Progettazione di Algoritmi **Anno Accademico 2021–2022**

Note per la Lezione 28

Ugo Vaccaro

In questa lezione riguardiamo l'algoritmo di Algoritmo di Kruskal per MST e ne forniamo una implementazione efficiente. Ricordiamo che l'algoritmo di Kruskal, informalmente, procede nel seguente modo:

> Aggiungi a T uno ad uno gli archi del grafo, in ordine di costo crescente, saltando gli archi che creano cicli con gli archi già aggiunti.

Per un'implementazione efficiente dell'algoritmo di Kruskal abbiamo quindi bisogno di un sistema efficiente per stabilire se l'aggiunta di un arco (u, v) all'albero T crea un ciclo o meno. Sia $C(u)$ la componente connessa del grafo (V, T) cui u appartiene ($C(u)$ =insieme dei nodi raggiungibili da u) e sia $C(v)$ la componente connessa cui v appartiene. L'aggiunta di (u, v) creerà un ciclo se e solo se u e v appartengono alla stessa componente connessa, ovvero se e solo se $C(u) = C(v)$, non creerà un ciclo se $C(u)$ e $C(v)$ sono diverse (ovvero sono disgiunte). Nell'ipotesi che (u, v) non crei un ciclo, possiamo aggiungere l'arco (u, v) a T, ed in tal caso le componenti connesse $C(u)$ e $C(v)$ si fondono in un'unica componente connessa.

Abbiamo quindi bisogno di un metodo efficiente per fare le seguenti cose:

- Dati due vertici u e v, stabilire se essi appartengono o meno alla stessa componente connessa
- Gestire la natura dinamica di tali componenti connesse (le componenti si possono creare a causa di un'inserzione di un'arco, cresscono per l'aggiunta di nodi, si possono fondere in componenti più grandi, etc.)

Studiamo quindi, in via preliminare, la questione di progettare Strutture dati per Insiemi Disgiunti, che ci sarÃ utile per risolvere i problemi di cui sopra.

Problema: Gestire una collezione dinamica $\mathcal{S} = \{S_1, \ldots, S_r\}$ di insiemi a due a due disgiunti, dove:

• Ogni insieme $S_i \subseteq \{x_1, x_2, \ldots, \}$

• Ogni insieme S_i nella collezione $\mathcal{S}$ ha un *rappresentante*, denotato con $rap[S_i]$, che in generale é un elemento esso stesso in S_i, arbitrario ma fissato.

Nell'applicazione che a noi poi interessa, gli S_i sono le componenti connesse che via via l'algoritmo di Kruskal crea e $\{x_1, x_2, \ldots, \}$ è l'insieme dei vertici del grafo. Le operazioni che intendiamo supportare sono:

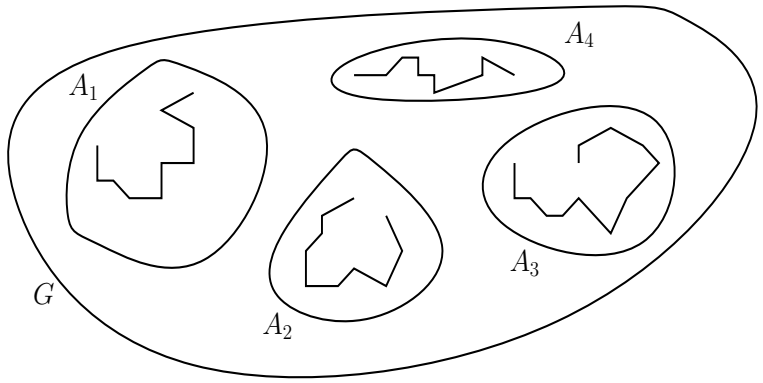
MAKE-SET(x): aggiunge alla collezione $\mathcal{S}$ un nuovo insieme $S = \{x\}$ il cui unico elemento é x. Vale che $rap[S] = x$. Poiché gli insiemi in $\mathcal{S}$ devono essere disgiunti, x non deve appartenere a nessun altro insieme della collezione.

UNION(x, y): sostituisce gli insiemi S_x ed S_y di $\mathcal{S}$, che contengono gli elementi x e y, con un nuovo insieme $S = S_x \cup S_y$, ed assegna un qualche valore a $rap[S]$.

