Union-Find

Moreno Marzolla
Dip. di Scienze dell'Informazione
Università di Bologna

marzolla@cs.unibo.it
http://www.moreno.marzolla.name/

Struttura dati per insiemi disgiunti

- Motivazioni
  - In alcune applicazioni siamo interessati a gestire insiemi disgiunti di oggetti
- Operazioni fondamentali:
  - Creare un insieme a partire da un singolo elemento
  - Unire due insiemi
  - Identificare l'insieme a cui appartiene un elemento
- Struttura dati
  - Una collezione $S = \{ S_1, S_2, ..., S_k \}$ di insiemi dinamici disgiunti
  - Gli insiemi contengono complessivamente $n \geq k$ elementi
  - Ogni insieme è identificato da un rappresentante univoco

Scelta del rappresentante

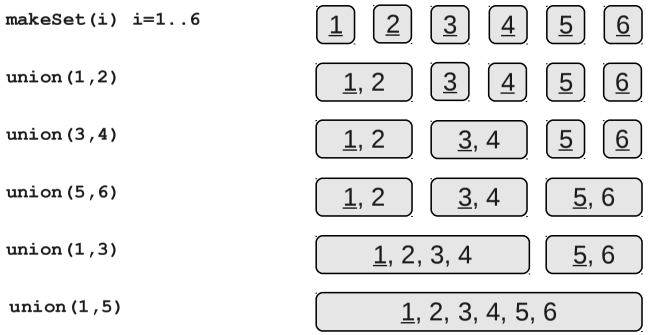
- Il rappresentante di S_i può essere un qualsiasi membro dell'insieme S_i
  - Operazioni di ricerca del rappresentante su uno stesso insieme devono restituire sempre lo stesso oggetto
  - Solo in caso di unione con altro insieme il rappresentante può cambiare

Operazioni su strutture Union-Find

- `makeSet(elem x)`
  - Crea un insieme il cui unico elemento (e rappresentante) è *x*
  - *x* non deve appartenere ad un altro insieme esistente
- `find(elem x)→name`
  - Restituisce il rappresentante dell'unico insieme contenente *x*
- `union(name x, name y)`
  - Unisce i due insiemi rappresentati da *x* e da *y*
  - Assumiamo che il nome del nuovo insieme sia *x* (questa ipotesi non è strettamente necessaria)
  - I vecchi insiemi devono essere distrutti

Esempio

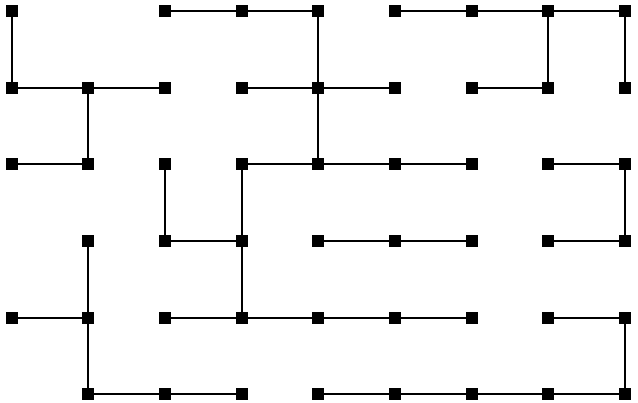
(i valori sottolineati indicano il rappresentante)



Esempio di utilizzo

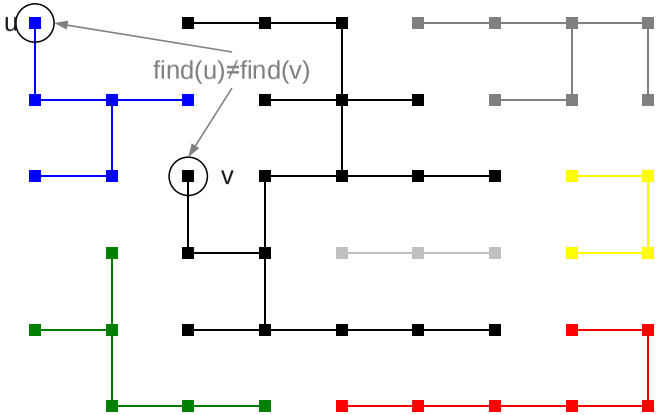
- Determinare le componenti connesse di un grafo non orientato dinamico, soggetto ad inserimenti di archi
- Algoritmo
  - Si inizia creando un insieme per ogni nodo del grafo
  - Per ogni arco *(u,v)* esegui `union(find(u),find(v))`
  - Alla fine, avremo l'insieme delle componenti connesse
- Complessità
  - Costo: `|V| makeSet + |E| union + 2|E| find`
  - |V| = numero di nodi del grafo; |E| = numero di archi
  - Questo algoritmo è interessante per la capacità di gestire grafi dinamici (in cui gli archi vengono aggiunti)

