# Gli alberi di copertura

## Algoritmi 2 – Lezione 9

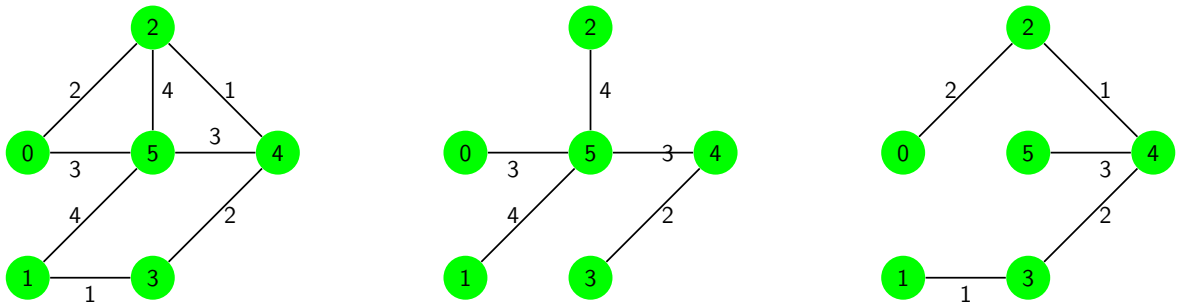
A. Monti
Sapienza Università di Roma

Corso: Algoritmi 2

# Indice

# Introduzione: Il Problema del MST

Dato un grafo connesso e pesato l'obiettivo è trovare un sottoinsieme degli archi del grafo che connetta tutti i nodi con il costo totale minimo, ovvero la somma minima dei pesi degli archi. Questo sottoinsieme di archi forma un **albero di copertura**. Nota che, poiché si tratta di un albero, il sottoinsieme non contiene cicli.




In figura a sinistra abbiamo un grafo connesso e pesato $G$. Al centro, un albero di copertura di $G$ non minimo (costo totale 16). A destra, il minimo albero di copertura di $G$ (costo totale 9).

# Introduzione all'algoritmo di Kruskal

- Il problema di trovare un albero di copertura a costo minimo è centrale in informatica, con numerose applicazioni, ad esempio nella progettazione di reti e nelle telecomunicazioni.
- Per risolvere questo problema sono stati sviluppati diversi algoritmi efficienti.
- Tra questi, l'algoritmo di Kruskal è una soluzione basata su un approccio **greedy**.
- La sua strategia consiste nel selezionare, a ogni passo, l'arco con il peso minimo che non forma cicli.

# Passaggi dell'algoritmo di Kruskal

1. Si ordinano tutti gli archi del grafo in ordine non decrescente rispetto al peso.
2. Si considerano gli archi uno alla volta, seguendo l'ordine ottenuto.
3. Per ciascun arco $(u, v)$, si verifica se la sua aggiunta al sottoinsieme corrente di archi genera un ciclo.
4. Se non si forma alcun ciclo, l'arco viene aggiunto al Minimo Albero di Copertura; altrimenti, viene scartato.
5. Il processo continua fino a ottenere un albero con $n - 1$ archi, ovvero un albero di copertura.

# Dimostrazione di correttezza (1/2)

L'algoritmo di Kruskal trova sempre un Minimo Albero di Copertura (MST).
La dimostrazione si basa su un ragionamento per assurdo.

**Ipotesi per assurdo**: Supponiamo che l'algoritmo di Kruskal **non** produca un MST.
Sia $T$ l'albero generato da Kruskal e $T^*$ un MST diverso da $T$, scelto tra quelli che differiscono da $T$ nel minor numero di archi.

**La contraddizione**:
Consideriamo il primo arco $e$ che **non** appartiene a $T^*$ scelto da Kruskal .

- Se aggiungiamo $e$ a $T^*$, si forma un ciclo $C$.
- In questo ciclo esiste almeno un arco $e' \in T^*$ che non è in $T$.
- Poiché Kruskal ha scelto $e$ prima di $e'$, segue che: $peso(e) \leq peso(e')$.

