Algoritmi e strutture dati

Roberto Cordone

A. A. 2015-16

Capitolo 4

Implementazioni delle partizioni

Nota: queste dispense sono un rapido riassunto delle lezioni svolte nel dicembre 2015 e gennaio 2016. In buona parte ripetono gli argomenti delle dispense del prof. Goldwurm, con alcune aggiunte, qualche piccola modifica nella presentazione, qualche esempio. Chi notasse errori, incoerenze, oscurità o avesse dubbi è invitato a contattarmi per segnalarli.

Si tratta di rappresentare una partizione di un insieme A dato in una collezione di sottoinsiemi disgiunti fra loro, ognuno dei quali ha un elemento rappresentativo. Per esempio, se $A = \{a_1, a_2, a_3, a_4\}$, una possibile partizione è $P = \{\{a_1, \bar{a}_4\}, \{a_2, \bar{a}_3\}\}$, dove gli elementi rappresentativi sono evidenziati barrandoli.

In termini di algebra eterogenea:

$$([A, \mathrm{PART}(A)], [\mathrm{UNION}, \mathrm{FIND}, \mathrm{ID}, a_1, \ldots, a_m])$$

dove

- la funzione `FIND` riceve una partizione e un elemento e restituisce l'elemento rappresentante del sottoinsieme di cui l'elemento dato fa parte nella partizione data;
- la funzione `UNION` riceve una partizione e due elementi e restituisce una partizione nella quale i due sottoinsiemi di cui fanno parte gli elementi dati sono fusi in uno solo.
- Un'implementazione banale consiste nel descrivere ogni sottoinsieme della partizione con una lista di elementi, magari tenendo in cima il rappresentante:
  - la funzione `UNION` consiste nel fondere due liste: questo richiede tempo $O(n)$, perché si tratta di spostare tutti gli elementi di una lista nell'altra. Si può fare meglio se si usa una lista circolare con un puntatore all'ultimo elemento, che permette di appendere una lista all'altra in tempo costante $O(1)$;
  - la funzione `FIND`, però, richiede di scorrere tutte le liste per trovare l'elemento, in modo da sapere quale lista lo contiene, e quindi determinarne il rappresentante: tempo $O(n)$.
- Un'altra implementazione banale consiste nel tenere in un vettore l'indicazione del rappresentante per ciascun elemento:

– la funzione FIND richiede tempo costante, perché basta consultare il vettore per avere la risposta cercata;

– la funzione UNION consiste nello scorrere il vettore e cambiare il rappresentante di uno dei due sottoinsiemi, sostituendolo con il rappresentante dell'altro, in modo da fondere i due sottoinsiemi: questo richiede tempo $O(n)$.

4.4 I Merge-Find set (MF-set)

- Una rappresentazione sofisticata cerca un compromesso fra l'efficienza delle due funzioni. Si tratta di vedere la partizione come una foresta, in cui gli elementi sono nodi e in particolare i rappresentanti sono le radici degli alberi. Ogni nodo punta un altro nodo, tranne i rappresentanti che puntano a NULL. Si veda la Figura 4.1.

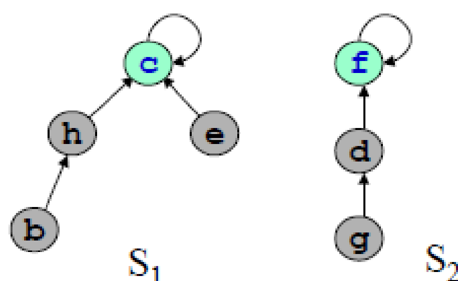

Figura 4.1: Una partizione vista come foresta radicata

- L'implementazione pratica può sfruttare i puntatori, ma in questo caso conviene usare la rappresentazione a vettore, perché la dimensione complessiva della struttura non cambia mai durante l'uso (contrariamente ai dizionari che subiscono inserimenti e cancellazioni, per cui una struttura a vettore può essere troppo rigida). Le radici punteranno, anziché a NULL a un indice numerico particolare, come 0 o −1 (se 0 è un indice valido per il vettore) oppure punteranno all'indice dell'elemento stesso (l'elemento i avrà `P[i].father == i`). Nel seguito, useremo 0 per coerenza con le dispense.

```
struct elemento {
  A  *a;       /* elemento */
  int father;  /* indice dell'elemento padre */
};

typedef struct elemento MFset[n];

MFset P;
```

dove una partizione P è vista come un vettore di strutture che contengono un elemento e l'indice dell'elemento padre. Spesso, invece di un vettore di strutture, si usano più vettori separati, uno per ciascun campo.

```
A  *a[n];       /* vettore degli elementi */
```

```
int father[n];  /* vettore degli indici padre */
```

