1. Code con priorità e insiemi disgiunti

Le strutture di dati visti finora, con l'esclusione dei grafi, sono state utilizzate per realizzare i tipi di dato astratto sequenza, insieme e dizionario. Esistono dei casi, tuttavia, in cui questi tipi di dato non soddisfano le peculiari esigenze di un algoritmo, magari perché mancano di una particolare operazione, oppure perché la loro generalità li rende inefficienti per compiti specifici.

Può essere quindi necessario progettare nuovi tipi di dato, e nuove strutture per contenerli. L'elenco di strutture di dati speciali è lungo, e contiene per esempio gli alberi *trie*, gli *heap binomiali*, gli *heap di Fibonacci*, ecc. Per motivi di spazio, ci limiteremo ad esaminare due esempi molto diffusi, ovvero code con priorità basate su *heap* binari e insiemi disgiunti.

1.1 Code con priorità

Una *coda con priorità* (*priority queue*, in inglese) è una particolare struttura di dati, i cui elementi sono associati a *priorità*, valori numerici confrontabili tramite una relazione "$\leq$" di ordinamento totale. Il nome deriva dall'interpretazione in cui tale coda contiene elementi che devono essere serviti rispettandone la priorità.

> **Esempio (Pronto Soccorso)** Nella sala d'attesa del pronto soccorso di un ospedale, un paziente colpito da un incidente che può risultargli fatale sarà soccorso prima di altri pazienti, affetti da malanni meno gravi, che erano già in attesa prima di lui. ■

La mappatura fra il concetto di priorità e i valori numerici dipende da come viene utilizzata la coda. In alcuni casi, si dà la priorità ai valori numerici più bassi (*priorità decrescente*), come nel caso di un sistema operativo che esegue processi in base alla loro scadenza (*earliest deadline first*); in altri casi, si dà la priorità ai valori numerici più alti (*priorità crescente*).

Questo capitolo descrive le code con priorità decrescente, che verranno utilizzate per risolvere il problema dei CAMMINI MINIMI [cfr. Capitolo 11]. In questo tipo di dati astratto, sono ammesse le operazioni insert(), che inserisce un nuovo elemento associato ad un valore di priorità; min(), che restituisce l'elemento con valore di priorità più basso; deleteMin(), che restituisce l'elemento con valore di priorità più basso dopo averlo cancellato; e infine decrease(), che diminuisce la priorità di

un elemento esistente. In una coda con priorità crescente, insert() rimane inalterato e le operazioni max(), deleteMax() e increase() sostituiscono le altre operazioni.

PRIORITYQUEUE

```
% Crea una coda con priorità vuota
PriorityQueue()

% Restituisce true se la coda con priorità è vuota
boolean isEmpty()

% Restituisce l'elemento minimo di una coda con priorità non vuota
item min()

% Rimuove e restituisce il minimo da una coda con priorità non vuota
item deleteMin()

% Inserisce l'elemento x con priorità p in una coda con priorità non piena
% Restituisce un oggetto PRIORITYITEM che identifica x all'interno della coda
PRIORITYITEM insert(item x, int p)

% Diminuisce la priorita' dell'oggetto identificato da y portandola al valore p
decrease(PRIORITYITEM y, int p)
```

Se più elementi hanno la stessa priorità, questa specifica non descrive in che ordine vengono estratti. La procedura insert() restituisce un oggetto che identifica l'elemento appena inserito nella coda con priorità. Questo identificatore è necessario unicamente per l'operazione decrease(), che dovendo modificare la priorità di un oggetto all'interno della coda ha bisogna di una maniera agile per ritrovare questo oggetto. Nel descrivere la realizzazione, quando questo non genera confusione, non faremo distinzione fra un elemento e la sua priorità.

