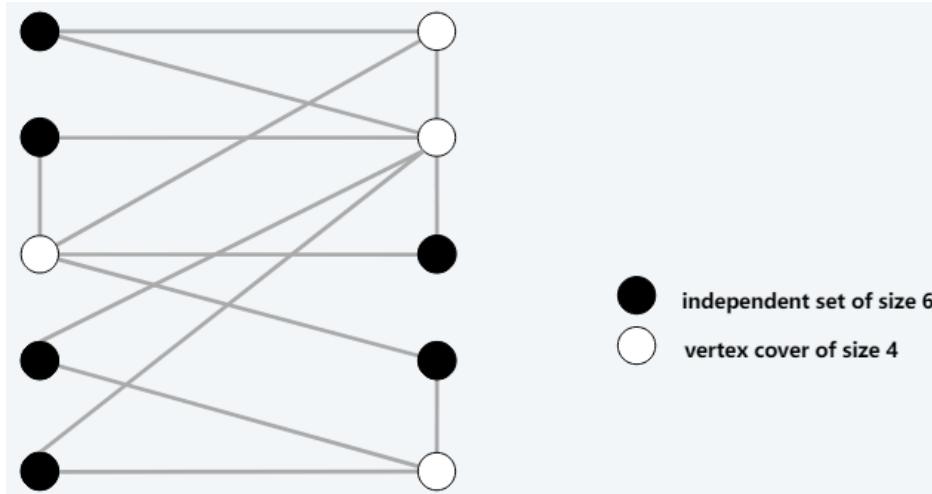


VERTEX COVER (COPERTURA DEI VERTICI)

Vertex-Cover: Dato un grafo $G=(V,E)$ e dato un intero k , c'è un sottoinsieme di k (o meno) vertici tali che ogni arco è incidente per almeno un vertice nel sottoinsieme?

Esempio: c'è un vertex cover di dimensione ≤ 4 ?

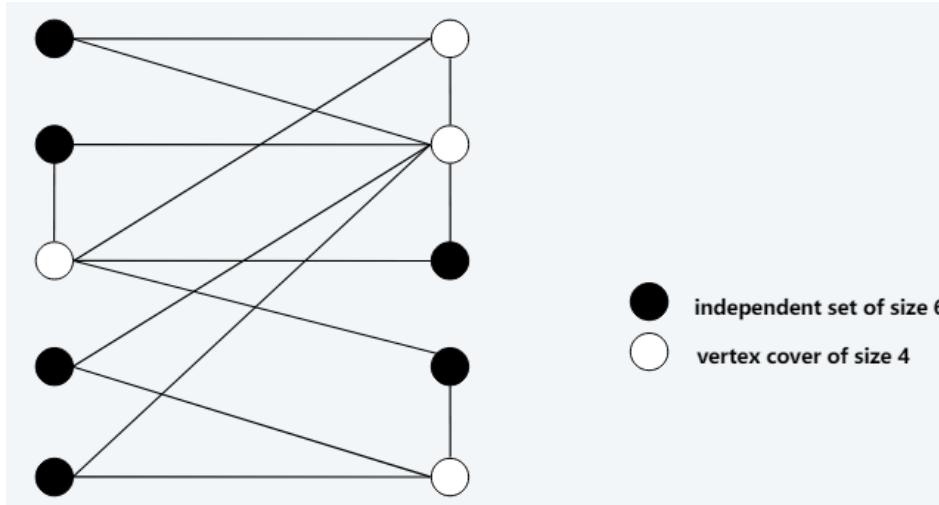
Esempio: c'è un vertex cover di dimensione ≤ 3 ?



VERTEX COVER E INDIPENDENT SET SI RIDUCONO L'UNO ALL'ALTRO

Teorema: INDIPENDENT-SET \equiv_P VERTEX-COVER

Dim: mostriamo che S è un indipendente set di dimensione k se e solo se $V-S$ è un vertex cover di dimensione $n-k$.



=>

Sia S un indipendente set di dimensione k

$V-S$ è di dimensione $n-k$

Consideriamo un arco arbitrario (u,v) appartenente ad E

S indipendente implica che

- o u non appartiene ad S , o v non appartiene ad S , o entrambe

- o u appartiene a $V-S$ o v appartiene a $V-S$, o entrambe

Pertanto $V-S$ copre (u,v)

<=

Sia $V-S$ un vertex cover di dimensione $n-k$

S è di dimensione k

Consideriamo un arco arbitrario (u,v) appartenente ad E

$V-S$ è un vertex cover implica che

- o u appartiene a $V-S$, o v appartiene a $V-S$, o entrambe

- o u non appartiene ad S , o v non appartiene ad S , o entrambe

Pertanto S è un indipendente set.

SET COVER

Set-Cover: Dato un insieme U di elementi, una collezione S di sottoinsiemi di U , e un intero k , ci sono un numero $\leq k$ di questi sottoinsiemi la cui unione è uguale a U ?

Semplice applicazione:

- m pezzi di software disponibili
- U è un insieme di n capacità che vorremmo avere nel nostro sistema.
- l'iesimo pezzo di software fornisce l'insieme S_i contenente parte delle capacità contenute in U
- Obiettivo: ottenere tutte le n capacità utilizzando il minor numero di pezzi di software.

$U = \{ 1, 2, 3, 4, 5, 6, 7 \}$	
$S_a = \{ 3, 7 \}$	$S_b = \{ 2, 4 \}$
$S_c = \{ 3, 4, 5, 6 \}$	$S_d = \{ 5 \}$
$S_e = \{ 1 \}$	$S_f = \{ 1, 2, 6, 7 \}$
$k = 2$	

a set cover instance

Ulteriore esempio: dato un universo $U=\{1,2,3,4,5,6,7\}$ e dato il seguente insieme, qual'è la dimensione minima di un set cover?

$U = \{ 1, 2, 3, 4, 5, 6, 7 \}$	
$S_a = \{ 1, 4, 6 \}$	$S_b = \{ 1, 6, 7 \}$
$S_c = \{ 1, 2, 3, 6 \}$	$S_d = \{ 1, 3, 5, 7 \}$
$S_e = \{ 2, 6, 7 \}$	$S_f = \{ 3, 4, 5 \}$

Dimensione minima:3

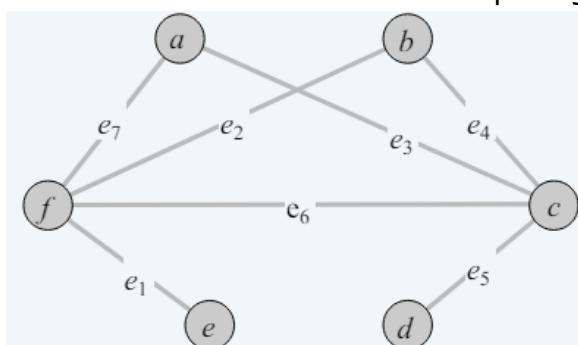
RIDUZIONE DI VERTEX COVER A SET COVER

Teorema: VERTEX-COVER \leq_P SET-COVER

Dim: data un'istanza di Vertex-Cover $G=(V,E)$ e k , costruiamo una istanza di Set-Cover (U,S,k) che ha un set cover di dimensione k se e solo se G ha un vertex cover di dimensione k .

Costruzione:

- Universo $U=E$
- Includiamo un sottoinsieme per ogni nodo $v \in V$: $S_v = \{ e \in E : e \text{ incidente } v \}$.



vertex cover instance
($k = 2$)

$U = \{ 1, 2, 3, 4, 5, 6, 7 \}$	
$S_a = \{ 3, 7 \}$	$S_b = \{ 2, 4 \}$
$S_c = \{ 3, 4, 5, 6 \}$	$S_d = \{ 5 \}$
$S_e = \{ 1 \}$	$S_f = \{ 1, 2, 6, 7 \}$

set cover instance
($k = 2$)

Lemma: $G = (V, E)$ contiene un vertex cover di dimensione k se e solo se (U, S, k) contiene un set cover di dimensione k .