# Dimostrazione di correttezza (2/2)

Costruiamo un nuovo albero $T'$ rimuovendo $e'$ da $T^*$ e aggiungendo $e$:

$$costo(T') = costo(T^*) - peso(e') + peso(e)$$

Dato che $peso(e) \leq peso(e')$, si ha:

$$costo(T') \leq costo(T^*)$$

Poiché $T^*$ è un MST, deve valere $costo(T') = costo(T^*)$, quindi anche $T'$ è un MST.

Tuttavia, $T'$ ha un maggior numero di archi in comune con $T$ rispetto a $T^*$, poiché $e \in T'$ e $e' \notin T'$. Questo contraddice la scelta di $T^*$ come MST più simile a $T$.

**Conclusione**: L'ipotesi iniziale è falsa. Pertanto, l'algoritmo di Kruskal genera sempre un Minimo Albero di Copertura.

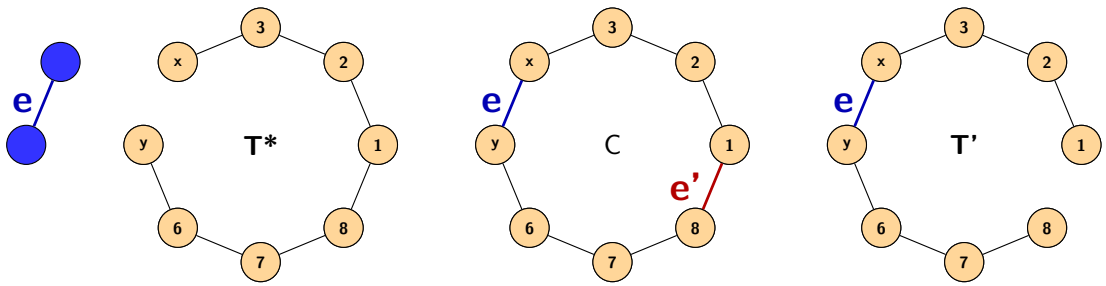



Figura: Passaggi chiave nella dimostrazione di Kruskal.

# La struttura dati Union-Find

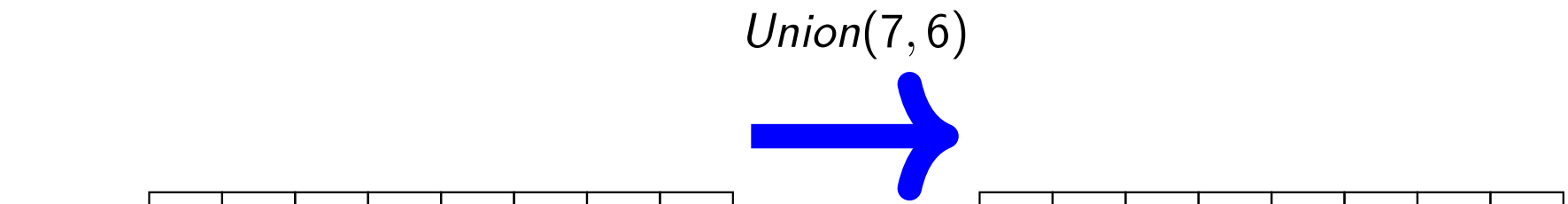
- Una delle principali sfide nell'implementare Kruskal è verificare rapidamente se un arco crea un ciclo.
- La **struttura dati Union-Find**, o Insiemi Disgiunti, è perfetta per questo scopo.
- Mantiene una partizione degli elementi in insiemi disgiunti.
- Offre due operazioni principali in tempo quasi costante:
  - **find(i)**: Restituisce l'identificatore del gruppo a cui appartiene l'elemento 'i'.
  - **union(i, j)**: Unisce i due gruppi a cui appartengono gli elementi $i$ e $j$.
- Un ciclo si verifica quando si tenta di unire due nodi che sono già nello stesso insieme.