1.1.1 Realizzazione con liste

Si può realizzare una coda con priorità utilizzando liste, ordinate o non ordinate. Nel caso di liste ordinate, le operazioni min() e deleteMin() hanno complessità $O(1)$, mentre insert() e decrease() sono $O(n)$. Nel caso di liste non ordinate, la complessità di insert() e decrease() può scendere ad $O(1)$, mentre quella di min() e deleteMin() sale a $O(n)$. In questa implementazione, l'oggetto di tipo PRIORITYITEM restituito dalla insert() è un puntatore alla cella che contiene l'elemento appena inserito.

1.1.2 Realizzazione con alberi bilanciati

Un'ulteriore possibilità è quella di utilizzare alberi bilanciati; in questo caso, tutte le operazioni hanno costo $O(\log n)$, a scapito di una più complessa organizzazione della struttura di dati e di una maggiore occupazione di memoria.

1.1.3 Realizzazione con *heap*

Se il numero di elementi presenti nella coda con priorità può essere limitato superiormente, allora è possibile rendere uniforme la complessità di deleteMin(), insert() e decrease() con una realizzazione sequenziale efficiente e particolarmente semplice, che usa un vettore chiamato *heap*.

Gli elementi della coda con priorità di dimensione n possono essere disposti in un vettore $H[1 \ldots n]$ (detto appunto *heap*) che può essere interpretato come un albero binario B. Ciascun nodo di B corrisponde ad una ben precisa posizione nel vettore H e memorizza un elemento della coda con priorità. L'albero B deve verificare le seguenti proprietà:

(1) se h è il livello massimo delle foglie, allora ci sono esattamente $2^h - 1$ nodi di livello minore di h;

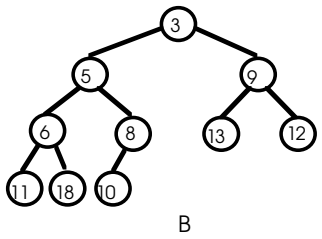

Figura 1.1: Un albero binario B che rappresenta uno *heap*.

(2) tutte le foglie di livello h sono addossate a sinistra;
(3) ogni nodo diverso dalla radice contiene un elemento della coda con priorità che è maggiore o uguale a quello contenuto nel padre.

Esempio (Albero B) L'albero B di Figura 1.1 contiene gli elementi della coda con priorità $C = \{5,10,8,11,13,12,9,18,3,6\}$ e verifica le proprietà (1)–(3). Il livello massimo delle foglie è 3, ci sono $2^3 - 1 = 7$ nodi di livello al più 2, e le 3 foglie di livello 3 sono addossate a sinistra. ■

Grazie alle proprietà (1) e (2), ciascun livello k di B (tranne l'ultimo) contiene tutti i possibili 2^k nodi, $0 \leq k < h$, mentre i nodi di livello massimo h sono addossati sul lato "sinistro" dell'albero. In altri termini, B è "quasi pieno", cioè non ha nodi "mancanti" tranne le foglie a destra di livello massimo. Inoltre, tutti i nodi interni hanno grado 2, tranne al più un nodo interno con grado 1.

È possibile passare da una rappresentazione ad albero ad una rappresentazione a vettore in modo molto semplice: si scorre l'albero per livelli, dall'alto verso il basso, e da sinistra verso destra: prima il nodo radice al livello 0, poi i suoi figli al livello 1, poi i nodi del livello 2, e così via fino all'ultimo livello h. Poiché ad ogni livello il numero di nodi raddoppia, si ha che:

- la radice di B viene memorizzata in $H[1]$;
- i figli sinistro e destro del nodo i-esimo (memorizzato in $H[i]$) vengono memorizzati nelle posizioni $H[2i]$ e $H[2i+1]$, rispettivamente;
- il padre del nodo $i > 1$ è memorizzato nella posizione $H[\lfloor i/2 \rfloor]$.