FIND-SET(x): ritorna il rappresentante $rap[S_x]$, dove S_x é quell'unico membro della famiglia $\mathcal{S}$ che contiene l'elemento x.

Vediamo una prima applicazione di tale struttura dati al seguente problema.

Dato un grafo $G = (V, E)$, una *componente connessa* di G é un sottoinsieme di vertici $A \subseteq V$ composto di vertici *tutti* connessi tra di loro, con la proprietá addizionale che $\forall\ v \in V - A, \forall\ u \in A$, vale che u e v non sono raggiungibili. Il problema consiste nell'identificare all'interno del grafo G le sue componenti connesse.



Abbiamo giá visto che un tale problema si puó risolvere mediante BFS o DFS. Vediamo un modo ulteriore per risolverlo.

Risoluzione del problema via strutture dati per insiemi disgiunti.

```
COMPONENTI-CONNESSE(G)
1.   FOR ogni vertice v ∈ V
2.       MAKE-SET(v)
3.   FOR ogni arco (u, v) ∈ E
4.       IF FIND-SET(u) ≠FIND-SET(v)
5.         THEN UNION(u, v)
```

L'algoritmo mantiene in ogni istante della sua esecuzione una collezione di insiemi disgiunti, ciascun insieme é composto da vertici tutti connessi tra di loro (ovvero, é una componente connessa). Inizialmente, gli insiemi sono composti ciascuno da un singolo vertice (istruzioni 1. e 2.). Nelle linee 3. e 4., per ogni arco (u, v) si controlla se i suoi due estremi si trovano in componenti connesse diverse (e quindi l'arco stesso le unisce), oppure se si trovano nella stessa componente connessa. Se vale il primo caso, nella istruzione 5. si fondono le due componenti connesse in un'unica componente connessa.

Quindi, ritornando al nostro problema originale:

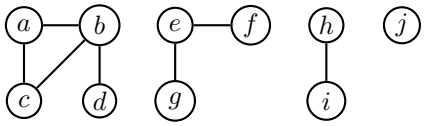
Problema: dato un grafo G e due vertici u e w in esso, sono i due vertici connessi in G?

Per risolvere il problema, basta controllare se i due vertici si trovano in una stessa componente connessa di G

```
STESSA-COMPONENTE(u, v)
1.   IF FIND-SET(u) = FIND-SET(v)
2.     THEN RETURN True
3.     ELSE RETURN False
```

Le complessitá di COMPONENTI-CONNESSE(G) e di STESSA-COMPONENTE(u, v) dipendono criticamente dalle complessitá delle operazioni MAKE-SET(x) UNION(x, y), e FIND-SET(x) su insiemi disgiunti.

Esempio:



Arco processato	Collezione di insiemi disgiunti									
insiemi iniziali	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b, d)	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e, g)	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e, g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a, c)	$\{a, c\}$	$\{b, d\}$			$\{e, g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(h, i)	$\{a, c\}$	$\{b, d\}$			$\{e, g\}$	$\{f\}$		$\{h, i\}$		$\{j\}$
(a, b)	$\{a, b, c, d\}$				$\{e, g\}$	$\{f\}$		$\{h, i\}$		$\{j\}$
(e, f)	$\{a, b, c, d\}$				$\{e, f, g\}$			$\{h, i\}$		$\{j\}$
(b, c)	$\{a, b, c, d\}$				$\{e, f, g\}$			$\{h, i\}$		$\{j\}$

Le componenti connesse correttamente individuate sono $\{a, b, c, d\}$, $\{e, f, g\}$, $\{h, i\}$, $\{j\}$.

Ritorniamo al problema di trovare una struttura dati efficiente che ci permetta di implementare in maniera efficiente le operazioni MAKE-SET(x), UNION(x, y) e FIND-SET(x).

Prima (semplice) rappresentazione: liste linkate.