Esempio



Esempio

(i colori indicano le componenti connesse)

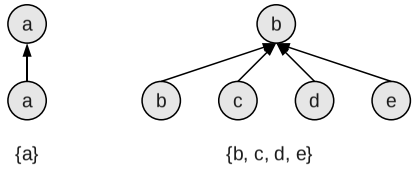


Implementazione di Union-Find

- Algoritmi elementari:
  - Algoritmo QuickFind: alberi di altezza uno
    - `makeSet()`,`find()`: O(1); `union()`: O(n)
  - Algoritmo QuickUnion: alberi generali
    - `makeSet()`,`union()`: O(1); `find()`: O(n)
- Algoritmi basati su euristiche di bilanciamento
  - QuickFind—Euristica sul peso
  - QuickUnion—Euristica sul rango
- Algoritmi basati su euristiche di compressione
  - QuickUnion: rango + compressione

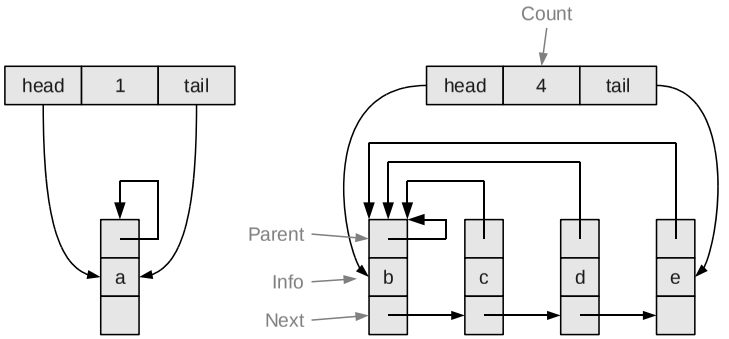
QuickFind

- Ogni insieme viene rappresentato con un albero di altezza uno
  - Le foglie dell'albero contengono gli elementi dell'insieme
  - Il rappresentante è la radice

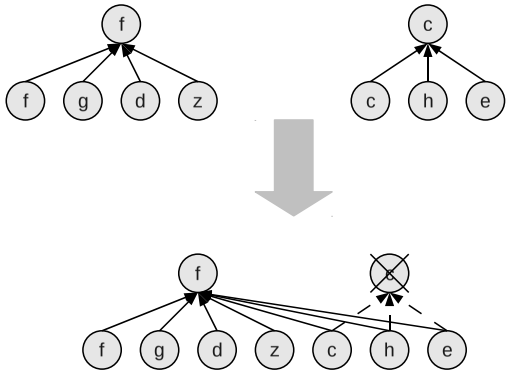


Nota implementativa

- È possibile rappresentare gli insiemi disgiunti tramite liste



QuickFind: Esempio—`union(find(g), find(h))`

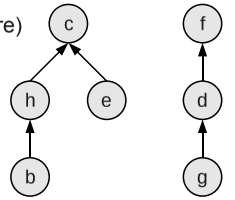


QuickFind

- Le operazioni `makeSet()` e `find()` richiedono tempo O(1):
  - `makeSet(x)`: crea un albero in cui l'unica foglia è *x* e il rappresentante di *x* è *x* stesso; costo O(1)
  - `find(x)`: restituisce il puntatore al padre di *x*; costo O(1)
- L'operazione `union(A,B)` richiede più tempo:
  - Tutte le foglie dell'albero B vengono spostate nell'albero A
  - Costo nel caso pessimo O(n), essendo n il numero complessivo di elementi in entrambi gli insiemi disgiunti
    - Infatti nel caso peggiore B ha n-1 elementi

QuickUnion

- Implementazione basata su foresta
  - Si rappresenta ogni insieme tramite un albero radicato generico
  - Ogni nodo dell'albero contiene
    - l'oggetto
    - un puntatore al padre (la radice non ha padre)
  - Il rappresentante è la radice



