

# ADVANCED TOPICS ON ALGORITHMS

## 1. ALGORITMI DI APPROSSIMAZIONE

LEZ1

- INTRO ALG.APPROX.
- MINIMUM VERTEX COVER PROBLEM (MIN CARDINALITY)
  - ALGORITMO PER CARDINALITY VERTEX COVER (2-APPROX)
- MINIMUM SET COVER PROBLEM (MINIMIZZARE COSTO)
  - ALGORITMO GREEDY PER SET COVER ( $H_n$ -APPROX, CON  $H_n = 1 + 1/2 + \dots + 1/n$ )

LEZ2

- MINIMUM STEINER TREE PROBLEM (STP)
  - METRIC STEINER TREE PROBLEM (METRIC STP)
    - ALGORITMO PER STP (2-APPROX)
  - TRAVELING SALESMAN PROBLEM (TSP)
    - METRIC TRAVELING SALESMAN PROBLEM (METRIC TSP)
      - ALGORITMO PER METRIC TSP (2-APPROX)
      - ALGORITMO DI CHRISTOFIDES PER METRIC TSP (3/2-APPROX)

LEZ3

- MINIMUM SET COVER PROBLEM (SC)
  - FORMULAZIONE IN PROGRAMMAZIONE LINEARE(LP) DI SET COVER PROBLEM
    - ALGORITMO TRAMITE LP-ROUNDING PER SET COVER (2-APPROX)
  - LP-DUALITY (PROGRAMMAZIONE LINEARE - DUALITA')
  - SET COVER TRAMITE DUAL-FITTING
    - ALGORITMO GREEDY PER SET COVER ( $H_n$ -APPROX)

LEZ4

- SCHEMA PRIMALE DUALE
  - ALGORITMO (F-APPROX) PER SET COVER PASSANTE PER PRIMALE DUALE
  - STEINER FOREST PROBLEM
    - FORMULAZIONE PROGRAMMAZIONE LINEARE(ILP) DI STEINER FOREST
    - ALGORITMO PER PROBLEMA STEINER FOREST (2-APPROX)

## 2. ALGORITMI PARAMETRIZZATI

LEZ5

- K-VERTEX COVER PROBLEM
  - ALGORITMO FORZA BRUTA (TEMPO ESECUZIONE PESSIMO)
  - ALGORITMO BOUNDED-SEARCH TREE (ALGORITMO FPT)
- FPT (FIXED PARAMETER TRACTABLE)
- KERNELIZATION
- KERNEL POLINOMIALE PER K-VERTEX COVER
- W[1]-HARDNESS

LEZ6

- K-PATH
- COLOR-CODING
  - ALGORITMO PROGRAMMAZIONE DINAMICA
- KERNEL DI DIMENSIONE PARI A 2K-VERTICI PER VERTEX COVER TRAMITE LP
  - ALGORITMO DI KERNELIZATION

LEZ7

- PARTY PROBLEM
- ALGORITMO DI PROG.DINAMICA PER WEIGHTED INDIPENDENT SET SU ALBERI
  - ALBERI PARTENDO DA UN GRAFO
    - TREE DECOMPOSITION
    - WIDTH (PROPRIETA' DI TREE DECOMPOSITION)
    - TREewidth
    - LEMMI UTILI
    - TREE DECOMPOSITION RIDONDANTE
  - ALGORITMO PROG.DINAMICA SU GRAFO CON LARGHEZZA DELL'ALBERO LIMITATA W (PER RISOLVERE WEIGHTED INDIPENDENT SET)

LEZ8

- COMPLESSITA' PARAMETRIZZATA
- RIDUZIONI PARAMETRIZZATE
- MULTICOLORED CLIQUE
- K-DOMINATING SET
- PROBLEMI DIFFICILI E OSSERVAZIONI SU DI ESSI
- CIRCUIT SATISFIABILITY
  - WEIGHTED CIRCUIT SATISFIABILITY
  - DEPTH E WEFT DI UN CIRCUITO
- W-GERARCHIA
- ESPOENTIAL TIME HIPOTESIS (ETH)
- TRASFERIMENTO LOWER BOUNDS
- STRONG ETH (SETH)

## 3. STRUTTURE DATI AVANZATE

LEZ9

- PROBLEMA: CONTARE IL NUMERO DI 1 IN UNA FINESTRA
- STRUTTURA DATI DGIM
- PROBLEMA: TROVARE GLI ELEMENTI PIU' FREQUENTI IN UNO STREAM
  - STICKY SAMPLING ALGORITHM

LEZ10

- PROBLEMA LOWEST COMMON ANCESTOR (LCA)
- PROBLEMA RMQ
- RIDUZIONE DI LCA QUERIES A RMQ
- SOLUZIONI AL PROBLEMA RMQ
- LEVEL ANCESTOR
- JUMP POINTERS
- LONG PATH DECOMPOSITION
- LONG PATH DECOMPOSITION + LADDERS
- LONG PATH DECOMPOSITION + LADDERS + JUMP POINTERS
- MACRO-MICRO TREES

LEZ11

- RANGE TREES
- IL PROBLEMA FRACTIONAL CASCADING
- CROSS LINKING (IDEA1)
- LA TECNICA FRACTIONAL CASCADING (IDEA2)
- LAYERED RANGE TREES (PER DIMENSIONE=2)

LEZ12

- NON CARICATA

## OVERVIEW

1. Algoritmi di approssimazione:
  - campo ben consolidato
  - approccio ampiamente utilizzato per problemi (NP-)difficili (NP-HARD)
  - tecniche interessanti: rounding (arrotondamento), dual-fitting, approccio primale-duale
2. Algoritmi parametrizzati:
  - Analisi multivariata di algoritmi
  - Nozioni raffinate di efficienza e hardness
  - Tecniche interessanti: color coding, kernelization, treewidth
3. Strutture dati avanzate:
  - nucleo principale di problemi algoritmici
  - DSs per dati geometrici, big data, static trees, stringhe
  - tecniche interessanti: fractional cascading (cascata frazionaria), indirection

## ALGORITMI DI APPROXIMAZIONE

Un algoritmo di  $\alpha$ -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  $\alpha$  rispetto al valore di una soluzione ottimale.

$\alpha$ : rapporto di approssimazione o fattore di approssimazione

problema di minimizzazione:

- $\alpha \geq 1$
- per ogni istanza  $x$ , la soluzione restituita  $s$  ha costo( $s$ )  $\leq \alpha \text{OPT}(x)$

problema di massimizzazione:

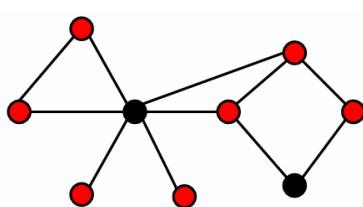
- $\alpha \leq 1$
- per ogni istanza  $x$ , la soluzione restituita  $s$  ha valore( $s$ )  $\geq \alpha \text{OPT}(x)$

## MINIMUM VERTEX COVER PROBLEM (MIN CARDINALITY)

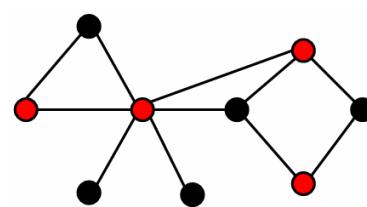
Input = un grafo non diretto  $G=(V,E)$

Soluzione accettabile:  $U \subseteq V$  tale che ogni arco  $(u,v) \in E$  è coperto, cioè  $u \in U$  oppure  $v \in U$

Misura da minimizzare: cardinalità di  $U$



a vertex cover of size 7



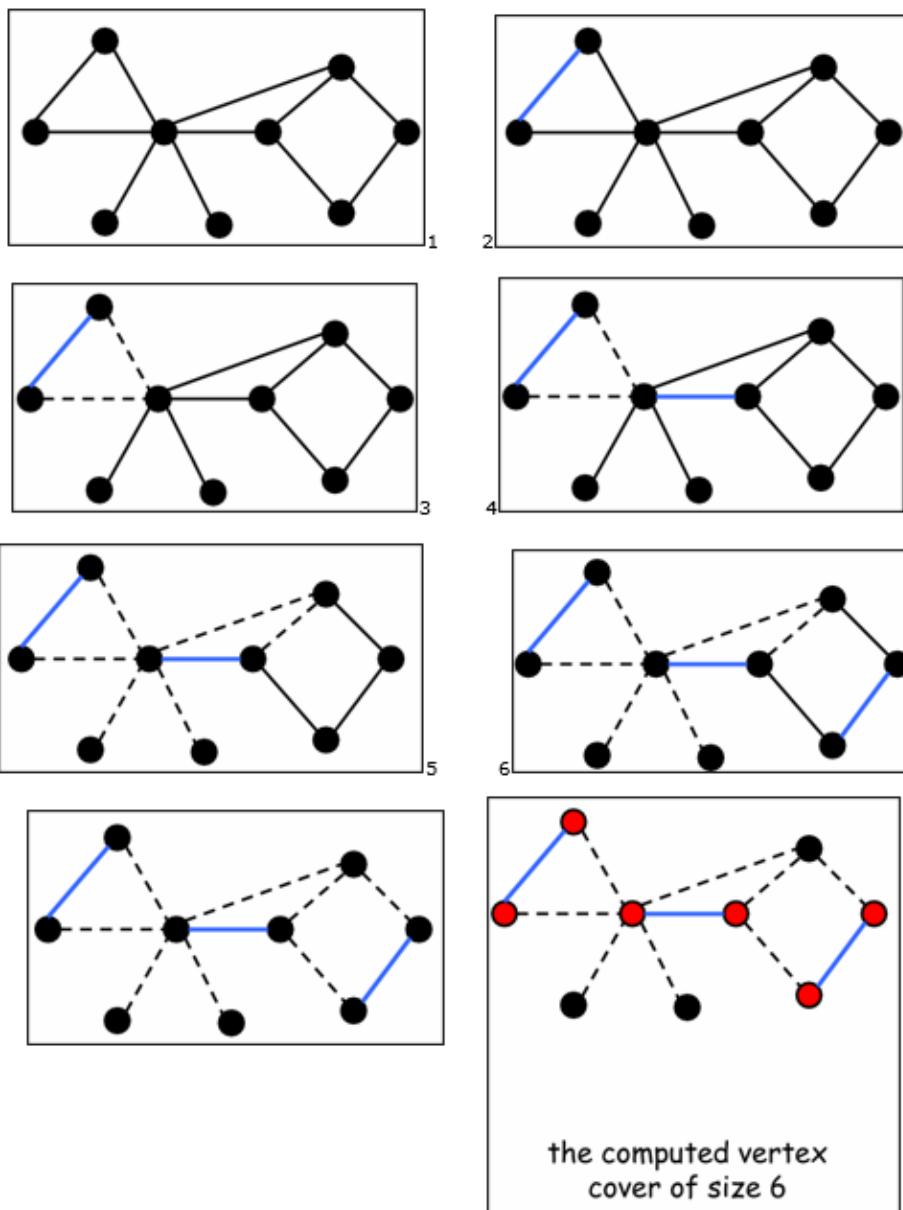
a better vertex cover of size 4

DEF (Matching): Dato un grafo  $G=(V,E)$ , un sottoinsieme degli archi  $M \subseteq E$  è un matching se non ci sono due archi in  $M$  che condividono uno stesso nodo (endpoint).

DEF (Matching massimale): Un matching  $M \subseteq E$  si dice massimale se per ogni arco  $e \in E \setminus M$ ,  $M \cup \{e\}$  non è un matching.

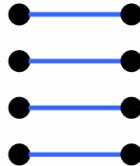
## ALGORITMO PER CARDINALITY VERTEX COVER (Algorithm 1.2)

Trovare un matching massimale in  $G$  e restituire in output un insieme di vertici matchati.



LEMMA: L'algoritmo restituisce un VC (Vertex Cover) fattibile.

- Dim: sia  $M \subseteq E$  il matching massimale calcolato dall'algoritmo



Gli archi in  $M$  sono chiaramente coperti. Per la massimalità di  $M$  qualsiasi altro arco  $(x,y)$  condivide un endpoint con un certo arco in  $M$  ... e quindi è coperto.

THM: l'algoritmo è una 2-approximazione per il problema VC

- Dim: la soluzione ritornata è un possibile VC (lemma precedente)

Sia  $M \subseteq E$  il matching massimale calcolato dall'algoritmo, e sia  $U$  il corrispondente VC  
qualsiasi soluzione ottima deve avere dimensione  $OPT$  almeno  $|M|$

Schema di lower bounding: la dimensione di un qualsiasi matching massimale è un lower bound alla dimensione di un VC ottimale

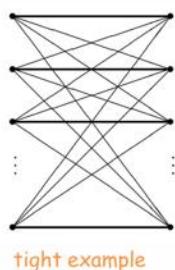
dunque:  $|U| = 2|M| \leq 2OPT$

#### TRE DOMANDE IMPORTANTI DA FARSI IN QUESTI CASI:

- 1) Il rapporto di approssimazione dell'algoritmo 1.2 può essere migliorato da una analisi migliore?
- 2) È possibile progettare un algoritmo di approssimazione con un rapporto apx migliore utilizzando lo schema di lower bound dell'algoritmo 1.2, cioè la dimensione di un matching massimale?
- 3) Esiste un altro schema di lower bound che può portare a un algoritmo di approssimazione migliore per VC?

## Risposte:

1. Can the approximation ratio of Algorithm 1.2 be improved by a better analysis? **NO**



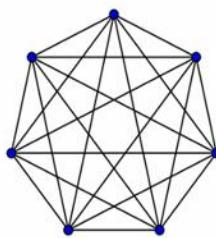
complete bipartite graph  $K_{n,n}$

Algorithm 1.2 will pick all the  $2n$  vertices

$OPT=n$  (one side is an optimal VC)

tight example

2. Can an approximation algorithm with a better apx ratio be designed using the lower bounding scheme of Algorithm 1.2, i.e. the size of a maximal matching? **NO**



complete graph  $K_n$  where  $n$  is odd

size of any maximal matching is  $(n-1)/2$

$OPT=n-1$

3. Is there some other lower bounding scheme that can lead to a better approximation algorithm for VC? **OPEN**

Partial answer:

Theorem

Assuming the [unique games conjecture](#) holds, if there exists an  $\alpha$ -approximation algorithm for the VC problem with  $\alpha < 2$ , then  $P=NP$ .

roughly: a particular problem (called unique games) is NP-hard

## MINIMUM SET COVER PROBLEM

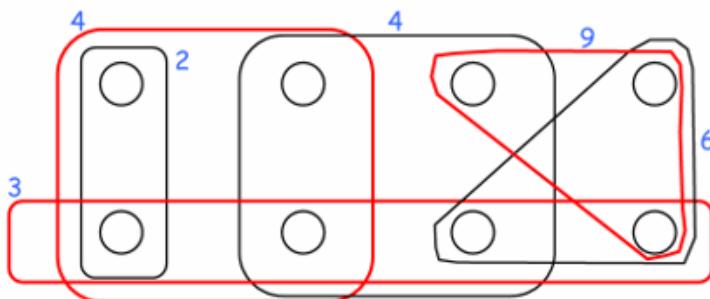
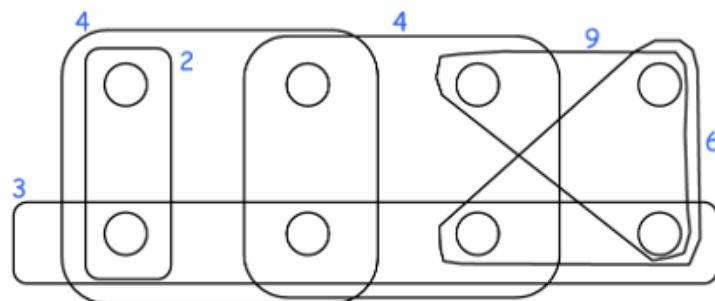
Input: Universo  $U$  di  $n$  elementi

Una collezione di sottoinsiemi di  $U$ ,  $S=\{S_1, \dots, S_k\}$

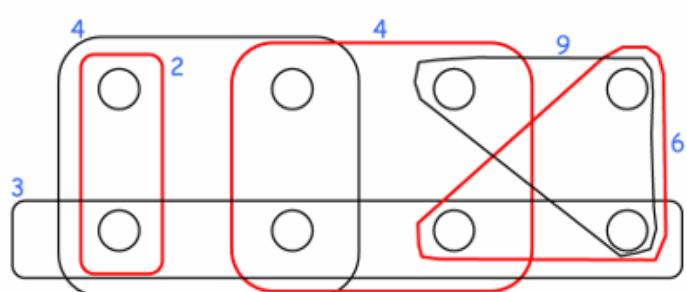
Ogni  $S \in S$  ha un costo positivo  $c(S)$

Soluzione fattibile: un sottoinsieme  $C \subseteq S$  che copre  $U$  (la cui unione è  $U$ )

Misura (da minimizzare): cost of  $C$ :  $\sum_{S \in C} c(S)$



a set cover of  
cost 16



a better set  
cover of cost 12

Strategia di tipo greedy: scegli l'insieme di cost-effective (costo effettivo) più conveniente e rimuovi gli elementi coperti, fino a quando tutti gli elementi sono coperti.

Sia  $C$  l'insieme degli elementi già trattati, allora il costo effettivo di  $S$  è  $c(S)/|S-C|$  (costo medio per il quale  $S$  copre i nuovi elementi)

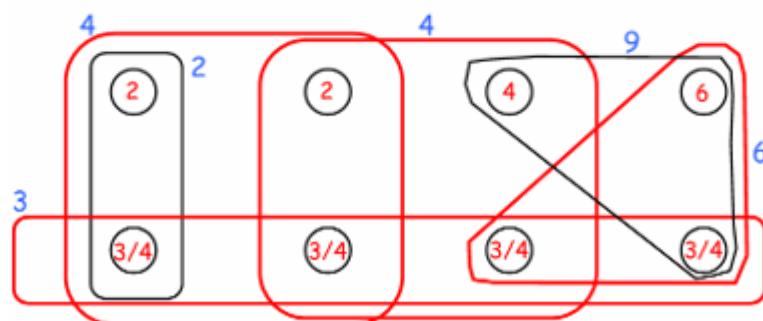
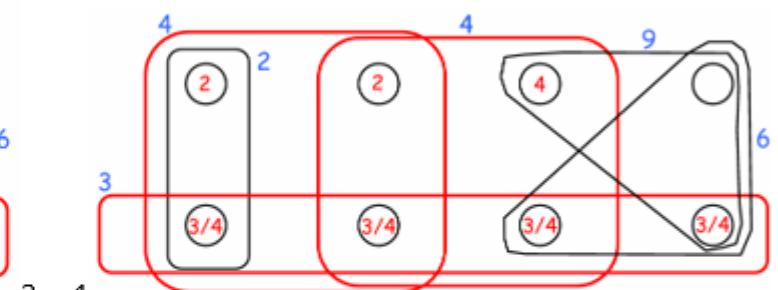
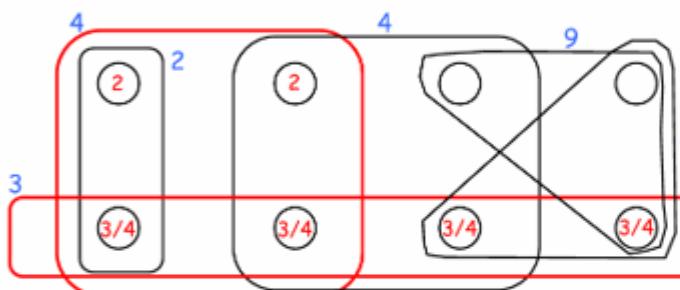
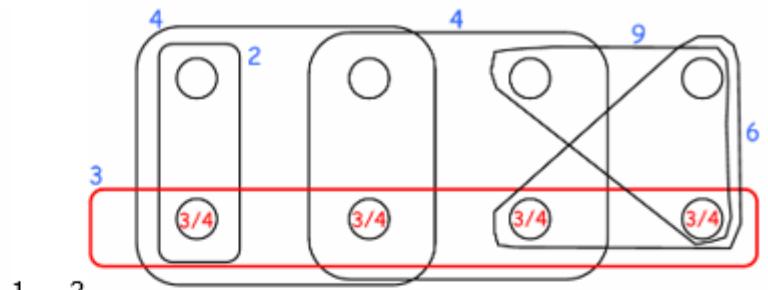
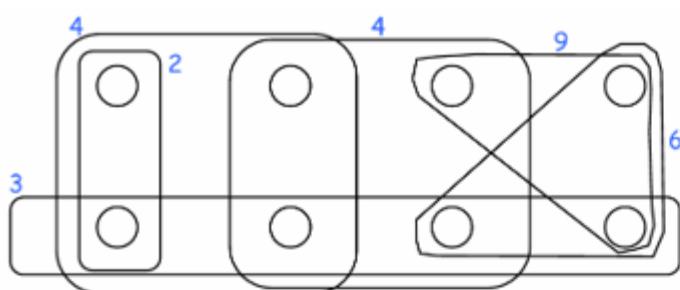
### ALGORITMO GREEDY SET COVER (Algorithm 2.2)

#### Algorithm 2.2 (Greedy set cover algorithm)

1.  $C \leftarrow \emptyset$
2. While  $C \neq U$  do
  - Find the most cost-effective set in the current iteration, say  $S$ .
  - Let  $\alpha = \frac{\text{cost}(S)}{|S-C|}$ , i.e., the cost-effectiveness of  $S$ .
  - Pick  $S$ , and for each  $e \in S - C$ , set  $\text{price}(e) = \alpha$ .
  - $C \leftarrow C \cup S$ .
3. Output the picked sets.

average cost at which  
e is covered

(Costo medio al  
quale e è coperto)



the computed set  
cover of cost 17

Numera gli elementi di U nell'ordine in cui sono stati coperti, risolvendo arbitrariamente i legami.

Sia  $e_1, \dots, e_n$  en questa numerazione:

LEMMA: Per ogni  $k \in \{1, \dots, n\}$ ,  $\text{price}(e_k) \leq \text{OPT}/(n-k+1)$

- Dim: ad ogni iterazione, gli insiemi rimanenti della soluzione ottimale possono coprire tutti gli elementi rimanenti  $C' = U - C$  con costo al più  $\text{OPT}$

Uno di questi insiemi rimanenti ha costo effettivo al più  $\text{OPT}/|C'|$

All'iterazione in cui  $e_k$  è coperto,  $C'$  contiene almeno  $n-k+1$  elementi.

Per l'algoritmo greedy:  $\text{price}(e_k) \leq \text{OPT}/|C'| \leq \text{OPT}/(n-k+1)$

TEOREMA: l'algoritmo greedy è un algoritmo con fattore di approssimazione  $H_n$  per il problema del Minimum Set Cover, dove  $H_n = 1 + 1/2 + \dots + 1/n$ .

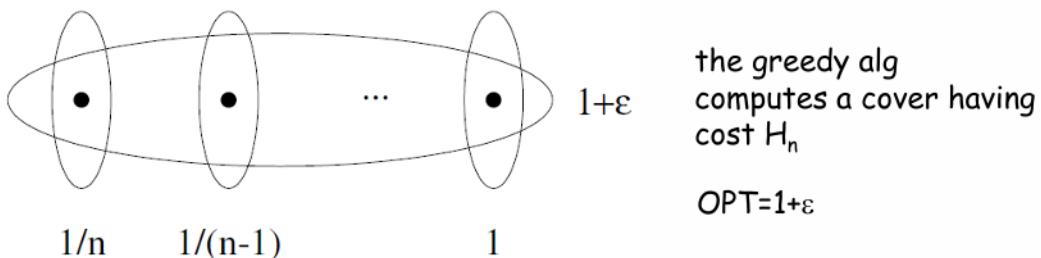
- Dim: poiché il costo di ogni insieme preso è distribuito tra i nuovi elementi coperti allora

$$\text{cost of the cover} = \sum_{k=1}^n \text{price}(e_k) \leq \sum_{k=1}^n \text{OPT}/(n-k+1) \leq H_n \text{OPT}$$

$$H_n = \sum_{k=1}^n \frac{1}{k} \leq \ln n + 1 \quad n\text{-th harmonic number}$$

ESEMPIO TIGHT:

tight example

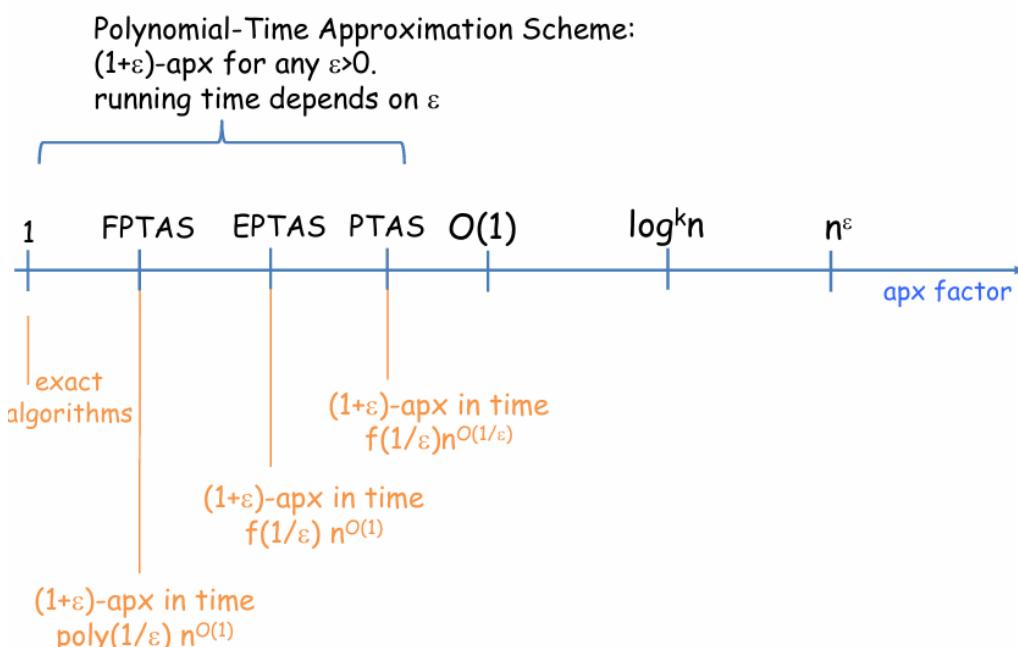


TEOREMA: esiste una qualche costante  $c > 0$  tale che se esiste un algoritmo di approssimazione  $(c \ln n)$ -apx per il problema del SC(Set Cover) non pesato, allora  $P=NP$

TEOREMA: se esiste un algoritmo di approssimazione  $(c \ln n)$ -apx per il problema unweighted SC, per qualche costante  $c < 1$ , allora c'è un algoritmo di tempo di tempo  $O(n^{O(\log \log n)})$  per ogni problema NP-completo.