# Union-Find: array diretto

Questa è l'implementazione più semplice da concepire, ma anche la meno efficiente. Un array $C$ memorizza direttamente il rappresentante di ogni elemento.

- *Crea*$(n)$: Inizializza un array $C$ di dimensione $n$, dove $C[i] = i$ per ogni $i$. Ogni elemento è il rappresentante di se stesso.
- *Find*$(u, C)$: Restituisce $C[u]$ che è il rappresentante dell'insieme contenente $u$.
- *Union*$(rap\_u, rap\_v, C)$: Unisce gli insiemi scorrendo l'intero array $C$. L'operazione trova tutti gli elementi che appartengono all'insieme $rap\_v$ e li riassegna all'insieme $rap\_u$.

Di seguito la modifica dell'array $C$ a seguito della fusione degli insiemi 7 e 6:



# Codice Python: array diretto

```
def Crea(n):
    C = [ i for i in range(n)]
    return C


def Find(u, C):
    rap_u = C[u]
    # Restituisco il rappresentante di u
    return rap_u


def Union (rap_u, rap_v, C):
        for i in range(len(C)):
            if C[i] == rap_v:
                C[i] = rap_u
```

**Prestazioni:**

- *Crea*: $\Theta(n)$
- *Find*: $O(1)$
- *Union*: $\Theta(n)$

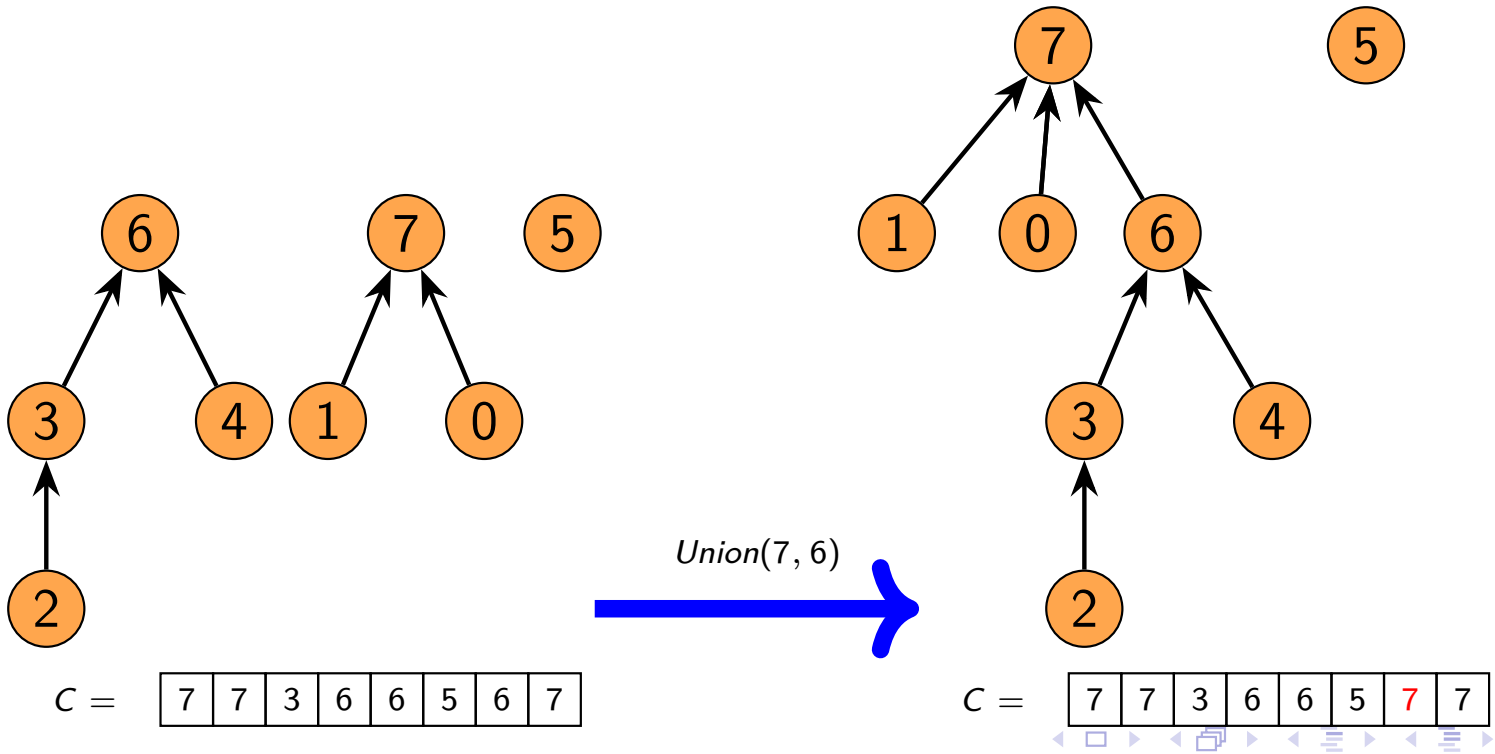
Come notato, l'elevata complessità di *Union* rende questa implementazione inefficiente. In un algoritmo come Kruskal, il costo totale di $n$ unioni sarebbe $O(n^2)$, rendendola impraticabile per grafi di grandi dimensioni.