Per comodità, nel seguito utilizzeremo le seguenti abbreviazioni per indicare le posizioni del padre e dei figli sinistro e destro, se esistono:

- $p(i) = \lfloor i/2 \rfloor$ (padre)
- $l(i) = 2i$ (figlio sinistro)
- $r(i) = 2i+1$ (figlio destro)

È facile verificare che il figlio sinistro (destro) di i esiste se e solo se $l(i) \leq n$ ($r(i) \leq n$). È possibile esprimere la proprietà (3) direttamente sul vettore *heap*: $H[i] \geq H[p(i)], \forall i > 1$.

Esempio (Heap H) Il vettore di Figura 1.2 rappresenta l'albero di Figura 1.1. ■

Queste definizioni, e lo pseudocodice che segue, sono relativi ad una coda con priorità decrescente. In questo caso, la struttura di dati viene chiamata *min-heap*. La struttura simmetrica per code con priorità crescente viene chiamata *max-heap*.

La realizzazione delle operazioni della coda con priorità è molto semplice se si immagina di operare nell'albero B anziché sul vettore H. Infatti, per eseguire min() basta leggere l'elemento contenuto nella radice, mentre per eseguire deleteMin() si cancella la foglia di livello massimo più a destra, sì da mantenere verificate le proprietà (1) e (2), se ne copia l'elemento nella radice, e si

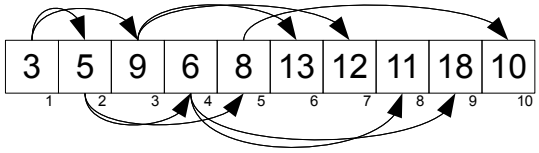

Figura 1.2: L'heap corrispondente all'albero binario di Figura 1.1.

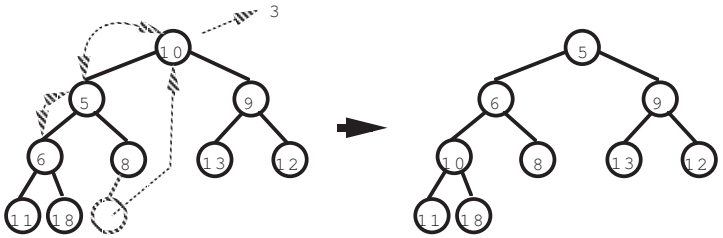

Figura 1.3: Cancellazione del minimo.

fa "scendere" tale elemento lungo un percorso radice-foglia, scambiandolo con il minimo degli elementi contenuti nei figli, fino a verificare la proprietà (3).

Esempio (deleteMin()) Volendo cancellare 3 dall'albero B di Figura 1.1, si cancella la foglia contenente 10, si ricopia 10 nella radice, si scambia 10 con 5 e successivamente 10 con 6 (Figura 1.3). ■

Per eseguire insert(), si aggiunge una foglia col nuovo elemento al livello massimo più a sinistra, per verificare le proprietà (1) e (2), facendo poi "salire" tale elemento lungo un percorso foglia-radice scambiandolo col padre, fino a verificare la proprietà (3). Lo stesso meccanismo si applica alla decrease(): la nuova priorità potrebbe essere inferiore a quella del padre e dei suoi antenati.

Esempio (insert) Volendo inserire l'elemento 4 nell'albero B di Figura 1.1, si aggiunge un figlio, con 4, al nodo contenente 8, si scambia 4 con 8 e successivamente 4 con 5 (Figura 1.4). ■

La proprietà (1) di B permette di mantenere il livello massimo h delle foglie limitato da $\lfloor \log n \rfloor$. Infatti, uno *heap* con altezza h ha un numero di nodi n compreso fra 2^h e $2^{h+1}-1$, da cui $h = \lfloor \log n \rfloor$.

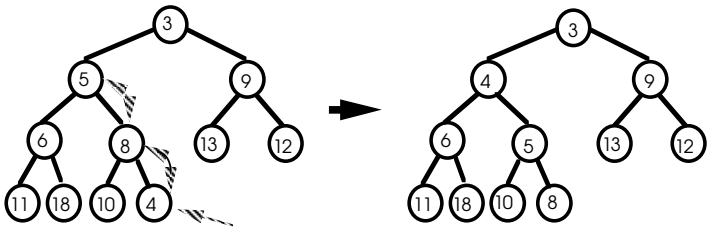

Figura 1.4: Inserimento di 4.