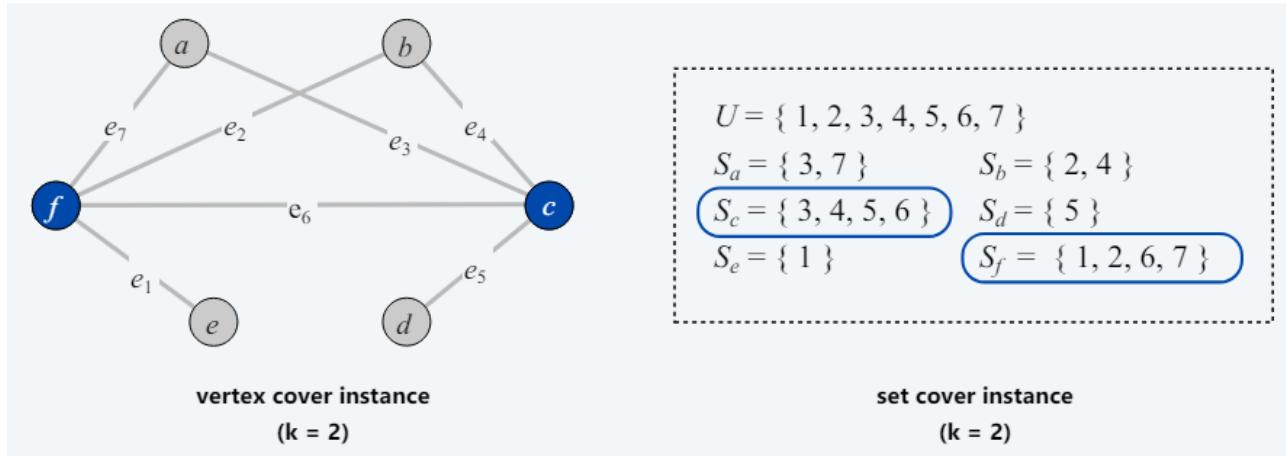
Dim:

=>

Sia $X \subseteq V$ un vertex cover di dimensione k in G .

Allora $Y = \{S_v : v \in X\}$ è un insieme set cover di dimensione k .

"yes" instances of VERTEX-COVER
are solved correctly

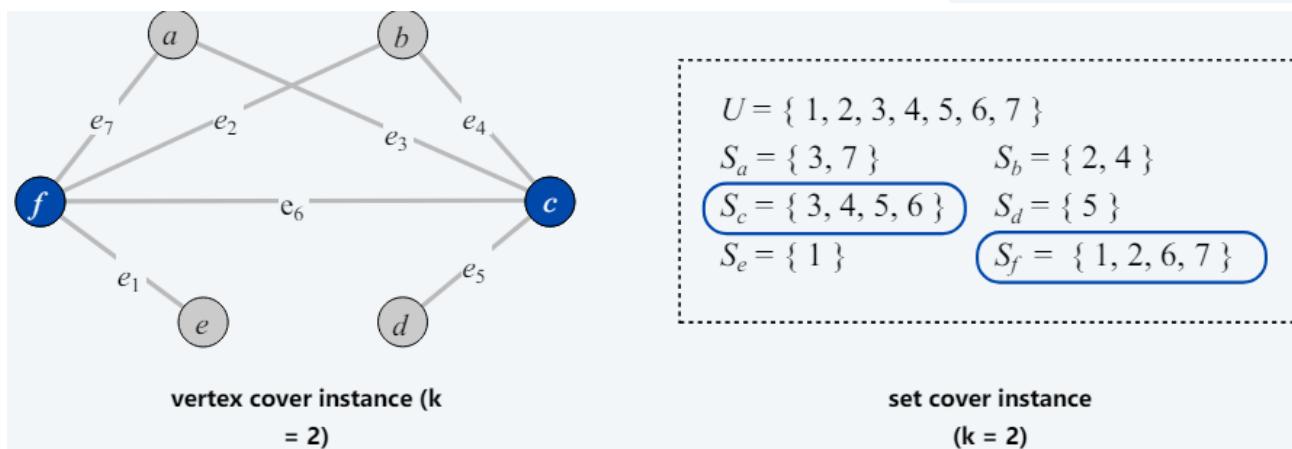


<=

Sia $Y \subseteq V$ un set cover di dimensione k in (U, S, k)

Allora $X = \{v : S_v \in Y\}$ è un vertex cover di dimensione K in G

"no" instances of VERTEX-COVER
are solved correctly



SODDISFACIBILITA' (SATISFIABILITY)

Letterale: una variabile booleana oppure la sua negazione

x_i or \bar{x}_i

Clausola: una disgiunzione di letterali

$C_j = x_1 \vee \bar{x}_2 \vee x_3$

Forma normale congiuntiva (CNF): una formula proposizionale Φ che è una congiunzione di clausole

$\Phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$

SAT

Sia Φ una formula CNF, quest'ultima ha una assegnazione che la rende vera?

3-SAT

SAT dove ogni clausola contiene esattamente 3 letterali.

$$\Phi = (\bar{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee x_4)$$

yes instance: $x_1 = \text{true}$, $x_2 = \text{true}$, $x_3 = \text{false}$, $x_4 = \text{false}$

SATISFIABILITY E' HARD

Ipotesi: Non esiste un algoritmo di tempo polinomiale per 3-SAT

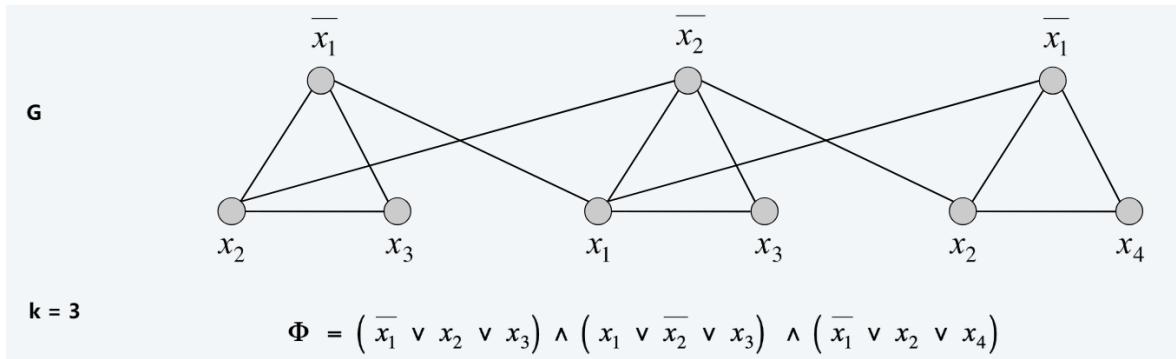
P vs NP: Questa ipotesi è equivalente alla congettura P ≠ NP

RIDUZIONE DI 3-SATISFIABILITY A INDEPENDENT SET

Teorema: Data una istanza Φ di 3-SAT, costruiamo una istanza (G, k) di Indipendent Set che ha una dimensione di indipendent set $k = |\Phi|$ se e solo se Φ è soddisfacibile.

Costruzione:

- G contiene 3 nodi per ogni clausola, uno per ogni letterale
- Collegare 3 letterali in una clausola in un triangolo
- Collegare ogni letterale alla sua negazione



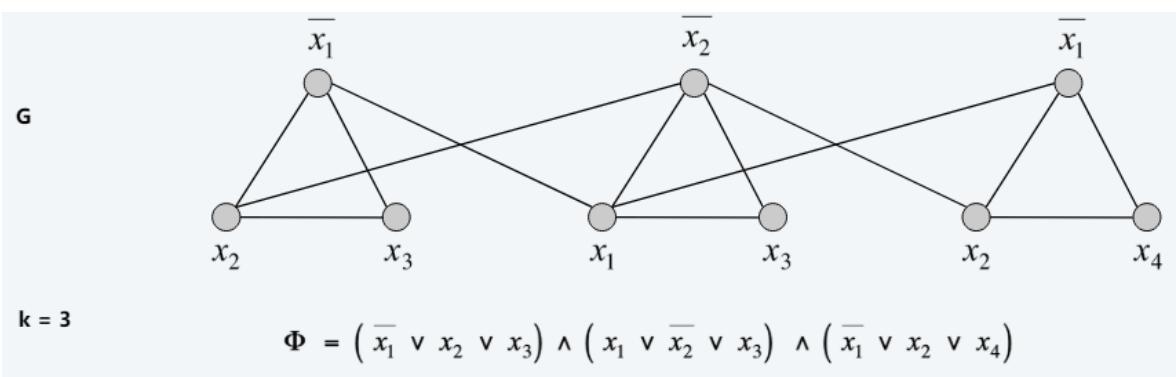
Lemma: Φ è soddisfacibile se e solo se G contiene un indipendent set di dimensione $k = |\Phi|$.

