Azzolini Riccardo 2019-05-14

Partizioni di A e operazioni Union e Find

1 Partizione

Una **partizione** di un insieme A è una famiglia $\{A_1, \ldots, A_m\}$

- di sottoinsiemi di A: $A_i \subseteq A$
- tra loro disgiunti: $\forall i \neq j, \quad A_i \cap A_j = \varnothing$
- la cui unione è l'insieme A: $\bigcup_{i=1,\ldots,m} A_i = A$

Ogni partizione $P = \{A_1, \ldots, A_m\}$ identifica una relazione di equivalenza R_P (e viceversa):

$$\forall x, y \in A, \quad x \; R_P \; y \iff \exists i, \;\; x, y \in A_i$$

Si può quindi scegliere un **elemento rappresentativo** $a_i \in A_i$, per cui $[a_i] = A_i$, e rappresentare P come tupla degli elementi rappresentativi:

$$P \equiv (a_1, \ldots, a_m)$$

La partizione nella quale ogni elemento è rappresentante di se stesso si chiama **partizione identità**.

2 Algebra eterogenea Partizioni di A

Dato un insieme $A = \{a_1, \ldots, a_m\}$, si definisce l'algebra eterogenea **partizioni di A**,

$$\mathrm{PA} = \langle [A, P(A)], [\mathrm{Union}, \mathrm{Find}] \rangle$$

con le operazioni

$$\text{Union} : A \times A \times P(A) \to P(A)$$
$$\text{Union}(x, y, P) = (P \setminus \{[x], [y]\}) \cup \{[x] \cup [y]\}$$

$$\text{Find} : A \times P(A) \to A$$
$$\text{Find}(x, P) = y \iff y = \text{Rappr}([x])$$

dove:

- $P(A)$ è l'insieme di tutte le possibili partizioni di A;
- Union toglie dalla partizione P i due sottoinsiemi $[x]$ e $[y]$, e ne aggiunge uno nuovo, la loro unione $[x] \cup [y]$ (quindi il numero di sottoinsiemi diminuisce di 1);
- Find trova l'elemento rappresentante della classe di equivalenza a cui appartiene x (ciò è possibile perché, per la definizione di partizione, ogni elemento appartiene a una e una sola classe di equivalenza).

Osservazione: $\text{Find}(x_1, P) = \text{Find}(x_2, P) \iff x_1 \; R_P \; x_2$

2.1 Possibili implementazioni

- *Liste concatenate*: sono semplici, ma il costo medio di ciascuna operazione è $O(|A|)$.
- *Foreste* (cioè insiemi) *di alberi*: il costo di ciascuna operazione è $O(\log |A|)$ se si effettua il bilanciamento degli alberi, e non ci sono svantaggi (l'implementazione risulta addirittura più semplice di quella su liste).

3 Partizioni e foreste di alberi

Si rappresenta ciascuna classe di equivalenza tramite un albero avente alla radice l'elemento rappresentativo.

Tali alberi sono con radice, ma non ordinati, e ogni nodo può avere un numero illimitato di figli. Inoltre, per l'implementazione dell'algebra eterogenea *partizioni di A*, è necessario solo risalire gli alberi dalle foglie alla radice. Per questi motivi, li si rappresenta mediante il vettore dei padri.

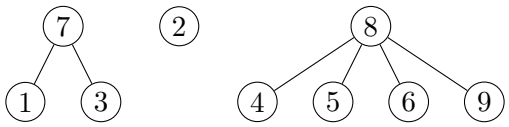
Nel vettore, un nodo radice viene indicato ponendolo come padre di se stesso (un'altra opzione sarebbe usare un valore convenzionale che non corrisponda a un indice del vettore).

3.1 Esempio

Dato l'insieme $A = \{1, \ldots, 9\}$, la partizione

$$P = \{\{1, 3, \underline{7}\}, \{\underline{2}\}, \{4, 5, 6, \underline{8}, 9\}\}$$

(dove gli elementi sottolineati sono quelli rappresentativi) potrebbe essere rappresentata dalla foresta di alberi



che corrisponde al vettore

	1	2	3	4	5	6	7	8	9
padre =	7	2	7	8	8	8	7	8	8

Esempi di operazioni:

- $\text{Find}(5, P) = 8$;
- $\text{Union}(1, 2, P) = \{\{1, 2, 3, \underline{7}\}, \{4, 5, 6, \underline{8}, 9\}\}$ (in questo caso, si è scelto 7 come rappresentante della nuova classe di equivalenza).