### THE APPROXIMATION GAME (IL GIOCO DELL'APPROSSIMAZIONE)

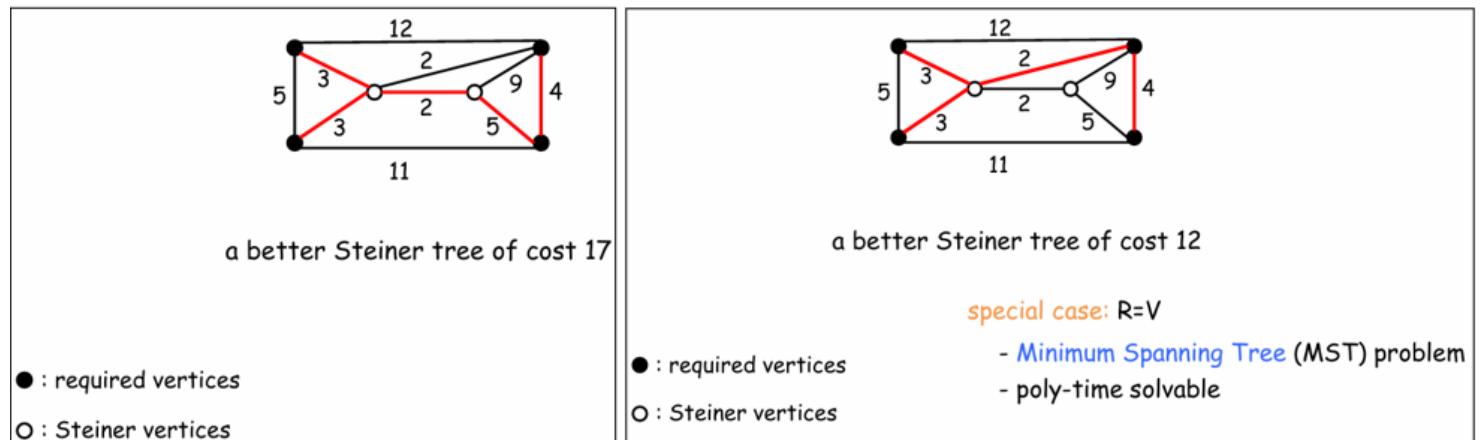
Migliorare e migliorare ancora il fattore di approssimazione.



MINIMUM STEINER TREE PROBLEM (Problema dell'albero di Steiner)Input: grafo non diretto  $G=(V,E)$  con archi di costo non negativoSottoinsieme di vertici detti required  $R \subseteq V$ ; i vertici in  $V-R$  sono chiamati vertici di SteinerSoluzione ammissibile: un albero  $T$  contenente tutti i vertici required e un sottoinsieme qualsiasi dei vertici di Steiner

Misura (da minimizzare):

$$\text{cost of } T : \sum_{e \in E(T)} c(e)$$



Passaggio ad istanze metriche passando al problema Metric Steiner Tree

METRIC STEINER TREE PROBLEM (Problema dell'albero di Steiner metrico)

Il problema è lo stesso di prima, ma abbiamo che:

- $G$  è completo
- Costi degli archi rispettano la diseguaglianza triangolare, vale a dire che per ogni  $u,v,w$  vale che:

$$c(u,v) \leq c(u,w) + c(w,v)$$

[ Abbiamo quindi una sottoclasse di istanze del problema generale (Minimum Steiner Tree problem)  
Perché passare al caso metrico? Perchè il caso metrico "cattura" tutta la difficoltà del problema risolvendo l'istanza metrica più semplice e poi ritrasformando la soluzione nell'istanza originale senza costi extra ]

TEOREMA: Esiste un fattore di approssimazione che preserva la riduzione dal problema dell'albero di Steiner al problema dell'albero di Steiner metrico.

- Dim: sia  $I$  un'istanza del problema ST(Steiner Tree) costituito dal grafo  $G=(V,E)$  e dai vertici required  $R$ , allora in tempo polinomiale abbiamo:

istanza  $I'$  del problema ST metrico, con  $G'=(V,E')$  completo; $c' = (u,v)$  in  $G' = \text{cost}$  di un qualunque  $u-v$  shortest path in  $G$  $R' = R$ Poiché per ogni  $(u,v) \in E$ ,  $c'(u,v) \leq c(u,v)$ ,  $\text{OPT}(I') \leq \text{OPT}(I)$ 

→ CONVERSIONE IN ISTANZA METRICA:

Uno Steiner tree  $T'$  di  $I'$  può essere convertito in tempo polinomiale in uno Steiner Tree  $T$  di  $I$  con al più lo stesso costo nel modo seguente:

1. sostituire ogni arco  $(u,v)$  di  $T'$  con il percorso più breve in  $G$
2. selezionare qualsiasi spanning tree  $T$  del sottografo ottenuto di  $G$

$$\text{cost}(T) \leq \text{cost}(T')$$

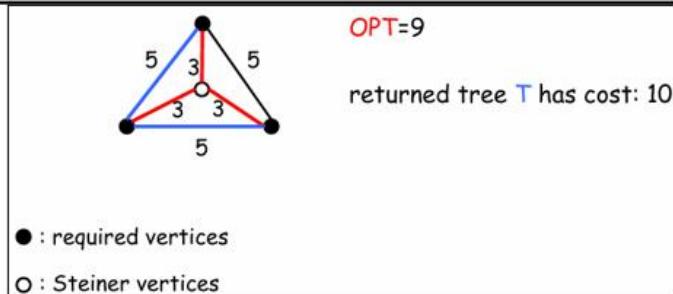
## ALGORITMO PER PROBLEMA METRICO ST

Funzionamento:

1. Prendo i nodi required
2. Guardo solo il sottografo indotto dai nodi required (mi dimentico degli altri)
3. Calcolo su esso il minimum spanning tree (MST)
4. Restituisco MST

### Algorithm

output a Minimum Spanning Tree (MST) of the subgraph  
of G induced by R

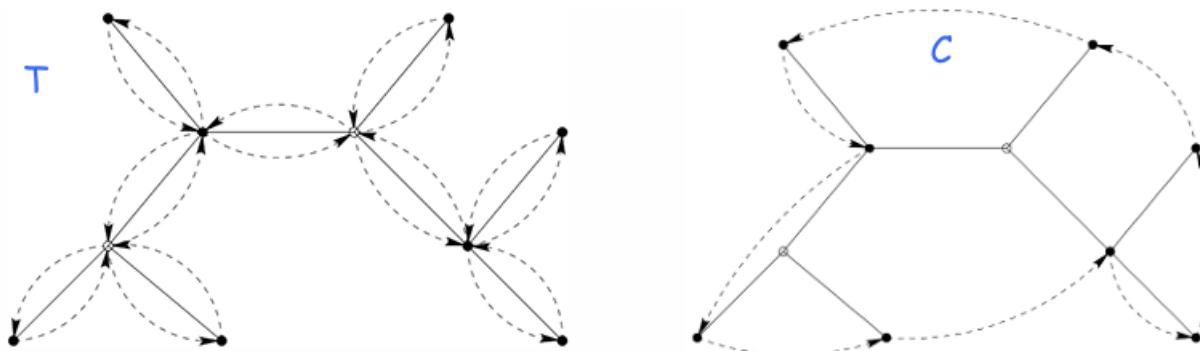


TEOREMA: l'algoritmo è una 2-approximazione per il problema metrico ST.

- Dim: sia T uno Steiner tree ottimo di costo OPT, e sia M l'MST su R.

Raddoppio gli archi di T ottenendo un grafo Euleriano di costo  $2\text{OPT}$

Considero un tour Euleriano di costo  $2\text{OPT}$



Come si ottiene ciclo hamiltoniano C su R a partire da T?

Si ottiene attraversando il tour euleriano e "saltando" i vertici di Stainer evitandoli in modo da "toccare" soltanto i vertici precedentemente visitati di R (faccio lo "shortcut").

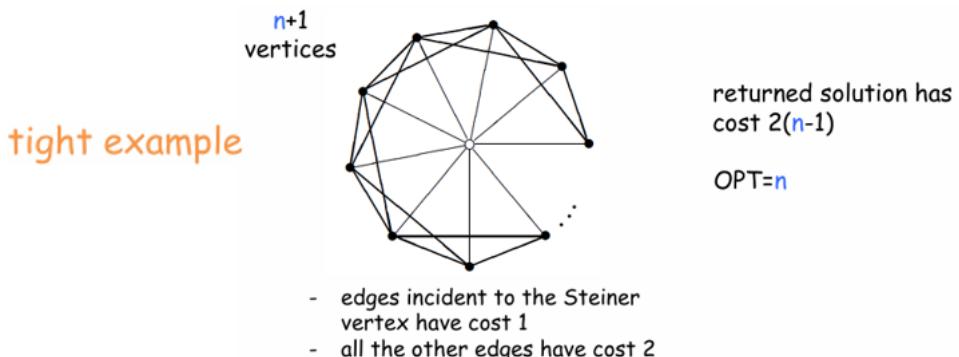
Ora C tocca soltanto i nodi required.

Per diseguaglianza triangolare:  $\text{cost}(C) \leq 2\text{OPT}$

Poiché C è un sottografo esteso di  $G[R]$ :  $\text{cost}(M) \leq \text{cost}(C)$

( $G[R] = \text{grafo indotto da } R$ )

ESEMPIO TIGHT:

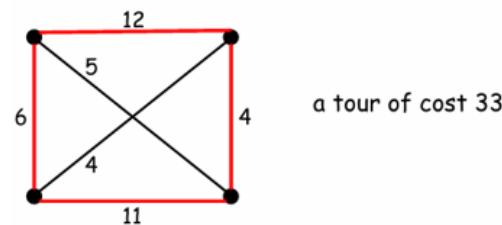
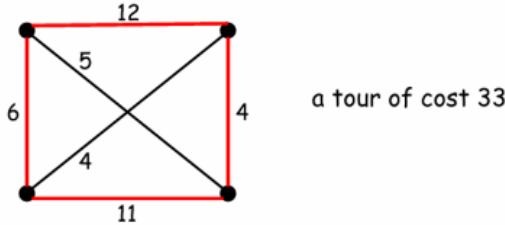


## TRAVELING SALESMAN PROBLEM (TSP) (PROBLEMA DEL COMMESO VIAGGIATORE)

Input: grafo completo non diretto  $G=(V,E)$  con pesi degli archi non negativi

Soluzione ammissibile: un ciclo  $C$  che visita ogni vertice esattamente una volta

Misura (da minimizzare): il costo del ciclo, ossia  $\text{cost of } C : \sum_{e \in E(C)} c(e)$



Anche per questo problema, come per il precedente, passiamo attraverso l'istanza metrica per facilitarci

## METRIC TSP (METRIC TRAVELING SALESMAN PROBLEM)

L'istanza metrica parte dal grafo completo non orientato, in più quest'ultimo deve rispettare la disugualanza triangolare, ossia per ogni  $u,v,w$  vale che  $c(u,v) \leq c(u,w) + c(w,v)$

**TEOREMA:** Per qualsiasi funzione calcolabile in tempo polinomiale  $\alpha(n)$ , TSP non può essere approssimato entro un fattore di  $\alpha(n)$ , a meno che  $P=NP$ .

- Dim: per contraddizione: sia  $A$  un algoritmo  $\alpha(n)$ -apx.

Usiamo  $A$  per decidere il ciclo hamiltoniano.

Sia  $G$  un'istanza del ciclo hamiltoniano. Definiamo  $G'$  nel modo seguente:

- o  $G'=(V,E')$  è completo;
- o  $c(u,v)=1$  se  $(u,v) \in E(G)$ ;  $c(u,v)=n\alpha(n)$  altrimenti

Chiaramente:

- o se  $G$  ha un ciclo hamiltoniano, allora il tour ottimale di TSP in  $G'$  costa  $n$
  - o se  $G$  non ha un ciclo hamiltoniano, allora il tour ottimale di TSP ha un costo  $> n\alpha(n)$
- => $G$  ha un ciclo hamiltoniano se  $A$  restituisce un tour di costo  $n$

**NOTA:** mentre nello Steiner Tree le istanze metriche catturano la difficoltà del problema, nel caso del TSP invece succede il contrario. Ossia se non ho istanza metrica non otterrò alcuna approssimazione, cioè se voglio risolvere il TSP per istanze non metriche non posso approssimare.

## ALGORITMO PER METRIC TSP – FACTOR2

Funzionamento:

1. Sì parte dall'istanza metrica
2. Prendo il grafo  $G$  metrico completo, su questo trovo l'MST denominandolo  $T$  (questo MST tocca tutti i nodi e ha costo totale minimo)
3. Raddoppio gli archi di  $T$  e trovo un circuito euleriano che prende il nome di  $\tau$  (tau)
4. A questo punto potrei avere che sto attraversando un vertice più di una volta (cosa che non voglio), per evitarlo eseguo la tecnica di shortcut vista prima secondo cui attraverso  $\tau$  e ogni volta che  $\tau$  mi porta su un vertice già visitato lo salto e vado su un nodo ancora non visitato (applicando così lo shortcut)
5. Restituisco ciò che ho ottenuto dopo il passo 4

### **Algorithm (metric TSP – factor 2)**

1. Find an MST  $T$  of  $G$
2. Double every edge of  $T$  to obtain an Eulerian graph
3. Find an Eulerian tour  $\tau$  on this graph
4. Output the tour that visits vertices of  $G$  in the order of their first appearance in  $\tau$ . Let  $C$  be this tour.

TEOREMA: Questo algoritmo è un algoritmo 2-approssimato per il TSP metrico.

- Dim: rimuovere un arco da un tour ottimale in TSP ci dà uno spanning tree di G

Quindi:  $\text{cost}(T) \leq \text{OPT}$

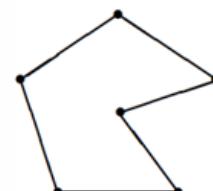
Abbiamo:  $\text{cost}(C) \leq \text{cost}(\tau) = 2\text{cost}(T) \leq 2\text{OPT}$

ESEMPIO TIGHT:

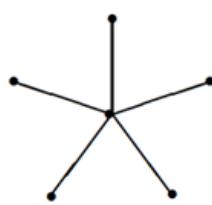
tight example



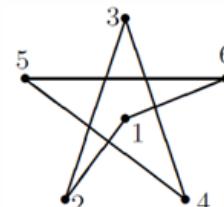
- n vertices
- think edges have cost 1  
(star+(n-1)-cycle)
- all the other edges have cost 2



optimal tour of cost  $\text{OPT}=n$



feasible MST



returned tour of cost  $2n-2$   
(for the feasible specified order)

### ALGORITMO DI CHRISTOFIDES PER METRIC TSP – FACTOR 3/2

Idea: trovare un sottografo/tour euleriano più economico. "Se trovo un circuito euleriano che costa meno di 2 volte ottimo allora forse riesco a migliorare l'approssimazione".

Ricorda: un grafo è Euleriano se e solo se tutti i vertici hanno grado pari

Ricorda: in ogni grafo non orientato, il numero di vertici di grado dispari è pari

Funzionamento:

1. Trovo un MST  $T$  del grafo  $G$
2. Sia  $V'$  un sottoinsieme dei nodi contenente solo nodi di grado, calcolo il minimum cost perfect matching (denominato)  $M$  su  $V'$ . Aggiungo poi  $M$  a  $T$  e ottieni un grafo euleriano (un minimum cost perfect matching è un matching che tocca tutti i vertici, e di tutti questi matching voglio quello che minimizza la somma dei pesi degli archi ; è una generalizzazione del max-flow e si risolve in tempo polinomiale)
3. Trovo un tour euleriano (denominato)  $\tau$  su questo grafo ottenuto
4. Genero il tour che visita i vertici di  $G$  nell'ordine della loro prima apparizione in  $\tau$ . Lascio che  $C$  sia questo tour.

### **Algorithm** (metric TSP – factor 3/2)

1. Find an MST  $T$  of  $G$
2. Compute a minimum cost perfect matching ,  $M$ , on the set  $V'$  of odd-degree vertices of  $T$ . Add  $M$  to  $T$  and obtain an Eulerian graph
3. Find an Eulerian tour  $\tau$  on this graph
4. Output the tour that visits vertices of  $G$  in the order of their first appearance in  $\tau$ . Let  $C$  be this tour.

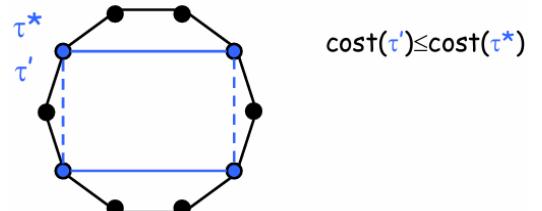
LEMMA: sia  $V' \subseteq V$ , tale che  $|V'|$  è pari, e sia  $M$  un minimum cost perfect matching su  $V'$ .

Allora,  $\text{cost}(M) \leq \text{OPT}/2$ .

- Dim: sia  $\tau^*$  un TSP ottimale del costo  $\text{OPT}$ .

sia  $\tau'$  il tour su  $V'$  ottenuto dopo lo shortcutting su  $\tau^*$ .

$\tau'$  è l'unione di 2 perfect matching su  $V'$ , chiamati  $M_1$  e  $M_2$ .

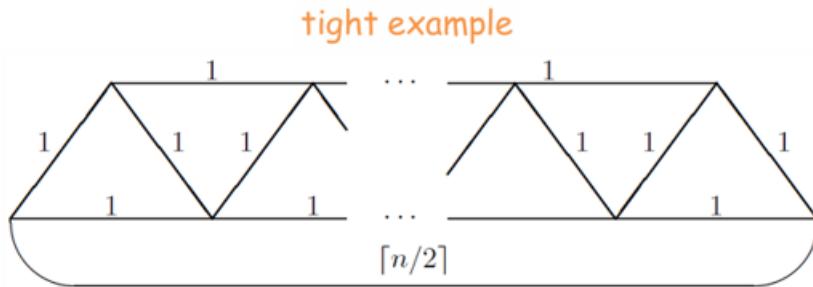


$$\text{cost}(M) \leq \min\{\text{cost}(M_1), \text{cost}(M_2)\} \leq 1/2(\text{cost}(\tau')) \leq 1/2(\text{OPT})$$

TEOREMA: l'algoritmo di Christofides è una  $3/2$  approssimazione per il metric TSP.

- Dim: abbiamo  $\text{cost}(C) \leq \text{cost}(\tau) = \text{cost}(T) + \text{cost}(M) \leq \text{OPT} + 1/2(\text{OPT}) \leq 3/2(\text{OPT})$

ESEMPIO TIGHT:



- $n$  vertici con  $n$  numero dispari
- MST fattibile: un percorso di  $n-1$  archi
- matching: un singolo arco di costo  $[n/2]$  parte intera superiore  
 $\text{OPT}=n$   
 Ritorna tour di costo  $n-1+[n/2]$  parte intera superiore

### MINIMUM SET COVER PROBLEM

Input: insieme  $U$  di  $n$  elementi

una collezione di sottoinsiemi di  $U$  che chiamiamo  $S$ , con  $S = \{S_1, \dots, S_k\}$

ogni  $s \in S$  ha costo positivo  $c(s)$

Soluzione ammissibile: un sottoinsieme  $C \subseteq S$  che copre  $U$  (la cui unione è  $U$ )

Misura (da minimizzare): cost of  $C$ :  $\sum_{S \in C} c(S)$

Def: per frequenza di un elemento  $e$  si intende il numero di insiemi nell'input che lo contengono.

Per "f" si intende la frequenza dell'elemento più frequente (quindi la frequenza massima)

### FORMULAZIONE IN PROGRAMMAZIONE LINEARE INTERA (Integer Linear Programming) DEL PROBLEMA SC

$$\begin{aligned}
 & \text{minimize} && \sum_{S \in S} c(S)x_S \\
 & \text{(PER OGNI OGGETTO ABBIAMO IL VINCOLO:)} \\
 & \text{subject to} && \sum_{S: e \in S} x_S \geq 1 \quad e \in U \\
 & && \text{SE FACCIAMO LA SOMMA DI TUTTI GLI } x_S \text{ APPARTENENTI A } S \text{ QUESTO DEVE ESSERE } \geq 1 \\
 & && (x_S \text{ VARIABILE DI SCELTA}) \quad x_S \in \{0,1\} \quad S \in S \\
 & && \text{relax with} \\
 & && x_S \geq 0 \quad \& \quad x_S \leq 1 \\
 & && \text{( RIDONDANTE PERCHE' VOLENDO MINIMIZZARE NON CONVIENE MAI METTHERE A UNA VARIABILE UN VALORE } \geq 1 \text{ )} \\
 & && \text{redundant}
 \end{aligned}$$

Possiamo rilassare la condizione che impone valori interi dell'ILP passando alla formulazione LP-relaxtion seguente:

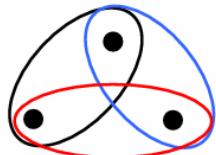
$$\begin{aligned}
 & \text{LP-relaxation} \\
 & \text{minimize} && \sum_{S \in S} c(S)x_S \\
 & \text{subject to} && \sum_{S: e \in S} x_S \geq 1 \quad e \in U \\
 & && x_S \geq 0 \quad S \in S \\
 & && \text{a feasible solution is} \\
 & && \text{a fractional SC} \\
 & && \text{OPT}_f: \text{cost of the min fractional SC}
 \end{aligned}$$

Inoltre, vale la relazione seguente:

$$\text{OPT}_f \leq \text{OPT}$$

PERCHE' A DESTRA CONTIENE PIU' SOLUZIONI AMMISSIBILI DI QUELLO A SINISTRA, E STIAMO MINIMIZZANDO. QUINDI LE SOLUZIONI EXTRA IN LP-RELAXTION SONO PIU' PICCOLE RISPETTO ALLE SOLUZIONI CONDIVISE CON ILP

Esempio:



OPT=2

OPT<sub>f</sub>=1.5

set all  $x_S$  to  $\frac{1}{2}$

- 3 elements
- 3 sets
- all sets have cost 1

Digressione: quali metodi conosci per risolvere un problema di programmazione lineare?

1. Metodo del simplesso (simplex algorithm), nel caso peggiore impiega tempo esponenziale ma in pratica ha costo quasi lineare
2. Metodo dell'ellissoide (ellipsoid method), nel caso peggiore ha costo polinomiale ma in pratica non è molto buono come metodo

### ALGORITMO PER SET COVER TRAMITE LP-ROUNDING (ALG 14.1)

Funzionamento:

1. Trova una soluzione ottimale per il rilassamento LP.
2. Scegli tutti gli insiemi  $S$  per i quali  $x_S \geq 1/f$  in questa soluzione

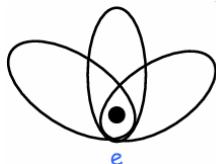
#### Algorithm 14.1 (Set cover via LP-rounding)

1. Find an optimal solution to the LP-relaxation.
2. Pick all sets  $S$  for which  $x_S \geq 1/f$  in this solution.

ALGORITMO DI APPROXIMAZIONE PER SET COVER, CHE PASSANDO PER LA PROGRAMMAZIONE LINEARE RISOLVE IL PROBLEMA SET COVER CON UNA  $f$ -APPROXIMAZIONE

Teorema: l'algoritmo è un algoritmo di approssimazione  $f$  per il problema SC.

- Dim: La soluzione calcolata è una soluzione ammissibile



si prenda un elemento 'e', e si considerino al massimo gli insiemi  $f$  che contengono 'e', poiché 'e' è coperto nella soluzione frazionaria, esiste almeno un insieme  $S$  con  $x_S >= 1/f$   
allora 'e' è coperto nella soluzione intera calcolata.

il processo di arrotondamento aumenta ogni

$x_S$  di un fattore al più  $f$ , Allora:  $\text{cost of the computed cover} \leq f \cdot OPT_f \leq f \cdot OPT$

Caso speciale: Weighted Vertx Coverproblem

(un caso speciale del set cover in cui scegliamo i vertici per coprire gli archi (gli insiemi sono i vertici))

Input: un grafo non diretto  $G=(V,E)$  in cui ogni vertice  $v$  ha costo  $c(v)$

Soluzione ammissibile:  $U \subseteq V$  tale che ogni arco  $(u,v) \in E$  è coperto (cioè  $u \in U$  oppure  $v \in U$ )

Misura(da minimizzare):  $\text{cost}(U) : \sum_{v \in U} c(v)$

$f$ : frequenza dell'elemento più frequente = 2



The LP-rounding algorithm is a 2-approximation algorithm for the weighted VC problem

## ESEMPIO TIGHT:

visualizza un'istanza di set cover come un ipergrafo:

- gli insiemi corrispondono ai vertici
- gli elementi corrispondono agli iperbordi
- un insieme/vertice  $v$  copre un elemento/iperbordo  $X \subseteq V$  se  $v \in X$

Siano  $V_1, \dots, V_k$  k insiemi di cardinalità n ciascuno. L'ipergrafo ha:

- insieme dei vertici:  $V = V_1 \cup \dots \cup V_k$
- $n^k$  hyperedges: ogni hyperedge preleva un vertice da  $V_i$
- tutti gli insiemi/vertici hanno costo 1

$f=k$  (la frequenza f è pari a k perché ogni oggetto è contenuto in k insiemi)

$\text{OPT}_f = n$  (imposta ogni variabile set/vertice a  $1/k$ )

la soluzione arrotondata restituita ha un costo  $kn$

$\text{OPT} = n$  (scegli  $V_1$ )

## LP-DUALITY (PROGRAMMAZIONE LINEARE – DUALITÀ)

Introduzione pratica alla dualità:

$\text{minimize}$ $7x_1 + x_2 + 5x_3$  $\text{subject to}$ $x_1 - x_2 + 3x_3 \geq 10$ $5x_1 + 2x_2 - x_3 \geq 6$ $x_1, x_2, x_3 \geq 0$  <b><math>z</math>: value of the optimal solution</b>
---

$z$  è al massimo  $\alpha$ ? (do una prova tramite certificato)

Certificato SÌ: qualsiasi soluzione fattibile di valore  $\leq \alpha$

$x=(2,1,3)$  soluzione ammissibile di valore  $7*2+1*1+5*3=30$

quindi  $z \leq 30$

Is  $z$  at least  $\alpha$ ? è  $z$  almeno sopra un certo valore alfa?

$$\begin{array}{l} 7x_1 + x_2 + 5x_3 \geq \\ x_1 - x_2 + 3x_3 \geq 10 \end{array} \quad \rightarrow \quad z \geq 10$$

riga sopra  $\geq$  riga sotto confrontando i termini prima delle  $x$

la soluzione ottima deve avere almeno valore 10

Is  $z$  at least  $\alpha$ ?

$$\begin{array}{l} 7x_1 + x_2 + 5x_3 \geq \\ y_1 x_1 - x_2 + 3x_3 \geq 10 \\ y_2 5x_1 + 2x_2 - x_3 \geq 6 \end{array} \quad \rightarrow \quad z \geq 16$$

faccio la somma di questi due (10+6)

faccio la somma tra 10 e 6 ottenendo  $z \geq 16$

Primale e duale sono:

### primal program

$\text{minimize}$	$7x_1 + x_2 + 5x_3$
$\text{subject to}$	$x_1 - x_2 + 3x_3 \geq 10$
	$5x_1 + 2x_2 - x_3 \geq 6$
	$x_1, x_2, x_3 \geq 0$

### dual program

$\text{maximize}$	$10y_1 + 6y_2$
$\text{subject to}$	$y_1 + 5y_2 \leq 7$
	$-y_1 + 2y_2 \leq 1$
	$3y_1 - y_2 \leq 5$
	$y_1, y_2 \geq 0$

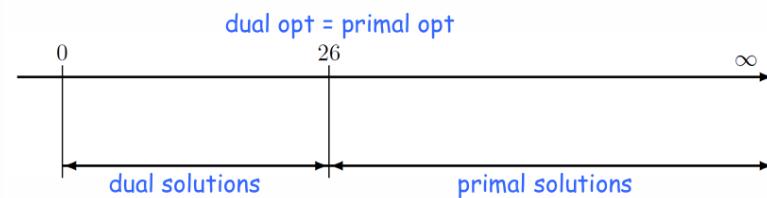
NB: Ogni soluzione ammissibile del duale dà una limitazione inferiore alla soluzione ottima del primale.