Dim:

=> considerare un assegnamento soddisfacibile per Φ

- Selezionare un letterale true da ogni clausola/triangolo
- C'è un indipendent set di dimensione $k = |\Phi|$

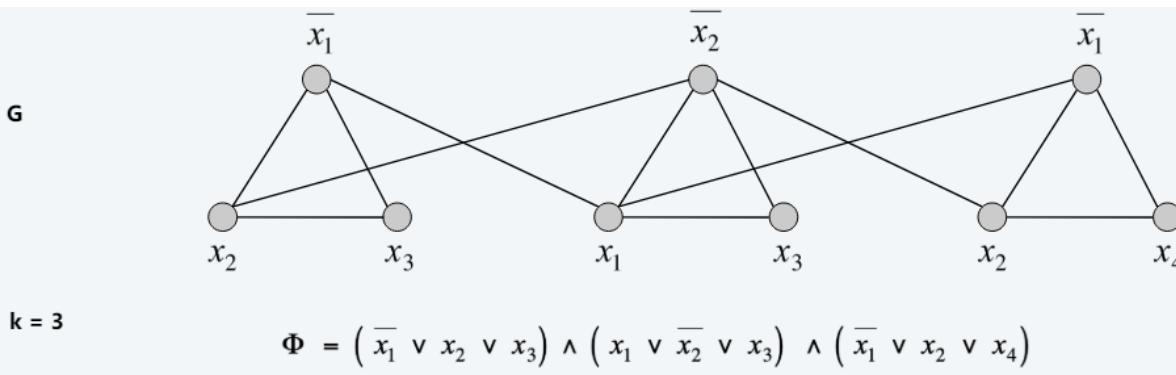
“yes” instances of 3-SAT
are solved correctly



\leq sia s un indipendent set di dimensione k

- s deve contenere esattamente un nodo in ogni triangolo
- Imposta questi letterali a true (e i letterali rimanenti in modo coerente)
- Tutte le clausole in Φ sono soddisfatte

“no” instances of 3-SAT
are solved correctly



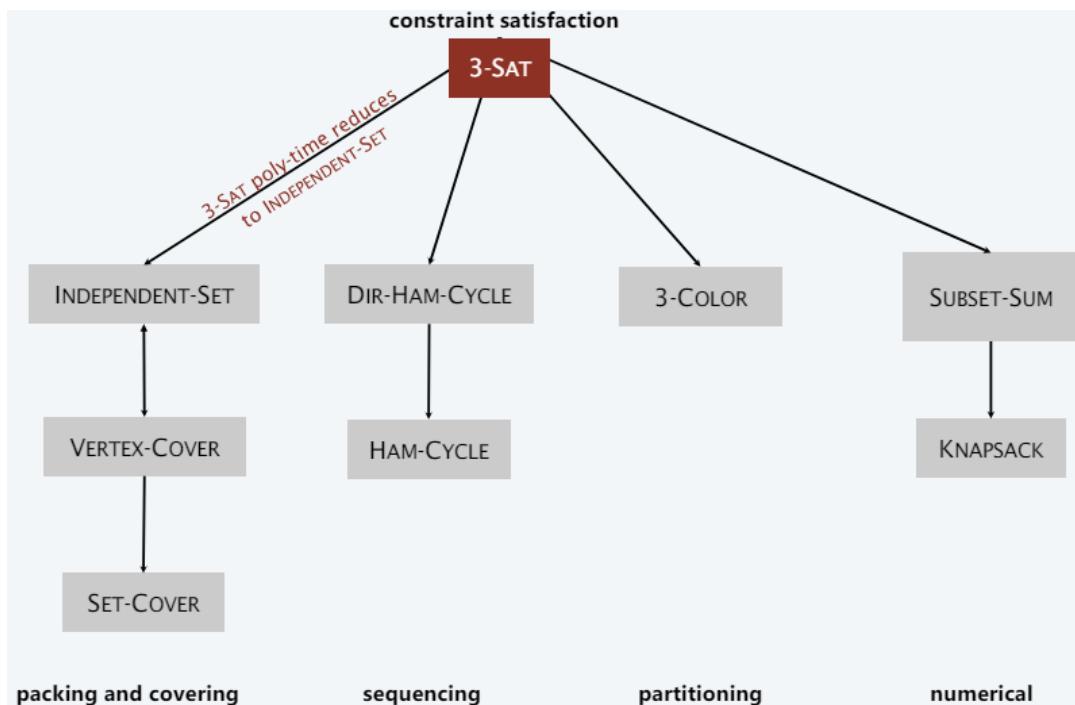
RICAPITOLANDO

Strategie basilari di riduzione:

1. SEMPLICE EQUIVALENZA: $\text{INDEPENDENT-SET} \equiv_{\text{P}} \text{VERTEX-COVER}$.
2. CASO SPECIALE A CASO GENERALE: $\text{VERTEX-COVER} \leq_{\text{P}} \text{SET-COVER}$.
3. ENCODING CON GUDGET : $\text{3-SAT} \leq_{\text{P}} \text{INDEPENDENT-SET}$

Transitività: se $X \leq_{\text{P}} Y$ e $Y \leq_{\text{P}} Z$ allora $X \leq_{\text{P}} Z$ (dim:comporre i due algoritmi)

Esempio: $\text{3-SAT} \leq_{\text{P}} \text{INDEPENDENT-SET} \leq_{\text{P}} \text{VERTEX-COVER} \leq_{\text{P}} \text{SET-COVER}$



Nota: Decision problem, Search problem e Optimization problem sono tre problemi riducibili l'uno all'altro.

Teorema: $\text{VERTEX-COVER} \equiv_{\text{P}} \text{FIND-VERTEX-COVER}$

con FIND-VERTEX-COVER=trovare un vertex cover di dimensione $\leq k$
e VERTEX-COVER=esiste un vertex cover di dimensione $\leq k$?

Teorema: $\text{FIND-VERTEX-COVER} \equiv_{\text{P}} \text{FIND-MIN-VERTEX-COVER}$

con FIND-VERTEX-COVER=trovare un vertex cover di dimensione $\leq k$
e FIND-MIN-VERTEX-COVER =trovare un vertex cover di dimensione minima

LEZ18 – 14/05/2024 (PvsNP, NP-completezza, co-NP)

PROBLEMA DI DECISIONE

Il problema X è un insieme di stringhe

- L'istanza s è una stringa
- L'algoritmo A risolve il problema X: $A(s) = \text{"yes"}$ se s appartiene a X, "no" se s non appartiene a X

Def: l'algoritmo A si esegue in tempo polinomiale se per ogni stringa s, $A(s)$ termina in un numero di step $\leq p(\text{lunghezza di } s)$, dove $p()$ è una qualche funzione polinomiale.

Def: P=insieme di problemi di decisione per cui esiste un algoritmo di tempo polinomiale (su una macchina di Turing deterministica).

NP

Def: L'algoritmo C(s,t) è un certificato per il problema X se per ogni stringa s quest'ultima appartiene a X se e solo se esiste una stringa t per cui $C(s,t)=\text{yes}$.

Def: NP=insieme dei problemi di decisione per cui esiste un certificato di tempo polinomiale

- $C(s,t)$ è un algoritmo di tempo polinomiale
- Il certificato t è di dimensione polinomiale: $|t| \leq p|s|$ per qualche $p()$ polinomiale

CERTIFICAZIONI E CERTIFICATI : SODDISFACIBILITÀ

SAT: Data una formula Φ in CNF, questa ha un'assegnazione di verità che la rende vera?

3SAT: SAT dove ogni clausola contiene esattamente tre letterali.

Certificato: un'assegnazione di valori veri alle variabili booleane.

Certificazione: controlla che ogni clausola in Φ ha almeno un letterale true.

Esempio:

$$\text{instance } s \quad \Phi = (\bar{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee x_4)$$

$$\text{certificate } t \quad x_1 = \text{true}, x_2 = \text{true}, x_3 = \text{false}, x_4 = \text{false}$$

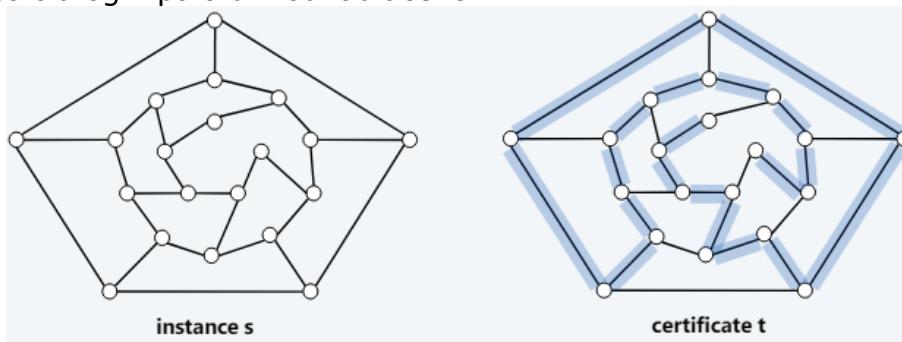
Conclusioni: $\text{SAT} \in \text{NP}$, $\text{3-SAT} \in \text{NP}$.

CERTIFICAZIONI E CERTIFICATI: HAMILTON PATH

Hamilton-Path: dato un grafo non diretto $G=(V,E)$, esiste un semplice path P che visita tutti i nodi?

Certificato: una permutazione π di n nodi.

Certificazione: controlla che π contiene ogni nodo in V esattamente una volta, e che G contiene un arco tra ogni paio di nodi adiacenti.



Conclusioni: $\text{Hamilton-Path} \in \text{NP}$.

DUE PROBLEMI CHE PROBABILMENTE NON APPARTENGONO A NP

1. PEDINE: data la posizione delle pedine di una scacchiera nxn, si può garantire che le pedine nere vincano?
2. CO-LONGEST-OATH: Dato un grafo non diretto $G=(V,E)$, la lunghezza del più lungo percorso semplice è $\leq k$?

ALCUNI PROBLEMI CHE SI TROVANO IN NP

NP: Problemi di decisione per cui esiste un certificato di tempo polinomiale.