Poiché deleteMin() e insert() provocano scambi di elementi lungo un solo percorso radice-foglia, entrambi possono essere realizzati in modo da richiedere al più $O(\log n)$ passi.

Vediamo ora in dettaglio le procedure in pseudocodice che realizzano la coda con priorità basata su *heap*. Per semplicità, le priorità saranno rappresentate da numeri interi, ma qualunque tipo dotato di una relazione di ordinamento $\leq$ potrebbe andare bene. La struttura di dati è mantenuta in un vettore *heap* di record di tipo PRIORITYITEM; ognuno di questi record contiene, oltre ad una coppia elemento-priorità, la posizione attuale del record all'interno del vettore. Questa informazione è necessaria per realizzare decrease().

```
PRIORITYITEM
int priority                          % Priorità
item value                            % Elemento
int pos                               % Posizione nel vettore heap
```

Oltre al vettore *heap* H, la struttura di dati contiene un intero *capacity* che indica la dimensione massima del vettore, inizializzata al momento di costruire la struttura di dati, e un intero *dim* che indica quanti elementi sono attualmente presenti nel vettore e corrisponde alla posizione (nel vettore *heap*) della foglia di B che sta più a destra fra quelle di livello massimo. Il valore 0 in *dim* indica una coda con priorità vuota. La funzione min() è molto semplice: se la coda con priorità non è vuota, è sufficiente restituire il primo elemento del vettore.

```
PRIORITYQUEUE
int capacity                          % Numero massimo di elementi nella coda
int dim                               % Numero attuale di elementi nella coda
PRIORITYITEM[] H                      % Vettore heap

PRIORITYQUEUE PriorityQueue(int n)
    PRIORITYQUEUE t = new PRIORITYQUEUE
    t.capacity = n
    t.dim = 0
    t.H = new PRIORITYITEM[1...n]
    return t

item min()
    precondition: dim > 0
    return H[1].value
```

Nella insert(), si incrementa il valore di *dim* per far spazio al nuovo elemento x. Questo corrisponde ad aggiungere una foglia nel livello massimo di B, nella posizione più a sinistra fra quelle disponibili. In questa posizione, l'elemento x potrebbe non rispettare la proprietà (3), ovvero potrebbe essere più piccolo di suo padre. Il cursore i viene inizializzato a *dim*; se i non coincide con la radice e l'elemento contenuto in $H[i]$ è più piccolo di quello di suo padre, questi vengono scambiati dall'operazione swap() e il cursore i risale alla posizione del padre $p[i]$. Il ciclo **while** si ripete fino a quando la proprietà (3) non viene rispettata. Nel caso pessimo, la risalita continua fino alla radice ($i = 1$). La procedura termina restituendo l'oggetto PRIORITYITEM in cui l'elemento è stato inserito.

PRIORITYITEM insert(**item** x, **int** p)

precondition: $dim < capacity$

$dim = dim + 1$
$H[dim] =$ **new** PRIORITYITEM()
$H[dim].value = x$
$H[dim].priority = p$
$H[dim].pos = dim$
int $i = dim$
while $i > 1$ **and** $H[i].priority < H[p(i)].priority$ **do**
  swap$(H, i, p(i))$
  $i = p(i)$
return $H[i]$

La funzione swap(), oltre a scambiare le posizioni i e $p(i)$, deve anche tenere aggiornata la posizione dei record mano a mano che vengono spostati.

swap(PRIORITYITEM$[]$ H, **int** i, **int** j)

$H[i] \leftrightarrow H[j]$
$H[i].pos = i$
$H[j].pos = j$

Per realizzare deleteMin() useremo una procedura ausiliaria minHeapRestore(), che sarà utilizzata anche nella sezione successiva. deleteMin() si basa su un semplice trucco: sovrascriviamo la prima posizione dello *heap* (quella che contiene il minimo) con l'elemento che si trova nella foglia più a destra del livello massimo, e decrementiamo *dim* ad indicare che il numero di elementi presenti è diminuito di 1. L'albero risultante è "quasi" uno *heap*: i sottoalberi radicati nei figli sinistro e destro della radice rispettano la proprietà (3), mentre l'unica possibile eccezione è proprio la radice, che potrebbe essere maggiore dei propri figli.