NB: Ogni soluzione fattibile del primordiale dà un limite superiore alla soluzione ottimale del duale.

NB: Allora due soluzioni con lo stesso valore devono essere entrambe ottimali!

Soluzione ottima caso precedente:  $x=(7/4, 0, 11/4)$   $y=(2, 1)$  entrambe di valore 26

primal program	dual program
minimize $7x_1 + x_2 + 5x_3$	maximize $10y_1 + 6y_2$
subject to $x_1 - x_2 + 3x_3 \geq 10$	subject to $y_1 + 5y_2 \leq 7$
$5x_1 + 2x_2 - x_3 \geq 6$	$-y_1 + 2y_2 \leq 1$
$x_1, x_2, x_3 \geq 0$	$3y_1 - y_2 \leq 5$
	$y_1, y_2 \geq 0$



primal program	dual program
minimize $\sum_{j=1}^n c_j x_j$	maximize $\sum_{i=1}^m b_i y_i$
subject to $\sum_{j=1}^n a_{ij} x_j \geq b_i \quad i=1, \dots, m$	subject to $\sum_{i=1}^m a_{ij} y_i \leq c_j \quad j=1, \dots, n$
$x_j \geq 0 \quad j=1, \dots, n$	$y_i \geq 0 \quad i=1, \dots, m$

Teorema (LP-duality theorem)

Il programma primale ha un ottimo finito se e solo se il suo duale ha un ottimo finito. Inoltre, se  $x=(x_1, \dots, x_n)$  e  $y=(y_1, \dots, y_m)$  sono soluzioni ottimali rispettivamente per i programmi primali e duali, allora vale che:

$$\sum_{j=1}^n c_j x_j = \sum_{i=1}^m b_i y_i$$

Teorema (teorema della dualità debole)

Se  $x=(x_1, \dots, x_n)$  e  $y=(y_1, \dots, y_m)$  sono soluzioni ammissibili rispettivamente per i programmi primale e duale, allora vale che:

$$\sum_{j=1}^n c_j x_j \geq \sum_{i=1}^m b_i y_i$$

- dim:

since for  $y$  is feasible and  $x_j$ 's are nonnegative

$$\begin{aligned} \sum_{j=1}^n c_j x_j &\geq \sum_{j=1}^n \left( \sum_{i=1}^m a_{ij} y_i \right) x_j = \\ &= \sum_{i=1}^m \left( \sum_{j=1}^n a_{ij} x_j \right) y_i \geq \sum_{i=1}^m b_i y_i \end{aligned}$$

since for  $x$  is feasible and  $y_i$ 's are nonnegative

## SET COVER TRAMITE DUAL-FITTING

**ILP:**

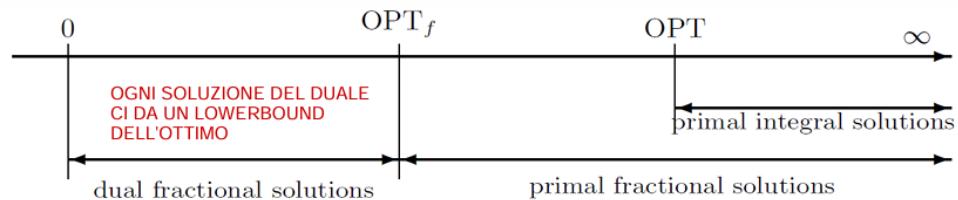
$$\begin{array}{ll} \text{minimize} & \sum_{S \in \mathcal{S}} c(S)x_S \\ \text{subject to} & \sum_{S: e \in S} x_S \geq 1 \quad e \in U \\ & x_S \in \{0,1\} \quad S \in \mathcal{S} \end{array}$$

### LP-relaxation

$$\begin{array}{ll} \text{minimize} & \sum_{S \in \mathcal{S}} c(S)x_S \\ \text{subject to} & \sum_{S: e \in S} x_S \geq 1 \quad e \in U \\ & x_S \geq 0 \quad S \in \mathcal{S} \end{array}$$

### dual program

$$\begin{array}{ll} \text{maximize} & \sum_{e \in U} y_e \\ \text{subject to} & \sum_{e: e \in S} y_e \leq c(S) \quad S \in \mathcal{S} \\ & y_e \geq 0 \quad e \in U \end{array}$$



1) prendiamo il problema di programmazione lineare intera

2) lo rilassiamo

3) scriviamo il duale corrispondente a partire dal problema di programmazione lineare

OPT = Soluzione ottima del set cover (quindi del programma lineare intero)

Strategia greedy: scegliere l'insieme più conveniente e rimuovere gli elementi coperti, fino a coprire tutti gli elementi.

→ Sia  $C$  l'insieme degli elementi già coperti. rapporto costo-efficacia di  $S$ :  $c(S)/|S-C|$  (costo medio al quale  $S$  copre i nuovi elementi)

### Algorithm 2.2 (Greedy set cover algorithm)

1.  $C \leftarrow \emptyset$
2. While  $C \neq U$  do
 

Find the most cost-effective set in the current iteration, say  $S$ .  
 Let  $\alpha = \frac{\text{cost}(S)}{|S-C|}$ , i.e., the cost-effectiveness of  $S$ .  
 Pick  $S$ , and for each  $e \in S - C$ , set  $\text{price}(e) = \alpha$ .  
 $C \leftarrow C \cup S$ .
3. Output the picked sets.

ALG.GREEDY ASSEGNA UN  
PREZZO AD OGNI OGGETTO

average cost at which  
e is covered

$$\text{for each } e \in U, \quad y_e = \frac{\text{price}(e)}{H_n} \quad (\text{H}_n = \text{numero armonico ordine } n)$$

Lemma: Il vettore  $y$  definito sopra è una soluzione fattibile per il programma duale.

Teorema: L'algoritmo greedy è un algoritmo di approssimazione  $H_n$  per il problema SC

- dim:  $\text{cost of the cover} = \sum_{e \in U} \text{price}(e) = H_n \sum_{e \in U} y_e \leq H_n \text{OPT}_f \leq H_n \text{OPT}$

### SCHEMA PRIMALE DUALE

L'idea di alto livello dell'approccio è la seguente:

- 1) L'algoritmo inizia con una soluzione primale intera non ammissibile e una soluzione duale fattibile.
- 2) Iterativamente si migliora la soluzione duale e si migliora la fattibilità della soluzione primale intera. Questo fino a quando non si ottiene una soluzione primale intera fattibile.
- 3) analisi: dimostrare la garanzia di approssimazione utilizzando il valore della soluzione duale come limite inferiore.

### ALGORITMO DI F-APPROSSIMAZIONE PER SET COVER PROBLEM CHE PASSA PER IL PRIMALE DUALE (E NON PER IL ROUNDING COME LA SCORSA VOLTA)

Partiamo dal problema set cover definito la scorsa lezione, ossia:

Input= insieme  $U$  di  $n$  elementi

una collezione di sottoinsiemi di  $U$  che chiamiamo  $S$ , con  $S=\{S_1, \dots, S_k\}$

ogni  $s \in S$  ha costo positivo  $c(s)$

Soluzione ammissibile= un sottoinsieme  $C \subseteq S$  che copre  $U$  (la cui unione è  $U$ )

Misura (da minimizzare)= cost of  $C$ :  $\sum_{S \in C} c(S)$

Def frequenza=per frequenza di un elemento e si intende il numero di insiemi nell'input che lo contengono.

Def  $f$ = Per "f" si intende la frequenza dell'elemento più frequente (quindi la frequenza massima) e sia come in lezione precedente:

$$\text{ILP:} \quad \begin{aligned} & \text{minimize} && \sum_{S \in S} c(S)x_S \\ & \text{subject to} && \sum_{S: e \in S} x_S \geq 1 \quad e \in U \\ & && x_S \in \{0,1\} \quad S \in S \end{aligned}$$

#### LP-relaxation

$$\begin{aligned} & \text{minimize} && \sum_{S \in S} c(S)x_S \\ & \text{subject to} && \sum_{S: e \in S} x_S \geq 1 \quad e \in U \\ & && x_S \geq 0 \quad S \in S \end{aligned}$$

#### dual program

$$\begin{aligned} & \text{maximize} && \sum_{e \in U} y_e \\ & \text{subject to} && \sum_{e: e \in S} y_e \leq c(S) \quad S \in S \\ & && y_e \geq 0 \quad e \in U \end{aligned}$$

Data una soluzione duale  $y$ , diciamo che un insieme  $S$  è stretto(tight) se  $\sum_{e \in S} y_e = c(S)$

Idea: scegliere solo insiemi tight e non sovraccaricare nessun insieme.

### ALGORITMO PER SET COVER – FATTORE $f$

#### Algorithm 15.2 (Set cover – factor $f$ )

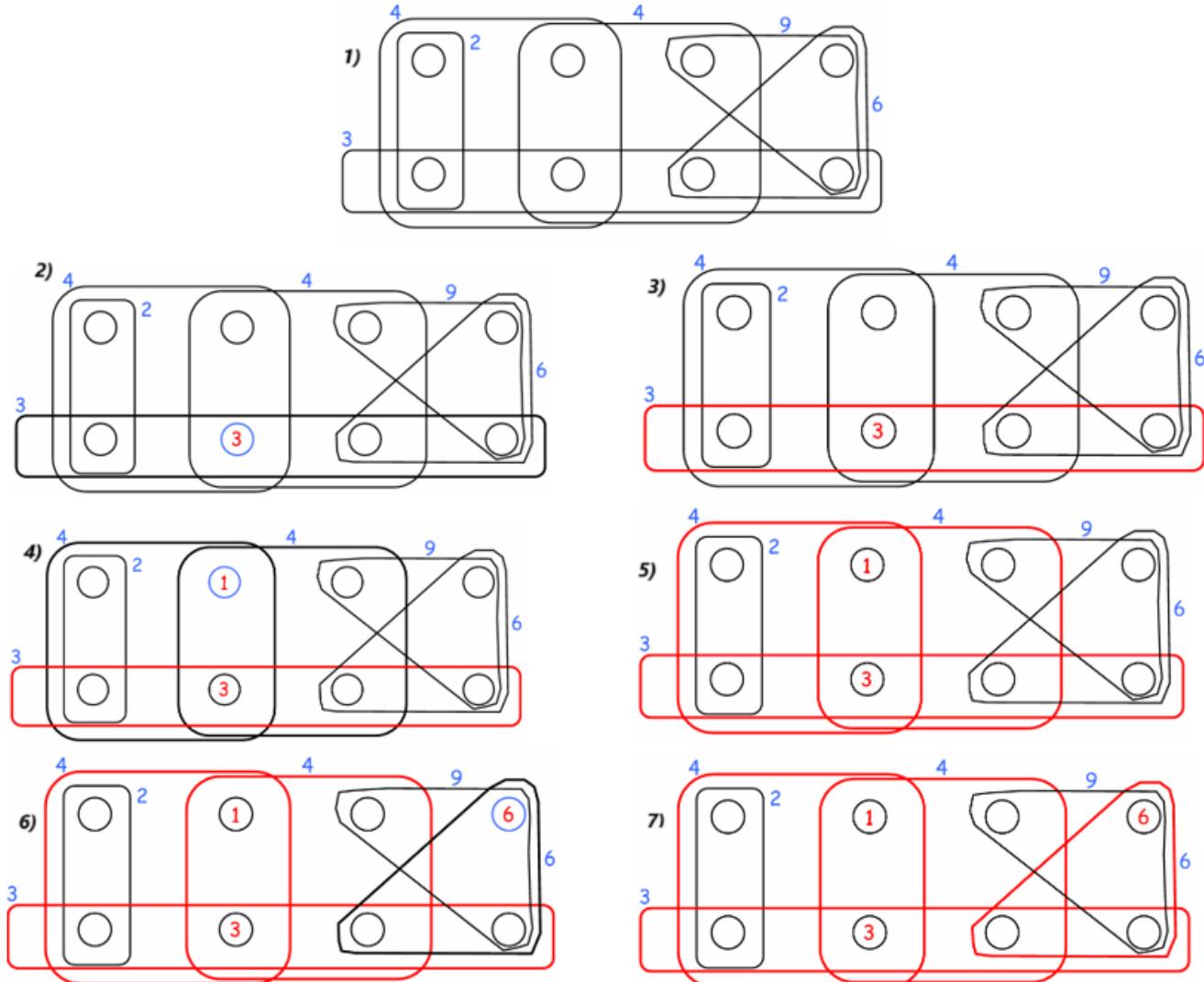
1. Initialization:  $x \leftarrow 0; y \leftarrow 0$
2. Until all elements are covered, do:
  - Pick an uncovered element, say  $e$ , and raise  $y_e$  until some set goes tight.
  - Pick all tight sets in the cover and update  $x$ .
  - Declare all the elements occurring in these sets as "covered".
3. Output the set cover  $x$ .

Questo algoritmo si basa sul selezionare in set cover solo gli insiemi tight garantendo sempre di non sforare mai il costo dell'insieme.

## Funzionamento algoritmo:

1. cominciamo con soluzione duale ammissibile (tutti  $y_e=0$ ) e soluzione non ammissibile (tutti  $x=0$ )
2. finché tutti gli elementi sono coperti do:  
prendere un elemento 'e' qualsiasi non ancora coperto ed alzare variabili  $y_e$ (migliorando valore duale) finché qualche insieme non diventa tight
3. a questo punto uno o più insiemi sono tight, prendere tutti insiemi tight e mettere a 1 la corrispondente variabile
4. quindi otteniamo e restituiammo il set cover ammissibile x

Visivamente:



somma valori rossi = soluzione duale (10 è soluzione del duale , 3+6+1)

Teorema: l'algoritmo è una f-approssimazione per il problema Set Cover.

- Dim: La copertura calcolata è chiaramente fattibile.

Affermiamo che:  $\sum_{S \in \mathcal{S}} c(S)x_S \leq f \sum_{e \in U} y_e$



(Pensalo come denaro che puoi usare per acquistare la soluzione primaria scelta)

ogni elemento e:

- ha  $f * y_e$  quantità di denaro
- paga  $y_e$  per ogni set scelto  $S$  contenente e

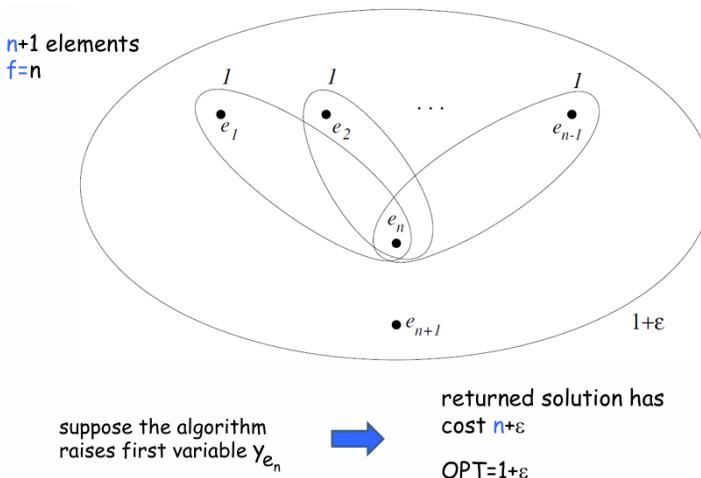
poiché ogni e è al massimo in f insiemi, e ha abbastanza soldi per i suoi pagamenti

poiché ogni insieme scelto  $S$  è tight,  $S$  è completamente pagato dagli elementi che contiene

poiché y è ammissibile allora

$$\sum_{e \in U} y_e \leq OPT$$

Esempio tight:



### STEINER FOREST PROBLEM

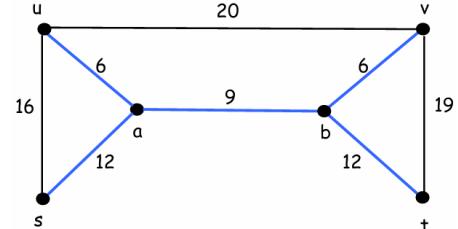
(Generalizzazione dello steiner tree, utilizza tecnica diversa più complicata sempre tramite primale duale)

Input: grafo non diretto  $G=(V,E)$  con costi degli archi non negativi  
collezione di sottoinsiemi disgiunti di  $V$  chiamata  $S_1, \dots, S_k$

Soluzione ammissibile: una foresta  $F$  in cui ogni coppia di vertici appartenenti allo stesso insieme  $S_i$  è collegata.

Misura (da minimizzare) = cost of  $F : \sum_{e \in E(F)} c(e)$

Esempio di Steiner forest di costo 45 con  $s_1=\{u,v\}$  e  $s_2=\{s,t\}$  è il seguente:



### METRIC STEINER FOREST PROBLEM

$G$  è completo, inoltre i costi degli archi soddisfano la diseguaglianza triangolare per ogni  $u, v, w$  t.c  $c(u, v) \leq c(u, w) + c(w, v)$

$r$  è funzione requisito di connettività tale che:

$$r(u, v) = \begin{cases} 1 & \text{if } u \text{ and } v \text{ belong to the same } S_i \\ 0 & \text{otherwise} \end{cases}$$

una funzione  $f$  su tutti i tagli(cuts) in  $G$ , per ogni  $S \subseteq V - \text{taglio}(S, S' = V \setminus S)$ :

$$f(S) = \begin{cases} 1 & \text{if } \exists u \in S \text{ and } v \in S' \text{ such that } r(u, v) = 1 \\ 0 & \text{otherwise} \end{cases}$$

### FORMULAZIONE INTEGER LINEAR PROGRAMMING (ILP) DEL PROBLEMA STEINER FOREST

$$\text{minimize } \sum_{e \in E} c_e x_e$$

se prendiamo sottoinsieme  $S$  qualsiasi,

LP-relaxation

$$\text{subject to } \sum_{e: e \in \delta(S)} x_e \geq f(S) \quad S \subseteq V$$

ci sono 2 casi:  $f(S)$  vale 0 oppure  $f(S)$  vale 1. per

$$\text{minimize } \sum_{e \in E} c_e x_e$$

( $x_e = 0$  se l'arco non viene preso, 1 altrimenti)

$x_e \in \{0, 1\}$   $e \in E$   
relax with  $x_e \geq 0 \text{ and } x_e \leq 1$   
redundant

vincoli in cui  $f(s)=1$  voglio che se guardo tutti archi  $\delta(S)$  almeno uno di questi devo averlo preso (questo è il significato del vincolo che utilizza la somma [sommatoria per ogni  $e$  di  $x_e \geq f(s)$ ]])

$$\text{subject to } \sum_{e: e \in \delta(S)} x_e \geq f(S) \quad S \subseteq V$$

$$x_e \geq 0 \quad e \in E$$

$\delta(S)$ : edges crossing the cut ( $S, S' = V \setminus S$ )

### LP-relaxation

$$\text{minimize} \quad \sum_{e \in E} c_e x_e$$

$$\text{subject to} \quad \sum_{e: e \in \delta(S)} x_e \geq f(S) \quad S \subseteq V$$

$$x_e \geq 0 \quad e \in E$$

### dual program

$$\text{maximize} \quad \sum_{S \subseteq V} f(S) y_S$$

$$\text{subject to} \quad \sum_{S: e \in \delta(S)} y_S \leq c_e \quad e \in E$$

$$y_S \geq 0 \quad S \subseteq V$$

L'arco 'e' ha a che fare con  $y_S$  duale se  $y_S > 0$  ed  $e \in \delta(S)$ .

$S$  è stato incrementato in una soluzione duale se  $y_S > 0$

- oss: incrementare  $S$  o  $S'$  ha lo stesso effetto
- oss: nessun vantaggio nell'incrementare un insieme  $S$  con  $f(S)=0$  (supponiamo di non aumentare mai tali insiemi)

L'arco  $e$  è tight(stretto) se la quantità totale di dual sembra uguale al suo costo

- oss: il programma duale cerca di massimizzare la somma dei duali soggetti al vincolo per cui nessun arco è troppo stretto

In qualsiasi punto, gli archi attualmente selezionati formano una foresta  $F$

$S$  non è soddisfatto se  $f(S)=1$  ma non c'è alcun arco selezionato che attraversa il taglio  $(S, S')$

$S$  è attivo se è un insieme minimo non soddisfatto in  $F$

- oss: se  $F$  non è ammissibile, allora ci deve essere un insieme attivo

Lemma: L'insieme  $S$  è attivo se e solo se è un componente连通的 nella foresta attualmente selezionata e  $f(S)=1$ .

Quindi idea algoritmo per Steiner forest problem: foresta vuota, ogni volta inizio a selezionare archi e ad ogni istante di tempo ho foresta (anche non ammissibile) che collega alcuni nodi.

$S$  non è soddisfatto se  $f(s)=1$  ma nella soluzione corrente non ho archi che soddisfano  $f(s)=1$ .

### ALGORITMO PER STEINER FOREST

#### Algorithm 22.3 (Steiner forest)

1. (Initialization)  $F \leftarrow \emptyset$ ; for each  $S \subseteq V$ ,  $y_S \leftarrow 0$ .
2. (Edge augmentation) while there exists an unsatisfied set do:  
simultaneously raise  $y_S$  for each active set  $S$ , until some edge  $e$  goes tight;  
 $F \leftarrow F \cup \{e\}$ .
3. (Pruning) return  $F' = \{e \in F \mid F - \{e\} \text{ is primal infeasible}\}$

discard all redundant edges

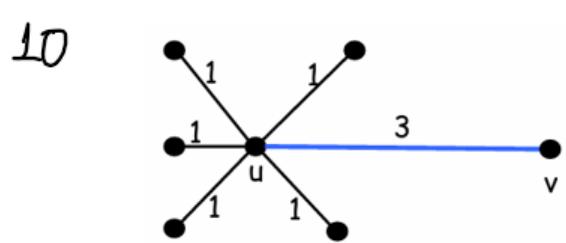
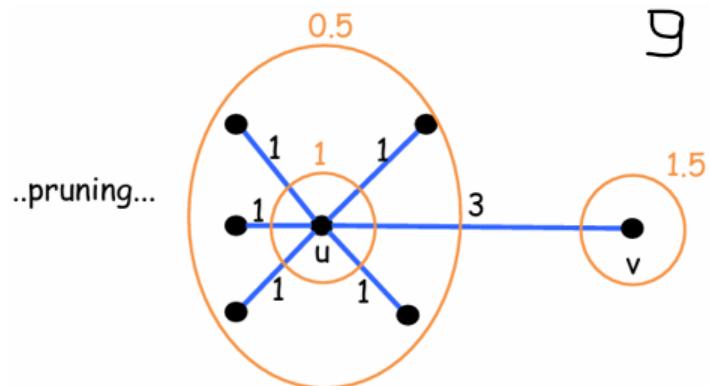
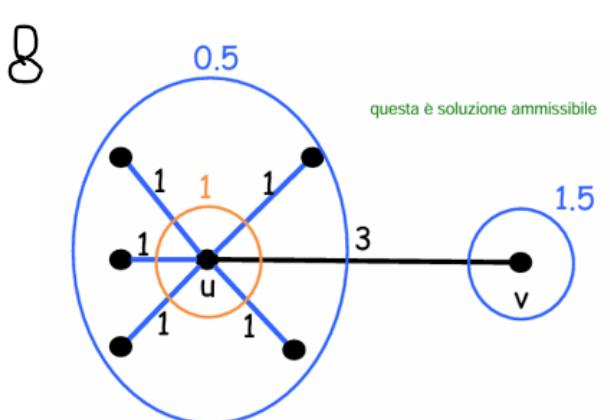
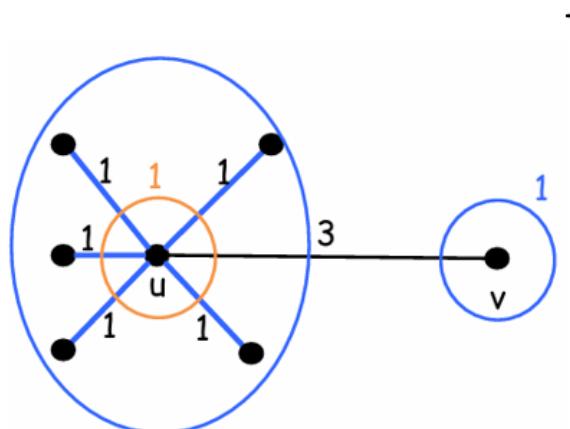
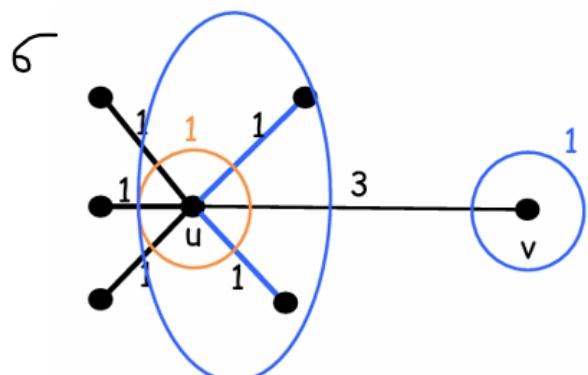
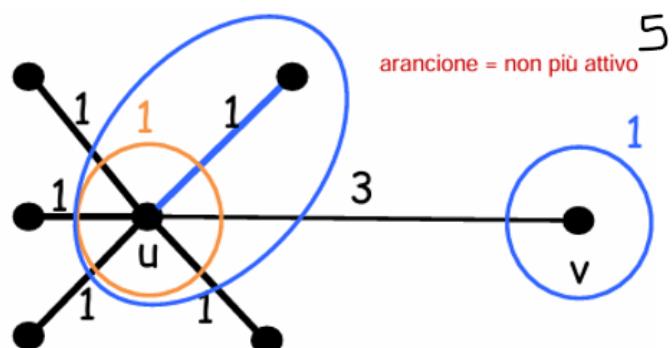
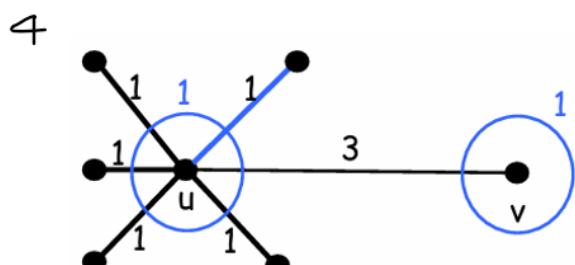
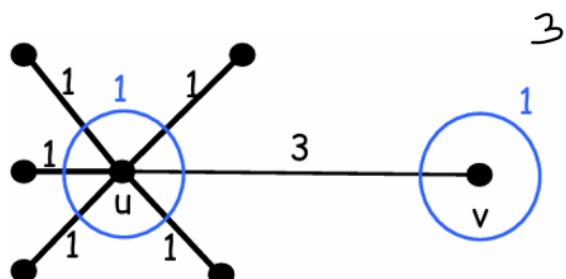
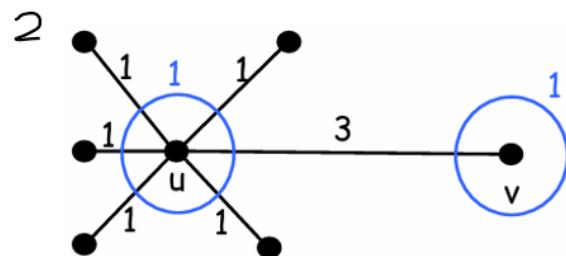
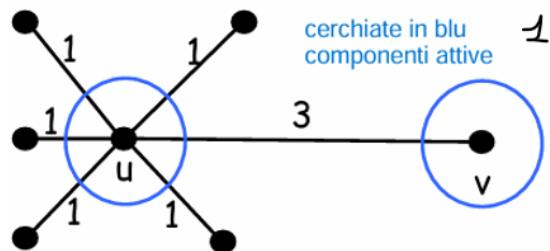
an edge  $e \in F$  is redundant if  $F - \{e\}$  is also a feasible solution