P , NP E EXP

P: Problemi decisionali per i quali esiste un algoritmo in tempo polinomiale.

NP: Problemi di decisione per i quali esiste un certificatore in tempo polinomiale.

EXP: Problemi di decisione per i quali esiste un algoritmo in tempo esponenziale.

Proposizione: $P \subseteq NP$

Proposizione: $NP \subseteq EXP$

Fatto: $P \neq EXP \Rightarrow P \neq NP$, o $NP \neq EXP$, o entrambi.

LA DOMANDA PRINCIPALE: P vs NP

$P = NP$? Il problema della decisione è così semplice come il problema della certificazione?



Se sì, ci sono algoritmi efficienti per 3-SAT, TSP, VERTEX-COVER, FACTOR, ...

Se no, non ci sono algoritmi efficienti possibili per 3-SAT, TSP, VERTEX-COVER, ...

Opinione: probabilmente no.

ALTRI RISULTATI POSSIBILI

$P = NP$ ma solo $\Omega(n^{100})$ algoritmi per 3-SAT

$P \neq NP$ ma solo $O(n^{\log n})$ algoritmi per 3-SAT

$P = NP$ è indipendente (di ZFC axiomatic set theory).

TRASFORMAZIONI POLINOMIALI

Def(Cook): il problema X polinomiale si riduce al problema Y se le istanze arbitrarie del problema X possono essere risolte usando:

- Numero polinomiale di passi computazionali standard, più
- Numero polinomiale di chiamate ad oracolo che risolve il problema Y

Def(Karp): il problema X polinomiale si trasforma in problema Y se data qualsiasi istanza x di X, possiamo costruire un'istanza y di Y tale che x è un'istanza sì di X se e soltanto se y è un'istanza sì di Y.

Nota: La trasformazione polinomiale è una riduzione polinomiale con una sola chiamata all'oracolo per Y, esattamente alla fine dell'algoritmo per X. Quasi tutte le precedenti riduzioni erano di questa forma.

Domanda aperta. Questi due concetti sono gli stessi rispetto a NP?

NP-COMPLETO

NP-Completo: un problema $Y \in NP$ con la proprietà che per ogni problema $X \in NP$, $X \leq_P Y$.

Proposizione. Supponiamo $Y \in NP$ -completo. Allora $Y \in P$ se e solo se $P = NP$.

Dim: $\leq_P P = NP$ allora $Y \in P$ perché $Y \in NP$

\Rightarrow Supponiamo $Y \in P$, consideriamo un problema $X \in NP$. Da $X \leq_P Y$ abbiamo $X \in P$
Questo implica $NP \subseteq P$, noi sappiamo già che $P \subseteq NP$, pertanto $P = NP$.

Domanda fondamentale: Ci sono problemi naturali NP-completi?

NP-COMPLETEZZA

Osservazione: una volta stabilito il primo problema "naturale" NP-completo si ha un effetto domino.

Per dimostrare che $Y \in NP$ -completo:

- 1) Mostrare che $Y \in NP$
- 2) Scegliere un problema $X \in NP$ -completo.
- 3) Dimostrare che $X \leq_P Y$

Proposizione: Se $X \in NP$ -completo, $Y \in NP$, e $X \leq_P Y$ allora $Y \in NP$ -completo.

Dim: considerare un problema $W \in NP$. Allora $W \leq_P X$ e $X \leq_P Y$. Da transitività abbiamo che $W \leq_P Y$, dunque $Y \in NP$ -completo.

ALCUNI PROBLEMI NP-COMPLETI

- Generici problemi semplici NP-completi ed esempi paradigmatici:
- Problemi di packing/covering: SET-COVER, VERTEX-COVER, INDEPENDENT-SET.
- Problemi di soddisfacimento vincoli: CIRCUIT-SAT, SAT, 3-SAT.
- Problemi di sequenziamento: HAMILTON-CYCLE, TSP.
- Problemi di partizionamento: 3D-MATCHING, 3-COLOR.
- Problemi numerici: SUBSET-SUM, KNAPSACK.

Pratica: La maggior parte dei problemi NP sono noti per essere in P o NP-complete.

NP-intermedio? FACTOR, DISCRETE-LOG, GRAPH-ISOMORPHISM,

Teorema(Ladner): A meno che $P = NP$, esistono problemi in NP che non sono né in P né in NP-complete.

CO-NP

Asimmetria di NP: Abbiamo bisogno di certificati brevi soltanto per istanzi "sì".

Esempio1: SAT vs UN-SAT

- Possiamo provare che una formula CNF è soddisfacibile specificando un'assegnazione
- Come possiamo provare che una formula non è soddisfacibile?

SAT. Given a CNF formula Φ , is there a satisfying truth assignment?

UN-SAT. Given a CNF formula Φ , is there no satisfying truth assignment?

Esempio2: HAMILTON-CYCLE vs NO-HAMILTON-CYCLE

- Possiamo provare che un grafo è Hamiltoniano specificando una permutazione
- Come possiamo provare che un grafo non è Hamiltoniano?

HAMILTON-CYCLE. Given a graph $G = (V, E)$, is there a simple cycle Γ that contains every node in V ?

No-HAMILTON-CYCLE. Given a graph $G = (V, E)$, is there no simple cycle Γ that contains every node in V ?

Domanda: come si classificano UN-SAT e NO-HAMILTON-CYCLE?

- $SAT \in NP$ -COMPLETO e $SAT \equiv_P UN-SAT$
- $HAMILTON-CYCLE \in NP$ -COMPLETO e $HAMILTON-CYCLE \equiv_P NO-HAMILTON-CYCLE$
- Ma né UN-SAT né NO-HAMILTON-CYCLE sono noti per essere in NP

NP e CO-NP

NP: Problemi decisionali per i quali esiste un certificatore di tempo polinomiale

Esempi: SAT, HAMILTON-CYCLE e COMPOSITES.

Def: Dato un problema di decisione X, il suo complemento è lo stesso problema con le risposte "sì" e "no" invertite.

CO-NP: complementi di problemi decisionali in NP.

Esempi: UN-SAT, NO-HAMILTON-CYCLE, PRIMES

NP = CO-NP??

Domanda fondamentale aperta: $NP=CO-NP?$

- Le istanze "sì" hanno certificato se e solo se ce l'hanno le istanze "no"?
- Opinione consensuale: no.

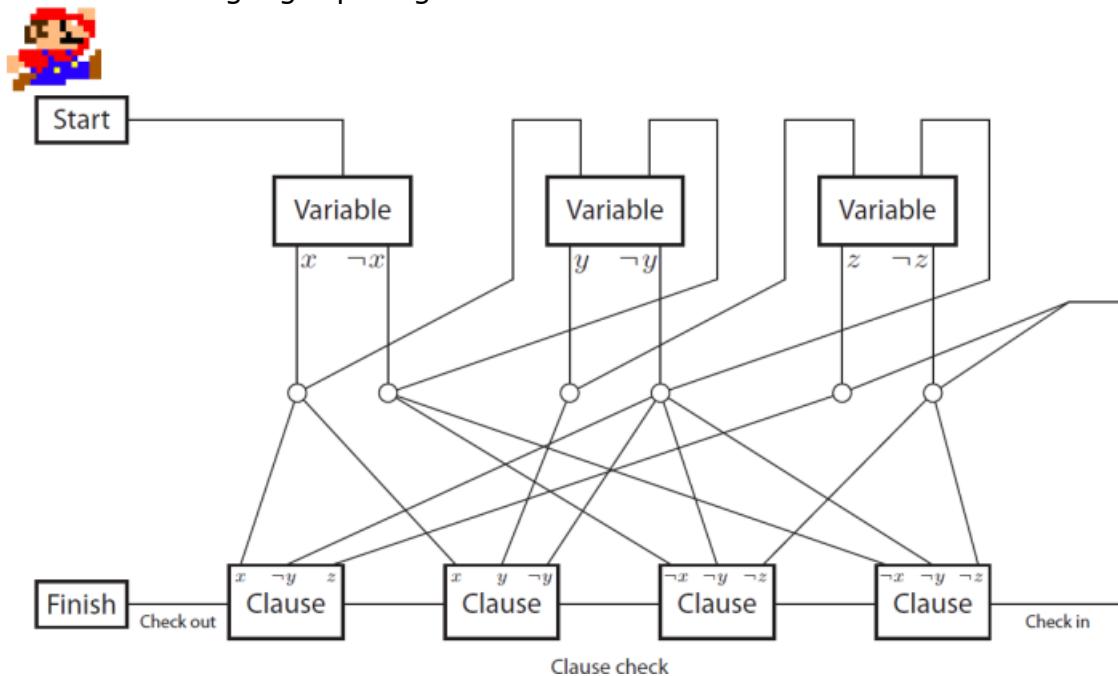
Teorema: se $NP \neq CO-NP$ allora $P \neq NP$

LEZ19 – 16/05/2024 (Qualche altra riduzione: Super Mario, Peg Solitaire e Tetris sono NP-completi)

SUPER MARIO RIDUZIONE DA 3-SAT

Idea: data un'istanza di 3-SAT costruiamo un livello/istanza di Super Mario che è risolvibile se e solo se la formula è soddisfacibile.