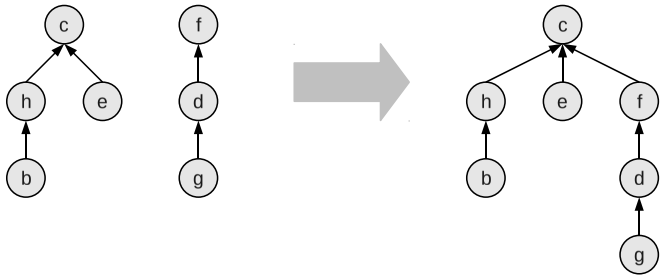
{c, e, h, b}

{f, d, g}

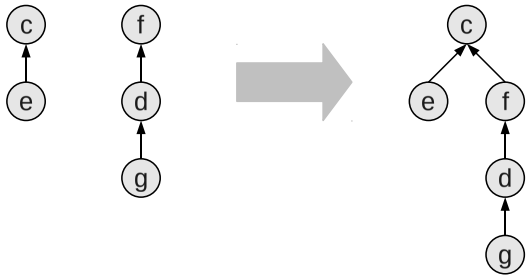
QuickUnion

- `makeSet(x)`
  - Crea un albero con un unico nodo *x*
  - Costo: O(1) nel caso pessimo
- `find(x)`
  - Risale la lista dei padri di *x* fino a trovare la radice e restituisce la radice come oggetto rappresentante
  - Costo: O(n) nel caso pessimo
- `union(A,B)`
  - Appende l'albero B ad A, rendendo la radice di B figlia della radice di A
  - Costo: O(1) nel caso pessimo

QuickUnion: Esempio–`union(c, f)`



QuickUnion: Esempio–`union(c, f)`



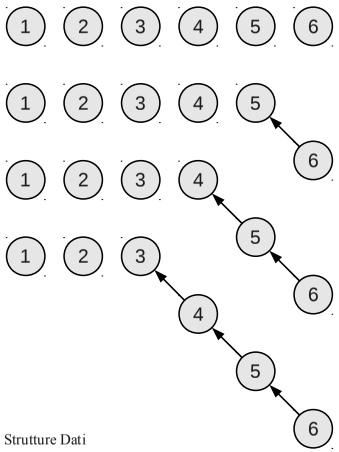
Caso pessimo per find()

`makeSet(i) i=1..6`

`union(5,6)`

`union(4,5)`

`union(3,4) ...`



Riepilogo

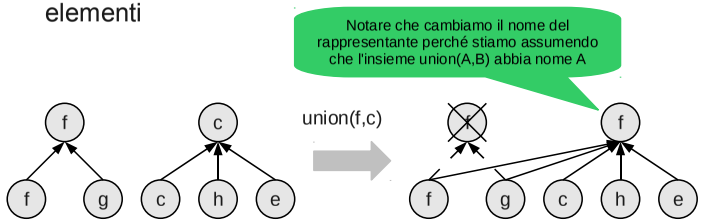
	QuickFind	QuickUnion
makeSet	O(1)	O(1)
union	O(n)	O(1)
find	O(1)	O(n)

Considerazioni

- Quando usare....
  - QuickFind?
    - Quando le `union()` sono rare e le `find()` frequenti
  - QuickUnion?
    - Quando le `find()` sono rare e le `union()` frequenti
- È importante sapere che esistono tecniche euristiche che permettono di migliorare questi risultati

QuickFind: Euristica sul peso

- Una strategia per diminuire il costo dell'operazione `union()` in QuickFind consiste nel:
  - Memorizzare nella radice il numero di elementi dell'insieme; la dimensione può essere mantenuta in tempo O(1)
  - Appendere l'insieme con meno elementi a quello con più elementi



Osservazioni / 1

- Ogni volta che una foglia di un albero QuickFind bilanciato acquista un nuovo padre, fa parte di un insieme che ha *almeno il doppio di elementi* di quello cui apparteneva
- Dimostrazione
  - union(A,B) con size(A) ≥ size(B)
    - Le foglie di B cambiano padre
    - size(A)+size(B) ≥ size(B)+size(B) = 2 size(B)
  - union(A,B) con size(A) ≤ size(B)
    - Le foglie di A cambiano padre
    - size(A)+size(B) ≥ size(A)+size(A) = 2 size(A)