Funzionamento dell'algoritmo:

1. All'inizio non ho nemmeno un arco preso nella foresta, quindi soluzione primale intera non ammissibile. Inoltre soluzione duale ammissibile con tutti gli  $y_S=0$ .
  2. Finché c'è un insieme non soddisfatto (finché sol. non ammissibile) prendere tutti insiemi attivi di  $s$  (uno per ogni componente connessa che ha  $f(s)=1$ ) e alzare le corrispondenti variabili duali tutte insieme finché un arco non diventa tight, a questo punto ci fermiamo e mettiamo questo arco dentro  $F$ . Ripetiamo...
  3. La soluzione ottenuta dalla fase 2 è ammissibile ma potrebbe avere costo troppo alto, rimuovere quindi da  $F$  tutti gli archi che sono ridondanti (ossia partendo da  $F$  ammissibile, se togliendo l'arco 'e'  $F$  rimane ammissibile allora lo togliamo).
- Ottieniamo soluzione  $F'$  che è la soluzione cercata (2-approximazione)

Esempio necessità di pruning:

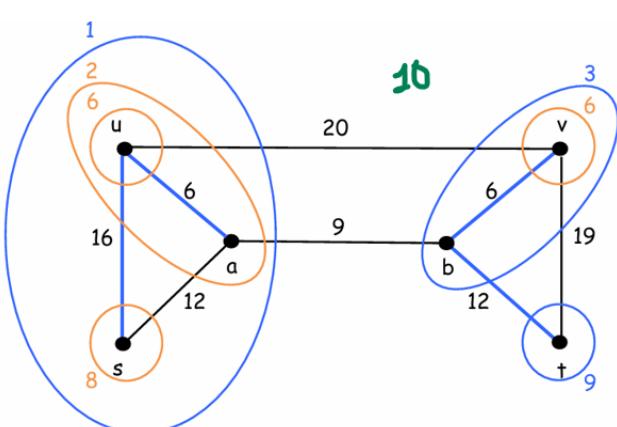
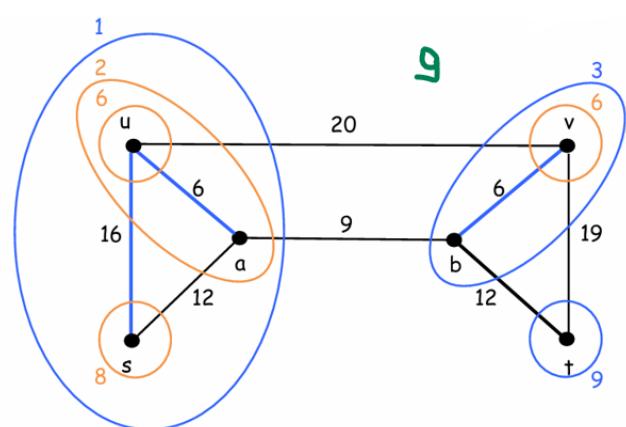
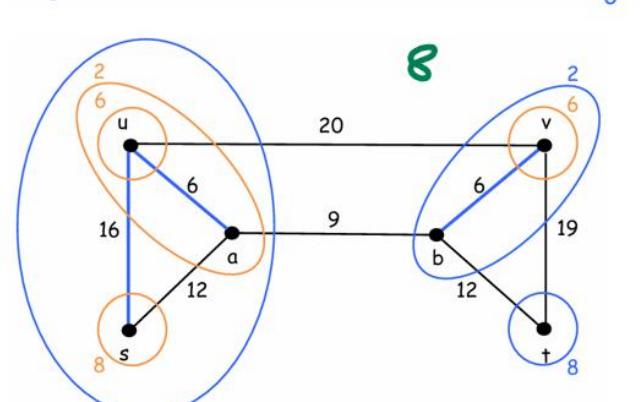
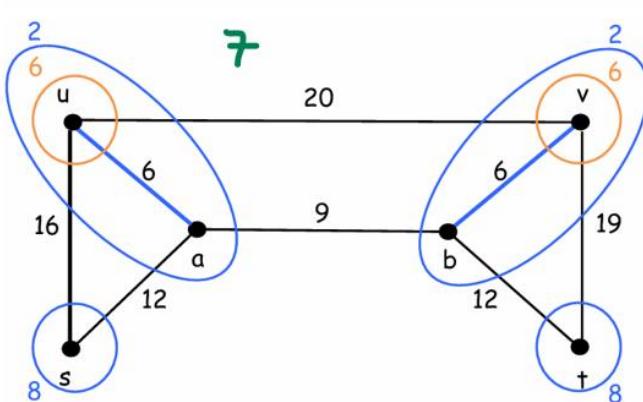
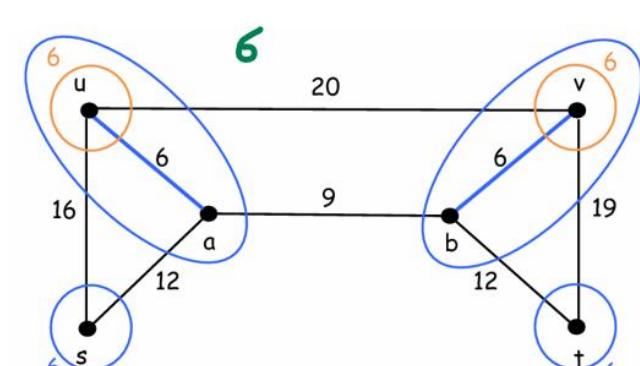
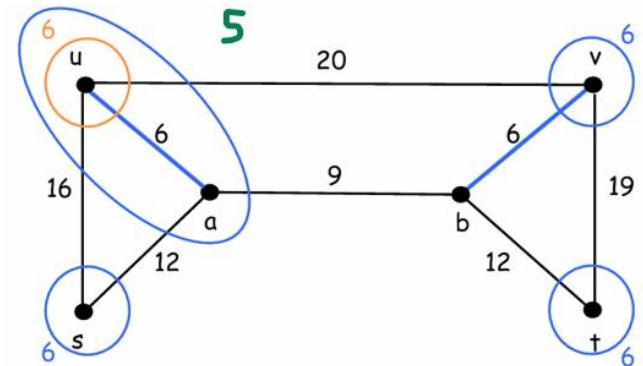
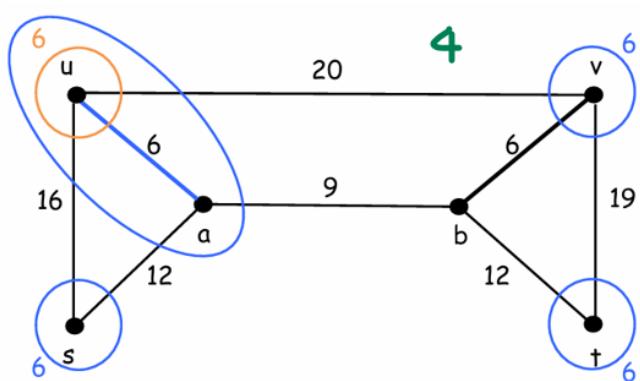
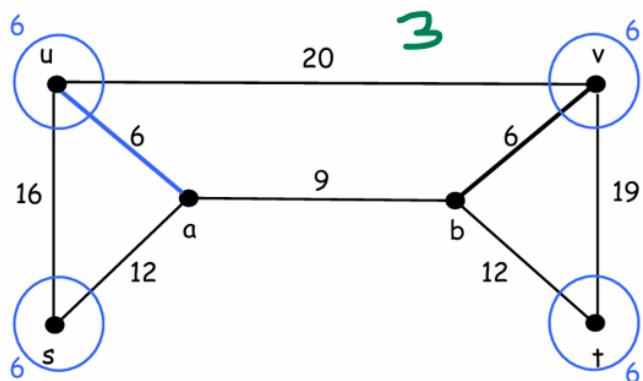
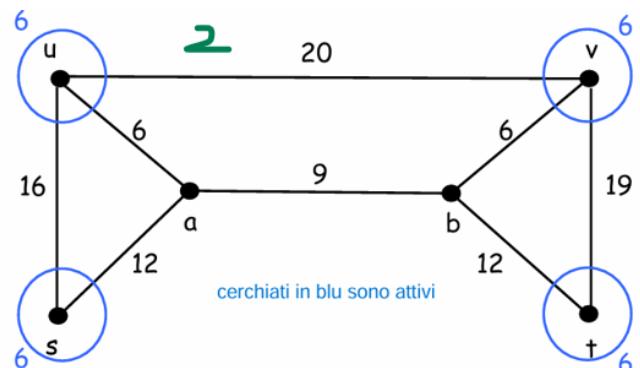
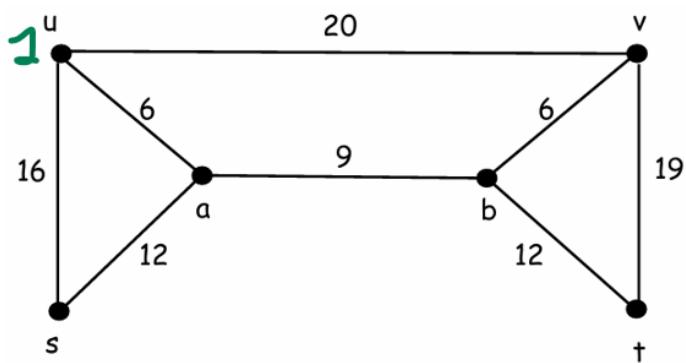
(S1={u,v})

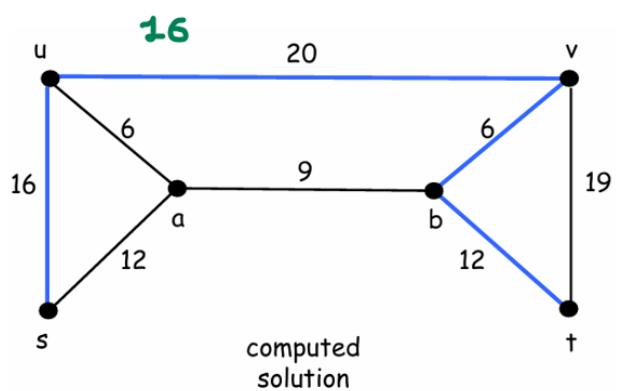
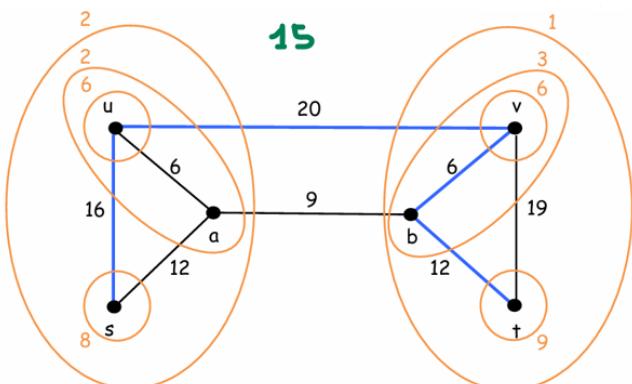
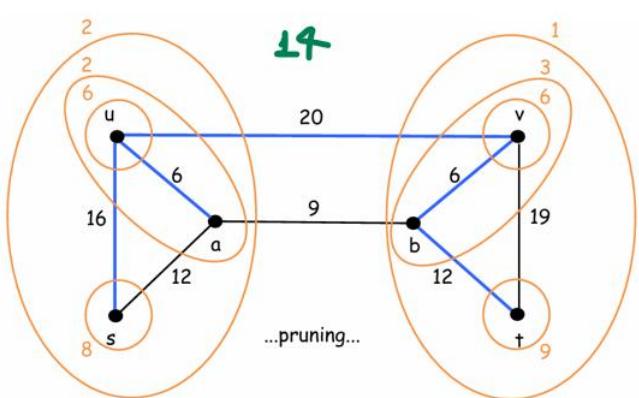
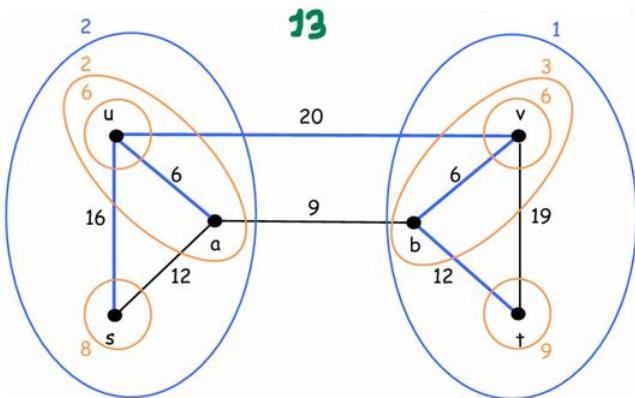
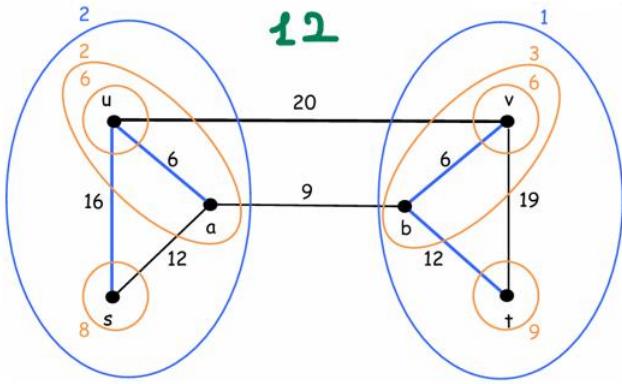
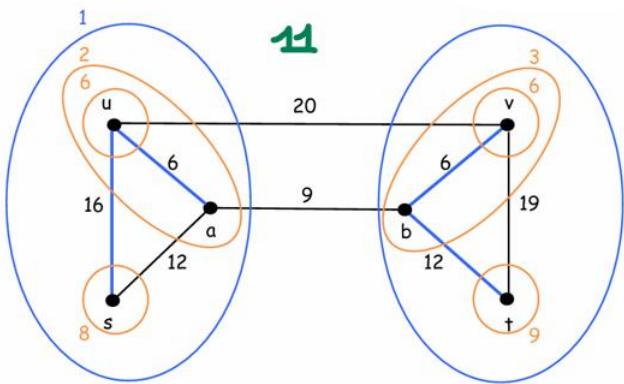


computed  
solution

Esempio di esecuzione dell'algoritmo:

$s1=\{u,v\}$   $s2=\{s,t\}$





Teorema: l'algoritmo è una 2-approximazione per il problema Steiner Forest.

## LEZ5 – 19/11/2024 (Parameterized algorithms, k-Vertex Cover problem con algoritmo, FPT, kernelization, polynomial kernel for k-Vertex Cover, W[1]-hardness)

Idea dietro gli algoritmi parametrizzati = quando studio complex algoritmi il parametro principale che si prende è la dimensione dell'istanza (nota: istanze difficili sono istanze grandi di solito).

Con algoritmi parametrizzati non tengo conto soltanto della dimensione dell'istanza come parametro principale ma tengo conto di più parametri per capire quali di questi rendono il problema difficile (oltre la dimensione dell'istanza).

Tre obiettivi tipici in progettazione algoritmi sono:

1. Risolvere problemi (NP-)hard
2. Algoritmi veloci (tempo polinomiale)
3. Calcolare le soluzioni esatte

di solito questi tre insieme non si possono avere e ne bisogna sceglierne due:

- combinazione 1. e 2. in algoritmi di approssimazione
- combinazione 2. e 3. in problemi in P
- Combinazione 1. e 3. si ottiene tramite alg. parametrizzati!

Idea: puntare ad algoritmi esatti, ma confinare la dipendenza esponenziale a un parametro.

Obiettivo: un algoritmo il cui tempo di esecuzione è polinomiale nella dimensione del problema  $n$  ed esponenziale nel parametro  $\Rightarrow$  algoritmo esatto in esecuzione veloce a condizione che  $k$  sia piccolo.

Parametro:  $k(x)$  intero non negativo associato all'istanza  $x$ .

Problema parametrizzato: un problema + un parametro (diciamo "problema  $P$  w.r.t. parametro  $k$ ").

### K-VERTEX COVER

Input: un grafo  $G=(V,E)$

un intero non negativo  $k$

Domanda: esiste un vertex cover  $S \subseteq V$  di dimensioni al massimo  $|S| \leq k$ ?

Parametro:  $k$

Esempio:  $k$  può effettivamente essere piccolo



→ Risolvibile tramite forza bruta:

1. provare tutti gli  $O(n^k)$  sottoinsiemi dei  $k$  vertici
2. per ogni sottoinsieme  $S$  controllare se  $S$  è un vertex cover (una copertura dei vertici)

Tempo di esecuzione tramite forza bruta:

$O(n^k m)$	<b>BAD</b>	$n^{f(k)}$	exponent depends on $k$
(CONSIDERATA INEFFICIENTE NEL MONDO DEGLI ALGORITMI PARAMETRIZZATI)			

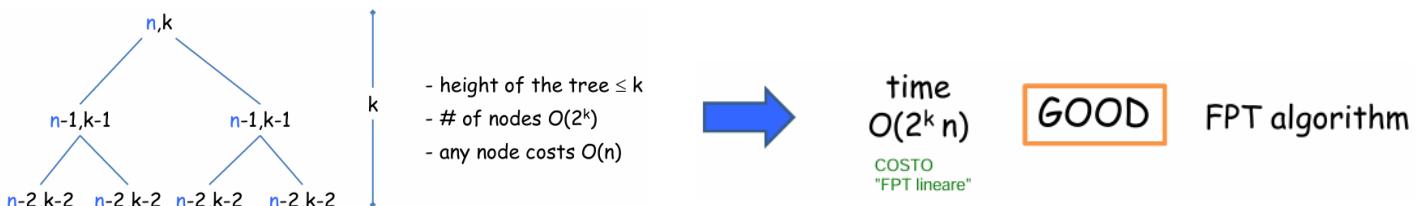
→ Bounded-search tree algorithm (Algoritmo ad albero di ricerca limitata):

1. Considera un qualunque arco non coperto  $e=(u,v)$  (se non c'è un arco scoperto restituisci TRUE)
2.  $u \in S$  oppure  $v \in S$  (oppure entrambi), indovina quale provando entrambe le possibilità
  - 2.1. aggiungi  $u$  a  $S$ , elimina  $u$  e tutti gli archi incidenti da  $G$   
(continua ricorsivamente su  $G$  con  $k'=k-1$ )
  - 2.2. aggiungi  $v$  a  $S$ , elimina  $v$  e tutti gli archi incidenti da  $G$   
(continua ricorsivamente su  $G$  con  $k'=k-1$ )
3. restituire l'OR dei due esiti

caso base:  $k=0$

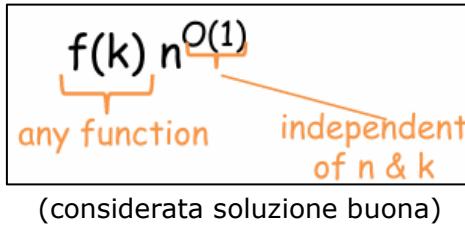
se c'è un arco (scoperto) in  $G$  restituisci FALSE, restituisci TRUE altrimenti.

Tempo di esecuzione: tramite analisi dell'albero della ricorsione



## FPT [DEFINIZIONE PRINCIPALE DI QUESTE 4 LEZIONI (IMPORTANISSIMA!)]

Un problema parametrizzato si dice **Fixed Parameter Tractable (FPT)** se può essere risolto in tempo:

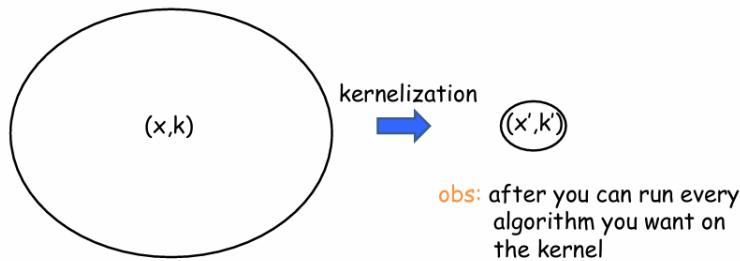


Per mostrare che un problema è FPT (e progettare un algoritmo FPT) posso utilizzare le tecniche seguenti:

- 1) bounded-search trees (vista in slide precedente)
- 2) kernelization (tecnica per progettare algoritmi FPT, è il prossimo argomento trattato)
- 3) color coding
- 4) iterative compression
- 5) algebraic techniques
- 6) treewidth

### KERNELIZATION

- Idea: pre-elaborazione di un'istanza per semplificarla in un'istanza equivalente molto più piccola.
- Kernelization: un algoritmo in tempo polinomiale che converte un'istanza  $(x, k)$  in un'istanza piccola ed equivalente  $(x', k')$  detta kernel.
  - o Equivalente vuol dire che la risposta di  $(x, k)$  è la stessa della risposta di  $(x', k')$ .
  - o Piccola perchè dimensione di  $(x', k')$   $\leq f(k)$ .



### TEOREMA (potente e molto importante)

Sono FPT tutti e soli i problemi parametrizzati che ammettono kernelization

(Un problema parametrizzato è FPT se e solo se esso ammette kernelization)

### KERNEL POLINOMIALE PER K-VERTEX COVER (vertex cover ammette kernel polinomiali)

Vediamo un algoritmo che prende istanza di vertex cover e la trasforma in una istanza equivalente polinomiale in  $k$ , questo algoritmo si basa sulle seguenti regole di riduzione:

- regola 1: se c'è un vertice  $v$  di grado  $\geq k+1$ , allora elimina  $v$  (e tutti i suoi archi incidenti) da  $G$  e decrementa il parametro  $k$  di 1. La nuova istanza è  $(G-v, k-1)$
- regola 2: se  $G$  contiene un vertice isolato (di grado 0)  $v$ , elimina  $v$  da  $G$ . La nuova istanza è  $(G-v, k)$

Lemma: se  $(G, k)$  è una istanza sì e nessuna delle regole sopra è applicabile a  $G$ , allora vale che

$$|E(G)| \leq k^2 \quad \text{e} \quad |V(G)| \leq 2^{k^2}.$$

Nota: applicare regole 1 e 2 una alla volta fino a che sono applicabili, quando non lo sono più passo alla 3.

- regola 3: sia  $(G, k)$  una istanza tale che le regole 1 e 2 non sono applicabili. Se  $k < 0$  oppure  $G$  ha più di  $k^2$  archi o più di  $2^{k^2}$  vertici, concludi che  $(G, k)$  è un'istanza NO.  
Genera un'istanza NO canonica.

Tempo d'esecuzione: con implementazione semplice in  $O(n^2)$

con implementazione intelligente in  $O(n+m)$

Per risolvere k-vertex cover: kernelization+ bound-search tree alg  $\rightarrow O(n+m+2^k k^2)$

## IN GENERE, QUALE PUO' ESSERE IL PARAMETRO K? ALCUNI ESEMPI:

- 1) la dimensione k della soluzione che stiamo cercando
- 2) il grado massimo del grafo di input
- 3) la lunghezza delle stringhe nell'input
- 4) il numero di mosse in un gioco di puzzle
- 5) il budget in un problema di aumento
- 6) ....

## ESEMPI DI PROBLEMI NP-HARD CHE SONO FPT:

- 1) Trovare un vertex cover di dimensione k
- 2) trovare un percorso di lunghezza k
- 3) trovare k triangoli disgiunti
- 4) Disegno di un grafico nel piano con k incroci di archi
- 5) Trovare percorsi disgiunti che collegano k coppie di vertici
- 6) Trovare la clique massima in un grafo di grado massimo k
- 7) ...

## W[1]-HARDNESS

Evidenza negativa simile alla NP-completezza. Se un problema è W[1]-difficile, allora il problema non è FPT a meno che FPT=W[1].

Alcuni esempi di problemi W[1]-hard:

- 1) Trovare una clique/indipendent set di dimensione K
- 2) Trovare un insieme dominante di dimensione K
- 3) Trovare k insiemi disgiunti a coppie
- 4) Dato un grafo G, trovare k vertici che coprono almeno s archi (copertura parziale dei vertici)
- 5) Data una formula booleana, decidere se può essere soddisfatta assegnando TRUE ad al massimo a k variabili

## DOMANDE DA FARMI QUANDO HO A CHE FARE CON ALGORITMI PARAMETRIZZATI

- 1) Il problema è FPT? (FPT vs W[1]-HARDNESS)
- 2) Qual è la migliore dipendenza  $f(k)$  dal parametro?
- 3) Qual è la migliore dipendenza possibile da  $k$  nell'esponente?

## NOTA LEZIONE

In questa lezione quindi abbiamo visto due esempi di tecniche per ottenere algoritmi FPT, ossia kernelization e bound-search tree che abbiamo usato insieme per ottenere la complessità finale relativa alla risoluzione del k-vertex cover.

### K-PATH

Per descrivere color coding introduciamo il problema K-Path.

Input: un grafo  $G=(V,E)$

un intero  $k$  non negativo

Domanda:  $G$  ha un cammino semplice lungo  $k$  vertici? (cammino deve toccare ogni nodo una sola volta!!)

Parametro:  $k$

Oss: NP-difficile poiché contiene il percorso hamiltoniano come caso speciale

**Theorem [Alon, Yuster, Zwick 1994]**

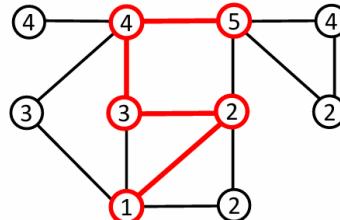
k-Path can be solved in time  $2^{O(k)} n^{O(1)}$ .

Questo algoritmo usa la tecnica del color coding, che migliora la complessità degli algoritmi precedenti che avevano tempo  $k^O(k) n^O(1)$ .

### COLOR CODING

Idea generale:

1. Assegnare ad ogni nodo un colore tra una palette di  $\{1, \dots, k\}$  colori in modo uniforme e indipendentemente casuale



2. Sui nodi colorati, cercare se esiste un cammino(path) colorato 1-2-...-k e invia come output 'YES' se esiste, altrimenti ritorna 'NO'

Oss1: Se non c'è un cammino lungo  $k$  allora non esiste alcun percorso colorato 1-2-...-k quindi sicuramente risponde 'NO'

Oss2: Se c'è un cammino lungo  $k$  allora c'è una certa probabilità che questo percorso sia colorato 1-2-...-k.

