

Corso di laurea in Informatica

Progettazione d'algoritmi

Struttura dati per insiemi disgiunti

Angelo Monti



SAPIENZA
UNIVERSITÀ DI ROMA

- Union-Find, noto anche come Disjoint Set Union (DSU), è una struttura dati per gestire insiemi disgiunti.
 - È utilizzato per operazioni di unione e ricerca efficienti su insiemi disgiunti.
-
- Le tre operazioni fondamentali di Union-Find sono:
 1. **Crea(S)**: restituisce una struttura dati Union-Find sull'insieme S di elementi dove ciascun elemento è in un insieme separato.
 3. **Find(x,C)**: Restituisce il nome dell'insieme della struttura dati C a cui appartiene l'elemento x.
 5. **Union(A, B, C)**: modifica la struttura dati C fondendo la componente A con la componente B e restituisce il nome della nuova componente.

Una gestione efficiente di insiemi disgiunti è utile in diversi contesti tra cui l'evoluzione di un grafo nel tempo attraverso l'aggiunta di archi. In questo caso gli insiemi disgiunti rappresentano le componenti connesse del grafo.

L'operazione ***find(u)*** consente di determinare a quale componente connessa appartiene il nodo u . Questa operazione può essere sfruttata per determinare se due nodi u e v appartengono alla stessa componente, semplicemente controllando se $find(u) == find(v)$.

Se viene aggiunto l'arco (u,v) al grafo, si verifica innanzitutto se u e v sono nella stessa componente connessa. Se $A=find(u)$ e $B=find(v)$ risultano distinti, allora l'operazione ***Union(A, B)*** può essere usata per unire le due componenti.

Discutiamo brevemente su cosa intendiamo con nome dell'insieme (ad esempio quello ritornato dalla funzione *find* su un elemento x). C'è un'ampia flessibilità nella scelta, come risulta dalle varie implementazioni la cosa importante è che $find(u) = find(v)$ se e solo se u e v sono nello stesso insieme. La scelta fatta nelle implementazioni che seguono qui è quella di scegliere come nome dell'insieme quello di un particolare elemento dell'insieme stesso.

Come primo approccio possiamo pensare di **assegnare all'insieme il nome dell'elemento massimo in esso contenuto.**

Probabilmente il modo più semplice di implementare questa struttura dati per n elementi è di mantenere il vettore C delle n componenti.

All'inizio ogni elemento è in un insieme distinto vale a dire $C[i] = i$. Quando la componente i viene fusa con la componente j , se $i > j$ allora tutte le occorrenze di j nel vettore C verranno sostituite da i (se $i < j$ accadrà il contrario).

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

```
def Find(u,C):  
    return C[u]
```

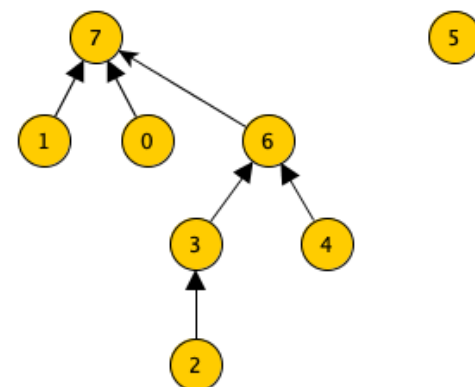
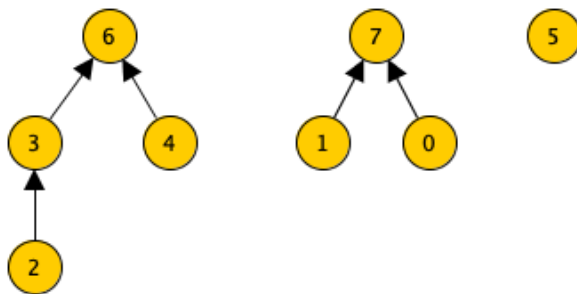
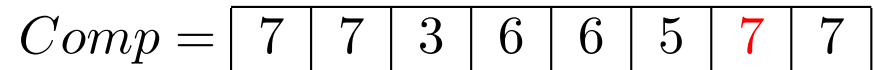
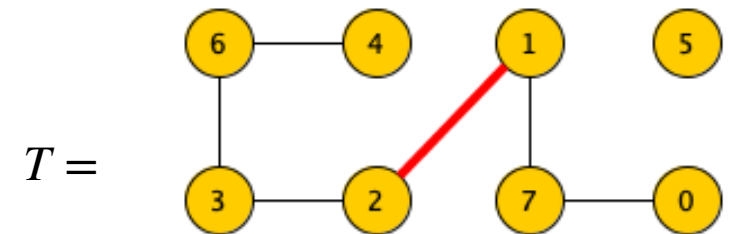
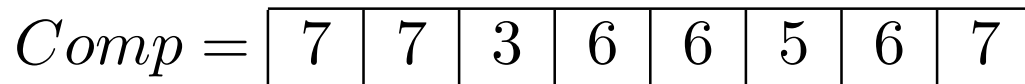
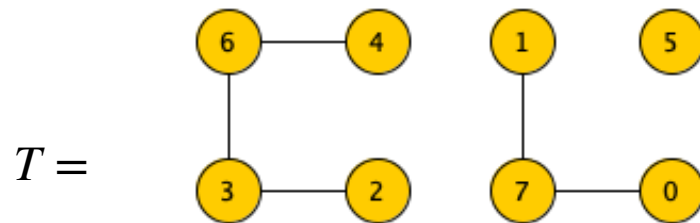
```
def Union (a,b,C):  
    if a > b:  
        for i in range(len(C)):  
            if C[i]==b: C[i]=a  
    else:  
        for i in range(len(C)):  
            if C[i]==a: C[i]=b
```