Osservazioni / 2

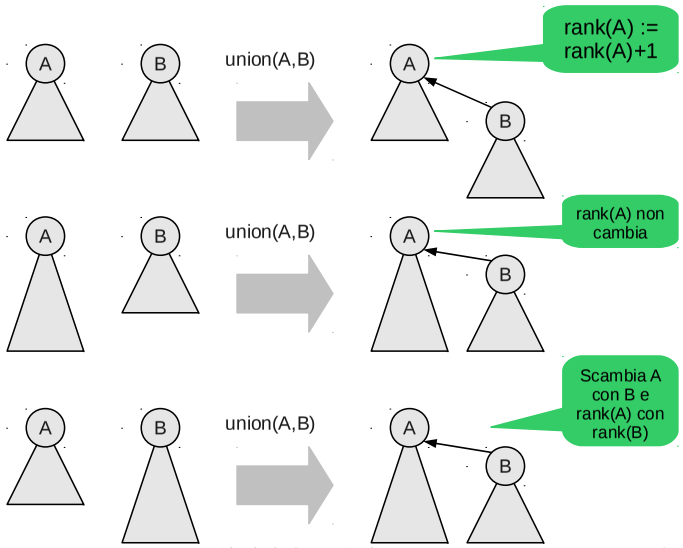
- Alberi QuickFind con bilanciamento sulla union supportano n operazioni `makeSet()`, m operazioni `find()`, ed al più (n-1) operazioni `union()` in tempo totale O(m + n log n).
  L'occupazione di memoria è O(n)
- Dimostrazione
  - n `makeSet()` ed m `find()` costano O(n+m)
  - Dimostriamo che le (n-1) `union()` richiedono tempo complessivo O(n log n)

Costo (n-1) union: metodo dei crediti

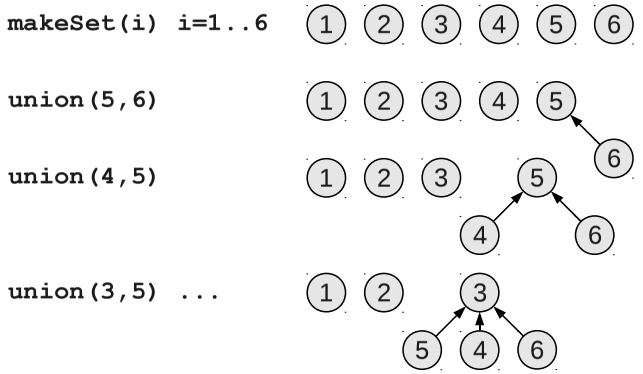
- Associamo ($\log_2$ n) crediti ad ogni elemento
  - Vengono “pagati” da makeSet()
- Ogni volta che colleghiamo un nodo ad un nuovo padre, consumiamo un credito di quel nodo
- ($\log_2$ n) crediti sono sufficienti
  - Ad ogni union *in cui l'elemento cambia padre*, la dimensione dell'insieme cui l'elemento entra a far parte è almeno il doppio dell'insieme originario
  - Dopo k union, ogni elemento appartiene ad un insieme con almeno $2^k \leq n$ elementi
  - Quindi al più ci possono essere $k \leq (\log_2 n)$ cambiamenti di paternità per ogni elemento

QuickUnion
Euristica “union by rank”

- Il problema degli alberi QuickUnion è che possono diventare troppo alti
  - quindi rendere inefficienti le operazioni `find()`
- Idea:
  - Rendiamo la radice dell'albero più basso figlia della radice dell'albero più alto
- Ogni radice mantiene informazioni sul proprio rango
  - il rango *rank(x)* di un nodo *x* è il numero di archi del cammino più lungo fra *x* e una foglia sua discendente
  - rango ≡ altezza del sottoalbero radicato sul nodo