# Union-Find: vettore dei padri

Questa implementazione rappresenta gli insiemi come una foresta di alberi. L'array $C$ memorizza il "genitore" di ogni elemento, e la radice di ogni albero è il rappresentante del suo insieme.

- *Crea*($n$): Inizializza un array $C$ di dimensione $n$ con $C[i] = i$.
- *Find*($u$, $C$): Risale la catena dei padri a partire da $u$ fino a raggiungere la radice (l'elemento che punta a se stesso), restituendo il rappresentante di $u$
- *Union*($rap\_u$, $rap\_v$, $C$): Unisce gli insiemi a cui appartengono $u$ e $v$ rendendo la radice di uno (ad esempio $rap\_v$) figlia della radice dell'altro ( $rap\_u$).

Di seguito la modifica dell'array $C$ a seguito della fusione degli insiemi 7 e 6:



# Codice Python: vettore dei padri

```
def Crea(n):
    C = [ i for i in range(n)]
    return C

def Find(u, C):
    rap_u = u
    while rap_u != C[rap_u]:
       # Risalgo nell'albero
        rap_u = C[rap_u]
    # Restituisco il rappresentante di $u$
    return rap_u

def Union (rap_u, rap_v, C):
    # Il rappresentante dell'insieme contenente v diventa rap_u
    C[rap_v] = rap_u
```

**Prestazioni:**

- *Crea*: $\Theta(n)$
- *Find*: $O(n)$ nel caso peggiore, poiché gli alberi possono diventare sbilanciati.
- *Union*: $O(1)$

Per essere utilizzata in Kruskal, l'operazione completa di unione richiede prima due chiamate a *Find* per ottenere i rappresentanti dei nodi, portando la complessità totale dell'operazione a $O(n)$.

# Union-Find: unione per rango

Per l'operazione *Find*, l'efficienza è massimizzata evitando di produrre alberi sbilanciati. Questo obiettivo si ottiene applicando all'operazione di *Union* **l'ottimizzazione dell'unione per rango o dimensione** che consiste nel collegare la radice dell'albero più piccolo a quella dell'albero più grande allo scopo di mantenere gli alberi poco profondi.

Per implementare questa operazione ogni nodo $u$ del vettore $C$ è rappresentato da una coppia $[u, size]$ dove *size* (nel caso delle radici) è il numero di nodi nell'insieme.