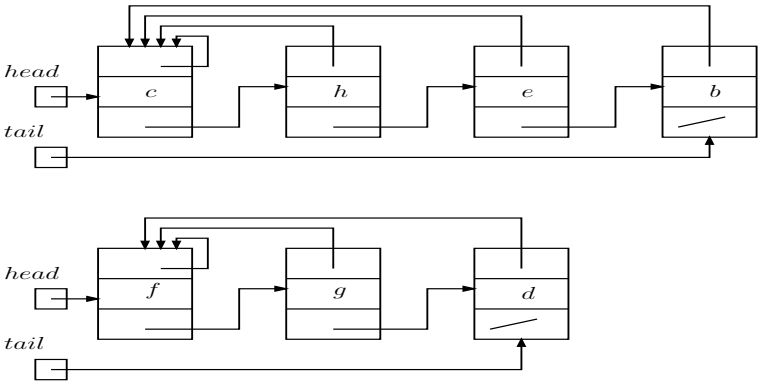
• Ogni insieme S_i della collezione $\mathcal{S}$ é rappresentato mediante una lista linkata;

• L'elemento nella testa della lista é il *rappresentante* dell'insieme corrispondente;

• Ciascun oggetto nella lista linkata contiene un elemento dell'insieme, un puntatore all'oggetto contenente il successivo elemento dell'insieme, ed un puntatore al rappresentante dell'insieme;

• Ciascuna lista mantiene un puntatore *head* alla testa della lista (rappresentante dell'insieme) ed un puntatore *tail* all'ultimo elemento della lista.

Esempio: $S_1 = \{b, c, e, h\}$, $S_2 = \{d, f, g\}$



Complessitá di MAKE-SET(x) UNION(x, y), e FIND-SET(x)

Rappresentando ciascun insieme S nella collezione $\mathcal{S}$ mediante una lista linkata, avremo:

MAKE-SET(x)
`crea una lista con l'unico elemento` x
tail `e` *head* `puntano entrambi ad` x

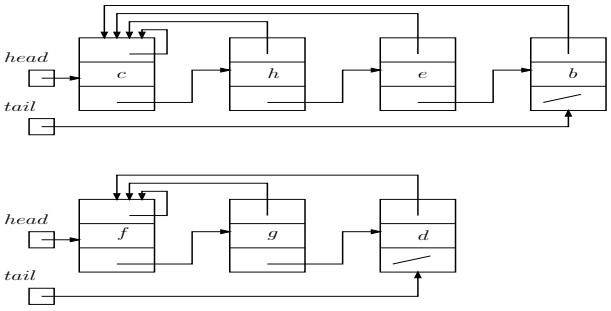
Complessitá: $\Theta(1)$

FIND-SET(x)
`RETURN`(puntatore di x al rappresentante)

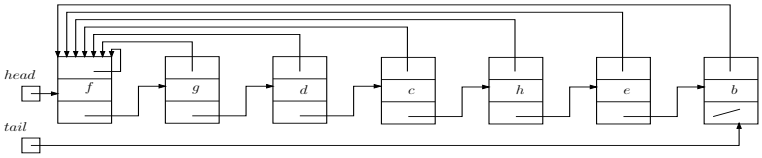
Complessitá: $\Theta(1)$

UNION(x, y)
`a) appendi la lista contenente` x `alla fine della lista contenente` y
`b) aggiorna` *head*`,` *tail* `ed i puntatori degli elementi della nuova lista a` *head*

Vediamo un esempio, Siano dati gli insiemi $S_1 = \{b, c, e, h\}$, $S_2 = \{d, f, g\}$, rappresentati dalle due liste seguenti:



Supponiamo ora di eseguire UNION(e, f). Avremo la seguente lista come risultato:



Procediamo ora con l'analisi dell'operazione UNION(x, y), avendo deciso di rappresentare gli insiemi della nostra collezione mediante liste a puntatori. La parte dispendiosa di UNION(x, y) é la `b)`. In essa occorre aggiornare tutti i puntatori alla testa della lista degli elementi presenti nella lista di x. Il nuovo valore di tali puntatori deve essere pari alla **nuova** testa della lista, che é quella della lista contenente y. Questa operazione richiede tempo proporzionale alla lunghezza della lista di x.

Immaginiamo ora una situazione particolarmente *sfavorevole*: abbiamo inizialmente n oggetti $x_1, x_2, \ldots, x_n$, su cui eseguiamo in sequenza MAKE-SET(x_1), ..., MAKE-SET(x_n). Il costo di ciascun MAKE-SET(x_i) é $\Theta(1)$, per un totale di $\Theta(n)$. Supponiamo ora di eseguire in sequenza le operazioni UNION(x_1, x_2), UNION(x_2, x_3), UNION(x_3, x_4), ..., UNION(x_{n-1}, x_n). Quanto costa effettuare tutte queste operazioni?