Probabilità di successo (lower bound): viene inviato 'YES' con probabilità almeno pari a  $k^{(-k)}$

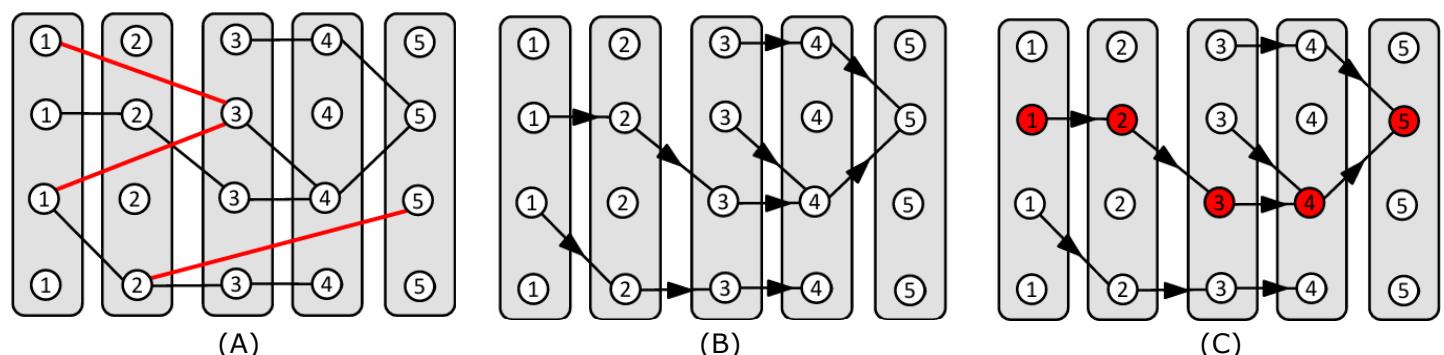
### AUMENTARE LE PROBABILITÀ DI SUCCESSO: RIPETIZIONI INDEPENDENTI (INDEPENDENT REPETITIONS)

Se la probabilità di successo è almeno  $p$ , allora la probabilità che l'algoritmo non dica 'YES' dopo 1/p ripetizioni vale al massimo:

$$(1-p)^{1/p} \leq (e^{-p})^{1/p} = 1/e \approx 0.38 \text{ ("probabilità di sbagliare")}$$

### TROVARE UN PERCORSO COLORATO 1-2-...-k (IN COLOR CODING)

- A. Gli archi che collegano classi di colore non adiacenti vengono rimossi
- B. Gli archi rimanenti vengono diretti verso la classe più grande
- C. Tutto ciò di cui abbiamo bisogno per verificare se esiste un percorso diretto dalla classe 1 alla classe  $K$



## COLOR CODING MIGLIORATO

Probabilità di successo passata da quella vista precedentemente [in color coding (1)] alla seguente:

$$\frac{k! k^{n-k}}{k^n} = \frac{k!}{k^k} \geq \frac{(k/e)^k}{k^k} = e^{-k} \rightarrow \text{YES with probability } e^{-k}. \quad (\text{Probabilità migliorata})$$

Assegnare colori da  $\{1, \dots, k\}$  vertici  $V(G)$  in modo uniforme e indipendentemente casuale (come prima).

Miglioramento: Ripetendo l'algoritmo  $100((e)^k)$  volte, la probabilità di una risposta errata è al massimo  $(1/e)^{100}$ .

Come trovare un path colorato? 2 modi:

- Provare tutte le permutazioni in tempo  $k! n^{O(1)}$

- Programmazione dinamica in tempo  $2^k n^{O(1)}$

## TROVARE UN CAMMINO BEN COLORATO TRAMITE PROGRAMMAZIONE DINAMICA (ALGORITMO COLOR CODING)

- Sottoproblemi

Per ogni  $v \in V$  e per ogni sottoinsieme non vuoto t.c.  $S \subseteq \{1, \dots, k\}$ , definiamo  $\text{Path}(v, S)$  come segue:

$\text{Path}(v, S)$ : esiste un percorso  $P$  che termina in  $v$  tale che ogni colore di  $S$  appare in  $P$  esattamente una volta e nessun altro colore appare in  $P$ ?

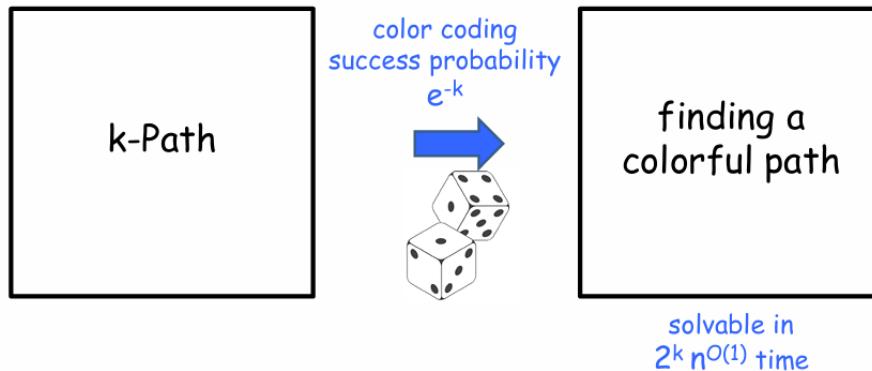
- o oss1: esiste un cammino ben colorato se e solo se  $\text{Path}(v, \{1, \dots, k\}) = \text{TRUE}$  per qualche  $v$
- o oss2: # di sottoproblemi pari a  $2^k n$

- Caso base:  $|S|=1$ ,  $\text{Path}(v, S)=\text{TRUE}$  se e solo se  $S=\{\text{col}(v)\}$

- Se  $S>1$  allora abbiamo che

$$\text{Path}(v, S) = \begin{cases} \text{OR}_{\substack{(u,v) \in E \\ \text{if } \text{col}(v) \in S}} \text{Path}(u, S - \{\text{col}(v)\}) & \text{if } \text{col}(v) \in S \\ \text{FALSE} & \text{otherwise} \end{cases}$$

IDEA: quindi per risolvere k-Path coloriamo i vertici e poi cerchiamo il cammino ben colorato



TEOREMA: Esiste un algoritmo randomizzato per k-Path che viene eseguito in tempo

$$(e2)^k n^{O(1)}$$

che o segnala un errore o trova un percorso di  $k$  vertici.

Inoltre, l'algoritmo trova una soluzione di un'istanza YES con probabilità costante.

## OTTENERE UN KERNEL DI DIMENSIONE 2K-VERTICI PER VERTEX COVER CON PROGRAMMAZIONE LINEARE

Una formulazione in programmazione lineare intera (ILP) di vertex cover:

### LP-relaxation

$$\text{minimize} \quad \sum_{v \in V} x_v$$

$$\text{minimize} \quad \sum_{v \in V} x_v$$

$$\text{subject to} \quad x_u + x_v \geq 1 \quad e = (u, v) \in E$$

$$v \in V$$

$$x_v \in \{0, 1\}$$

$$\text{subject to} \quad x_u + x_v \geq 1 \quad e = (u, v) \in E$$

$$v \in V$$

$$x_v \geq 0$$

relax with  
 $x_v \geq 0 \text{ & } x_v \leq 1$

redundant

a feasible solution is  
a **fractional VC**

$OPT_f$ : cost of the min fractional VC

$$OPT_f \leq OPT$$

Sia  $x$  una soluzione frazionaria ottimale. 3 classi in cui partizionare i vertici:

$$V_0 = \{ v \in V : x_v < \frac{1}{2} \}$$

$$V_{0.5} = \{ v \in V : x_v = \frac{1}{2} \}$$

$$V_1 = \{ v \in V : x_v > \frac{1}{2} \}$$

Teorema (Nemhauser-Trotter) = Esiste un minimum vertex cover  $S$  di  $G$  tale che  $V_1 \subseteq S \subseteq V_1 \cup V_{0.5}$

Kernelization (algoritmo):

calcolare una soluzione frazionaria ottimale  $x$  del rilassamento LP per l'istanza  $(G, k)$  di Vertex Cover.

Definire  $V_0, V_{0.5}, V_1$  come prima.

Se vale che  $\sum_{v \in V} x_v > k$  concludiamo che  $(G, k)$  è un'istanza no.

Altrimenti, scegliere in modo greedy  $V_1$  nel Vertex Cover, eliminare i vertici in  $V_1$  e  $V_0$  (e tutti i relativi archi incidenti). La nuova istanza è  $(G' = G - (V_1 \cup V_0), k' = k - |V_1|)$ .

Teorema:  $k$ -Vertex Cover ammette un kernel di al più  $2k$  vertici.

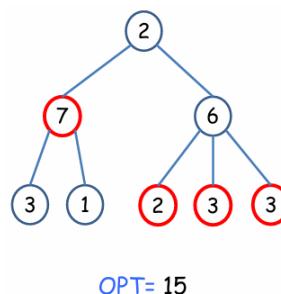
**LEZ7 – 26/11/2024 (party problem, algoritmo programmazione dinamica per weighted independent set on trees, tree decomposition, width, TREewidth, programmazione dinamica su grafo con treewidth w per risoluzione del problema weighted independent set)**

**PARTY PROBLEM**

Problema: invitare persone a un party

Massimizzare: fattore di divertimento totale delle persone invitate

Vincolo: tutti devono divertirsi, non invitare un collega e il suo boss direttamente contemporaneamente!



**input:** a tree with weights on the nodes

**goal:** an independent set of maximum total weight

**ALGORITMO DI PROGRAMMAZIONE DINAMICA PER WEIGHTED INDEPENDENT SET SU ALBERI**

Sottoproblemi = per ogni nodo v di T:

- $T_v$  è il sottoalbero di T radicato in v
- $A[v]$  è il peso di un maximum weighted independent set di  $T_v$
- $B[v]$  è il peso di un maximum weighted independent set di  $T_v$  che non contiene v

Goal: determinare  $A[r]$  per la radice r

Formulazione:

- Se v è una foglia allora  $A[v]=w_v$  E  $B[v]=0$
- Se v è un nodo interno con figli  $u_1, \dots, u_d$  allora vale che  $B[v]=\sum_{i=1}^d A[u_i]$

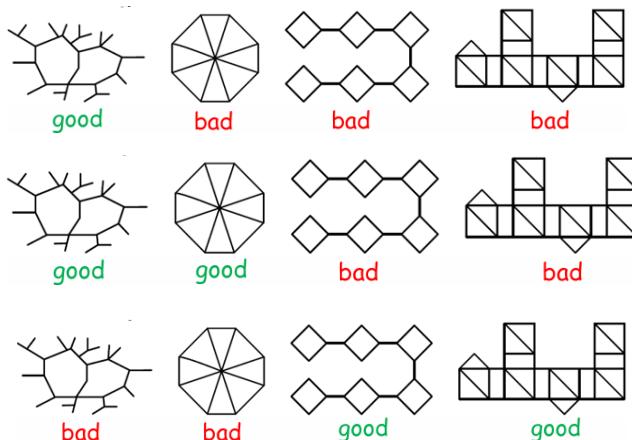
Ordine sottoproblemi: bottom up.

$$A[v]=\max\{ B[v], w_v + \sum_{i=1}^d B[u_i] \}$$

**GENERALIZZAZIONE ALBERI**

Come potremmo definire che un grafo è "ad albero" ("treelike")? Ecco tre definizioni possibili:

- DEF 1: "ad albero" se il numero di cicli è limitato (troppo poco generale)
- DEF 2: "ad albero" se la rimozione di un numero limitato di vertici lo rende aciclico
- DEF 3: "ad albero" se parti di dimensioni limitate sono collegate in modo simile ad un albero



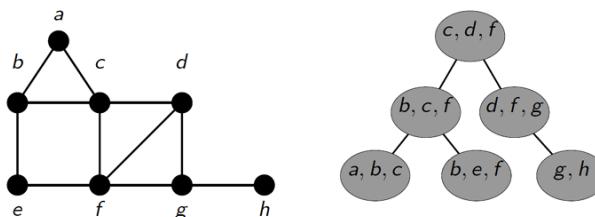
Vediamo definizione più accurata passante per tree decomposition:

**TREE DECOMPOSITION (SCOMPOSIZIONE/DECOMPOSIZIONE DAL ALBERO)**

Una scomposizione ad albero ( $T, \{V_t : t \in T\}$ ) di un grafo  $G=(V,E)$  consiste in un albero T (su un insieme di nodi diverso da G) e in un pezzo("bag")  $V_t \subseteq V$  associato a ciascun nodo t di T che soddisfa le seguenti tre proprietà:

1. [Node coverage] ogni nodo di G appartiene ad almeno un pezzo  $V_t$ ;
2. [Edge Coverage]: per ogni arco e di G, esiste un pezzo detto  $V_t$  contenente entrambi gli endpoint di e;
3. [Coherence]: Siano  $t_1, t_2$  e  $t_3$  tre nodi di T tali che  $t_2$  giace sul percorso da  $t_1$  e  $t_3$ . Allora, se un nodo v di G appartiene sia a  $V_{t_1}$  che a  $V_{t_3}$ , appartiene anche a  $V_{t_2}$ .

the width of  $(T, \{V_t : t \in T\})$ :  $\max_t |V_t|-1$



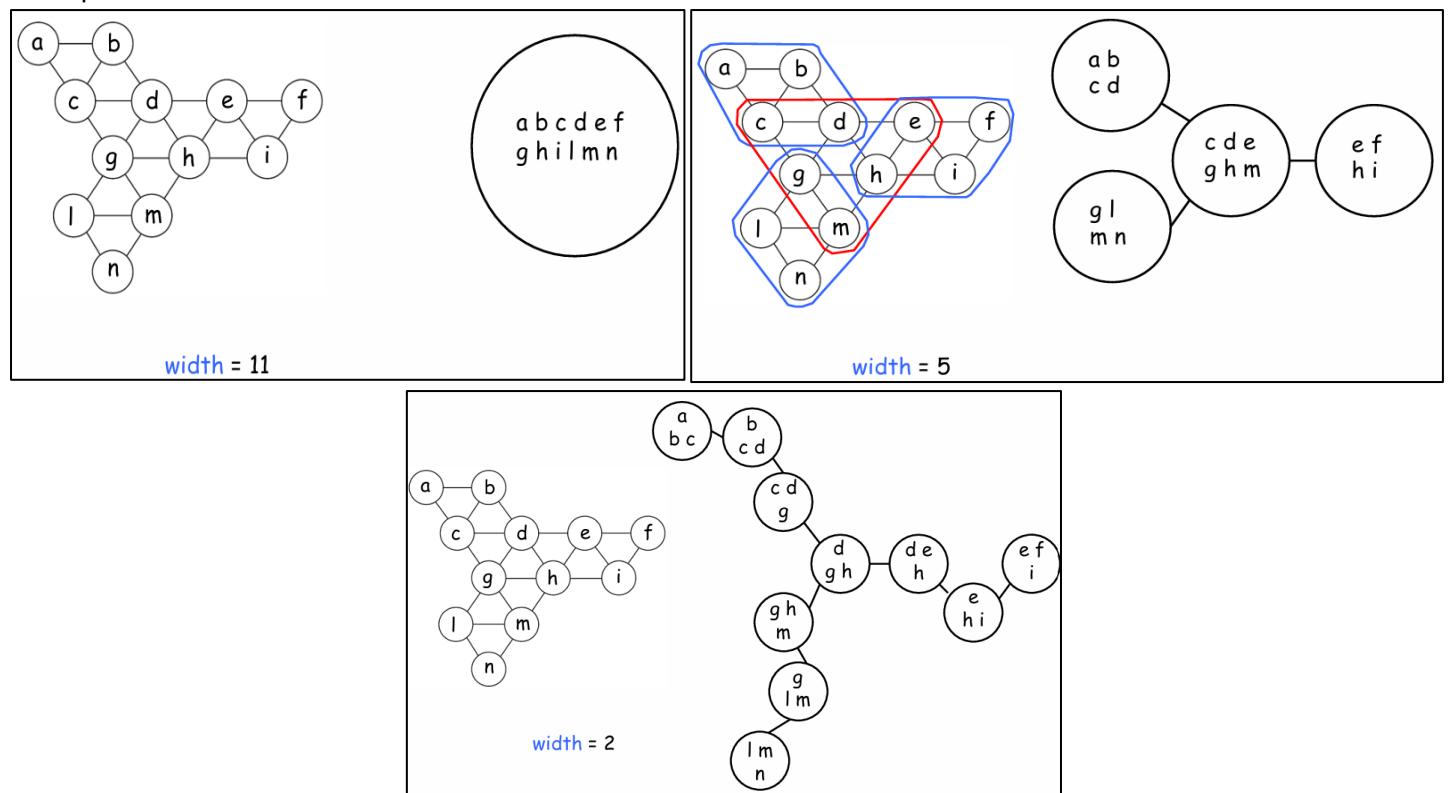
( A sx grafo, a destra la sua tree decomposition che chiamiamo T )

## WIDTH (PROPRIETA' DI TREE DECOMPOSITION)

Informalmente, **width=(size della bag più grande) - 1**; dove una bag ("pezzo") nel disegno sopra è un nodo grigio che compone la tree decomposition. Formalmente:

$$\text{the width of } (T, \{V_t : t \in T\}) : \max_t |V_t| - 1$$

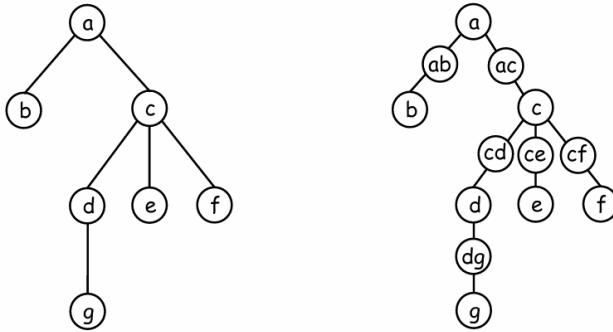
Esempi:



## TREEDWIDTH

**Per treewidth intendiamo la width della migliore tree composition di un grafo**

the **treewidth** of  $G$ : width of the best tree decomposition of  $G$



the **treewidth** of a tree is 1

Più treewidth è piccola maggiore è la vicinanza ad un albero.

## ALCUNI LEMMI E COSE DA SAPERE:

Sia  $T'$  un sottoalbero di  $T$ .

$G_{T'} =$  sottografo di  $G$  indotto dai nodi in tutti i pezzi associati ai nodi di  $T'$ , cioè l'insieme ottenuto dall'unione di tutte le bag.

Eliminazione di un nodo  $t$  da  $T$

→ **LEMMA:** Supponiamo che  $T-t$  ha componenti  $T_1, \dots, T_d$ . Allora i sottografi

$$G_{T_1} - V_t, G_{T_2} - V_t, \dots, G_{T_d} - V_t,$$

non hanno nodi in comune, e non ci sono archi tra loro.

(Cosa significa  $G_{T_1} - V_t$ ? Significa che sto rimuovendo gli archi della bag  $V_t$ ).

→ LEMMA: Siano  $x$  e  $y$  i due componenti di  $T$  dopo l'eliminazione dell'arco  $(x,y)$ . Allora i due sottografi

$$G_x - (V_x \cap V_y) \text{ and } G_y - (V_x \cap V_y)$$

non hanno nodi in comune e non ci sono archi tra di loro.

### DECOMPOSIZIONE RIDONDANTE DELL'ALBERO

Def: una tree decomposition  $(T, \{V_t : t \in T\})$  è ridondante se c'è un arco  $(x,y)$  con  $V_x \subseteq V_y$  (bag di  $x$  contenuta in tutta la bag di  $y$ ).

Come ottenere una decomposizione dell'albero non ridondante:

- ogni volta che una decomposizione dell'albero  $(T, \{V_t : t \in T\})$  è ridondante "contrarre" l'arco  $(x,y)$  "piegando" il pezzo  $V_x$  nel pezzo  $V_y$   
(fondere la bag più piccola in quella più grande)

LEMMA: Qualsiasi tree decomposition non ridondante di un grafo con  $n$  nodi ha al più  $n$  pezzi.

### PROGRAMMAZIONE DINAMICA SU GRAFO CON LARGHEZZA DELL'ALBERO LIMITATA W (PER RISOLVERE WEIGHTED INDEPENDENT SET)

La radice di  $T$  si trova in un nodo  $r$ , inoltre per ogni nodo  $t$  guarderò:

1.  $T_t$  che è il sottoalbero di  $T$  radicato in  $t$
2.  $G_t$  che è il sottografo di  $G$  indotto dai nodi di tutti i pezzi associati ai nodi di  $T_t$

Sottoproblemi = Per ogni nodo  $t$ , e per ogni  $U \subseteq V_t$ :

$$f_t(U) = \text{peso massimo di un independent set } S \text{ nel grafo } G_t \text{ con vincolo } S \cap V_t = U$$

Osservazione:  $f_t(U) = -\infty$  (o indefinito) se  $U$  non è un Independent Set.

Numero di sottoproblemi:  $2^{(w+1)}$  per ogni nodo  $t$  ( $|t| = \text{numero di bag}$ )

$n2^{(w+1)}$  complessivo per la decomposizione non ridondante dell'albero

Goal: calcolare  $\max_{U \subseteq V_r} f_r(U)$

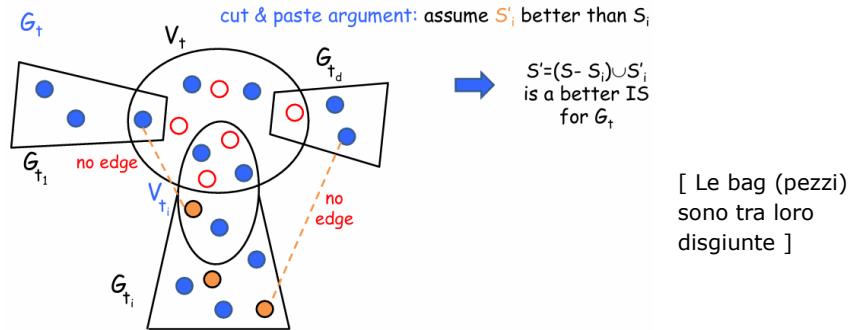
#### ALCUNE DEFINIZIONI + LEMMA

$f_t(U)$  = peso massimo di un independent set  $S$  in  $G_t$ , soggetto al requisito che  $S \cap V_t = U$   
sia  $S$  un IS di peso massimo in  $G_t$  soggetto al requisito che  $S \cap V_t = U$ , cioè  $w(S) = f_t(U)$

Supponiamo che  $T$  abbia figli  $t_1, \dots, t_d$ : allora  $S_i$  è l'intersezione di  $S$  e dei nodi di  $G_{t_i}$

- LEMMA:  $S_i$  è un IS di peso massimo di  $G_{t_i}$ , soggetto a  $S_i \cap V_t = U \cap V_{t_i}$ .

claim:  $S_i$  is opt for  $G_{t_i}$ , subject to  $S_i \cap V_t = U \cap V_{t_i}$



- Peso di un  $S_i$  ottimo:  $\max\{f_{t_i}(U_i) : U_i \cap V_t = U \cap V_{t_i} \text{ and } U_i \subseteq V_{t_i} \text{ is an IS}\}$

#### Risoluzione

→ caso in cui  $t$  è una foglia di  $T$ :  $f_t(U) = w(U)$

→ caso in cui  $t$  ha figli  $t_1, \dots, t_d$  in  $T$ :

$$\star \quad f_t(U) = w(U) + \sum_{i=1}^d \max\{f_{t_i}(U_i) - w(U_i \cap U) : U_i \cap V_t = U \cap V_{t_i} \text{ and } U_i \subseteq V_{t_i} \text{ is an IS}\}$$

Tempo per calcolare  $ft(U)$ :

- per ognuno dei  $d$  figli  $t_i$  e per ogni  $U_i \subseteq V_{t_i}$  controllare in tempo  $O(w)$  se  $U_i$  è un IS e se è consistente dia con  $V_t$  che con  $U$   $O(2^{w+1} w d)$
  - ci sono  $2^{w+1}$  possibili  $U$  per ogni nodo  $t$   $O(4^{w+1} w d)$
- ⇒ sommando tutti i nodi otteniamo il tempo totale di esecuzione:

total running time:  
 $O(4^{w+1} w n)$

### RICAPITOLANDO – COME CALCOLARE UNA TREE-DECOMPOSITION?

- Calcolare una treewidth di un grafo in input è un problema NP-hard.
- Esiste un algoritmo che, dato un grafo con treewidth  $w$ , produce una tree decomposition con width pari a  $4w$  in tempo  $O(f(w) m n)$

**LEZ8 – 02/12/2024 (Complessità parametrizzata, riduzioni parametrizzate, multicolored clique, k-Dominating Set, Circuit Satisfiability, Weighted circuit Satisfiability, Depth & Weft di un circuito, W-Gerarchia, Esponential Time Hypothesis, Trasferimento lower bounds, Strong ETH)**

“Lezione su come teoria permette di dimostrare che certe cose non si possono fare (lower bounds)”.

Che tipo di risultati negativi è possibile dimostrare?

- Possiamo dimostrare che un problema (ad esempio  $k$ -clique) non è FPT
- Possiamo dimostrare che un problema (ad esempio  $k$ -vertex cover) non ha algoritmi di tempo  $2^{o(k)}n^{O(1)}$

Oss: possiamo assumere che  $P \neq NP$

(se  $P=NP$ ,  $k$ -clique può essere risolto in tempo polinomiale e quindi è FPT)

Nota: Questi risultati sulle complessità si chiamano “lower bound condizionati”.

Idea: sviluppare una teoria che fornisca la prova che un problema parametrizzato è difficile (ad esempio, non FPT).

### COMPLESSITÀ PARAMETRIZZATA

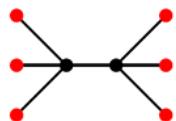
Per dimostrare una teoria di complessità per problemi parametrizzati servono 2 ingredienti:

- Un concetto di riduzione appropriato
- Una ipotesi appropriata(hardness)

Oss: per problemi parametrizzati non possiamo usare il concetto di riduzione polinomiale perché non funziona!

**Example:**  $G$  has an Independent Set of size  $k$  iff has a Vertex Cover of size  $n-k$

IS problem

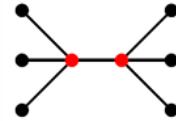


Complexity:



NP-complete

Vertex Cover problem



NP-complete

no  $n^{o(k)}$ -time  
algorithm is known

a  $O(2^k n^{O(1)})$   
algorithm exists



## RIDUZIONI PARAMETRIZZATE

Riduzioni parametrizzate: dato un problema P per ridurlo al problema Q usiamo una funzione  $\phi$  che mappa un'istanza  $(x, k)$  di P in un'istanza  $(x', k') = \phi(x, k)$  di Q, tale che:

1.  $(x, k)$  è un'istanza YES di P se e solo se  $(x', k')$  è un'istanza YES di Q;
2.  $(x', k')$  può essere calcolato in tempo  $f(k)n^{O(1)}$ ;
3.  $k' \leq g(k)$  per qualche funzione  $g$  (punto 3. il più importante!).