- Crea() costo $\Theta(n)$
- Find() costo $\Theta(1)$
- Union() costo $\Theta(n)$

Nella implementazione appena vista la *FIND* costa $O(1)$ e la *UNION* $\Theta(n)$.
Meglio bilanciare i costi: rendere meno costosa la *UNION* anche a costo di pagare qualcosa in più per la *FIND*.

IDEA: Uso il vettore dei padri:

- **FIND:** quando voglio sapere in che componente si trova un nodo devo semplicemente risalire alla sua radice. $O(n)$
- **UNION:** quando fondo due componenti una diventa figlia dell'altra. $O(1)$



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

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

```
def Union (a,b,C):|
    if a > b:
        C[b]=a
    else:
        C[a]=b
```

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

Non voglio che l'operazione FIND abbia un costo elevato quindi è importante che i cammini per raggiungere le radici del vettore dei padri non diventino troppo lunghi.

Per evitare questo problema, è preferibile mantenere gli alberi bilanciati.

IDEA: quando eseguo la Union per fondere due componenti scelgo sempre come nuova radice la componente che contiene il maggior numero di elementi.

L'intuizione è che in questo modo per almeno la metà dei nodi presenti nelle due componenti coinvolte nella fusione la lunghezza del cammino non aumenta.

IDEA: quando eseguo la *Union* per fondere due componenti scelgo sempre come nuova radice l'insieme che contiene il maggior numero di elementi.

Fondendo le componenti con quest'accorgimento garantiamo la seguente proprietà:

Se un insieme ha altezza h allora l'insieme contiene almeno 2^h elementi.

Dalla proprietà deduciamo che l'altezza delle componenti non potrà mai superare $\log_2 n$ (perché in caso contrario avrei nella componente più di n nodi il che è assurdo).

Implementando quindi la *Union* con l'accorgimento appena descritto la *find* richiederà tempo $O(\log n)$ (anziché $O(n)$).

In questa implementazione della Union-Find devo fare in modo che ai nodi radice sia associato anche il numero di elementi che la componente contiene.

Ogni elemento è caratterizzato da una coppia (x, numero) dove x è il nome dell'elemento e **numero** è il numero di nodi nell'albero radicato in x

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

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

```
def Union (a, b, C):  
    tota, totb = C[a][1], C[b][1]  
    if tota >= totb:  
        C[a]=(a, tota + totb)  
        C[b]=(a, totb)  
    else:  
        C[a]=(b, tota)  
        C[b]=(b, tota + totb)
```

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

PROPRIETA': Se una componente ha altezza h allora la componente contiene almeno 2^h nodi.

PROVA: Assumiamo per assurdo durante una delle fusioni si sia formata una nuova componente di altezza h che non rispetta la proprietà. Considera la prima volta che ciò accade e siano ca e cb le componenti che si fondono. Possono essere accadute due cose:

- *ca e cb erano componenti con la stessa altezza* allora avevano entrambe altezza $h - 1$ ed ognuna aveva almeno 2^{h-1} elementi (perché nelle fusioni precedenti la proprietà era sempre stata verificata). Quindi il numero totale di elementi della nuova componente è $2^{h-1} + 2^{h-1} = 2^h$ e la proprietà è verificata.
- *ca e cb avevano diverse altezze* allora l'altezza dopo la fusione è quella della componente di altezza maggiore che doveva essere già di altezza h e conteneva da sola già 2^h elementi.