Operazione	Numero di oggetti aggiornati
MAKE-SET(x_1)	1
MAKE-SET(x_2)	1
MAKE-SET(x_3)	1
$\vdots$	$\vdots$
MAKE-SET(x_n)	1
UNION(x_1, x_2)	1
UNION(x_2, x_3)	2
UNION(x_3, x_4)	3
$\vdots$	$\vdots$
UNION(x_{n-1}, x_n)	$n-1$

In totale, nelle $2n$ operazioni vengono aggiornati $\sum_{i=1}^{n-1} i = \Theta(n^2)$ puntatori. É come dire che ogni operazione costa, "in media", $\Theta(n)$. Come rimediare al problema?

Nel caso peggiore, la implementazione della operazione di UNION che abbiamo appena visto richiede tempo $O(n)$ in quanto puó accadere di appendere liste lunghe a liste corte, e conseguentemente occorrerá aggiornare il puntatore al rappresentante per ciascun elemento della lista lunga.

Supponiamo ora di mantenere per ciascuna lista anche un contatore che valuta gli elementi presenti nella lista, e di appendere sempre la lista *piú corta alla lista piú lunga*, in modo da minimizzare il numero di puntatori alla

testa della lista da aggiornare. Chiamiamo questo metodo **euristica per unione pesata**.

Analisi dell' euristica per unione pesata:

Una sequenza qualsiasi di m operazioni di MAKE-SET, UNION, e FIND-SET, n delle quali sono MAKE-SET, prende tempo $O(m + n \log n)$

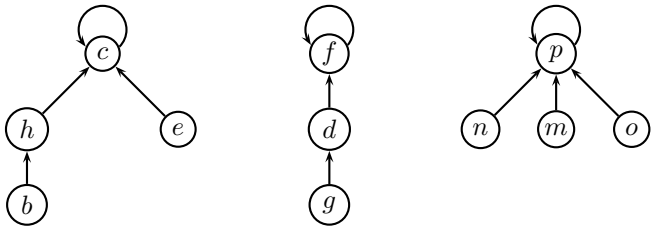
Mostreremo il risultato stimando il numero di aggiornamenti di puntatori che vengono effettuati durante le m operazioni. Innanzitutto calcoliamo, per ogni oggetto x, una limitazione superiore al numero di volte che il puntatore di x alla testa della lista viene aggiornato. Ricordiamo che se tale puntatore di x viene aggiornato, ció vuol dire che in una qualche operazione di UNION, l'elemento x si trovava nella lista piú corta. Pertanto, la prima volta che il puntatore di x alla testa della lista viene aggiornato durante una qualche operazione di UNION, ció implica che la lista risultante (contenente x) deve avere almeno 2 elementi. La seconda volta che il puntatore di x alla testa della lista viene aggiornato durante una qualche operazione di UNION comporta che la lista risultante deve contenere almeno 4 elementi. Continuando nel procedimento, possiamo dedurre che dopo che il puntatore di x alla testa della lista é stato aggiornato per la k-esima volta da una qualche operazione di UNION, allora la lista risultante deve contenere almeno 2^k elementi. Poiché una lista non puó contenere piú di tutti gli elementi, vale che $2^k \leq n$, ovvero $k \leq \lceil \log n \rceil$. Pertanto, $\forall\, x$, il puntatore di x alla testa della lista durante *tutte* le operazioni di UNION viene aggiornato al piú $\log n$ volte. Occorre anche contare gli aggiornamenti di puntatori *head* e *tail*, e gli aggiornamenti del contatore della lunghezza della lista. Questi prendono tempo $O(1)$ per ogni operazione di UNION, per un totale di $O(\log n)$ operazione per elemento. Essendo gli elementi in numero di n, otteniamo che la esecuzione di tutte le operazioni di UNION prende al piú tempo $O(n \log n)$. Infine, ricordiamo che ogni operazione di MAKE-SET e FIND-SET prende tempo $O(1)$, ve ne sono $O(m)$ di esse, per un tempo totale per eseguire l'intera sequenza di operazioni pari a $O(m) + O(n \log n) = O(m + n \log n)$.