Attuazione idea: trasformazione tramite l'utilizzo di gadget, 1 stanza gadget per ogni variabile e una stanza gadget per ogni clausola.



PEG SOLITAIRE RIDUZIONE DA HAMILTONIAN CYCLE.

NP-completo per decidere se la board può essere "pulita".



Riduzione da ciclo Hamiltoniano in un grafo diretto planare di grado 3.

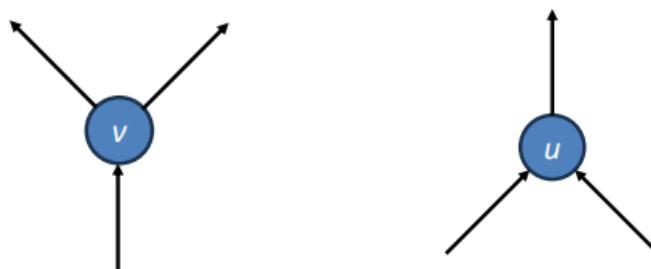
(Planare = può essere disegnato in un piano in modo che nessun arco si incroci con un altro).

Input: un grafo diretto planare G dove ogni vertice ha grado esattamente 3.

Goal: G ha un ciclo Hamiltoniano diretto?

Questo è NP-completo.

Ci sono solo due tipi di vertici (1-in 2-out gadget vertex grado):



1-in 2-out
degree vertex

2-in 1-out
degree vertex

Affermazione: è possibile pulire la board se e solo se G ha una ciclo hamiltoniano.
dim:

\leq

- se esiste un ciclo hamiltoniano C
 - prima cancellare i bordi che non appartengono a C
 - eliminare i pegs rimanenti passando per C

\geq

Intuitivamente se vuoi pulire la board devi giocare i gadgets nel modo corretto

(Questa riduzione è spiegata molto bene nelle slides con scatch quindi vedila da li)

Nota lezione: Gadget crossover utile su grafi planari in cui non ci devono essere incroci

TETRIS RIDUZIONE DA 3-PARTITION

3-PARTITION:

Input: una collezione A di n interi positivi $a_1 \dots a_n$

Domanda: è possibile partizionare A in $n/3$ collezioni $A_1 \dots A_{(n/3)}$ in modo che ogni partizione abbia somma dei valori dei propri elementi uguale alle altre?

Fatto1: 3-Partition è NP-completo, anche se $t/4 < a_i < t/2$

Oss: se assumiamo $t/4 < a_i < t/2$ abbiamo $|A_i| = 3$ per ogni A_i

Fatto2: 3-Partition è forte NP-Hard, è NP-Completo anche se ogni a_i è polinomialmente limitato in n ($n =$ numero di numeri)

Per effettuare la riduzione,

Input: una configurazione iniziale della board più l'intera sequenza di pezzi

Goal: Puoi pulire la board?

Si utilizzano ai gadget, tra cui initiator, filler(ai times) e terminator.

(Per maggiori info vedi slides fatte molto bene)

LEZ20 - 21/05/2024 (esercitazione)

SUSPICIOUS COALITION PROBLEM: RIDUZIONE DA VERTEX COVER

LECTURE PLANNING PROBLEM

LEZ21- 23/05/2024 (algoritmi di approssimazione: load balancing & center selection)

Supponiamo di dover risolvere un problema di ottimizzazione NP-hard. Che cosa dovrei fare?

Sacrificare una delle tre caratteristiche desiderate.

- i. Risoluzione in tempo polinomiale.
- ii. Risolve istanze arbitrarie del problema.
- iii. Trova la soluzione ottimale al problema. (sacrificata in algoritmi di approssimazione)

$\hat{\rho}$ -approximation algorithm:

- Funziona in tempo polinomiale.
- Risolve istanze arbitrarie del problema
- Trova la soluzione che è nel ratio $\hat{\rho}$ dell'ottimo.

Sfida. Necessità di dimostrare che il valore di una soluzione è vicino a quello ottimale, senza nemmeno sapere qual'è il valore ottimale.

α -APPROXIMATION ALGORITHM

Un algoritmo di α -approssimazione per un problema di ottimizzazione è un algoritmo di tempo polinomiale che per tutte le istanze del problema produce una soluzione il cui valore è entro un certo fattore di α rispetto al valore di una soluzione ottimale.

α : rapporto di approssimazione o fattore di approssimazione

problema di minimizzazione:

- $\alpha \geq 1$
- per ogni soluzione restituita x , $\text{costo}(x) \leq \alpha \text{OPT}(x)$

problema di massimizzazione:

- $\alpha \leq 1$
- per ogni soluzione restituita x , $\text{valore}(x) \geq \alpha \text{OPT}(x)$

LOAD BALANCING

Input: m macchine identiche; $n \geq m$ jobs, job j ha tempo di elaborazione t_j .

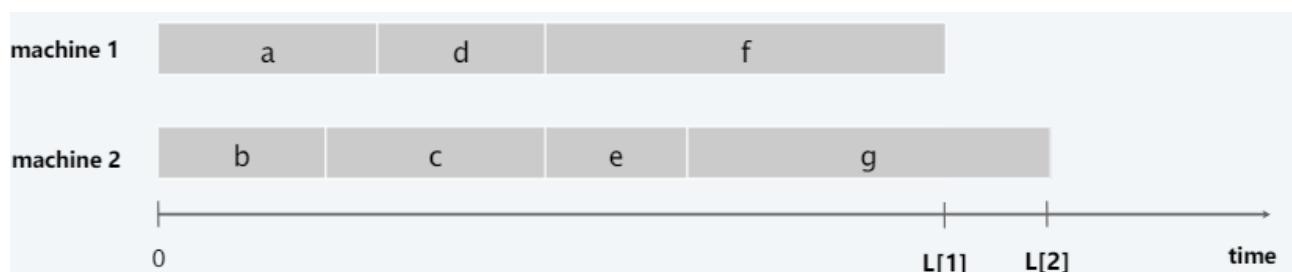
- Il job j deve essere eseguito in modo contiguo su una macchina.
- Una macchina può elaborare al massimo un job alla volta.

Def: Sia $S[i]$ il sottoinsieme dei jobs assegnati alla macchina i .

Il carico della macchina i è $L[i] = \sum_{j \in S[i]} t_j$.

Def: Il makespan è il carico massimo su qualsiasi macchina $L = \max_i L[i]$.

Bilanciamento del carico: Assegnare ogni lavoro a una macchina per minimizzare makespan.



Load balancing su due macchine è NP-hard:

Dim: $\text{PARTITION} \leq_P \text{LOAD-BALANCE}$.

ALGORITMO LIST-SCHEDULING PER LOAD BALANCING:

- Considera n jobs in un ordine fissato
- Assegna job j alla macchina i il cui carico è finora il più piccolo.

LIST-SCHEDULING ($m, n, t_1, t_2, \dots, t_n$)

FOR $i = 1$ TO m

$L[i] \leftarrow 0$. \leftarrow load on machine i

$S[i] \leftarrow \emptyset$. \leftarrow jobs assigned to machine i

FOR $j = 1$ TO n

$i \leftarrow \operatorname{argmin}_k L[k]$. \leftarrow machine i has smallest load

$S[i] \leftarrow S[i] \cup \{j\}$. \leftarrow assign job j to machine i

$L[i] \leftarrow L[i] + t_j$. \leftarrow update load of machine i

RETURN $S[1], S[2], \dots, S[m]$.

Implementazione: $O(n \log n)$ usando una coda con priorità per i carichi $L[k]$.

➤ Analisi list scheduling:

Teorema [Graham 1966] L'algoritmo greedy è un algoritmo 2-approximation

- Prima analisi del caso peggiore di un algoritmo di approssimazione.
- Necessità di confrontare la soluzione risultante con makespan ottimale L^* .

Lemma1: per tutti i k il makespan ottimale $L^* \geq t_k$

Dim: alcune macchine devono elaborare il job che richiede più tempo.