Nota: se Q è FPT allora anche P è FPT

Equivalentemente: se P non è FPT allora Q non è FPT

Esempio: da Independent Set a Clique

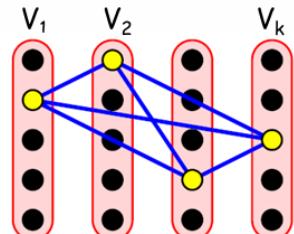
Non esempio: da Independent Set a Vertex Cover

### MULTICOLORED CLIQUE

Input: un grafo  $G=(V,E)$  che ha vertici colorati con  $k$  colori ( $k$  interi non negativi)

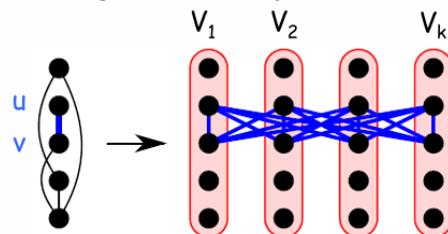
Parametro:  $k$

Scopo: vogliamo capire se  $G$  ha una clique di dimensione  $k$  col vincolo che ogni nodo nella clique deve avere un colore diverso



Teorema: Esiste una riduzione parametrizzata da Clique a Multicolored Clique:

- per ogni vertice  $v$  di  $G$ ,  $G'$  ha  $k$  vertici  $v_1, \dots, v_k$  (uno per ogni colore)
- se  $u$  e  $v$  sono adiacenti in  $G$ , collega tutte le copie di  $u$  con tutte le copie di  $v$



$$G \text{ has a } k\text{-clique} \iff G' \text{ has a multicolored } k\text{-clique}$$

(Similarmente si può avere riduzione da  $k$ -clique a multicolored  $k$ -Independent set)

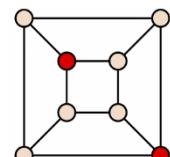
### K-DOMINATING SET (K-DS) (problema np-hard)

Input: un grafo  $G=(V,E)$  ed un intero  $k$  non negativo.

Domanda: esiste un insieme  $U$  di vertici di dimensione  $|U| \leq k$  tale che ogni  $v \in V \setminus U$  è adiacente a un vertice  $u \in U$ ?

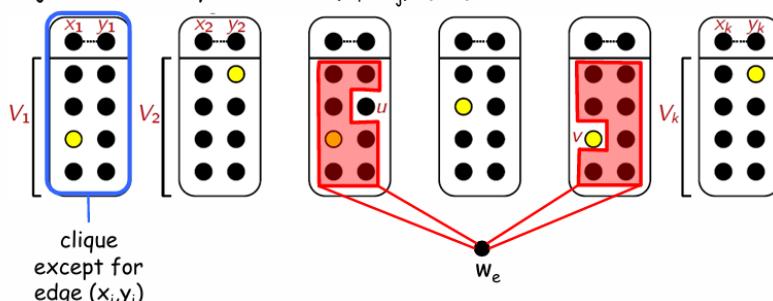
(Dato grafo non pesato trovare k-dominating set)

Parametro:  $k$ .



Teorema: esiste riduzione parametrizzata da Multicolored Independent Set a Dominating Set

- $G'$  has all vertices of  $G$  plus vertices  $x_i, y_i$ , for each color  $i$
- for each edge  $(u,v)$  in  $G$  with  $u \in V_i$  and  $v \in V_j$ , add a vertex  $w_e$  to  $G'$  adjacent to every vertex of  $(V_i \cup V_j) \setminus \{u,v\}$



- Insieme  $V_1$  di nodi colorati 1, insieme  $V_2$  di nodi colorati 2 etc...
- Per ogni gruppo inserisco due vertici:  $x$  e  $y$
- Idea: per ogni nodo metto tutti i possibili archi tranne arco  $x_1-y_1$  (clique except for edge  $(x_i, y_i)$ )
- Aggiungere nodo  $w_e$  da collegare opportunamente.

*Claim:* a k-DS must choose a vertex from each  $V_i$  and such vertices must form an independent set in  $G$ .

Nell'immagine i nodi gialli sono i nodi del k-dominating set (scelto un vertice giallo per ogni  $V_i$ )

## PROBLEMI HARD

Ci sono problemi che sono intrinsecamente difficili. Facendo riduzioni passiamo la difficoltà di un problema ad un altro. Alcuni problemi difficili quanto Clique sono Independent Set, Dominating Set (anche in bipartite graphs), Set Cover, Hitting Set, Connected Dominating Set, ...

→ Questo ci dice grossolanamente che nessuno di questi problemi può essere FPT

Come scegliamo l'equivalente di un problema difficile? Ci sono diverse ipotesi di hardness:

1. Ipotesi degli ingegneri: k-Clique non si può risolvere in tempo  $f(k)n^{O(1)}$  (non risolvibile in tempo FPT)
2. Ipotesi dei teorici: il k-step halting problem non può essere risolto in tempo FPT  
(halting problem: esiste cammino di una macchina di turing non deterministica che termina in k passi?)
3. Ipotesi del tempo esponenziale(ETH): 3-SAT con n variabili non può essere risolto in tempo  $2^{o(n)}$  ["2 alla o-piccolo!"]

Quale ipotesi è più plausibile? Le prime due sono equivalenti, mentre la terza è più forte delle altre due!

## ALCUNE OSSERVAZIONI

- 1) k-Clique e k-Step Halting problem possono essere ridotti l'uno all'altro  
(Ipotesi degli ingegneri e Ipotesi dei teorici sono equivalenti!)
- 2) k-Clique e k-Step Halting problem possono essere ridotti a k-Dominating Set
- 3) Esiste una riduzione parametrizzata da k-Dominating Set a k-Clique?

Probabilmente no. A differenza della NP-completatezza, dove la maggior parte dei problemi sono equivalenti, qui abbiamo una gerarchia di problemi difficili.

- Independent Set è W[1]-complete
- Dominating Set è W[2]-complete

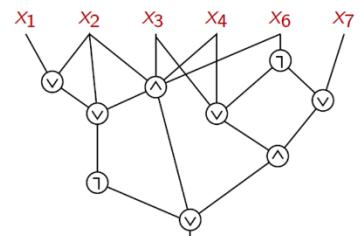
(Non importa se ci interessa solo se un problema è FPT o meno!)

Come si definiscono queste classi W[numero]?

"L'equivalente di SAT in questo caso è un circuito booleano"

→ Un circuito booleano consiste in un insieme di porte in input, porte negate, porte AND, porte OR, e una singola porta di output.

Definiamo i problemi circuit satisfiability e Weighted circuit satisfiability.



## CIRCUIT SATISFIABILITY

Dato un circuito booleano C, decidere se esiste un'assegnazione sugli input di C che rende l'output vero.

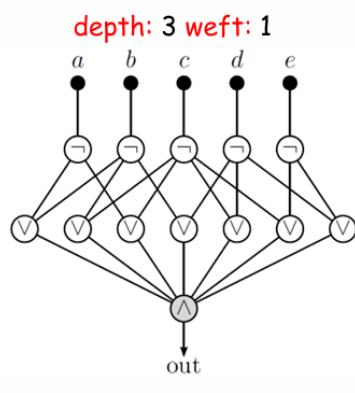
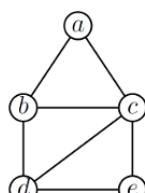
## WEIGHTED CIRCUIT SATISFIABILITY

Dato un circuito booleano C e un numero intero k, decidere se esiste un'assegnazione di peso k che rende l'output vero.

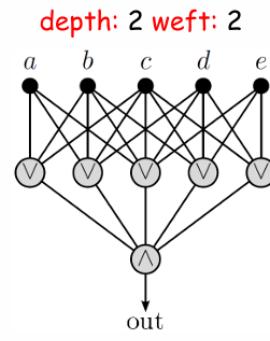
(Peso di un assegnamento = numero di variabili true)

**IMPORTANTE:** Su Weighted Circuit Satisfiability possiamo definire le varie classi W!

Sia k-Independent Set che k-Dominating Set possono essere ridotti a Weighted Circuit Satisfiability



k-Independent Set



k-Dominating Set

Idea: Dominating Set è più difficile di Independent Set perché abbiamo bisogno di un circuito più complicato

DEPTH di un circuito = la lunghezza massima di un cammino di tipo input-output.

Una porta si dice "large" se ha più di 2 input.

WEFT("trama") di un circuito = il numero massimo di porte di tipo large in un cammino di tipo input-output.

### W-HIERARCHY

Sia  $C[t; d]$  l'insieme di tutti i circuiti aventi weft(trama) al massimo 't' e profondità al massimo 'd'

Allora un problema  $P$  è nella classe  $W[t]$  se esiste una costante  $d$  e se esiste una riduzione parametrizzata da  $P$  a Weighted Circuit Satisfiability di  $C[t; d]$ .

Oss1: Independent Set è in  $W[1]$  e Dominating Set è in  $W[2]$

Fact: Independent Set è  $W[1]$ -complete

Fact: Dominating Set è  $W[2]$ -complete

Un problema è 'complete' per una data classe se ogni altro problema nella classe può essere ridotto ad essa. Pertanto una riduzione da DS a IS implicherebbe  $W[1]=W[2]$ .

Si pensa che  $W[1]=W[2]$ ? No, probabilmente NO.

### EXPONENTIAL TIME HIPOTESIS ED ALCUNE CONSEGUENZE IMPORTANTI

Cosa si può fare se assumiamo ETH? Vediamo....

Assumendo ETH vero, se prendiamo 3-SAT con  $n$  variabili non c'è nessun algoritmo di tempo  $2^{o(n)}$  che lo risolve.

#### Exponential Time Hypothesis (ETH)

There is no  $2^{o(m)}$ -time algorithm for m-clause 3-SAT

The textbook reduction from 3-SAT to 3-Coloring:

3-SAT formula  $F$   
 $n$  variables  
 $m$  clauses



a graph  $G$   
 $O(n+m)$  vertices  
 $O(n+m)$  edges

Lemma: se abbiamo istanza di SAT la possiamo ridurre in una istanza di 3-SAT equivalente dove il numero di variabili e il numero di clausole è equivalente.

### TRASFERIMENTO LOWER BOUNDS (PARTENDO DA UN ESEMPIO):

#### Exponential Time Hypothesis (ETH)

There is no  $2^{o(m)}$ -time algorithm for m-clause 3-SAT

The textbook reduction from 3-SAT to 3-Coloring:

3-SAT formula  $F$   
 $n$  variables  
 $m$  clauses



a graph  $G$   
 $O(n+m)$  vertices  
 $O(n+m)$  edges

Assumendo ETH vero, non esistono algoritmi di tempo  $2^{o(n)}$  per 3-coloring su un grafo con  $n$  vertici.

NOTA(TrasferimentoLowerBounds): esistono molte riduzioni simili da 3-SAT ad altri problemi basati su grafi.

Conseguenze: assumendo che ETH sia vero abbiamo che non esiste nessun algoritmo di tempo  $2^{o(n)}$  su un grafo con n vertici per i problemi seguenti:

- Independent Set
- Clique
- Dominating Set
- Vertex Cover
- Longest Path
- ...

Sappiamo che esistono molte riduzioni a partire da 3-SAT verso altri problemi basati su grafi.

Conseguenze su f(k): Assumendo ETH, non esistono algoritmi di tempo  $2^{o(k)} n^{O(1)}$  per grafi con n vertici per i problemi seguenti:

- k-Independent Set
- k-Clique
- k-Dominating Set
- k-Vertex Cover
- k-Path
- ...

(come prima ma con parametri k)

Assumendo ETH possiamo dimostrare che k-Clique non è FPT.

Teorema: non solo non risolvibile in FPT ma addirittura l'esponente di n deve essere lineare in k

#### STRONG ETH (SETH)

##### **Strong ETH (SETH)**

**There is no  $(2-\varepsilon)^n$ -time algorithm for CNF-SAT**

Per qualsiasi k fissato, un algoritmo di tempo  $n^{k-0.01}$  per k-DS violerebbe SETH.

assuming SETH:

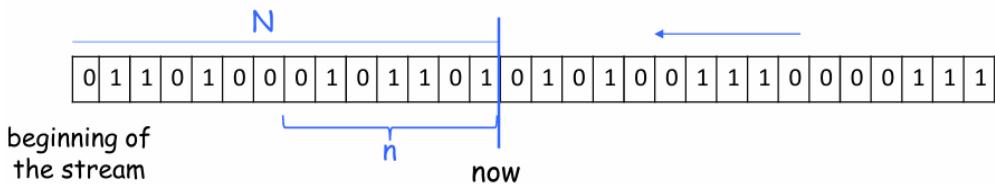
- no  $n^{2.99}$  time algorithm for 3-DS
- no  $n^{3.99}$  time algorithm for 4-DS
- no  $n^{4.99}$  time algorithm for 5-DS
- ...

#### NOTA LEZIONE

Inizialmente cercare di capire se siamo in FPT oppure no. Se non siamo in FPT allora vogliamo capire se siamo in W[1]-HARD.

CONTARE IL NUMERO DI 1 IN UNA FINESTRA

→ Il problema:



Obiettivo: elaborare un flusso di bit in ordine per rispondere a query del tipo: "quanti 1 ci sono negli ultimi n bit?"

Motivazione: (approssimativamente) contare gli eventi che soddisfano un certo criterio.

Esempio: I post/tweet sono contrassegnati con un flag=1 quando riguardano un argomento specifico. Le query possono essere utilizzate per rilevare se l'interesse per l'argomento cambia.

Sfida principale: il flusso è troppo grande per essere memorizzato interamente.

→ Costruire una struttura dati che mantenga una sequenza di N bit soggetti a:

- query(n) restituisce il numero di 1 negli ultimi n bits
- update(b) aggiunge il prossimo bit b alla sequenza

Nota: se vuoi la risposta esatta hai bisogno di  $\Omega(N)$  bits

→ Datar-Gionis-Indyk-Motwani's (DGIM) Data Structure:**DGIM data structure:**

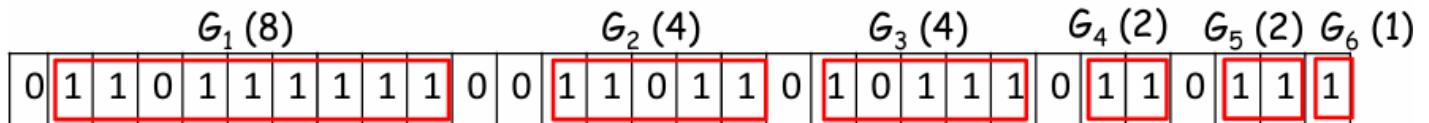
- **quality:**  $1+\epsilon$  approximated answers (for any  $\epsilon > 0$ )
- **size:**  $O(\epsilon^{-1} \log^2 N)$  bits
- **update time:**  $O(\log N)$
- **query time:**  $O(\epsilon^{-1} \log n)$

Idea DGIM Data Structure:

Sia  $B=1/\epsilon$

Raggruppare i bit della sequenza in gruppi  $G_1, \dots, G_T$  che soddisfino i seguenti punti:

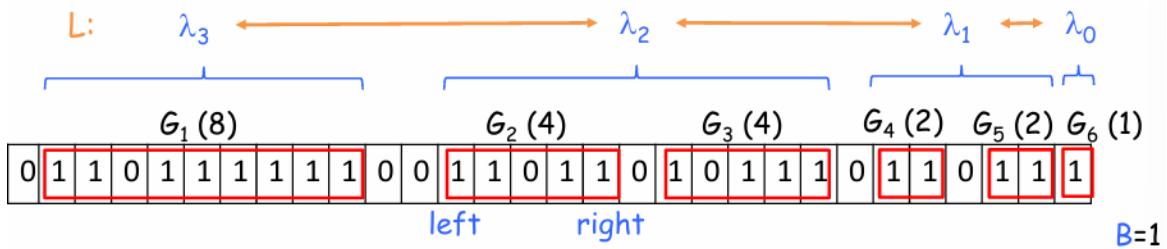
1. ogni  $G_i$  inizia e finisce con un bit di valore pari a 1;
2. tra gruppi adiacenti  $G_i$  e  $G_{i+1}$  ci sono solo bit di valore 0;
3. ogni  $G_i$  contiene  $2^k$  bit di valori pari a 1, per qualche  $k \geq 0$ ;
4. per qualsiasi  $1 \leq i < t$ , se  $G_i$  contiene  $2^k$  bit di valore pari a 1, allora  $G_{i+1}$  contiene  $2^k$  oppure  $2^{k-1}$  bit;
5. per ogni  $k$  eccetto quello più grande, il numero  $Z_k$  di gruppi contenenti  $2^k$  bits di valore 1 soddisfa  $B \leq Z_k \leq B+1$ . Per il  $k$  più grande abbiamo bisogno solo di  $Z_k \leq B+1$ .



$B=1$

Il gruppo  $G_j$  è una coppia di numeri interi ( $left, right$ ), tutti i gruppi adiacenti che hanno  $2^i$  1-bits (1bit=bit di valore pari a 1) sono mantenuti da una lista doppiamente collegata chiamata  $\lambda_i$   
 $\lambda_i$  memorizza: head, tail e size

L: una lista globale doppiamente collegata che memorizza tutte le liste  $\lambda_i$ .



Memorizzare  $G_j$  richiede  $O(\log N)$  bits

$$|L| = O(\log N)$$

$$|\lambda_i| \leq B+1 = O(\varepsilon^{-1})$$



overall size of the DS:  
 $O(\varepsilon^{-1} \log^2 N)$  bits

Operazione di update:

update( $b$ ): aggiungi il bit successivo  $b \in \{0,1\}$  alla sequenza

- Se  $b=0$  non fare nulla.
- Se  $b=1$  allora:
  1. Crea un nuovo gruppo con il nuovo 1-bit e aggiungilo a  $\lambda_0$ ;
  2. Se  $\lambda_0$  contiene  $B+2$  gruppi allora unisci i due gruppi più a sinistra formando così un nuovo gruppo di 2 1-bits e aggiungilo a  $\lambda_1$  come un nuovo gruppo più a destra (nota che  $\lambda_0$  ora ha  $B$  gruppi);
  3. Ripetere il passaggio 2 per  $\lambda_i$ ,  $i=1,2,\dots$ ;

tempo di update:

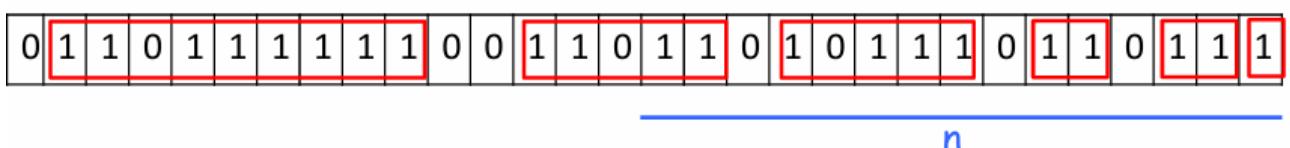
- la creazione/unione/spostamento di un gruppo richiede tempo  $O(1)$
- numero di iterazioni è  $O(|L|)$
- ⇒ tempo di aggiornamento complessivo:  $O(\log N)$

Operazione di query:

query( $n$ ): restituisce il numero di 1 negli ultimi  $n$  bits.

Funzionamento: trovare tutti i gruppi che intersecano gli ultimi  $n$  bits, poi restituire il numero di 1-bit che contengono.

Tempo: necessita di navigare tutti i gruppi dalla testa dello streaming, tempo =  $O(\varepsilon^{-1} \log n)$ .



Nota: l'operazione di query restituisce un valore approssimato, non del tutto esatto.

## TROVARE GLI ELEMENTI PIU' FREQUENTI IN UNO STREAM (UN'APPLICAZIONE DEL CAMPIONAMENTO)

→ Il Problema: dato un flusso di elementi, trova gli elementi la cui frequenza è superiore a una soglia data.

Dati due parametri  $0 < \varepsilon < \varphi < 1$ , e un flusso di  $n$  elementi  $x_1, x_2, \dots, x_n$ , Trovare:

- tutti gli elementi la cui frequenza è almeno  $\varphi n$  (nessun falso negativo).
- nessun elemento con frequenza inferiore a  $(\varphi - \varepsilon)n$ .

→ STICKY SAMPLING ALGORITHM (Manku e Motwani, 2002) – "Algoritmo di campionamento appiccicoso"

- randomizzato
- raggiungere i due obiettivi con probabilità  $1 - \delta$
- mantenere un campione di dimensione prevista pari a  $2\varepsilon^{-1} \log(\varphi^{-1} \delta^{-1})$

$0 < \delta < 1$ : parametro di errore definito dall'utente

avviso: lo spazio è indipendente dalla lunghezza del flusso(stream)  $n$ .

Funzionamento dell'algoritmo:

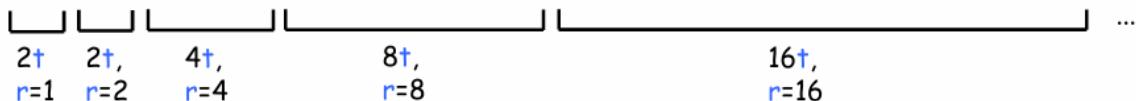
$$t = \varepsilon^{-1} \log(\varphi^{-1} \delta^{-1})$$

mantenere un campione  $S$ : insieme di coppie  $(x, f_e(x))$

- $f_e(x)$ : stima della frequenza reale  $f(x)$  dell'elemento  $x$

per gestire un flusso potenzialmente illimitato, il calcolo procede in finestre

- ogni finestra ha una dimensione e una frequenza di campionamento  $r$ ;
- la finestra successiva ha una grande e una frequenza che sono il doppio della precedente;
- all'inizio  $S$  è vuoto e  $R=1$ .



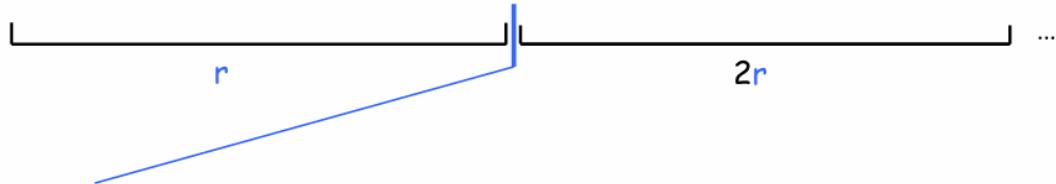
in una finestra data, quando arriva il prossimo elemento del flusso  $x$ :

- se  $x \in S$  allora incrementa  $f_e(x)$  di 1;
- altrimenti, inserisci  $(x, 1)$  in  $S$  con probabilità  $1/r$ .

Ogni volta che la frequenza di campionamento cambia da  $r$  a  $2r$  viene eseguita una fase di regolazione(adjusting step), il cui obiettivo è quello di trasformare lo stato di  $S$  in quello in cui sarebbe stato se il nuovo tasso  $2r$  fosse stato utilizzato fin dall'inizio.

query( $n$ ): ritorna tutti gli elementi in  $S$  con frequenza stimata almeno  $(\varphi - \varepsilon)n$ .

ADJUSTING STEP:



for each element  $x \in S$ :

- lancia una moneta equa
- se esce croce allora
  - o lancia ripetutamente una moneta con probabilità di successo di  $1/(2r)$  finché non ottieni un successo;
  - o sia  $k$  il numero di lanci di moneta eseguiti;
  - o diminuire  $f_e(X)$  di  $k$
  - o se  $f_e(X) <= 0$  allora rimuovi  $x$  da  $S$ .

Oss:  $r$  è sempre una potenza di 2 => assumiamo  $r=2^i$

Lemma: Sia  $n$  il numero di elementi dello stream visti finora e assumiamo che la frequenza di campionamento corrente sia  $r$ . Allora  $1/r \geq t/n$ .

Lemma :  $2t/n \geq 1/r \geq t/n$

### Theorem

For any  $\varepsilon, \varphi, \delta \in (0,1)$ , with  $\varepsilon < \varphi$ , sticky sampling solves the frequent items problem with probability at least  $1 - \delta$  using a sample of expected size  $2\varepsilon^{-1} \log(\varphi^{-1} \delta^{-1})$ .

notice:  $f_e(x) \leq f(x)$   algorithm never return an item with  $f(x) < (\varphi - \varepsilon)n$ .

what about the size of  $S$ ?

worst case: all stream elements are distinct

### Riassumendo la lezione con appunti sparsi presi in presenza

Input:

1.  $0 < \text{epsilon} < \bar{f} < 1$  (parametro epsilon e parametro  $\bar{f}$  in input)
2.  $x_1, \dots, x_n$

Goal:

- 1) trova tutti elementi  $x$  t.c. la loro frequenza è almeno  $\bar{f}$  per  $n$
- 2) non tornare tutti gli elementi la cui frequenza è  $<=(\bar{f} - \text{epsilon})n$

Soluzione: sticky sample (algoritmo randomizzato basato su campionamento)

- a. matcha 1 e 2 con probabilità almeno  $1 - \delta$  ( $\delta = \text{err. prop}$ )
- b. dove size attesa dipende da parametri  $\bar{f}$ , epsilon, delta e non da  $n$

Definiamo struttura dati:

1. fissiamo  $t = \log(\bar{f}) / \text{epsilon}$
2. mantenere un insieme  $S$  di coppie  $(x, \tilde{f}_t)$
3. come funziona l'algoritmo? Funziona su finestre
  - a. ogni finestra ha un sample rate  $r \geq 1$
  - b. ogni finestra ha una size  $rt$  (size pari a "r per t")
4. a tempo iniziale  $S$  inizializzato come insieme vuoto

funzionamento algoritmo della struttura dati:

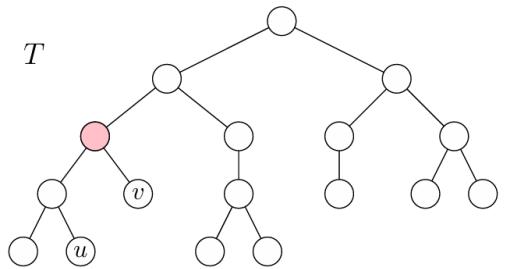
se c'è una tupla che include  $x$  in  $S$  allora incremento si 1 la sua frequenza ( $\tilde{f}_t$  di  $x$ )

se invece non c'è inserisco  $(x, 1)$  dentro  $S$  con probabilità  $1/r$

### PROBLEMA LOWEST COMMON ANCESTOR (LCA PROBLEM)

“Trovare l’antenato comune di profondità massima”.

Def: il lowest common ancestor (antenato comune più basso) LCA<sub>T</sub>(u,v) di u e di v in un albero radicato T è un vertice di depth(profondità) massima che è un antenato sia di u che di v (nodo rosa).



Il problema: dato T, costruire una struttura dati che permetti di pre-elaborare T per rispondere efficientemente alla query LCA seguente:

Query(u,v): ritorna LCA<sub>T</sub>(u,v)

Possibili soluzioni non efficienti hanno caratteristiche seguenti:

L’obiettivo è trovarne una migliore!