# Codice Python: unione per rango

```
def Crea(n):
    C = [ [i, 1]  for i in range(n)]
    return C

def Find(u, C):
    while u != C[u][0]:
        u = C[u][0]
    return u

def Union (rap_u, rap_v, C):
    if rap_u != rap_v:
        # l'albero più piccolo diventa figlio del più grande
        piccolo, grande  = rap_u, rap_v
        if C[rap_u][1] > C[rap_v][1]:
            piccolo, grande = grande, piccolo
        C[piccolo][0] = C[grande][0]
        C[grande][1] += C[piccolo][1]
```

**Prestazioni:**

- *Crea*: $\Theta(n)$
- *Find*: $O(\log n)$
- *Union*: $O(1)$

Per essere utilizzata in Kruskal, l'operazione completa di unione richiede prima due chiamate a *Find* per ottenere i rappresentanti dei nodi, portando la complessità totale dell'operazione a $O(\log n)$.

# Correttezza del bilanciamento per rango

**Teorema:** In una struttura Union-Find con **fusione per rango**, ogni albero di altezza $h$ contiene **almeno** $2^h$ nodi.

**Dimostrazione (per induzione su $h$):**

- **Caso base:** Se $h = 0$, l'albero è composto da un solo nodo (nessun figlio). Contiene quindi $1 = 2^0$ nodo. Proprietà verificata.
- **Passo induttivo:** Supponiamo vera la proprietà per tutte le altezze $\leq h - 1$.

  Consideriamo un albero di altezza $h$. Tale albero può formarsi solo unendo due alberi di altezza $h-1$ (fusione di insiemi con stesso rango).
  - Per ipotesi induttiva, ciascuno dei due sottoalberi contiene almeno $2^{h-1}$ nodi.
  - Dopo la fusione, il nuovo albero contiene almeno:

$$2^{h-1} + 2^{h-1} = 2^h \quad \text{nodi.}$$

**Conclusione:** ogni albero di altezza $h$ contiene almeno $2^h$ nodi. Ne segue che: $n \geq 2^h \quad \Rightarrow \quad h \leq \log_2 n$

# Implementazione dell'Algoritmo di Kruskal

```
def kruskal(G):
    '''
    Restituisce l'albero\foresta di copertura di costo minimo di G
    '''
    n = len(G)
    # Crea la lista degli archi di G
    #  u < v evita di aggiungere lo stesso arco due volte.
    E=[ (c, u, v) for u in range(n) for v, c in G[u] if u < v]
    # Ordina la lista degli archi per peso crescente
    E.sort()
    # Inizializza l'albero di copertura minimo
    T = [ [ ] for _ in range(n) ]
    C = Crea(n)
    for c,u,v in E:
        # Trova i rappresentanti dei nodi u e v
        rap_u = Find(u, C)
        rap_v = Find(v, C)
        if rap_u != rap_v:
            # I due nodi non sono nella stessa componente
            # e l'arco (u,v)  viene aggiunto al MST
            T[u].append(v)
            T[v].append(u)
            # Le due componenti vengono fuse
            Union(rap_u,rap_v, C)
    return T
```

# Complessità dell'algoritmo di Kruskal

- **Creazione e Ordinamento degli Archi** : Questa è la prima fase dell'algoritmo, dove vengono preparati tutti gli archi del grafo. L'operazione di estrazione degli archi ha costo $O(m)$, quella di ordinamento ha costo $O(m \log m) = O(m \log n)$

- **Costruzione dell'MST con Union-Find**: In questa fase, l'algoritmo esamina gli archi ordinati e decide quali includere nell'MST.
  - **Inizializzazione:** L'inizializzazione della struttura dati Union-Find con *Crea*$(n)$ ha una complessità di $O(n)$.
  - **Ciclo principale:** Il ciclo *for* itera $m$ volte, una per ogni arco. All'interno del ciclo, vengono eseguite le operazioni *Find* e *Union*.
    - Le due chiamate a Find costano $O(\log n)$ ciascuna, per un totale di $O(\log n)$ per iterazione.
    - La chiamata a *Union* costa $O(1)$ per iterazione.
    - Quindi, il costo totale del ciclo per tutti gli $m$ archi è $O(m \log n)$.