Esempio

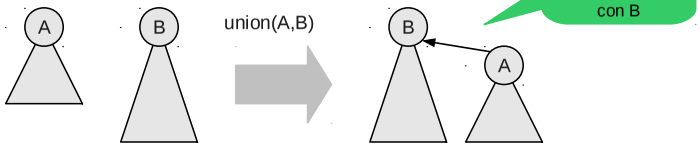


Proprietà alberi QuickUnion bilanciati

- Un albero QuickUnion bilanciato in altezza avente il nodo *x* come radice ha $n \geq 2^{rank(x)}$ nodi
- Dimostrazione: induzione sul numero di operazioni `union()` effettuate
  - Base (0 operazioni union): tutti gli alberi hanno rango zero (singolo nodo) quindi hanno esattamente $2^0=1$ nodi
  - Induzione: consideriamo cosa succede prima e dopo una operazione `union(A,B)`
    - A U B denota l'insieme ottenuto dopo l'unione
    - rank(A U B) è l'altezza dell'albero che denota A U B
    - |A U B| è il numero di nodi dell'albero A U B, e risulta |A U B| = |A| + |B| perché stiamo unendo sempre insiemi disgiunti

Passo induttivo

caso rank(A) < rank(B)

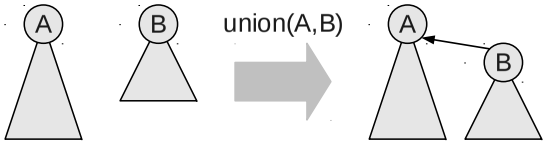


- |A U B| = |A|+|B|
- rank(A U B) = rank(B)
  - perché l'altezza dell'albero A U B è uguale all'altezza dell'albero B
- Per ipotesi induttiva, $|A| \geq 2^{rank(A)}$, $|B| \geq 2^{rank(B)}$
- Quindi

$$|A \cup B| = |A| + |B| \geq 2^{rank(A)} + 2^{rank(B)} > 2^{rank(B)} = 2^{rank(A \cup B)}$$

Passo induttivo

caso rank(A) > rank(B)

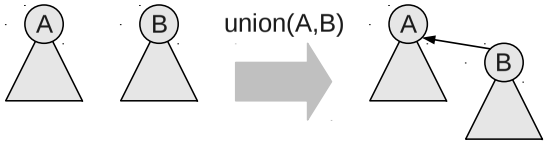


- |A U B| = |A|+|B|
- rank(A U B) = rank(A)
  - perché l'altezza dell'albero A U B è uguale all'altezza dell'albero A
- Per ipotesi induttiva, $|A| \geq 2^{rank(A)}$, $|B| \geq 2^{rank(B)}$
- Quindi

$$|A \cup B| = |A| + |B| \geq 2^{rank(A)} + 2^{rank(B)} > 2^{rank(A)} = 2^{rank(A \cup B)}$$

Passo induttivo

caso rank(A) = rank(B)



- |A U B| = |A|+|B|
- rank(A U B) = rank(A) +1
- Per ipotesi induttiva, $|A| \geq 2^{rank(A)}$, $|B| \geq 2^{rank(B)}$
- Quindi

$$|A \cup B| = |A| + |B| \geq 2^{rank(A)} + 2^{rank(B)} = 2 \times 2^{rank(A)} = 2^{rank(A)+1} = 2^{rank(A \cup B)}$$

Teorema

- Durante una sequenza di operazioni `makeSet()`, `union()` e `find()`, l'altezza di un albero QuickUnion bilanciato è ≤ ($\log_2$ n), essendo n il numero di operazioni `makeSet()`
- Dimostrazione
  - L'altezza di un albero QuickUnion A è rank(A)
  - Da quanto appena visto, $2^{rank(A)} \leq n$
  - Quindi altezza = rank(A) ≤ ($\log_2$ n)
- Quindi:
  - `makeSet()` ha costo O(1) nel caso pessimo
  - `union()` ha costo O(1) nel caso pessimo
  - `find()` ha costo O(log n) nel caso pessimo

QuickUnion
Euristica “union by size”

- Sia size(A) il numero di nodi contenuti nell'albero A
- QuickUnion con bilanciamento sulla cardinalità: in caso di operazione union(A,B)
  - se size(A) ≥ size(B), rendi B figlio della radice di A
  - se size(A) < size(B), rendi A figlio della radice di B
- Si dimostra (allo stesso modo appena visto) che applicando il bilanciamento sulla cardinalità, l'altezza degli alberi è sempre minore o uguale ($\log_2$ n)
  - Valgono quindi gli stessi costi del bilanciamento sull'altezza

Riepilogo

	QuickUnion	QuickFind	QuickFind bilanciato	QuickUnion by size/rank
makeSet	O(1)	O(1)	O(1)	O(1)
union	O(1)	O(n)	O(1)	O(1)
find	O(n)	O(1)	O(log n) amm.	O(log n)