Trivial solutions:

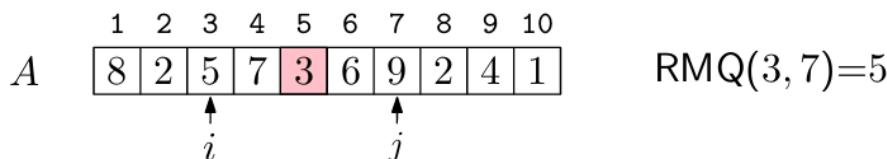
$n = \# \text{ of nodes}$

- Preprocessing time: none Size:  $O(n)$  Query time:  $O(n)$
- Preprocessing time:  $O(n^3)$  Size:  $O(n^2)$  Query time:  $O(1)$
- Preprocessing time:  $O(n^2)$  Size:  $O(n^2)$  Query time:  $O(1)$

### PROBLEMA RMQ CORRELATO AL PROBLEMA LCA

Dato un array A= $a_1, \dots, a_n$ , costruire una struttura dati in grado di pre-elaborare un array A per rispondere alla Range Minimum Query seguente:

RMQ(i,j): ritorna un elemento in  $\min_{k=1, \dots, j} a_k$

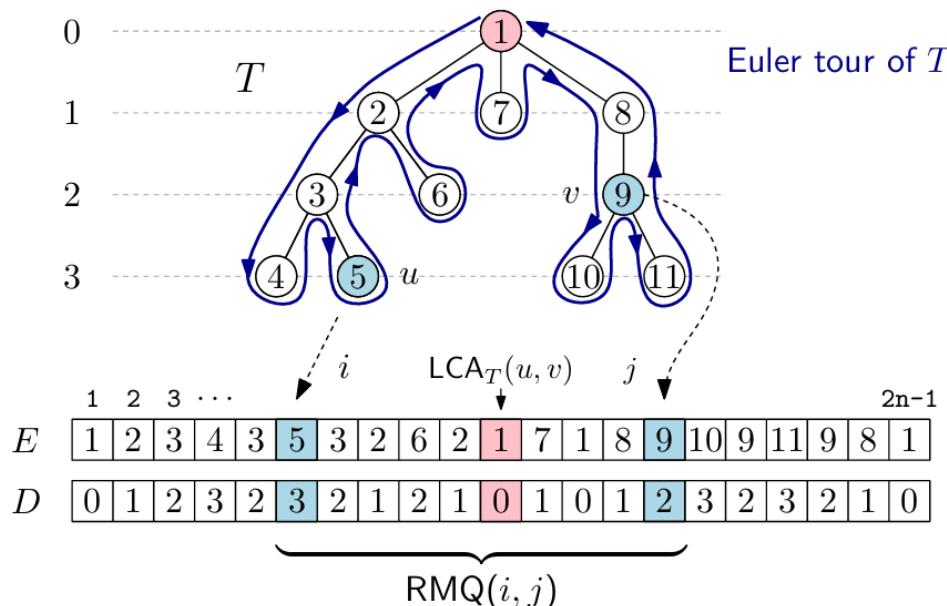


Alcune soluzioni banali:

- Preprocessing time: none Size:  $O(n)$  Query time:  $O(n)$
- Preprocessing time:  $O(n^3)$  Size:  $O(n^2)$  Query time:  $O(1)$
- Preprocessing time:  $O(n^2)$  Size:  $O(n^2)$  Query time:  $O(1)$

### RIDUZIONE DI LCA QUERIES A RMQ

Esiste riduzione da Lowest Common Ancestor Query a Range Minimum Query.



E = vettore elementi  
D = vettore dept

Sia  $u, v \in T$ , e sia  $i$  (rispettivamente  $j$ ) l’indice della prima occorrenza di  $u$  (rispettivamente  $v$ ) in E.  
 $\Rightarrow \text{LCA}_T(u, v) = \text{E}[\text{RMQ}(i, j)]$

## SOLUZIONI AL PROBLEMA RMQ

### 1) SOLUZIONE SPARSE TABLE

#### “Sparse Table” Solution to RMQ

For  $i = 1, \dots, n$  and  $\ell = 2^0, 2^1, \dots, 2^{\lfloor \log n \rfloor}$ , define:

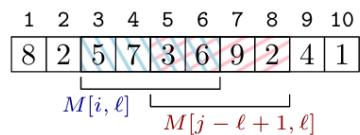
$$M[i, \ell] = \arg \min_{i \leq k < i + \ell} a_k$$

#### Preprocessing:

$$M[i, \ell] = \begin{cases} i & \text{if } \ell = 1 \\ \arg \min_{k \in \left\{ M\left[i, \frac{\ell}{2}\right], M\left[i + \frac{\ell}{2}, \frac{\ell}{2}\right] \right\}} a_k & \text{if } \ell > 1 \end{cases}$$

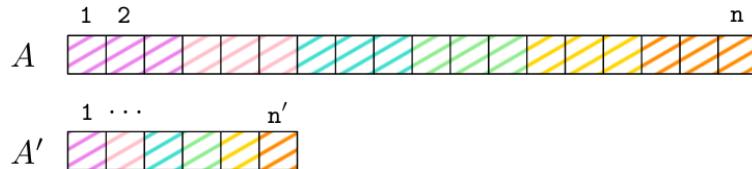
#### Answering a query:

Let  $\ell = 2^{\lfloor \log(j-i+1) \rfloor}$        $\text{RMQ}(i, j) = \arg \min_{k \in \{M[i, \ell], M[j-\ell+1, \ell]\}} a_k$



- Preprocessing time:  $O(n \log n)$
- Size:  $O(n \log n)$
- Query time:  $O(1)$

### 2) SOLUZIONE CON ORACOLO PIU' COMPATTO



### 3) CASO SPECIALE (si basa su blocchi con valore adiacenti tra loro distanti di un valore pari a +/-1)



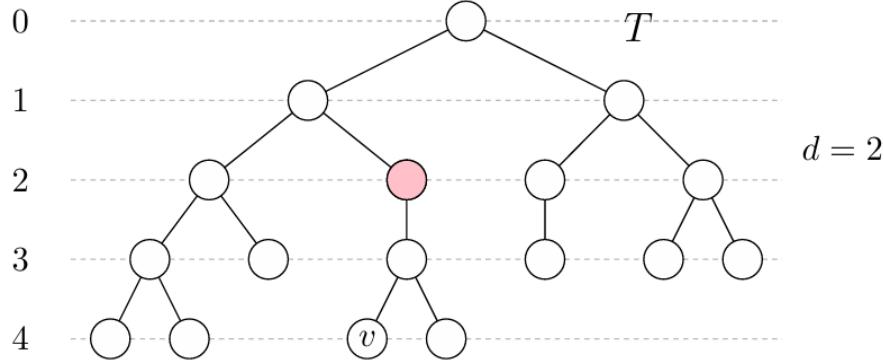
Recap soluzioni:

## RMQ Solutions: Recap

Size	Preprocessing Time	Query Time	Notes
$O(n)$	$O(n)$	$O(n)$	
$O(n^2)$	$O(n^3)$	$O(1)$	
$O(n^2)$	$O(n^2)$	$O(1)$	
$O(n \log n)$	$O(n \log n)$	$O(1)$	Sparse Table
$O(n)$	$O(n)$	$O(\log n)$	
$O(n \log \log n)$	$O(n \log \log n)$	$O(1)$	
$O(n)$	$O(n)$	$O(1)$	$\pm 1$ RMQ
$O(n)$	$O(n)$	$O(1)$	General case

## LEVEL ANCESTOR

Def: sia  $v$  il vertice di profondità  $d_v$  in  $T$ . Per  $d \leq d_v$ , una Level Ancestor Query  $LA(v, d)$  su un vertice  $v$  chiede di segnalare l'antenato di  $v$  di profondità pari a  $d$ .



Il problema: Dato  $T$ , costruire una struttura dati in grado di pre-elaborare  $T$  per rispondere alle Level Ancestor Queries.

Soluzioni banali:

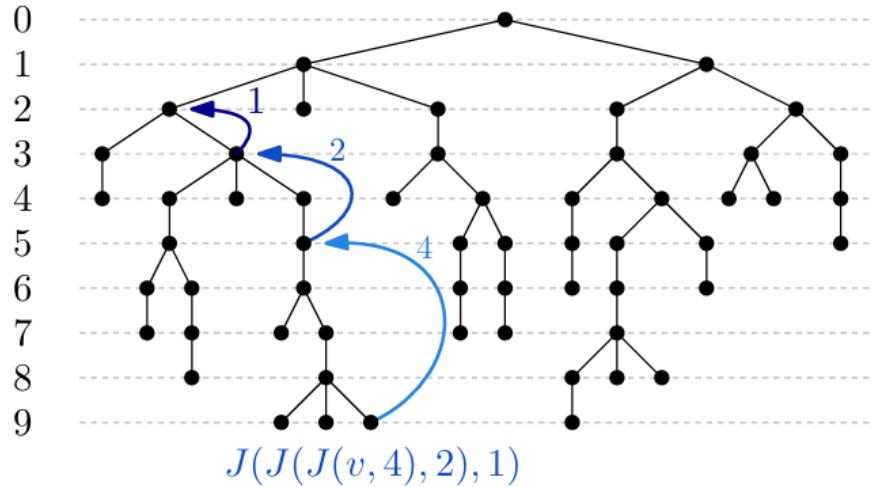
Trivial solutions:

$n = \# \text{ of nodes}$

- Preprocessing time: none Size:  $O(n)$  Query time:  $O(n)$
- Preprocessing time:  $O(n^3)$  Size:  $O(n^2)$  Query time:  $O(1)$
- Preprocessing time:  $O(n^2)$  Size:  $O(n^2)$  Query time:  $O(1)$

## JUMP POINTERS

Idea: per ogni vertice  $v$  e per ogni  $\ell = 0, 1, \dots, \log(d_v)$  salvare  $J(v, \ell) = LA(v, d_v - 2^\ell)$



Costruzione jump pointers:

**With a DFS visit of  $T$ :**

- Maintain a stack  $S$  that stores all the ancestors of the current vertex  $v$  of the visit
- $S$  can be updated in  $O(1)$  per traversed edge
- When vertex  $v$  is visited, its ancestor at depth  $d$  in  $T$  is the  $(d_v - d)$ -th vertex from the top of the stack

**Or using dynamic programming:**

$$J(v, \ell) = \begin{cases} \text{parent}(v) & \text{if } \ell = 0 \\ J(J(v, \ell - 1), \ell - 1) & \text{if } \ell > 0 \end{cases}$$

**Time complexity:**  $O(n + n \log h) = O(n \log n)$

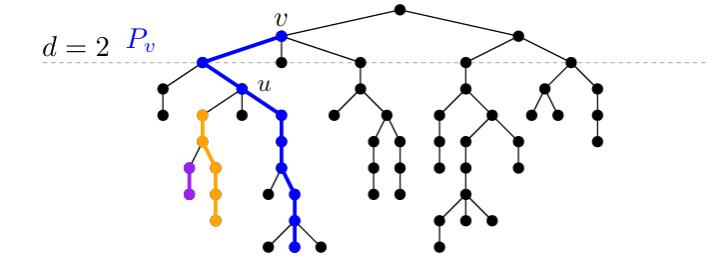
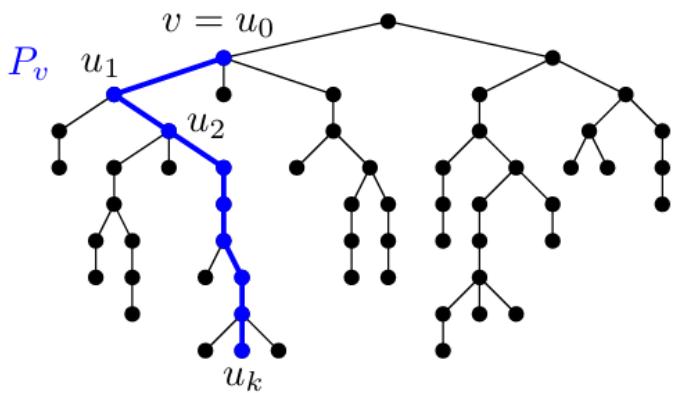
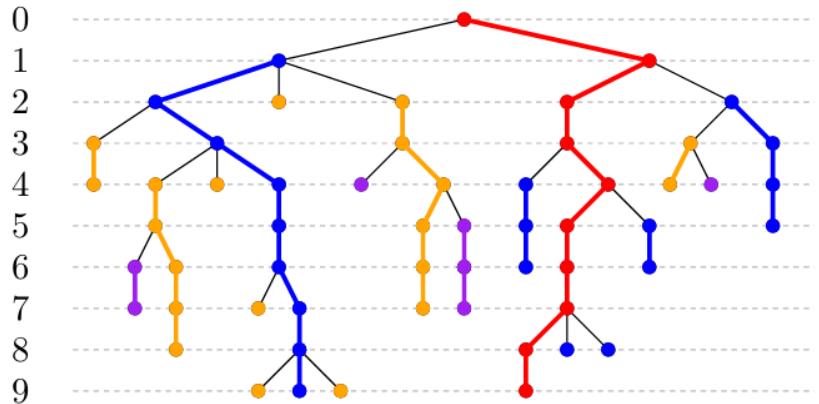
Tempi jump pointers:

Size	Preprocessing Time	Query Time	Notes
$O(n \log n)$	$O(n \log n)$	$O(\log n)$	Jump Pointers

### LONG PATH DECOMPOSITION

Partizionare  $T$  in una collezione di cammini  $D$ . Definiamo ricorsivamente:

- Selezionare uno dei più lunghi percorsi  $P$  di tipo radice-foglia in  $T$
- Selezionare cammini ricorsivamente da ogni albero della foresta  $T \setminus P$



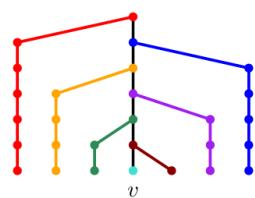
To report  $\text{LA}(u, d)$ :

- Let  $v = \tau(u)$
- If  $d \geq d_v$ : return  $A_v[d - d_v]$ .
- If  $d < d_v$ : return  $\text{LA}(\text{parent}(v), d)$ . (recursively)

Time:  $O(\# \text{recursive calls}) = O(\#\text{paths in } D \text{ from } v \text{ to the root})$ .

Nota: il numero di cammini distinti in  $D$  incontrati nel cammino  $P$  da  $v$  alla radice in  $T$  sono  $O(\sqrt{n})$

Tight:



Time:  $\Omega(\sqrt{n})$

Tempi Long Path Decomposition:

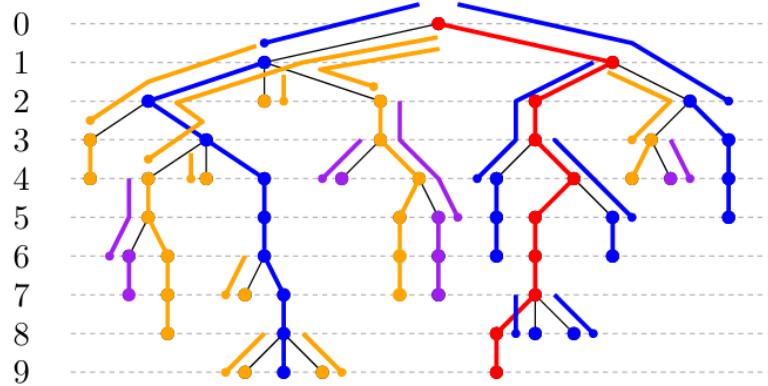
Size	Preprocessing Time	Query Time	Notes
$O(n)$	$O(n)$	$O(\sqrt{n})$	Long Path Dec.

## LONG PATH DECOMPOSITION + LADDERS

Sia  $\eta(P_v)$  il numero di vertici del cammino  $P_v \in D$ .  
 Estendere ogni cammino  $P_v \in D$  in una ladder ("scala")  $L_v$  con  $\eta(P_v)$  più vertici attraverso la radice (se loro esistono).

Per ogni ladder  $L_v = <v' = u_0, u_1, \dots, v = u_j, \dots, u_k>$ :

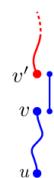
- Memorizzare in  $v$  un array  $B_v$  di lunghezza  $k+1$  dove  $B_v[i]$  contiene (un riferimento a)  $u_i$
- Ogni  $u_i$  con  $i >= j$  memorizza un riferimento  $\tau(u_i)$  a  $v$



La lunghezza di  $B_v$  è al massimo il doppio della lunghezza di  $A_v \Rightarrow$  la dimensione totale è almeno  $O(n)$ .

Per ritornare  $LA(u, d)$ :

- Sia  $v = \tau(u)$  e sia  $v' = B_v[0]$
- Se  $d >= d_{v'}$  allora ritorna  $B_v[d - d_{v'}]$
- Se  $d < d_{v'}$  allora ritorna  $LA(v', d)$  (ricorsivamente)



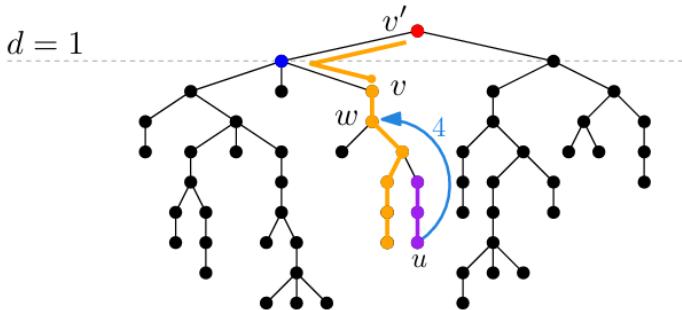
Quante chiamate ricorsive?

L'altezza del vertice interrogato raddoppia ad ogni interazione  $\Rightarrow O(\log n)$  iterazioni

Tempi Long Path Decomposition + Ladders:

Size	Preprocessing Time	Query Time	Notes
$O(n)$	$O(n)$	$O(\log n)$	+ Ladders

## LONG PATH DECOMPOSITION + LADDERS + JUMP POINTERS



$$0 < d_u - d = 2^{\ell_k} + 2^{\ell_{k-1}} + \dots + 2^{\ell_1}$$

To report  $LA(u, d)$ :

- Let  $w = J(u, \ell_k)$ ,  $v = \tau(w)$  and  $v' = B_v[0]$ .
- Return  $B_v[d_{v'} - d]$ .

Query time:  $O(1)$

Jump Pointers  
 Space usage:  $O(n + \underbrace{n \log n}_{\text{Ladders}}) = O(n \log n)$

**A trick to reduce space:**

- Only store jump pointers  $J(v, \ell)$  in the leaves  $v$  of  $T$ .
- For each node  $u$  of  $T$ , store a reference to a leaf  $\lambda_u$  in the subtree of  $T$  rooted at  $u$ .
- $LA(u, d) = LA(\lambda_u, d)$

Space usage:  $O(n + L \log n)$ , where  $L = \# \text{leaves of } T$ .

Tempi Long Path Decomposition + Ladders + Jump Pointers:

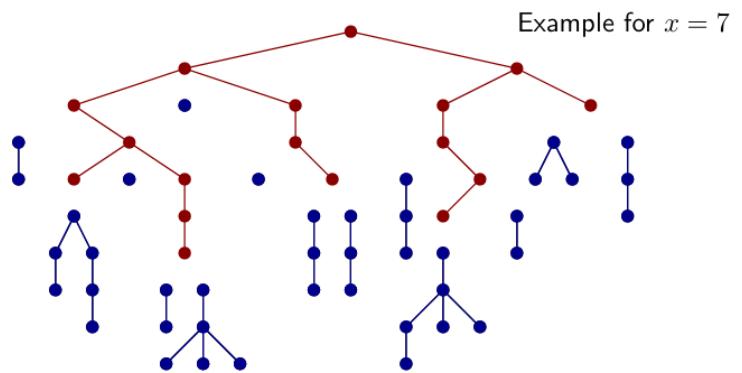
Size	Preprocessing Time	Query Time	Notes
$O(n + L \log n)$	$O(n + L \log n)$	$O(1)$	+ Ladders, JP

If only we had  $O(\frac{n}{\log n})$  leaves...

## MACRO-MICRO TREES

Trovare l'insieme  $M$  di tutti i vertici di profondità massima con almeno  $x = (1/4)\log n$  discendenti.

Dividere  $T$  in un macro-tree  $T'$  contenente tutti gli antenati dei vertici in  $M$  e diversi micro-tree in  $T \setminus T'$ .



Example for  $x = 7$

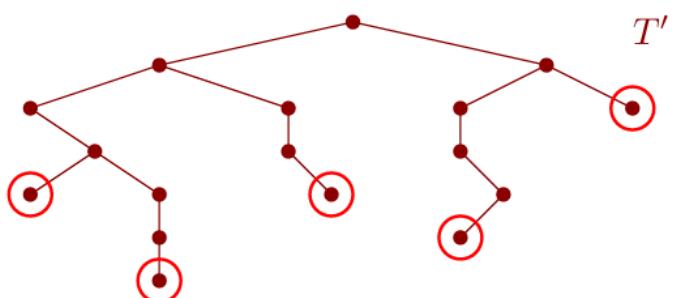
### → Gestione del macro-tree:

Quante foglie in  $T'$ ? le foglie di  $T'$  sono i vertici in  $M$ . Ogni vertice in  $M$  ha almeno  $(1/4)\log n$  discendenti in  $T$ .

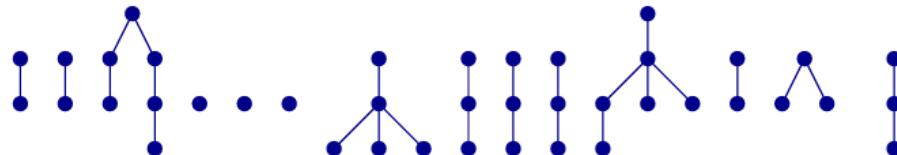
**Build the previous LA oracle  $\mathcal{O}'$  on  $T'$ .**

**Size/build time:**  $O(n + |M| \log n) = O(n)$ .

**Query time:**  $O(1)$

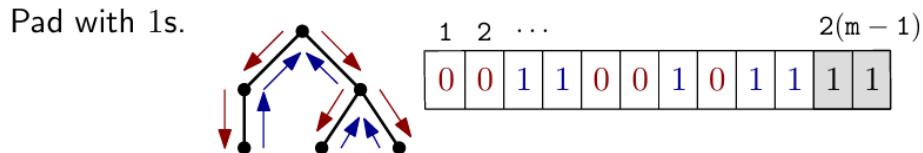


### → Gestione dei micro-tree:



Un albero radicato su  $\leq m$  vertici può essere rappresentato in modo univoco con un array di  $2(m-1)$  bits.

Eseguire DFS. Scrivere 0 quando un arco è attraversato verso le foglie, e 1 quando è attraversato verso la radice.



Quanti tipi diversi di micro-tree?  $O(\sqrt{n})$  tipi di micro-tree.

Per ognuno degli  $O(\sqrt{n})$  tipi di micro-tree  $T_i$ , costruire un oracolo  $O_i$  con dimensione/preprocessing time  $O(|T_i|^2)$  e query time  $O(1)$ .

Per ogni vertice  $u$  di  $T$  che appartiene ad un micro tree:

- Memorizzare in  $u$  l'indice  $i$  del tipo del suo micro-tree
- Memorizzare in  $u$  il vertice  $\mu(u)$  in  $T_i$  corrispondente ad  $u$
- Memorizzare in  $u$  la radice  $\rho(u)$  del suo micro-tree

Dimensione/tempo totale =  $O(n)$ .

### → Rispondere a una query: To answer $LA(u, d)$ :

- If  $u$  is in the macro tree  $T'$ : query  $\mathcal{O}'$  for  $LA(u, d)$ .
- If  $u$  is in a micro-tree  $T''$ :
  - If  $d < d_{\rho(u)}$ : query  $\mathcal{O}'$  for  $LA(\text{parent}(\rho(u)), d)$ .
  - Otherwise:
    - Let  $i$  be the type of the micro-tree containing  $u$ .
    - Query  $O_i$  for  $LA(\mu(u), d - d_{\rho(u)})$ .

Query time:  $O(1)$

(and map it back to a vertex in  $T''$ )

## RECAP SOLUZIONI

Size	Preprocessing Time	Query Time	Notes
$O(n)$	–	$O(n)$	
$O(n^2)$	$O(n^3)$	$O(1)$	
$O(n^2)$	$O(n^2)$	$O(1)$	
$O(n \log n)$	$O(n \log n)$	$O(\log n)$	Jump Pointers
$O(n)$	$O(n)$	$O(\sqrt{n})$	Long Path Dec.
$O(n)$	$O(n)$	$O(\log n)$	+ Ladders
$O(n + L \log n)$	$O(n + L \log n)$	$O(1)$	+ Ladders, JP
$O(n)$	$O(n)$	$O(1)$	+ Macro-Micro trees

Appunti presi in presenza:

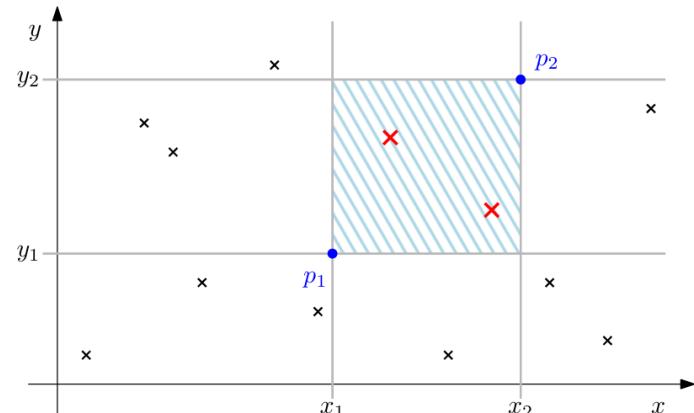
1. Long-Path Dec. : idea=Dimezzi distanza
2. +Ladders : idea=Raddoppi altezza

## **LEZ11 – 10/12/2024**

### RANGE TREES

Input = un insieme  $S$  di  $n$  punti  $D$ -dimensionali  
 Goal = costruire struttura dati che, dati  $p_1 \in \mathbb{Z}^D$  e  $p_2 \in \mathbb{Z}^D$ , può:

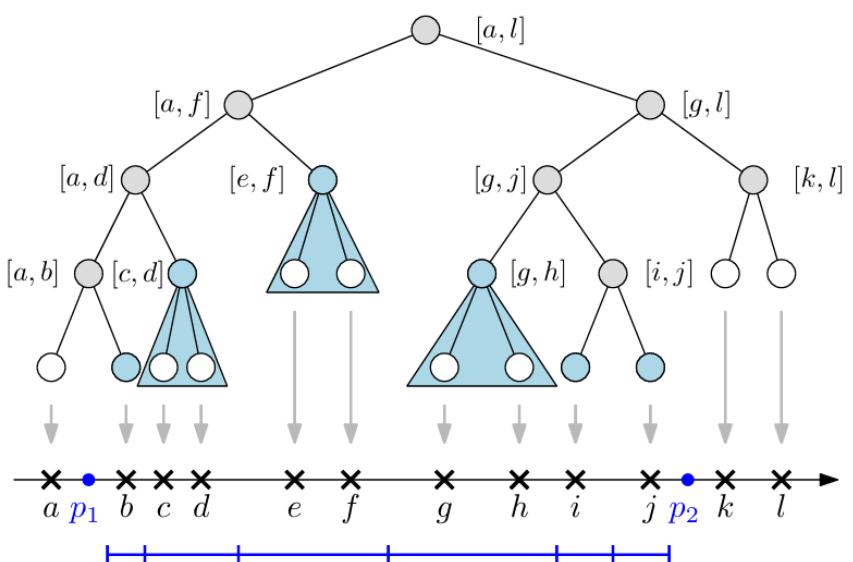
1. Ritornarne il numero di punti  $q \in S$  t.c.  $p_1 \leq q \leq p_2$
2. Ritornare l'insieme dei punti  $q \in S$  t.c.  $p_1 \leq q \leq p_2$
3. Ritornare il punto  $q \in S$  t.c.  $p_1 \leq q \leq p_2$  con la  $D$ -esima coordinata più piccola
4. ...