É possibile ottenere una implementazione piú veloce delle operazioni su insiemi disgiunti usando una diversa rappresentazione, in cui:

- Ogni insieme $S \in \mathcal{S}$ é rappresentato da un albero;
- Ogni elemento di S é un nodo dell'albero, con un unico puntatore al suo padre;
- La radice dell'albero rappresentante l'insieme S contiene il rappresentante dell'insieme S.

Esempio di rappresentazione di insiemi mediante alberi

$\mathcal{S} = \{S_1, S_2, S_3\}$, $S_1 = \{b, c, e, h\}$, $S_2 = \{d, f, g\}$, $S_3 = \{n, m, o, p\}$



Implementazione di MAKE-SET(x) UNION(x, y), e FIND-SET(x)

Nella rappresentazione ad alberi:

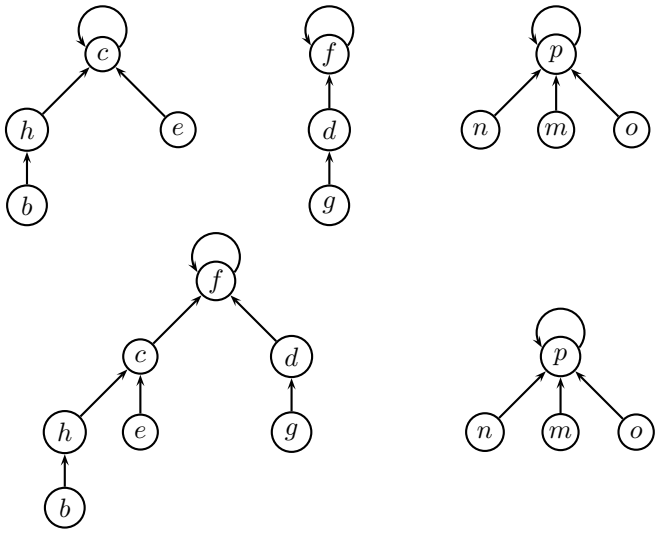
MAKE-SET(x)
`crea una albero con il solo nodo` x

FIND-SET(x)
segui i puntatori al padre da x fino alla radice dell'albero, restituisci il puntatore alla radice dell'albero

UNION(x, y)
esegui FIND-SET(x) e FIND-SET(y)
poni il puntatore al padre del nodo restituito da FIND-SET(x) uguale al nodo restituito da FIND-SET(y)

(in altri termini, in UNION(x, y) rendiamo il nodo radice dell'albero contenente l'elemento x figlio della radice dell'albero contenente l'elemento y)

Esempio: $S_1 = \{b, c, e, h\}$, $S_2 = \{d, f, g\}$, $S_3 = \{n, m, o, p\}$, e UNION(e, g)



Osservazioni:

La rappresentazione di insiemi disgiunti mediante alberi non produce, da sola, algoritmi per le operazioni di UNION, FIND, e MAKE-SET con complessitá inferiore a quelle che abbiamo precedentemente ottenuto mediante la rappresentazione di insiemi mediante liste, ed usando l'euristica per unione pesata.

Accoppiando peró la rappresentazione ad alberi con due nuove euristiche, si ottengono i migliori algoritmi noti per le tre operazioni di UNION, FIND, e MAKE-SET.

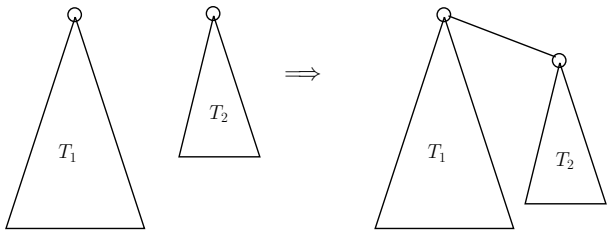
Le euristiche che analizzeremo sono **l'euristica dell' unione per rango**, e **l'euristica della compressione dei cammini**.

Prima euristica: unione per rango.

Nella euristica dell'unione per rango, procediamo in maniera simile all'euristica dell'unione pesata usata nella rappresentazione di insiemi disgiunti mediante liste.

Piú precisamente, manteniamo un valore (chiamato **rango**) associato ad ogni nodo dell' albero nella struttura. Tale valore corrisponde ad una limitazione superiore all'altezza del nodo nell'albero.