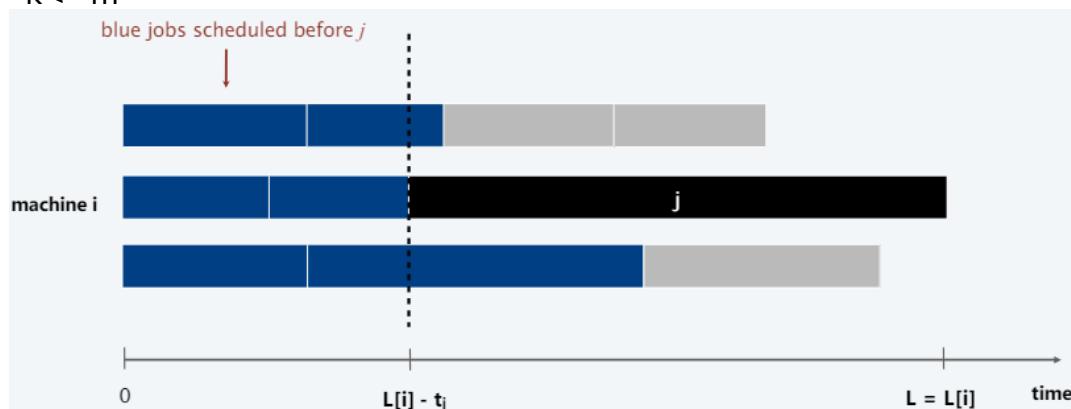
Lemma2: il makespan ottimale $L^* \geq (1/m) \sum k t_k$

Dim: il tempo totale di elaborazione è $\sum k t_k$
una delle m macchine deve fare almeno $1/m$ frazione del lavoro totale

Teorema: l'algoritmo greedy è un algoritmo 2-approximation.

Dim: si consideri il carico $L[i]$ della macchina collo di bottiglia i .

- sia j l'ultimo job programmato sulla macchina i
 - quando il job j è assegnato alla macchina i , i ha avuto il carico più piccolo.
- Il suo carico prima dell'assegnazione è $L[i] - t_j$; quindi $L[i] - t_j \leq L[k]$ per tutti i $1 \leq k \leq m$



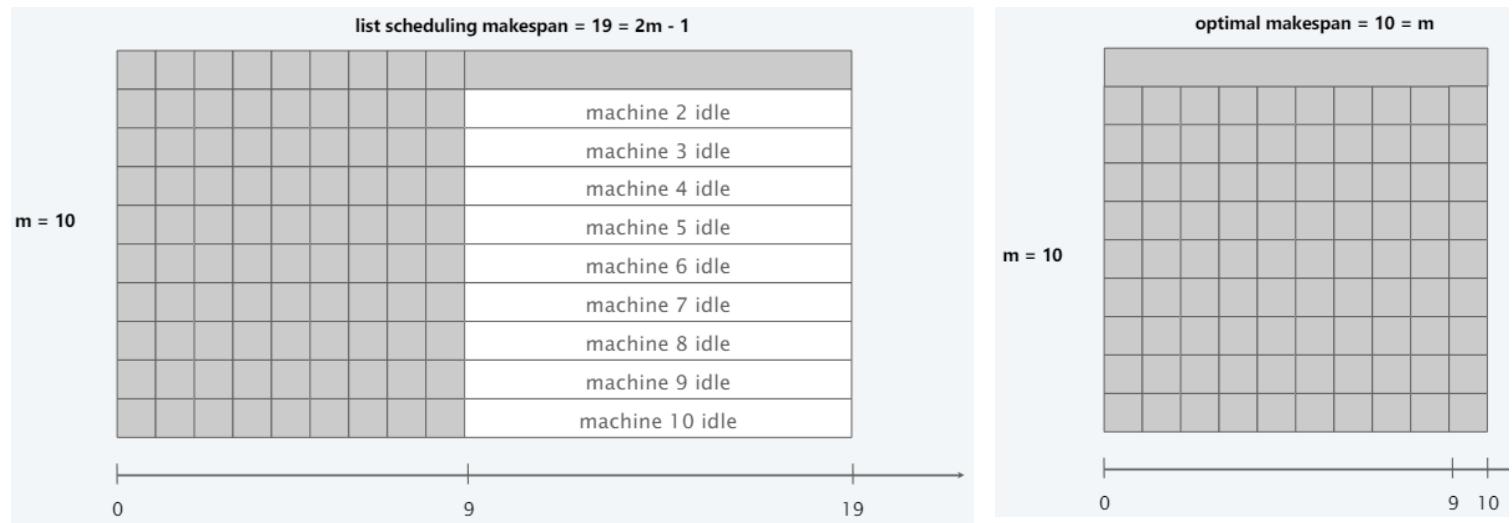
- somma disuguaglianze su tutti i k e dividere per m :

$$\begin{aligned}
 L[i] - t_j &\leq \frac{1}{m} \sum_k L[k] \\
 &= \frac{1}{m} \sum_k t_k \\
 \text{Lemma 2} \rightarrow &\leq L^*.
 \end{aligned}$$

- $L = L[i] = (L[i] - t_j) + t_j \leq 2L^*$

Q. La nostra analisi è stretta? In sostanza sì.

Es: m macchine, i primi m ($m - 1$) jobs hanno lunghezza 1, ultimo job ha lunghezza m .



BILANCIAMENTO DEL CARICO: REGOLA LPT(Longest Processing Time)

Tempo di elaborazione più lungo(LPT): Ordinare n jobs in ordine decrescente di tempi di elaborazione; quindi eseguire l'algoritmo di pianificazione delle liste.

LPT-LIST-SCHEDULING $(m, n, t_1, t_2, \dots, t_n)$

SORT jobs and renumber so that $t_1 \geq t_2 \geq \dots \geq t_n$.

FOR $i = 1$ TO m

$L[i] \leftarrow 0$. \leftarrow load on machine i

$S[i] \leftarrow \emptyset$. \leftarrow jobs assigned to machine i

FOR $j = 1$ TO n

$i \leftarrow \operatorname{argmin}_k L[k]$. \leftarrow machine i has smallest load

$S[i] \leftarrow S[i] \cup \{j\}$. \leftarrow assign job j to machine i

$L[i] \leftarrow L[i] + t_j$. \leftarrow update load of machine i

RETURN $S[1], S[2], \dots, S[m]$.

Osservazione: se la bottleneck machine i ha solo 1 job, allora è ottimale.

Dim: qualsiasi soluzione deve fare schedule di quel job.

Lemma3: se ci sono più di m jobs, $L^* \geq 2tm+1$

Dim:

- Considerare i tempi di elaborazione dei primi $m+1$ jobs $t_1 \geq t_2 \geq \dots \geq t_{m+1}$.
- Ognuno richiede almeno tempo t_{m+1} .
- Ci sono $m+1$ jobs e m macchine, quindi almeno una macchina ottiene due job

Teorema: la regola LPT è un algoritmo $3/2$ -approximation.

Dim[simile a dim per list scheduling]:

- Considerare il carico $L[i]$ della bottleneck machine i
- Sia j l'ultimo job programmato sulla macchina i (assumendo macchina i ha almeno 2 job, abbiamo $j \geq m+1$)
$$L = L[i] = (L[i] - t_j) + t_j \leq (3/2)L^*$$

La nostra analisi $3/2$ è stretta? No.

Teorema[Graham 1969]: la regola LPT è una $4/3$ -approximation.

Dim: analisi più sofisticata dello stesso algoritmo.

L'analisi di Graham $4/3$ è stressa? Si.

Esempio: m macchine

$N=2m+1$ jobs

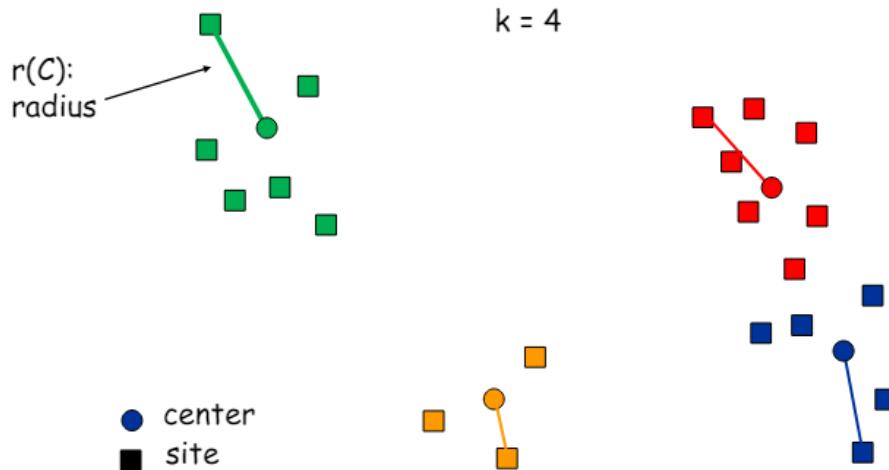
2 jobs di lunghezza $m, m+1, \dots, 2m-1$ e un ulteriore job di lunghezza m

Dunque $L/L^* = (4m-1)/(3m)$

CENTER SELECTION

K-Center Problem

- Input. insieme di n siti s_1, \dots, s_n e intero $k > 0$.
- Problema di selezione centrale (center selection problem): Selezionare k centri C in modo da ridurre al minimo la distanza massima da un sito al centro più vicino.



Una variante: i centri sono costretti ad essere collocati sulle posizioni dei siti.

Notazione:

$\text{dist}(x,y)$ = distanza tra x e y

$\text{dist}(s_i, C) = \min_{c \in C} \text{dist}(s_i, c)$ = distanza da s_i al centro più vicino

$r(C) = \max_i \text{dist}(s_i, C)$ = raggio di copertura minimo

Goal: trovare un insieme di centri C che minimizza $r(C)$, soggetto a $|C|=k$.