La procedura minHeapRestore(A, i, dim) ha il compito di rettificare questa situazione; parte dall'assunto che gli alberi radicati in $l(i)$ e $r(i)$ (se esistenti) rispettino già le proprietà dello *heap*, e verifica se uno dei figli di i è minore di i stesso. Se questo accade ($i \neq min$), gli elementi in $A[i]$ e $A[min]$ vengono scambiati. A questo punto, il problema può essersi spostato nella posizione min: siamo certi che i sottoalberi radicati in $l(min)$ e $r(min)$ sono *heap* (lo erano prima di eseguire questa chiamata di minHeapRestore()), ma è possibile che l'elemento min violi la proprietà (3) dello *heap*, ovvero che $A[min]$ sia maggiore di $A[l(min)]$ o $A[r(min)]$. La procedura minHeapRestore() viene quindi chiamata ricorsivamente su min. La chiamata ricorsiva termina quando i e i suoi figli rispettano le proprietà di uno *heap*.

Il codice della deleteMin() e della minHeapRestore() è mostrato di seguito; si noti che deleteMin() chiama minHeapRestore() sulla radice, subito dopo aver eseguito lo scambio con l'ultimo elemento.

Concludiamo con la procedure decrease(), che prende in input un oggetto di tipo PRIORITYITEM e ne aggiorna la priorità, riposizionandolo nel vettore *heap* se necessario. Poiché la priorità è diminuita, viene confrontata con quella del padre e scambiata se necessario.

```
minHeapRestore(PRIORITYITEM[] A, int i, int dim)
    int min = i
    if l(i) ≤ dim and A[l(i)].priority < A[min].priority then min = l(i)
    if r(i) ≤ dim and A[r(i)].priority < A[min].priority then min = r(i)
    if i ≠ min then
        swap(A, i, min)
        minHeapRestore(A, min, dim)
```

```
item deleteMin()
    precondition: dim > 0
    swap(H, 1, dim)
    dim = dim − 1
    minHeapRestore(H, 1, dim)
    return H[dim + 1]
```

```
decrease(PRIORITYITEM x, int p)
    precondition: p < x.priority
    x.priority = p
    int i = x.pos
    while i > 1 and H[i].priority < H[p(i)].priority do
        swap(H, i, p(i))
        i = p(i)
```

1.1.4 Heapsort

L'impiego più famoso della struttura di dati *heap* permette di realizzare un algoritmo di ordinamento di complessità ottima.

Esempio (Ordinamento) Si supponga che la sequenza di n elementi da ordinare sia memorizzata nel vettore A. È possibile ordinare gli elementi di A utilizzando una coda con priorità decrescente, nella quale sono dapprima inseriti in sequenza tutti gli elementi, e dalla quale sono poi estratti, sempre in sequenza, gli elementi minimi.

```
sort(item[] A, int n)
    PRIORITYQUEUE Q = PriorityQueue(n)
    for i = 1 to n do                          % Inserisce gli elementi nello heap
        Q.insert(A[i], A[i])
    for int i = 1 to n do                      % Ricopia gli elementi in ordine
        A[i] = Q.deleteMin()
```

Se la coda con priorità è realizzata con uno *heap*, la complessità della procedura sort() è $O(n \log n)$. Per quanto descritto nell'Esempio 5.5, una limitazione inferiore alla complessità del problema dell'ordinamento utilizzando confronti è $\Omega(n \log n)$. La procedura sort() ha quindi complessità ottima. ■