Nella euristica dell'unione per rango, ogni qualvolta si dovrá effettuare una operazione di UNION, la radice dell'albero con il *rango minore* punterá alla radice con il *rango maggiore*.



Seconda euristica: compressione dei cammini.

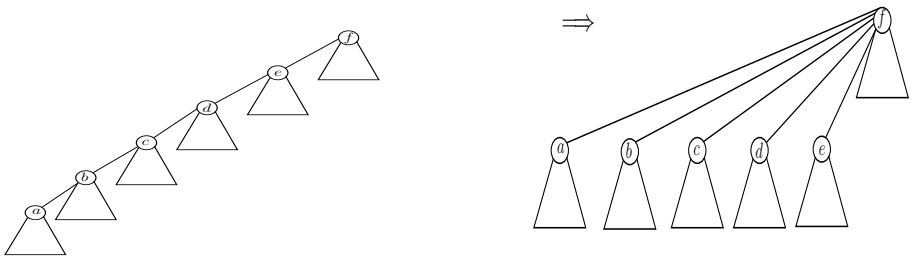
Ricordiamo la implementazione della operazione di FIND-SET:

FIND-SET(x)
`segui i puntatori al padre da` x `fino alla radice dell'albero, restituisci il puntatore alla radice dell'albero`

Siano $x_1 = x, x_2, \ldots, x_k = r$ i nodi dell'albero sul percorso da x fino alla radice r dell'albero, che vengono visitati durante l'esecuzione di FIND-SET(x).

Nella euristica della compressione dei cammini, *tutti* i nodi $x_1 = x, \ldots, x_{k-1}$ avranno il loro puntatore al padre assegnato ad r (ovvero, tutti i nodi $x_1 = x, \ldots, x_{k-1}$ diventeranno figli della radice)

Esempio di compressione di cammini durante una operazione di FIND-SET(a)



Ulteriori dettagli

• per ciascun nodo x, manteniamo un valore intero $rango[x]$, che sará sempre $\geq$ del numero di archi sul piú lungo percorso da x ad un discendente foglia y di x (altezza di x).

• Quando un insieme di cardinalitá 1 é inizialmente creato con una qualche operazione di MAKE-SET(x), il valore $rango[x]$ viene posto a 0.

• Ciascuna operazione di FIND-SET non cambierá i valori $rango[x]$, per ogni nodo x.

• Quando eseguiremo una operazione di UNION su due alberi, confronteremo i ranghi delle radici dei due alberi, e renderemo la radice con rango maggiore padre della radice con rango minore. Se le radici hanno uguale rango, sceglieremo in maniera arbitraria chi rendere padre.

Pseudocodice:

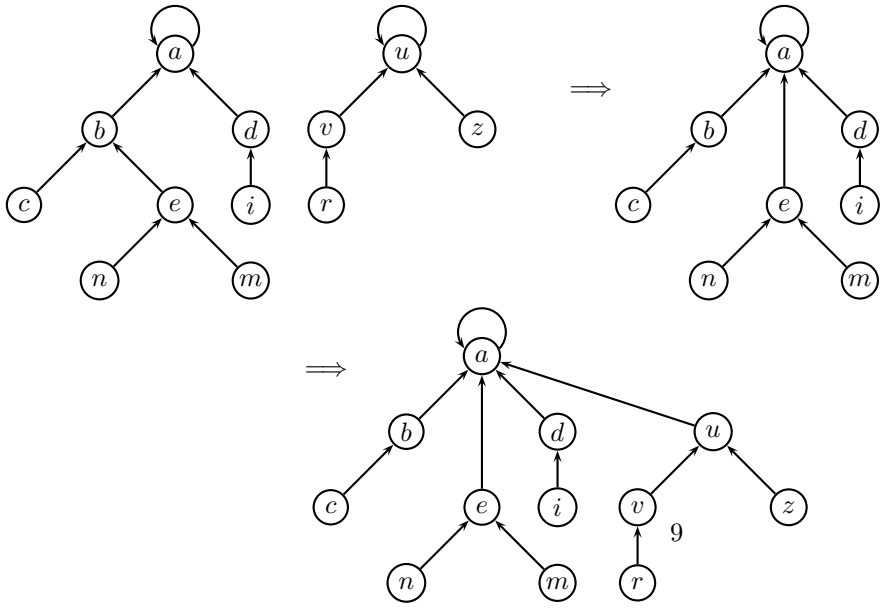
```
MAKE-SET(x)
Parent[x] ← x
rango[x] ← 0
```

```
FIND-SET(x)
IFx ≠ Parent[x]
THEN Parent[x] ←FIND-SET(Parent[x])
RETURNParent[x]
```