Distance function proprietà:

$\text{dist}(x,x)=0$

[identity]

$\text{dist}(x,y)=\text{dist}(y,x)$

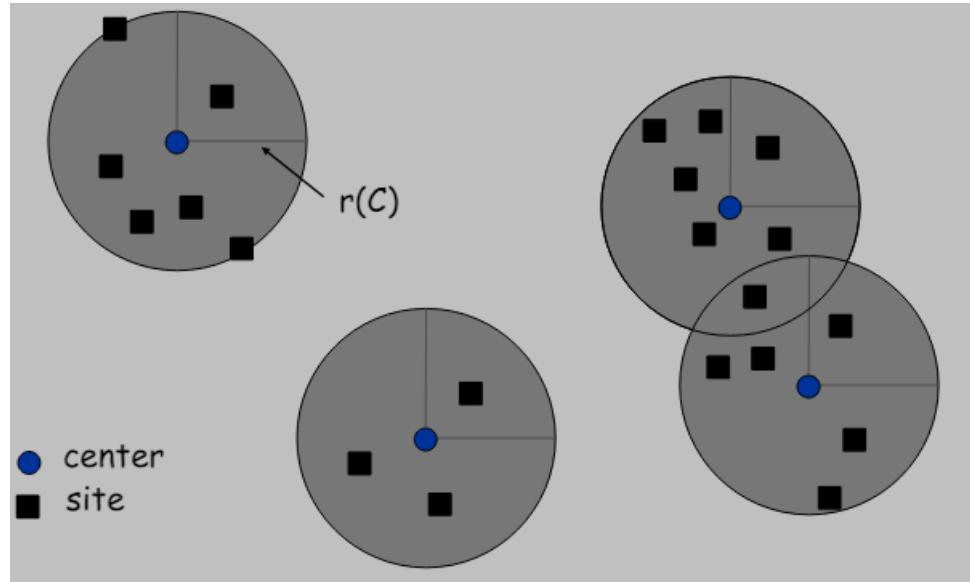
[symmetry]

$\text{dist}(x,y) \leq \text{dist}(x,z) + \text{dist}(z,y)$

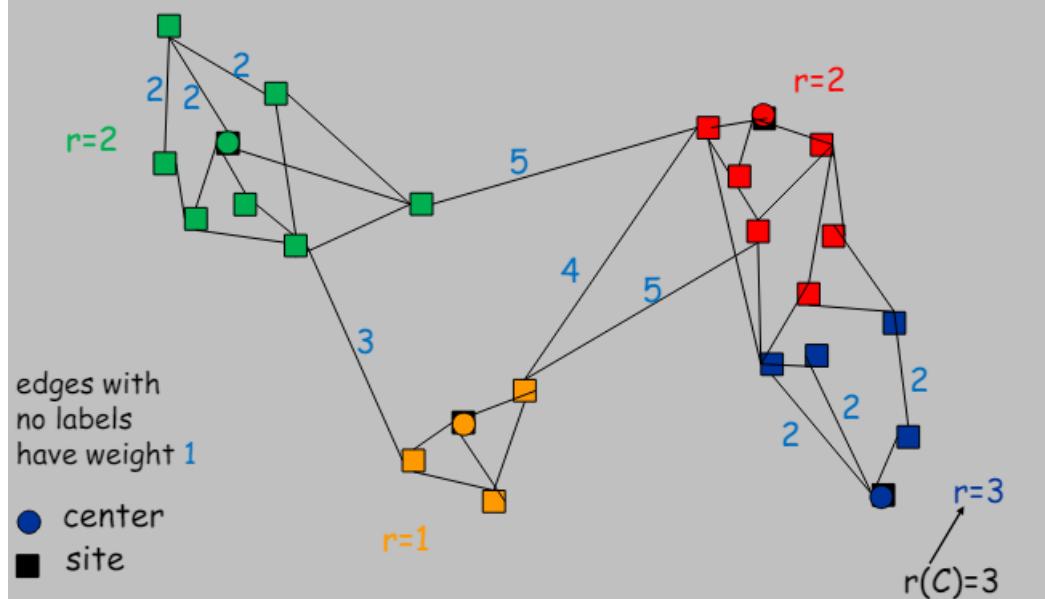
[triangle inequality]

Esempi:

- 1) Ogni sito è un punto nel piano, un centro può essere un qualsiasi punto nel piano, $\text{dist}(x,y)=\text{distanza euclidea}$



- 2) Ogni sito è un vertice in un grafo pesato non diretto, un centro può essere un qualsiasi vertice, $\text{dist}(x,y)=(\text{pesata}) \text{ distanza in } G \text{ tra } x \text{ e } y$.



CENTER SELECTION: GREEDY ALGORITHM

Algoritmo greedy: Ripetutamente sceglie il prossimo centro selezionando il sito più lontano da qualsiasi altro centro esistente.

```
Greedy-Center-Selection( $k, n, s_1, s_2, \dots, s_n$ ) {
     $C = \emptyset$ 
    repeat  $k$  times {
        Select a site  $s_i$  with maximum  $\text{dist}(s_i, C)$ 
        Add  $s_i$  to  $C$ 
    }
    return  $C$ 
}
```

Osservazione. Al termine tutti i centri in C sono accoppiati almeno $r(C)$ parti

➤ Analisi dell'algoritmo greedy

Teorema: Sia C^* un insieme ottimo di centri. Allora $r(C) \leq 2r(C^*)$

dim: complicata slide 33-34

Teorema: algoritmo greedy è un algoritmo 2-approximation per il problema di selezione dei centri.

Oss: algoritmo greedy mette sempre centri a siti, ma è ancora all'interno di un fattore di 2 della soluzione migliore che è permesso posizionare i centri dovunque.

Può esserci approssimazione migliore? Molto probabilmente no!

Teorema: finchè non si dimostra che $P=NP$ non c'è una ρ -approximation per il problema center-selection per ogni $\rho < 2$.

LEZ22- 28/05/2024 (Hash tables - Randomized implementation of dictionaries)

IL PROBLEMA DEL DIZIONARIO

Dato un universo U di possibili elementi, mantenere un sottoinsieme $S \subseteq U$ soggetto alle seguenti operazioni:

- make-dictionary(): Inizializza un dizionario vuoto.
- insert(u): aggiungere l'elemento $u \in U$ a S .
- delete(u): cancellare u da S , se u è attualmente in S .
- look-up(u): determinare se u è in S .

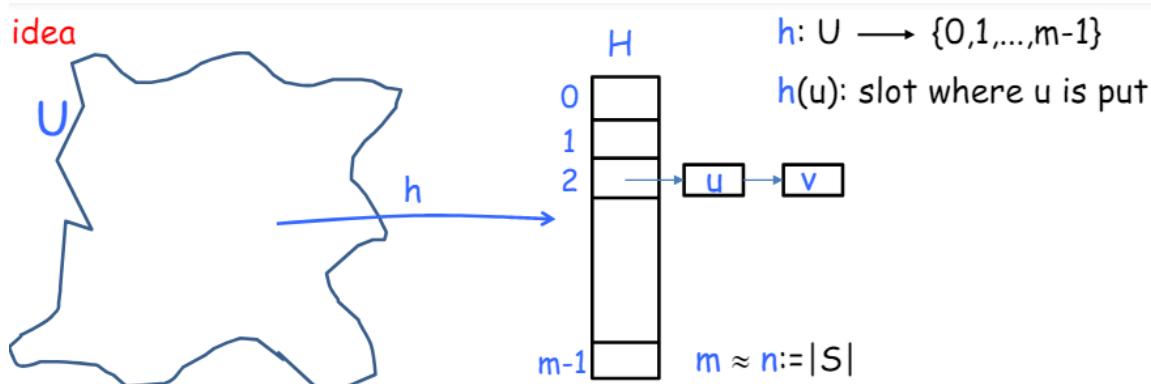
Sfida: L'Universo U può essere estremamente grande, quindi definire una matrice di dimensioni $|U|$ è impossibile.

Una soluzione: alberi bilanciati (ad es. AVL)

- Spazio $O(|S|)$
- $O(\log |S|)$ tempo per operazione

TABELLE HASH:

- Spazio $O(|S|)$
- $O(1)$ tempo previsto per ogni operazione



Collisione: quando $h(u)=h(v)$ ma $u \neq v$

$H[i]$: linked list di tutti gli elementi che h mappa nello slot i (hashing with chaining)

Inserti/Delete/Lookup di u :

- Calcolare $h(u)$
- Inserire/eliminare/cercare u scannerizzando la lista $H[h(u)]$

Goal: trovare una funzione h che "diffonde bene" gli elementi

SCEGLIERE UNA BUONA HASH FUNCTION

Obb: per ogni hash function deterministica si può trovare un insieme S dove tutti gli elementi di S sono mappati nello stesso slot $\Rightarrow \Theta(n)$ tempo per operazione

Idea per risolvere questo problema: usare la randomization (randomizzazione).