### → Caso $D=1$

- I punti sono interi
- Memorizzare i punti in un array ordinato (in tempo  $O(n \log n)$ )
- Eseguire query tramite ricerca binaria per  $p_1$  e  $p_2$
- Query time:  $O(\log n + k)$ , con  $k$  che è la dimensione dell'output
  - o  $K = \text{punti ritornati}$
  - o  $K = \Theta(1)$  se ci interessa soltanto il numero di punti
- Complessità spaziale:  $O(n)$

### Range Trees: $D = 1$



## Costruzione Range Tree per D=1:

- ordinamento preliminare di S (solo una volta)
- dividere S in S1 e S2 di circa  $n/2$  elementi ciascuno
- ricorsivamente costruire T1 e T2 a partire da S1 e S2 rispettivamente
- La radice di T ha T1 e T2 come suoi sottoalberi sinistro e destro
- Ritornare T

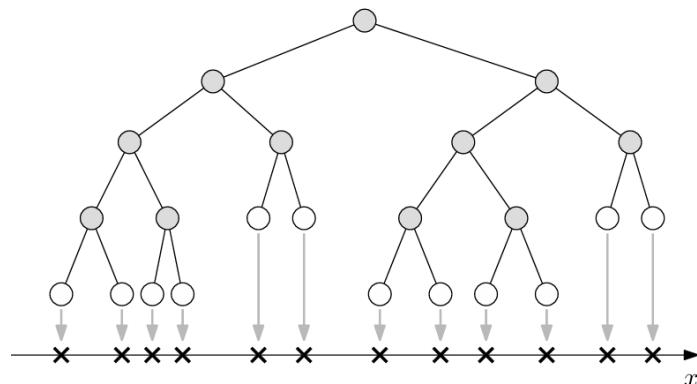
Tempo:  $O(n \log n)$

Se S già ordinato allora Tempo:  $O(n)$

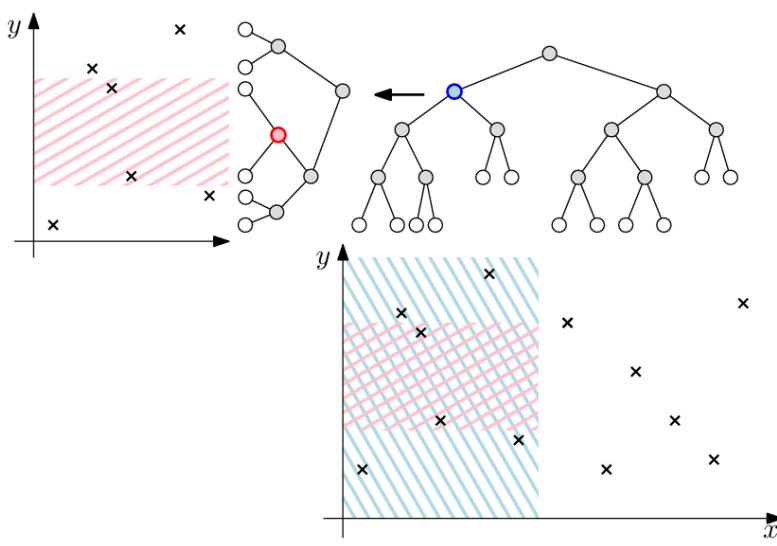
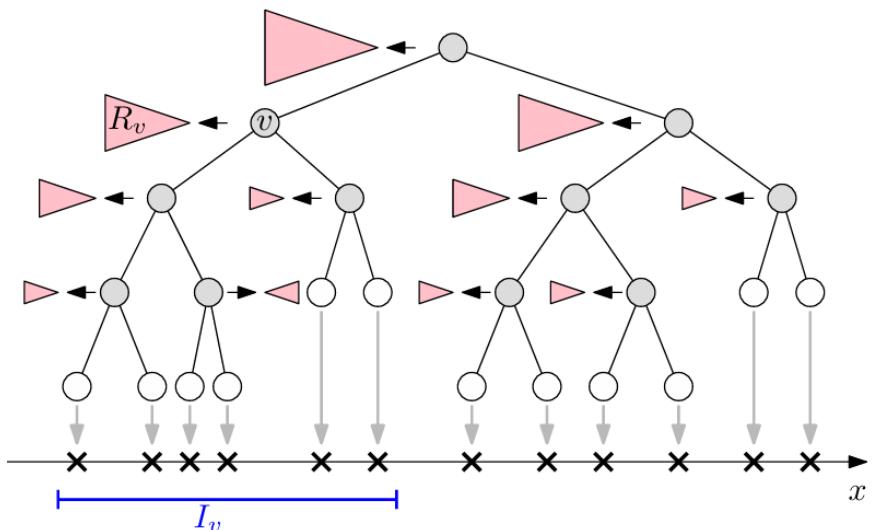
Complessità spaziale:  $O(n)$

### → Caso D=2

Costruire un range tree sull'insieme delle coordinate sull'asse x dei punti in S



Per ogni nodo  $v$  che rappresenta un intervallo  $I_v = [x_1, x_2]$ , costruire un range tree  $R_v$  sulle coordinate sull'asse delle y dei punti in S con coordinate sull'asse delle x in  $I_v$ .



## Costruzione Range Tree con D=2:

- ordinamento preliminare di S sulle x-coordinate
- dividere S in S1 e S2 di circa  $n/2$  elementi ciascuno
- ricorsivamente costruire T1 e T2 a partire da S1 e S2 rispettivamente
- La radice v di T ha T1 e T2 come suoi sottoalberi sinistro e destro
- Memorizzare, in v, un puntatore ad un nuovo 1D Range Tree su S
- Ritornare T

Tempo =  $O(n \log^2 n)$

Construction:

$S^y$  is the set  $S$  sorted on the  $y$ -coordinate

- Preliminarily sort  $S$  on the  $x$ -coordinate.
- Split  $S$  into  $S_1$  and  $S_2$  of  $\approx \frac{n}{2}$  elements each.
- Recursively build  $(T_1, S_1^y)$  and  $(T_2, S_2^y)$  from  $S_1$  and  $S_2$ , respectively.
- The root  $v$  of  $T$  has  $T_1$  and  $T_2$  as its left and right subtrees.
- Merge  $S_1^y$  and  $S_2^y$  into  $S^y$ .
- Store, in  $v$ , a pointer to a new 1D Range Tree on  $S^y$
- Return  $(T, S^y)$

**Time:**  $O(n \log n) + T(n)$ , where  $T(n) = 2 \cdot T(\frac{n}{2}) + O(n)$   
 $O(n \log n)$

To report the points  $p_1 = (x_1, y_1) \leq q \leq p_2 = (x_2, y_2)$ :

- Use  $T$  to find the  $h = O(\log n)$  subtrees  $R_1, \dots, R_h$  that store the points  $q = (x, y)$  with  $x_1 \leq x \leq x_2$ .
- For each tree  $R_j \in \{R_1, \dots, R_h\}$  representing the  $x$ -interval  $I_j$ :
  - Query  $R_j$  to report the number of/set of points  $q = (x, y)$  with  $x \in I_j$  and  $y_1 \leq y \leq y_2$ .

Time complexity:

$$O(\log n) \cdot O(\log n) + O(k) = O(\log^2 n + k)$$

Number of  $R_i$ s      Time to query  $R_i$       "size" of the output

Tempi e complessità dei Range Trees con  $D=2$ :

- Preprocessing time:  $O(n \log n)$
- Query time:  $O(\log^2 n + k)$
- Complessità spaziale:  $O(n \log n)$

#### → Caso $D > 2$

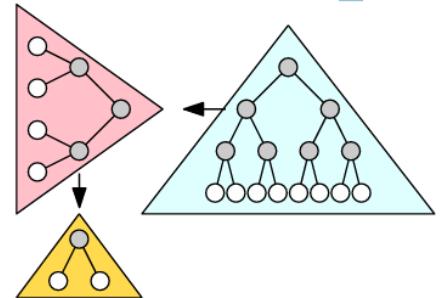
Per memorizzare i punti  $p = (x, y, z, w, \dots)$  in dimensioni  $D > 2$ :

Costruzione ricorsiva:

- Costruire un Range Tree  $T$  sulla prima coordinata  $x$  dei punti.
- Per ogni sottoalbero  $T_v$  di  $T$  associata con l'intervallo  $I_v = [x_1, x_2]$ :
  - Costruire un range tree  $R_v$  sulle ultime  $D-1$  coordinate  $(y, z, \dots)$  dell'insieme di punti  $p = (x, y, \dots)$  con  $x \in I_v$
  - Memorizzare, in  $v$ , un puntatore a  $R_v$

Tempo:  $O(n \log^{D-1} n)$

Spazio:  $O(n \log^{D-1} n)$



Query per dimensioni  $D > 2$ :

Let  $p_1 = (x_1, y_1, z_1, \dots), p_2 = (x_2, y_2, z_2, \dots)$ .

To report the points  $p_1 \leq q \leq p_2$ :

- Use  $T$  to find the  $h = O(\log n)$  subtrees  $R_1, \dots, R_h$  that store the points  $q = (x, y, z, \dots)$  with  $x_1 \leq x \leq x_2$ .
- For each tree  $R_j \in \{R_1, \dots, R_h\}$  representing the  $x$ -interval  $I_j$ :
  - Recursively query  $R_j$  to report the number/set of points  $q$  s.t.  $x \in I_j$  and  $(y_1, z_1, \dots) \leq q \leq (y_2, z_2, \dots)$ .

**Query time:**  $O(\log^D n + k)$ .

## Recap

$D$	Size	Preprocessing Time	Query Time	Notes
1	$O(n)$	$O(n \log n)$	$O(\log n + k)$	
2	$O(n \log n)$	$O(n \log n)$	$O(\log^2 n + k)$	
$> 2$	$O(n \log^{D-1} n)$	$O(n \log^{D-1} n)$	$O(\log^D n + k)$	

## FRACTIONAL CASCADING

### → Il problema

Input = k array ordinati  $A_1, \dots, A_k$  di  $n$  elementi ciascuno

Query: dato  $x$  riportare, per ogni  $i=1, \dots, k$ ,  $x$  se  $x \in A_i$  o il suo predecessore se  $x$  non appartiene ad  $A_i$ .

$A_1$	4   9   15   22   23   38   41   50   53   58
-------	---

$$x = 58$$

$A_2$	3   7   10   11   15   17   20   36   62   64
-------	---

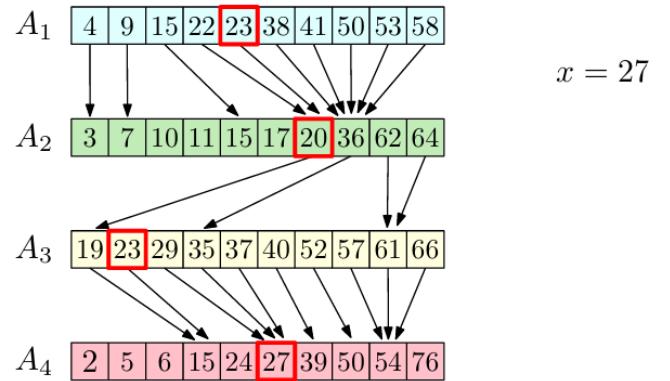
$A_3$	21   23   29   35   37   40   52   57   61   66
-------	---

$A_4$	2   5   6   15   24   27   39   50   54   76
-------	--

### → CROSS LINKING (prima idea)

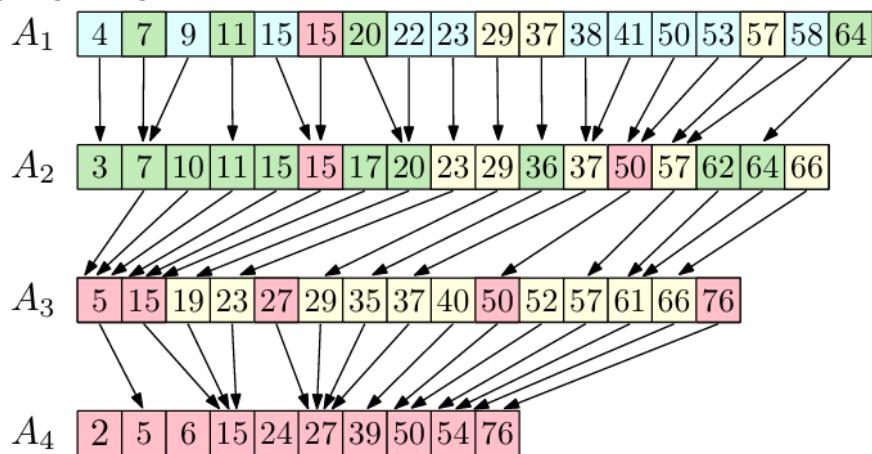
Mantenere i puntatori da  $A_i[j]$  al predecessore di  $A_i[j]$  in  $A_{i+1}$

Tempo caso peggiore:  $O(kn)$

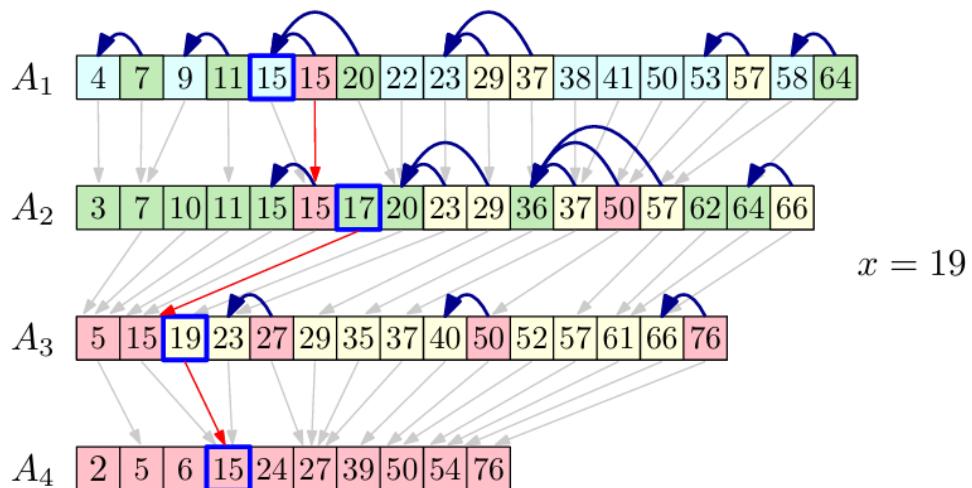


### → FRACTIONAL CASCADING (seconda idea)(tecnica)

Per  $i=k, k-1, \dots, 2$ : aggiungere ogni altro elemento di  $A_i$  in  $A_{i-1}$



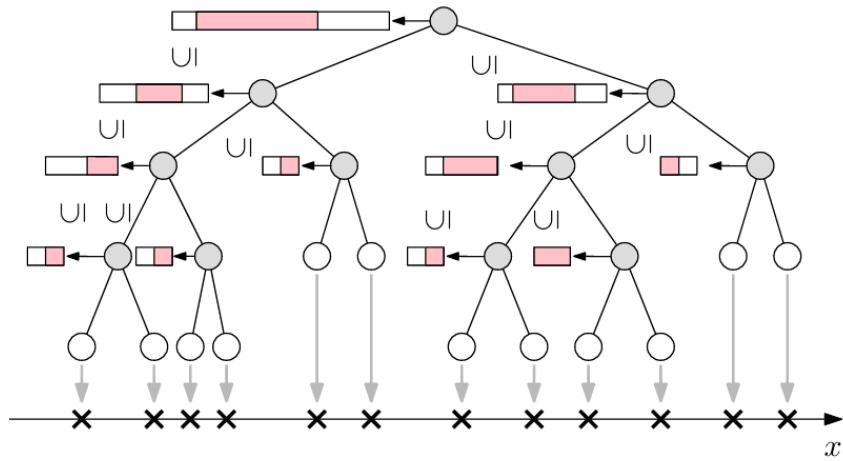
Mantenere i puntatori dai nuovi elementi aggiunti ad  $A_i$  ai loro predecessori tra gli elementi originali di  $A_i$



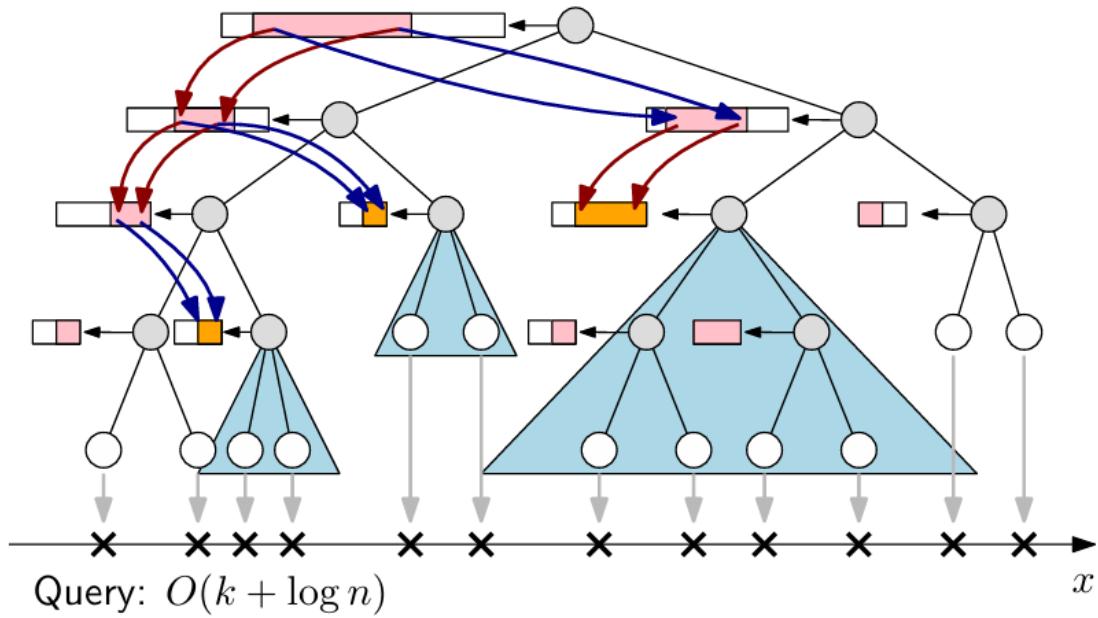
Size  $O(kn)$  Preprocessing  $O(kn)$  Query:  $O(k + \log n)$

## LAYERED RANGE TREES (PER D=2)

Riusa l'idea cross-linking dal fractional cascading



Per ogni elemento  $y$  nell'1D Range Tree di  $v$ , memorizzare un puntatore al predecessore di  $y$  nell'1D Range Tree del figlio sinistro/destro di  $v$ .



# Recap

$D$	Size	Preprocessing Time	Query Time	Notes
1	$O(n)$	$O(n \log n)$	$O(\log n + k)$	
2	$O(n \log n)$	$O(n \log n)$	$O(\log^2 n + k)$	
$> 2$	$O(n \log^{D-1} n)$	$O(n \log^{D-1} n)$	$O(\log^D n + k)$	
2	$O(n \log n)$	$O(n \log n)$	$O(\log n + k)$	with cross-linking
$> 2$	$O(n \log^{D-1} n)$	$O(n \log^{D-1} n)$	$O(\log^{D-1} n + k)$	with cross-linking

## 1. Problemi di String Matching

- **Definizione del problema:**

Data un’alfabeto  $\Sigma$ , un testo  $T \in \Sigma^*$  e un pattern  $P \in \Sigma^*$ , l’obiettivo è trovare una (o tutte le) occorrenza(i) di  $P$  in  $T$ .

*Esempio:* In “Bart played darts at the party” cercare “art” o, in un contesto biologico, individuare un pattern in una stringa formata dai simboli {A, C, G, T}.

- **Due modalità di query:**

- **One-shot:** sia il testo che il pattern fanno parte dell’input (si tratta di un classico problema di design di algoritmo).
- **Repeated:** il testo è statico e può essere preprocessato, mentre i pattern vengono forniti in tempo reale (il problema diventa di progettare una struttura dati per rispondere rapidamente alle query).

## 2. Tries (Alberi Prefisso)

- **Concetto Base:**

Un trie è una struttura dati che memorizza una collezione dinamica di  $k$  stringhe sull’alfabeto  $\Sigma$ , organizzata in modo da avere un percorso radice-foglia per ogni stringa (considerando anche un simbolo speciale di fine-stringa, ad esempio "\$").

- **Operazioni principali:**

- **Insert( $T$ ):** inserimento di una stringa  $T$ .
- **Delete( $T$ ):** cancellazione di una stringa  $T$ .
- **Find( $P$ ):** verifica se il pattern  $P$  è presente.
- **Contare/ritornare** le stringhe che iniziano con un prefisso  $P$ .
- **Predecessor( $T$ ):** dato  $T$ , trovare la stringa più grande (secondo l’ordine lessicografico) non minore di  $T$ .

- **Implementazioni e rappresentazioni:**

- **Array (dense):**  
Ogni nodo memorizza un array di dimensione  $|\Sigma|$  per puntare ai figli; spazio complessivo  $O(|\Sigma| \cdot n)$  e tempo di ricerca  $O(|P|)$ .
- **Array (sparse) o Alberi BST:**  
Utilizzando strutture più compatte che memorizzano solo i figli esistenti, lo spazio diventa  $O(n)$  e il tempo di accesso è  $O(|P| \log |\Sigma|)$ .
- **Weight-Balanced BST:**  
Qui si bilancia in base al “peso” (numero di foglie discendenti) permettendo query in tempo  $O(|P| + \log k)$ , risultando una soluzione ottimale nel caso statico.

- **Tecnica di Indirizzamento (Indirection):**

- Divide il trie in una parte “top” e diverse “bottom trees”, per ridurre ulteriormente i costi di query (ad esempio, eliminando il termine addizionale dipendente da  $|\Sigma|$ ).

- **Applicazioni dei Tries:**

- **Ordinamento di stringhe:** Inserendo le stringhe in un trie e visitandolo in ordine, si ottiene un ordinamento lessicografico.
- **Packet Routing:** Utilizzo dei tries per la ricerca del prefisso più lungo corrispondente a una destinazione, con applicazioni concrete in tabelle di routing.
- **Document Retrieval:** Pre-elaborazione di una collezione di documenti per rispondere rapidamente a query di ricerca, utilizzando simboli speciali per separare i documenti.

## 3. Compressed Tries (Radix Trees)

- **Concetto:**

- Si ottengono comprimendo (o “contrattando”) i cammini non branching: invece di avere molti nodi intermedi con un solo figlio, si memorizza un’unica etichetta di edge che rappresenta l’intera sequenza di caratteri.

- **Vantaggi:**

- Riduzione dell’occupazione di spazio e semplificazione della struttura, mantenendo invariate le operazioni di ricerca (usando il primo carattere dell’edge come chiave).

## 4. Suffix Trees

- **Definizione:**

- Il suffix tree di una stringa T è il trie compresso (o radix tree) di tutte le sue suffissi, generalmente terminata con un simbolo speciale "\$".
- *Esempio:* Per T = BANANAS\$, si costruisce un suffix tree in cui ogni foglia è etichettata con l'indice iniziale del suffisso corrispondente.

- **Operazioni:**

- **Ricerca di un pattern:** Si individua il nodo v corrispondente a P; tutti i suffissi che partono da v corrispondono alle occorrenze di P.
- **Lunghezza comune (LCP):** Utilizzando il Lowest Common Ancestor (LCA) tra due foglie, si può trovare rapidamente il prefisso comune più lungo tra due suffissi.
- **Longest Repeated Substring:** Si cerca il nodo più profondo (in termini di etichette sommate) che ha almeno due discendenti.
- **Ritrovamento di occorrenze aggiuntive:** Collegando le foglie con liste doppie, è possibile saltare rapidamente da un'occorrenza all'altra.

- **Complessità:**

- Costruzione in  $O(|T|)$  spazio (lineare rispetto alla lunghezza del testo).
- Query di ricerca in tempo  $O(|P| + \log |\Sigma| + \#matches)$  (se si memorizza il numero di foglie in ogni sottoalbero).

---

## 5. Suffix Arrays

- **Relazione con i Suffix Trees:**

- Un suffix array è una versione "ordinata" dei suffissi di T, memorizzando gli indici di inizio in ordine lessicografico.

- **Costruzione:**

- Si ordinano tutti i suffissi (con i loro indici iniziali) della stringa T = BANANAS\$, ottenendo una struttura lineare che, abbinata ad altri strumenti (come il LCP array), permette di risolvere problemi di pattern matching.

- **Utilizzo:**

- Possono essere impiegati per risolvere il problema dello string matching in modo comparabile ai suffix trees, spesso con costi spaziali inferiori.

---

## Applicazioni Comuni

- **String Sorting:**

- Ordinare una collezione di stringhe inserendole in un trie e visitandolo in ordine in tempo  $O(n + k \log |\Sigma|)$ .

- **Packet Routing:**

- Utilizzare i tries per effettuare il longest prefix matching, determinando rapidamente quale regola si applica a un indirizzo di destinazione.

- **Document Retrieval:**

- Preprocessare una collezione di documenti concatenati (utilizzando simboli speciali per separarli) per rispondere alle query di ricerca, individuando in tempo efficiente tutti i documenti che contengono un determinato pattern.

- **Problematiche di LCP e Longest Repeated Substring:**

- I suffix trees permettono di rispondere a query di longest common prefix (tramite LCA) e di individuare la sottostringa ripetuta più lunga.

---

## Considerazioni Finali

La lezione fornisce una visione completa su:

- Come strutturare e ottimizzare le operazioni fondamentali sullo string matching.
- Le differenze tra diverse implementazioni dei tries (dense, sparse, weight-balanced) e le relative trade-off in termini di spazio e tempo.
- La potenza dei suffix trees e suffix arrays nel risolvere problemi complessi di pattern matching e analisi delle stringhe.
- Applicazioni pratiche in ambiti come il routing e il retrieval dei documenti, che illustrano l'importanza di una buona progettazione delle strutture dati.

Questo riassunto evidenzia gli aspetti teorici (complessità, rappresentazione e operazioni) e pratici (applicazioni in sorting, routing e document retrieval) fondamentali per comprendere e utilizzare le strutture dati avanzate per il string matching.

## Representing Tries: Recap

	Space	Query Time
Array (dense)	$O( \Sigma  \cdot n)$	$O( P )$
Array (sparse) / BST	$O(n)$	$O( P  \log  \Sigma )$
Weight-balanced BST	$O(n)$	$O( P  + \log k)$
Indirection	$O(n)$	$O( P  + \log  \Sigma )$

Can be made dynamic with a time complexity of  $O(|T| + \log |\Sigma|)$  per insertion/deletion of  $T$