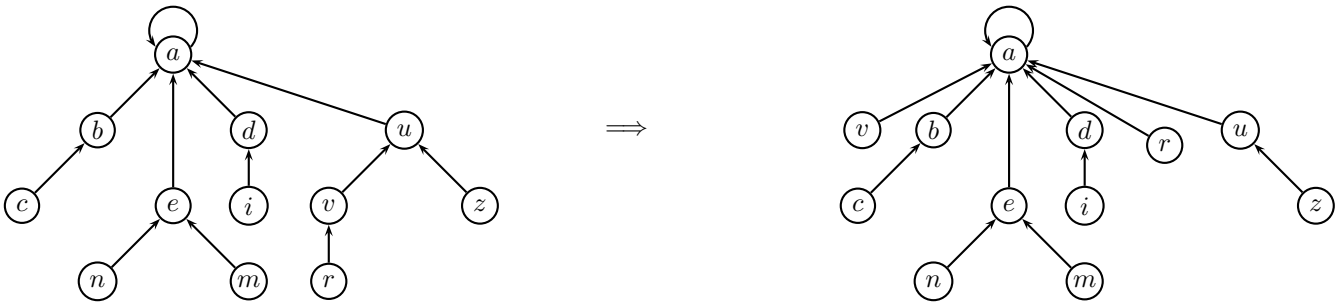
```
UNION(x, y)
LINK(Find-Set(x), Find-Set(y))
```

```
LINK(x, y)
IF rango[x] > rango[y]
THEN p[y] ← x    ELSE p[x] ← y
IF rango[x] = rango[y]
THEN rango[y] ← rango[y] + 1
```

Esempio: UNION(e, v). Ricordiamo che nell'eseguire la UNION(e, v) eseguiamo in via preliminare le operazioni FIND(e) e FIND(v), che *giÃ da sole possono modificare gli alberi*, come accade nell'esempio di sotto:



Esempio: FIND-SET(r)



Complessitá:

Usando la rappresentazione di insiemi mediante alberi, ed usando entrambe le euristiche di unione per rango e compressione di cammini, una sequenza qualsiasi di m operazioni di MAKE-SET, UNION, e FIND-SET, n delle quali sono MAKE-SET, prende tempo $O(m\alpha(n))$

La funzione $\alpha(n)$ é una funzione che cresce molto lentamente, ad esempio

$$\alpha(n) = \begin{cases} 0 & \text{per } 0 \leq n \leq 2 \\ 1 & \text{per } n = 3 \\ 2 & \text{per } 4 \leq n \leq 7 \\ 3 & \text{per } 8 \leq n \leq 2047 \\ 4 & \text{per } 2048 \leq n \leq 16^{512} \approx 10^{80} \end{cases}$$

Ritorniamo a Kruskal, che dice:

Aggiungi a T uno ad uno gli archi del grafo, in ordine di costo crescente, saltando gli archi che creano cicli con gli archi già aggiunti.

```
MST-KRUSKAL(G = (V, E), c)
1.  T ← ∅
2.  For ogni vertice v ∈ V
3.     Make-Set(v)
4.  ordina gli archi di E per costo c non decrescente
5.  For ogni arco (u, v) ∈ E, in ordine di peso non decrescente,
6.     If Find-Set(u) ≠Find-Set(v)
7.       Then T ← T ∪ (u, v)
8.         Union(u, v)
9.  return T
```

Analisi della complessità:

Il **For** delle linee 2.-3. richiede tempo $O(|V|$. La linea 4. richiede tempo $O(|E| \log |E|) = O(|E| \log |V|)$. All'interno del **For** della linea 5. effettueremo $2|E|$ chiamate a `Find-Set`, ed al più $|V|$ chiamate a `Union`. Dal risultato precedente, la parte di algoritmo da 5. a 9. richiede tempo $O(|E|\alpha(|V|)$, che sommata a $O(|E| \log |E|)$ dà un gran totale di $O(|E| \log |V|)$.