3.2 Implementazione

```
public class UnionFind {
    private int[] parent;
    private int count;

    public UnionFind(int n) {
        parent = new int[n];
        count = n;
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
    }

    public int size() { return count; }
```

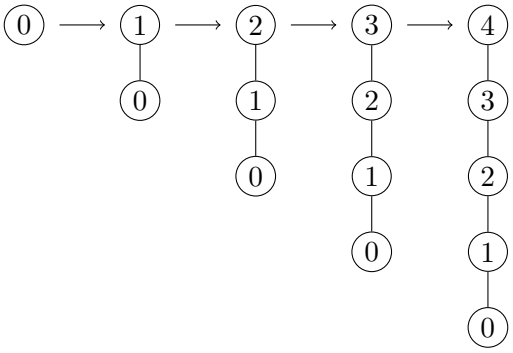
```
    public int find(int x) {
        while (parent[x] != x) {
            x = parent[x];
        }
        return x;
    }

    public void union(int x, int y) {
        int u = find(x);
        int v = find(y);
        if (u != v) {
            parent[v] = u;
            count--;
        }
    }
}
```

- `count` tiene traccia del numero di classi di equivalenza.
- Inizialmente viene creata la partizione identità.
- `find` risale dal nodo `x` fino alla radice dell'albero a cui appartiene, quindi la complessità è $O(h)$, cioè dipende dall'altezza h di tale albero:
  - $O(1)$ nel caso migliore, quando `x` è già la radice;
  - $\Theta(h)$ nel caso peggiore, se `x` è una foglia all'ultimo livello;
- `union` unisce i due alberi a cui appartengono `x` e `y` aggiungendo la radice del secondo ai figli della radice del primo, e poi decrementa il contatore del numero di classi di equivalenza. La sua complessità è asintoticamente uguale a quella di `find`, dato che esegue solo 2 chiamate a `find` e alcune operazioni a costo costante.

Osservazione: quest'implementazione non garantisce prestazioni $O(\log |A|)$, dato che può creare alberi degeneri. Ad esempio, data la partizione identità `parent` $= [0, 1, \ldots, n-2, n-1]$, l'esecuzione del ciclo

```
for (int i = 1; i < n; i++)
    p.union(i, 0);
```

produce l'albero degenere `parent` $= [1, 2, \ldots, n-1, n-1]$:



4 Foreste con bilanciamento

Nell'operazione di Union, i nodi dell'albero che viene posto come sottoalbero si allontanano dalla radice, facendo potenzialmente aumentare l'altezza complessiva.

Tra i due alberi che da unire, conviene quindi mettere come sottoalbero quello con meno nodi. In questo modo, si evita la formazione di alberi degeneri.

L'operazione di Union così modificata si dice **Union con bilanciamento**. Per poterla implementare, è necessario memorizzare la dimensione di ciascun albero: a tale scopo si usa un vettore parallelo `sz`, tale che

$$\texttt{sz}[i] = k \iff \underbrace{\texttt{parent}[i] = i}_{i \text{ è radice}} \wedge \#[i] = k$$

```java
public class UnionFindB {
    private int[] parent;
    private int[] sz;
    private int count;

    public UnionFindB(int n) {
        parent = new int[n];
        sz = new int[n];
        count = n;
        for (int i = 0; i < n; i++) {
            parent[i] = i;
            sz[i] = 1;
        }
    }

    public int size(); // Come in UnionFind
    public int find(int x); // Come in UnionFind
```

```
    public void union(int x, int y) {
        int u = find(x);
        int v = find(y);
        if (u == v) return;
        if (sz[u] < sz[v]) {
            parent[u] = v;
            sz[v] += sz[u];
        } else {
            parent[v] = u;
            sz[u] += sz[v];
        }
        count--;
    }
}
```

4.1 Altezza degli alberi

Teorema: Partendo dalla partizione identità, $[0, 1, \ldots, n-1]$, ed eseguendo unicamente Union con bilanciamento, si ottengono alberi con k nodi e altezza non superiore a $\lfloor \log_2 k \rfloor$.