Combinando le complessità di tutte le fasi, otteniamo: $O(n + m \log n)$

# Confronto tra le implementazioni Union-Find

| **Versione** | **Find** | **Union** | **Complessità totale** |
|---|---|---|---|
| Array diretto | $O(1)$ | $\Theta(n)$ | $\Theta(n^2)$ |
| Vettore dei padri | $O(n)$ | $O(1)$ | $O(n^2)$ |
| Unione per rango | $O(\log n)$ | $O(1)$ | $O(m \log n)$ |

# ESERCIZI

**Suggerimento:** Prima di cercare le soluzioni in rete, prova a ragionarci da solo: gli esercizi che seguono sono pensati per aiutarti a mettere alla prova la tua comprensione, la tua capacità di sviluppare una prova di correttezza e costruire un algoritmo corretto.
Solo dopo averci riflettuto, confronta la tua soluzione con altre possibili versioni.

# Esercizi

1. Sia $G$ un grafo connesso e pesato con pesi positivi. Sia $T$ un suo minimo albero di copertura e $T_s$ il suo albero dei cammini minimi a partire da un suo nodo $s$. Provare o confutare che la somma dei pesi degli archi di $T$ è uguale alla somma dei pesi degli archi di $T_s$.
2. Sia $G$ un grafo connesso e pesato. Sia $T$ un albero di copertura di costo minimo di $G$, e sia $T_s$ l'albero dei cammini minimi da un nodo $s$ verso tutti gli altri nodi di $G$. Dimostrare o fornire un controesempio che il costo di $T$ è al più il costo di $T_s$.
3. Sia $G$ un grafo connesso e pesato dove i pesi sono tutti diversi tra loro. Sia $T$ un minimo albero di copertura di $G$, sia $s$ un nodo e $T_s$ l'albero dei cammini minimi da $s$ verso tutti gli altri nodi di $G$. Dimostrare oppure fornire un controesempio che $T_s$ e $T$ condividono almeno un arco.
4. Sia $G$ un grafo connesso e pesato, e $G'$ il grafo che si ottiene da $G$ moltiplicando il peso di ciascun arco per una fissata costante $c > 0$.
   1. Sia $T$ un albero di copertura di costo minimo per $G$. Dimostra o confuta che $T$ sia un albero di copertura di costo minimo anche per il grafo G
   2. Sia $T_s$ l'albero dei cammini costruito a partire dal nodo $s$ per $G$. Dimostra o confuta che $T_s$ sia un albero dei cammini minimi anche per il grafo $G'$

# Esercizi

4 Immagina di dover progettare un sistema idrico per $n$ case identificate con gli interi gli interi $1, 2, \ldots n$. L'obiettivo è fornire acqua a ogni casa minimizzando il costo totale.
Hai a disposizione due opzioni per ogni casa:

- **Costruire un pozzo:** La costruzione di un pozzo nella casa $i$ ha un costo pari a $P[i]$.
- **Collegarla a un pozzo esistente:** La costruzione di una tubatura tra la casa $i$ e la casa $j$ ha un costo pari a $C[i, j]$. L'acqua può raggiungere una casa attraverso un pozzo costruito al suo interno, oppure attraverso una rete di tubature che la colleghi a un pozzo.

Il tuo compito è sviluppare un algoritmo che, dati il vettore dei costi dei pozzi $P$ e la matrice $n \times n$ dei costi delle tubature $C$, determini:

- In quali case costruire i pozzi.
- Quali tubature costruire per collegare tutte le case.

L'algoritmo deve risolvere il problema in tempo $O(n^2)$ e deve garantire che il costo totale sia minimo.

**Suggerimento:** Rappresentare il problema tramite un opportuno grafo pesato con $n + 1$ nodi.