- La funzione `FIND` consiste semplicemente nel risalire l'albero di cui fa parte l'elemento dato, fino a trovare la radice (Figura 4.2).

```
A *FIND (A *a, MFset P)
{
  k = KEY(a);
  while (P[k].father != 0)
    k = P[k].father;
  return P[k].a;
};
```

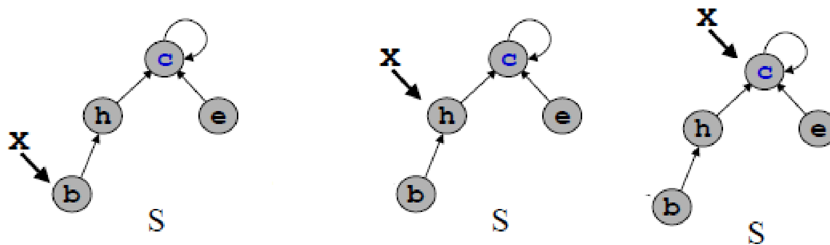

Figura 4.2: Ricerca del rappresentante di un nodo dato

La complessità dipende dal bilanciamento dell'albero e dal numero di figli: può essere qualsiasi fra

  - $O(1)$, per alberi in cui tutti gli elementi sono direttamente appesi alla radice;
  - $O(n)$, per alberi in cui ogni elemento è concatenato agli altri a formare una lista.

- La funzione `UNION` nella sua forma più semplice consiste nel cercare i due rappresentanti e attaccare uno di loro all'altro (per convenzione, ad esempio, il secondo al primo), come nella Figura 4.3.

```
MFset UNION (A *a1, A *a2, MFset P)
{
  v1 = FIND(a1);
  k1 = KEY(v1);
  v2 = FIND(a2);
  k2 = KEY(v2);
  if (k1 != k2) P[k2].father = k1;
  return P;
};
```

Questo richiede due volte il tempo della funzione `FIND` più tempo costante.

- Esercizio: si calcolino le componenti connesse del grafo che ha i seguenti lati: (d,g), (h,b), (b,g), (c,e), (i,l), (d,b), (e,g), (f,c), (e,d)

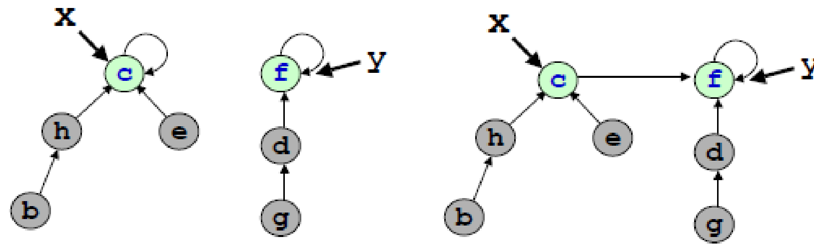

Figura 4.3: Fusione di due sottoinsiemi in uno solo

  - Risultano due alberi, uno di altezza 4 radicato in f e l'altro di altezza 1 radicato in i e contenente i e l.

- Tutto dipende quindi da quanto si riesce a tenere "piatto" ciascun albero della foresta, cioè a limitare l'altezza massima dei suoi alberi. Ci sono diversi modi per garantire questo risultato.

Bilanciamento basato sulla cardinalità

- L'idea base è appendere sempre l'albero di cardinalità minore a quello di cardinalità maggiore, anziché banalmente il secondo al primo. Si può dimostrare per induzione che questo limita l'altezza di ogni albero a $\lfloor \log_2 n \rfloor + 1$.
  - per $n = 1$, l'albero ha solo la radice, cioè un livello, che è $h(1) = 1 \leq \lfloor \log_2 1 \rfloor + 1$.
  - se la tesi vale per ogni $i < n$, significa che vale per i due sottoalberi che fondiamo in una applicazione della funzione `UNION`, che hanno cardinalità n_1 e n_2 positive e tali che $n_1 + n_2 = n$;
  - supponiamo $n_1 < n_2$, per cui appendiamo T_1 a T_2;
  - inoltre $n_1 < n_2$ e $n_1 + n_2 = n$ implica $2n_1 < n$, mentre $n_2 n$;
  - l'albero risultante dalla fusione ha altezza pari al massimo fra quella di T_1 aumentata di 1 e quella di T_2: $h(n) = \max(h(n_1) + 1, h(n_2))$
  - entrambe le altezze rispettano la tesi, per cui

$$\begin{cases} h(n) \leq h(n_1) + 1 \leq \lfloor \log_2 n_1 \rfloor + 2 < \lfloor \log_2 n/2 \rfloor + 2 = \lfloor \log_2 n \rfloor + 1 \\ h(n) \leq h(n_2) \leq \lfloor \log_2 n_2 \rfloor + 1 < \lfloor \log_2 n \rfloor + 1 \end{cases} \Rightarrow$$

$$\Rightarrow h(n) \leq \lfloor \log_2 n \rfloor + 1$$

    che è la tesi.