Approccio ovvio: per ogni u , scelgo $h(u)$ uniforme in modo randomico

- Look-up: dove inserisco u ? dobbiamo mantenere l'insieme di coppie $\{(u, h(u)): u \in S\}$

UNIVERSAL HASHING

Una famiglia H di funzioni hash è universale se per ogni $u, v \in U$ distinti vale che

$$\Pr_{h \in H}(h(u) = h(v)) \leq 1/m$$

TEOREMA

Sia H una famiglia di funzioni hash universali. Sia $S \subseteq U$ di n elementi. Sia $u \in S$.

Scegli una funzione random h da H , e sia X la variabile randomica che conta il numero di elementi di S mappati a $h(u)$.

Allora $E[X] \leq 1 + n/m$.

proof

$$\text{for each } s \in S \quad X_s \text{ r. v.} = \begin{cases} 1 & \text{if } h(s) = h(u) \\ 0 & \text{otherwise} \end{cases} \quad X = \sum_{s \in S} X_s$$

$$\begin{aligned} E[X] &= E\left[\sum_{s \in S} X_s\right] = \sum_{s \in S} E[X_s] = \sum_{s \in S} \Pr(h(s) = h(u)) \\ &= 1 + \sum_{s \in S \setminus \{u\}} \Pr(h(s) = h(u)) \leq 1 + n/m \end{aligned}$$



notice: $m = \Theta(n)$ expected $O(1)$ time per operation

PROGETTARE UNA FAMIGLIA UNIVERSALE DI FUNZIONI HASH

Table size: scegli m come un numero primo tale che $n \leq m \leq 2n$.

Integer encoding: identifica ogni elemento $x \in U$ con un intero base- m di r cifre:

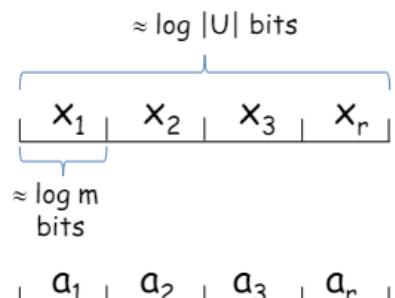
$$x = (x_1, x_2, \dots, x_r), x_i \in \{0, 1, \dots, m-1\}.$$

Hash function:

given $a \in U$, $a = (a_1, a_2, \dots, a_r)$

$$h_a(x) = \left(\sum_{i=1}^r a_i x_i \right) \bmod m$$

hash function family: $\mathcal{H} = \{h_a : a \in U\}$



word RAM model:

- manipolazione $O(1)$ parole macchina richiede tempo $O(1)$
 - ogni oggetto di interesse si inserisce in una parola macchina
- =>

Memorizzare ha richiede solo la memorizzazione del singolo valore a (1 parola macchina)
- calcolare $h_a(x)$ richiede tempo $O(1)$

TEOREMA

$H = \{h_a : a \in U\}$ è universale.

proof

Let $x = (x_1, x_2, \dots, x_r)$ and $y = (y_1, y_2, \dots, y_r)$ be two distinct elements of U .
We need to show that $\Pr[h_a(x) = h_a(y)] \leq 1/m$.

since $x \neq y$, there exists an integer j such that $x_j \neq y_j$.

we have $h_a(x) = h_a(y)$ iff

$$a_j \underbrace{(y_j - x_j)}_z = \underbrace{\sum_{i \neq j} a_i(x_i - y_i)}_{\alpha} \pmod{m}$$

we can assume a was chosen uniformly at random by first selecting all coordinates a_i where $i \neq j$, then selecting a_j at random. Thus, we can assume a_i is fixed for all coordinates $i \neq j$.

since m is prime & $z \neq 0$, z has a multiplicative inverse z^{-1} , i.e. $z z^{-1} = 1 \pmod{m}$

$$a_j = z^{-1} \underbrace{\sum_{i \neq j} a_i(x_i - y_i)}_{\alpha} \pmod{m}$$

→ $\Pr[h_a(x) = h_a(y)] \leq 1/m$.

UN'ALTRA FUNZIONE HASH UNIVERSALE (PIU' UTILIZZATA)

Scegli un primo $p \geq |U|$ (una volta).

Hash function: dati $a, b \in U$, $h_{ab}(x) = [(ax + b) \pmod{p}] \pmod{m}$

Hash function family: $H = \{h_{ab} : a, b \in U\}$

COME SCEGLIERE (DINAMICAMENTE) LA DIMENSIONE DELLA TABELLA

avviso: S cambia nel tempo e vogliamo usare spazio $O(|S|)$

parametri:

- n : # degli elementi attualmente presenti nella tabella, i.e. $n=|S|$;
- N : dimensione virtuale della tabella
- m : dimensione effettiva della tabella (numero primo tra N e $2N$)

Tecnica di raddoppio/dimezzamento:

- init $n=N=1$;
- ogniqualvolta $n>N$:
 - o $N:=2N$
 - o scegliere una nuova m
 - o re-hash tutti gli elementi (in tempo $O(n)$)
- ogniqualvolta $n<N/4$:
 - o $N:=N/2$
 - o scegliere una nuova m
 - o rieseguire tutte le voci (in $O(n)$ tempo)

=> $O(1)$ tempo ammortizzato per inserimento/cancellazione

IMPORTANZA ALGORITMI RANDOMIZZATI

Permette di risolvere efficacemente molti problemi pratici importanti tra cui i seguenti.

- Contare elementi distinti in uno stream:
dato uno stream di m elementi x_1, x_2, \dots, x_m dove ogni $x_i \in U$, ritornare il numero d di elementi distinti.
Soluzione semplice: spazio $O(d)$ oppure spazio $O(|U|)$
Contatori probabilistici: spazio $O(\log d)$
- Trovare elementi simili tra loro
Dati N elementi, trovare loro coppie la cui somiglianza è al di sopra di una soglia data.
Sfida principale: N è enorme ed è impossibile una soluzione $\Theta(N^2)$
Tecnica LSH: risolvere il problema in tempo e spazio $O(N \text{ polylog } N)$

VISTA DALL'ALTO ALGO2

1. GREEDY
 - 1.1 INTERVAL SCHEDULING & INTERVAL PARTITIONING
 - 1.2 UNION FIND
 - 1.2.1 STRUTTURE DATI PER UNION FIND: QUICKFIND E QUICKUNION
 - 1.3 MST (GRAFO PESATO E NON ORIENTATO, SPANNING TREE COSTO MINIMO)
 - 1.3.1 KRUSKAL
 - 1.3.2 PRIM
 - 1.4 CLUSTERING CON MASSIMO SPACING
2. PROGRAMMAZIONE DINAMICA
 - 2.1 WEIGHTED INDEPENDENT SET ON PATHS
 - 2.2 WEIGHTED INTERVAL SCHEDULING
 - 2.3 LONGEST INCREASING SUBSEQUENCE
 - 2.4 HOUSE COLORING PROBLEM
 - 2.5 SEGMENTED LEAST SQUARES
 - 2.6 KNAPSACK
 - 2.7 SEQUENCE ALIGNMENT
 - 2.8 SHORTEST PATH WITH NEGATIVE WEIGHTS
 - 2.8.1 ALGORITMO BELLMAN-FORD
- metà corso -----
3. FLUSSO
 - 3.1 MAX-FLOW/MIN-CUT PROBLEMS
 - 3.2 FORD-FULKERSON (correttezza e complessità)
 - 3.3 APPLICAZIONI.
 - 3.3.1 MATCHING SU GRAFI BIPARTITI
 - 3.3.2 CAMMINI ARCO-DISGIUNTI
 - 3.3.3 IMAGE SEGMENTATION
 - 3.3.4 BASEBALL-ELIMINATION
4. NP-COMPLETEZZA
 - 4.1 RIDUZIONI POLINOMIALI (CONFRONTI)
 - 4.2 P, NP, EXP, RELAZIONI
 - 4.3 PvsNP, NP-COMPLETEZZA, THM. COOK-LEVIN
 - 4.4 RIDUZIONI POLINOMIALI E PROBLEMI:
 - 4.4.1 SAT
 - 4.4.2 3SAT
 - 4.4.3 VERTEX-COVER
 - 4.4.4 INDEPENDENT SET
 - 4.4.5 (SUPER MARIO)
 - 4.4.6 (TETRIS)
 - 4.4.7 2-PARTITION
 - 4.4.8 3-PARTITION
 - 4.4.9 (3D-MATCHING)
5. ALGORITMI DI APPROSSIMAZIONE
 - 5.1 COSA SONO
 - 5.2 PERCHE' HA SENSO PROGETTARLI
 - 5.3 2-APX LOAD BALANCING
 - 5.4 3/2-APX LOAD BALANCING
 - 5.5 2-APX K-CENTER PROBLEM
6. TABELLE HASH