Dimostrazione: per induzione su k.

- Se $k = 1$, l'albero è formato dalla sola radice, quindi ha altezza 0, e $0 \leq \lfloor \log_2 1 \rfloor = 0$.
- Se $k > 1$, T ha k nodi ($|T| = k$) ed è il risultato di una Union di due alberi T_1 e T_2, con $|T_1| \leq |T_2|$. Allora:
  - siccome T_1 è più piccolo di T_2, si ha per forza $|T_1| \leq \lfloor \frac{k}{2} \rfloor$, e quindi

    $$h(T_1) \leq \left\lfloor \log_2 \frac{k}{2} \right\rfloor = \lfloor \log_2 k \rfloor - 1$$

    per ipotesi di induzione;
  - poiché T_1 ha almeno un nodo, $|T_2| \leq k - 1$, e per ipotesi di induzione

    $$h(T_2) \leq \lfloor \log_2 (k-1) \rfloor \leq \lfloor \log_2 k \rfloor$$

  L'altezza dell'albero T è

  $$h(T) = \max\{h(T_2),\, h(T_1) + 1\}$$

  perché, partendo dalla radice e scendendo alla foglia più lontana, si può

– rimanere in T_2, e in tal caso $h(T) = h(T_2)$;

– entrare nel nuovo sottoalbero T_1 dopo un passo, che implica $h(T) = h(T_1)+1$.

Di conseguenza, $h(T) \leq \lfloor \log_2 k \rfloor$. □

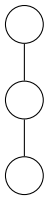
4.2 Complessità

Con l'aggiunta del bilanciamento, la complessità di Union rimane uguale a quella di Find, dato che le operazioni in più sono a costo costante.

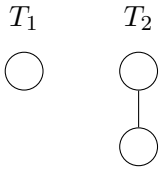
Siccome gli alberi che rappresentano la partizione sono bilanciati, Find (e quindi anche Union) ha prestazioni garantite $O(\log n)$.

4.3 Esempi di alberi impossibili

1. L'albero

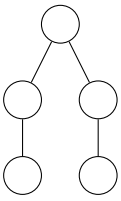


non può essere il risultato di una Union con bilanciamento, perché si potrebbe ottenere solo dai due alberi

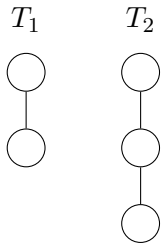


ponendo T_2, il più grande, come sottoalbero di T_1, il più piccolo, ma questa sarebbe una Union "sbilanciata".

2. Anche l'albero



non si può ottenere, perché dovrebbe essere l'unione di



ma T_2 è l'albero impossibile dell'esempio 1.

5 Compressione dei cammini

Utilizzando la Union con bilanciamento, l'esecuzione di $O(n)$ operazioni Union e Find ha costo totale $O(n \log n)$.

Si possono ottenere prestazioni migliori ricorrendo alla **compressione dei cammini**: durante l'esecuzione di Find, si memorizzano tutti i nodi incontrati nella risalita, e al termine li si pone direttamente come figli della radice. In questo modo, si avvicinano alla radice non solo tutti i nodi incontrati, ma anche i loro sottoalberi.

```
public int find(int x) {
    Stack<Integer> s = new Stack<Integer>();
    while (parent[x] != x) {
        s.push(x);
        x = parent[x];
    }
    while (!s.isEmpty()) {
        int v = s.top(); s.pop();
        parent[v] = x;
    }
    return x;
}
```

5.1 Complessità

Teorema (Tarjan): L'esecuzione di $O(n)$ operazioni Union e Find con bilanciamento e compressione dei cammini ha costo $O(n\,G(n))$, dove

$$G(n) = \min\{k \in \mathbb{N} \mid F(k) \geq n\}$$

è l'inversa di $F(k)$, un caso particolare della funzione di Ackermann, definita dall'equazione di ricorrenza

$$F(k) = 2^{F(k-1)}, \quad F(0) = 1$$

Osservazione: $G(n)$ cresce molto lentamente: $G(n) \leq 5$ per $n \leq 2^{65536}$. Di conseguenza, nella pratica il costo di $O(n)$ operazioni Union e Find è $O(n)$, ma è un problema aperto dimostrare se esistono implementazioni che abbiano costo veramente $O(n)$.

5.2 Esempio