- Questo meccanismo richiede di conservare nei MF-set anche la cardinalità degli alberi: basta aggiungere un campo `num` ad ogni struttura, che indica (per i soli elementi rappresentanti) la cardinalità dell'albero appeso.

```
MFset UNION (A *a1, A *a2, MFset P)
{
  v1 = FIND(a1);
  k1 = KEY(v1);
  v2 = FIND(a2);
  k2 = KEY(v2);
```

```
    if (k1 != k2)
    {
      if (P[k1].num < P[k2].num)
      {
        P[k1].father = k2;
        P[k2].num = P[k1].num + P[k2].num;
        P[k1].num = 0; /* non strettamente necessario */
      }
      else
      {
        P[k2].father = k1;
        P[k1].num = P[k1].num + P[k2].num;
        P[k2].num = 0; /* non strettamente necessario */
      }
    }

    return P;
  };
```

- Esercizio: si calcolino le componenti connesse del grafo che ha i seguenti lati: (d, g), (h, b), (b, g), (c, e), (i, l), (d, b), (e, g), (f, c), (e, d)
  - Risultano due alberi, uno di altezza 2 radicato in h e l'altro di altezza 1 radicato in i e contenente i e l.

Bilanciamento basato sull'altezza

- Invece di considerare la cardinalità, ci si può concentrare su ciò che effettivamente interessa, cioè l'altezza degli alberi, appendendo sempre l'albero di altezza minore a quello di altezza maggiore (in caso di parità, il secondo al primo);
- L'albero risultante ha altezza pari alla maggiore delle due, se sono diverse; a entrambe più 1 se sono uguali.
- Questo comporta un campo `rank`, analogo al campo `num` prima introdotto;

```
  MFset UNION (A *a1, A *a2, MFset P)
  {
    v1 = FIND(a1);
    k1 = KEY(v1);
    v2 = FIND(a2);
    k2 = KEY(v2);
    if (k1 != k2)
    {
      if (P[k1].rank < P[k2].num)
      {
        P[k1].father = k2;
        P[k1].rank = 0; /* non strettamente necessario */
      }
      else
      {
        P[k2].father = k1;
        if (P[k1].rank == P[k2].rank) P[k1].rank = P[k1].rank + 1;
```

```
      P[k2].num = 0; /* non strettamente necessario */
    }
  }

  return P;
};
```

- Anche questa strategia garantisce un'altezza logaritmica rispetto al numero di elementi.
- Esercizio: si calcolino le componenti connesse del grafo che ha i seguenti lati: (d, g), (h, b), (b, g), (c, e), (i, l), (d, b), (e, g), (f, c), (e, d)
  - Risultano gli stessi due alberi ottenuti con il bilanciamento in cardinalità.

Euristica di compressione dei cammini

- Ogni volta che si esegue la funzione `FIND` si percorre una catena di nodi che sarebbe meglio poter "spianare" appendendoli tutti alla radice. Questo è banalmente possibile: basta ripetere due volte la catena
  1. la prima volta per trovare la radice;
  2. la seconda volta per appendere ciascun nodo della catena alla radice.
- Questo comporta l'uso del campo `rank` e l'integrazione della funzione `FIND` come segue:

```
A *FIND (A *a, MFset P)
{
  /* si determina la radice dell'albero */
  k = r = KEY(a);
  while (P[r].father != 0)
    r = P[r].father;

  /* si appende direttamente ogni nodo della catena all'albero */
  while (P[k].father != 0)
  {
    p = P[k].father;
    P[k].father = r;
    k = p;
  }

  return P[r].a;
};
```

- Se si modifica in questo modo la funzione `FIND`, ovviamente, il suo tempo di esecuzione raddoppia. Tuttavia, dato che essa viene usata nella funzione `UNION`, una sequenza di `UNION` tenderà a creare una foresta molto più piatta, e quindi le esecuzioni successive diventano molto più rapide.
- Si può dimostrare che l'altezza della foresta tende a diventare praticamente costante al crescere di n. Infatti,

$$h(n) \leq \min \{k \in \mathbb{N} : n \leq F(k)\} \text{ con } F(k) = \begin{cases} 1 \text{ per } k = 0 \\ 2^{F(k-1)} \text{ per } k \geq 1 \end{cases}$$

- Siccome F cresce velocissima $(1, 2, 4, 16, 2^{16}, 2^{2^{16}}, \ldots)$, praticamente $(n) \leq 5$ in qualsiasi situazione.
- Esercizio: si calcolino le componenti connesse del grafo che ha i seguenti lati: (d, g), (h, b), (b, g), (c, e), (i, l), (d, b), (e, g), (f, c), (e, d)
  - Il lato (b, g) fa anche collegare direttamente g ad h;
  - Il lato (e, d) non fonde sottoinsiemi, ma applica la compressione e rende la foresta completamente piatta: due alberi di altezza 1