Union Find

Luca Tagliavini

April 13, 2021

Contents

0.1 Union Find

Risolve il problema dell'elaborazione di dati su *insiemi disgiunti*. Immaginiamo di avere due insiemi disgiunti, ossia contenenti elementi che appartengono solo ad uno o all'altro, matematicamente $U_1 \cap U_2 = \emptyset$. Le operazioni su questa struttura dati sono le basilari tra due insiemi:

- *make_set*: crea un insieme singoletto a partire da un singolo elemento
- *find*: cerca tra gli insiemi disponibili quello che contiene il dato elemento
- *union*: unisce due insiemi tra di loro

La struttura dati contiene un insieme dinamico di insiemi $S = \{S_1, S_2, \ldots, S_k\}$ tutti disgiunti tra loro. Tutti gli insiemi complessivamente contengono $n \geq k$ elementi (dove n e' la somma complessiva di tutti gli elementi degli insiemi, k e' il numero degli insiemi: dato che ogni insieme e' almeno singoletto l'assunzione e' ovvia). Oltretutto ogni insieme e' indentificato da un *rappresentante univoco*.

La scelta del rappresentate e' importante, in qunato deve rispettare due leggi:

- il rappresentante di S_i deve essere un qualunque valore contenuto in S_i
- chiamare *find* su $k, K_1 \in S_i$ deve restituire lo stesso rappresentante
- il rappresentatnte puo' cambiare solo dopo una operazione di unione

Un esempio di problema risolvibile tramite strutture *union find* sono quelli della gestione delle spedizioni o del tracciamento dei contatti su una PCB.

0.1.1 Operazioni

- *make_set(elem* $\overline{x}$*)* $\rightarrow$ *repr*: creo un insieme singoletto $\{\overline{x}\}$ ed uso il valore $\overline{x}$ come rappresentante univoco.
  Notare che per le proprieta' della struttura l'elemento $\overline{x}$ non deve essere gia' stato inserito nella struttura dati in precedenza.
- *find(elem x)* $\rightarrow$ *repr*: restituice il rappresentante dell'insieme che contiene l'elemento x.
- *union(repr x, repr y)* $\rightarrow$ *repr*: unisce i due insiemi indicati da x e y, e scegliamo come nuovo rappresentante canonico quello di x.

0.2 Implementazioni

- *quick-find*: usano alberi di altezza uno per ogni insieme, avendo tempi dell'ordine:
  make_set, *find*: $O(1)$, *union*: $O(n)$.

- *quick-union*: usano alberi di altezza dinamica per ogni insieme, avendo tempi dell'ordine:
  make_set, *union*: $O(1)$, *find*: $O(n)$.

- esistono poi implementazioni piu' serie di entrambe le tecniche per avere tempi logaritmici sulle operazioni di *find* o *union* che prima erano lineari.

0.2.1 Quick Find

Usando alberi di altezza uno per gli insiemi, quando si chiama *find(elem x)* sappiamo che x contiene un riferimento al padre (radice dell'albero), che e' il rappresentante dell'insieme, quindi il costo sara' chiaramenteo $O(1)$.

Per implementare *union(repr x, repr y)* possiamo invece cambiare semplicemente i puntatori di tutti i nodi di y in modo che puntino a x come padre. Nel caso peggiore, in cui uniamo $x = |\{\ldots\}| = 1$ con $y = |\{\ldots\}| = n - 1$ e percio' dovremo fare $n - 1$ assegnamenti per i figli di y: il costo sara' dunque $O(n)$.

0.2.2 Quick Union

Si rappresentano gli insiemi tramite un albero radicato generico, dunque ogni *union(repr x, repr y)* sara' semplicemente una aggiunta dell'albero y come ultimo nodo di x, il che ha chiaramente costo $O(1)$ (facendo $y.parent = x$).

Per la *find(elem x)* si parte dal nodo x al quale abbiamo un riferimento e saliamo l'albero fino alla radice la quale contiene il rappresentante, il che nel caso pessimo (quando l'albero e' lineare come una lista) avra' costo $O(n)$.

0.2.3 Euristiche di ottimizzazione

Euristiche su Quick Find:

L'operazione *union* e' poco efficiente nel caso in cui uniamo a un singoletto un insieme grande $n-1$, ma possiamo fare di meglio in questo caso. Anzi che fare l'unione classica come abbiamo definito prima, possiamo guardare il "peso" di ogni albero (mantenuto come valore nella radice in tempo costante $O(1)$ ad ogni operazione) e quando andiamo a fare l'union svolgiamo il cambiamento dei campi "parent" sull'insieme che ha peso minore, il che ci garantisce di svolgere meno operazioni nei casi pessimi. Quando si sposta il primo insieme nel secondo, la specifica sulla scelta del rappresentante non viene rispettata, quindi aggiorniamo la radice in modo tale che sia uguale a quella del primo albero.

Prendendo un insieme singoletto contenente x, esso potra' al massimo cambiare padre $\log_2 n$ volte, poiche' ogni volta che viene inserito in un altro insieme *raddoppia la grandezza dell'insieme di appartenenza di* x, e dunque al piu' il cambiamento sara' necessario al massiom $\log_2 n$ volte (ragionamento inverso degli algoritmi di ricerca ordinata ad esempio).

Ne segue attraverso alcuni passaggi enunciati nelle slide che il costo nel caso pessio e' $O(n/2) = O(n)$, tuttavia nel caso medio, facendo un costo ammortizzato si ha che il costo sara' $O(\log_2 n)$.

Euristiche su Quick Union;

L'operazione *find* e' particolarmente rallentata nel caso in cui l'albero assuma una profondita' eccessiva, accomunando la sua struttura a quella di una lista. Una euristica mirata a risolvere questo problema e' quella che tiene traccia in ogni radice del *rank* (rango) di un albero che ne indica la profondita' massima. Come nella euristica precedente grazie a questa metrica possiamo fare scelte intelligenti durante l'unione di due alberi. In particolare quello che faremo sara' unire y in x solo se $y.rank < x.rank$, altrimenti faremo l'opposto ricordandoci di cambiare poi le radici per mantenere la proprieta' del rappresentante.

In questo modo avremo alberi al massimo di profondita' logaritmica. L'operazione *find* che ha costo pessimo pari alla profondita' massima dell'albero avra' dunque un costo massimo $O(\log_2 n)$.