Benché la procedura sort() abbia complessità ottima se si usa uno *heap*, essa presenta lo svantaggio di richiedere un doppio trasferimento di elementi dal vettore A allo *heap* e dallo *heap* di nuovo al vettore A, con conseguente spreco di memoria. È possibile però effettuare l'ordinamento in loco, cioè direttamente nel vettore A senza un secondo vettore d'appoggio, dopo aver "ristrutturato"

A stesso in modo che verifichi le proprietà di un *max-heap*. Il motivo per cui è necessario un *max-heap* sarà chiarito a breve.

Dato il vettore $A[1 \dots n]$ in cui gli elementi sono disposti in un ordine qualunque, è possibile permutarli richiamando maxHeapRestore() più volte in modo che al termine il vettore A soddisfi le proprietà di *max-heap*. Poiché maxHeapRestore() lavora assumendo che gli alberi sottostanti siano già *max-heap*, dobbiamo partire dalle foglie e risalire verso la radice. Tutti gli elementi compresi fra $\lfloor n/2 \rfloor + 1$ e n sono foglie nell'albero B; sono quindi già dei *max-heap* per definizione, essendo composti da un solo elemento. Applicando maxHeapRestore() partendo da $\lfloor n/2 \rfloor$ e risalendo fino ad 1, si restaurano le proprietà di *max-heap* essendo certi che i sottoalberi siano *max-heap* (infatti, o sono foglie o sono nodi su cui abbiamo già chiamato maxHeapRestore()).

```
heapBuild(item[] A, int n)
    for i = ⌊n/2⌋ downto 1 do
        maxHeapRestore(A, i, n)
```

Invece di operare su una struttura di dati a parte e su associazioni elementi-priorità, la versione di maxHeapRestore() utilizzata qui lavora direttamente sul vettore di elementi preso in input.

```
maxHeapRestore(item[] A, int i, int dim)
    int max = i
    if l(i) ≤ dim and A[l(i)] > A[max] then max = l(i)
    if r(i) ≤ dim and A[r(i)] > A[max] then max = r(i)
    if i ≠ max then
        A[i] ↔ A[max]
        maxHeapRestore(A, max, dim)
```

Per valutare la complessità di heapBuild(), dobbiamo valutare la complessità di maxHeapRestore(), che dipende da i e dim. Sia m l'altezza del nodo i nel solito albero che rappresenta lo heap $A[i \dots dim]$, definita come segue: l'altezza di una foglia è zero, mentre l'altezza di un nodo che non è una foglia è uguale al massimo delle altezze dei figli più uno. Poiché la procedura si basa su una discesa dal nodo i fino ad una foglia, essa richiede tempo $O(m)$.

Si noti che in uno heap di n elementi ci sono al più $\lceil n/2^{m+1} \rceil$ nodi di altezza m. Poiché maxHeapRestore() richiede tempo $O(m)$ quando è richiamata su un nodo i di altezza m, la complessità di heapBuild() è

$$O\left(\sum_{m=0}^{\log n} \frac{n}{2^{m+1}} \cdot m\right),$$

che è dello stesso ordine di grandezza di

$$O\left(n \sum_{m=0}^{\log n} \frac{m}{2^m}\right).$$

Dato che

$$\sum_{m=0}^{\log n} \frac{m}{2^m} < \sum_{m=0}^{\infty} \frac{m}{2^m} = \frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2} = 2$$

la complessità di heapBuild() è $O(n)$.

Terminata la ristrutturazione, l'elemento massimo di $A[1 \dots n]$ si trova in $A[1]$. La posizione che gli compete è $A[n]$; facendo un'operazione deleteMax(), l'elemento in $A[n]$ viene sistemato nello heap $A[1 \dots n-1]$ e $A[n]$ diviene libera. È quindi possibile scrivere l'elemento massimo in $A[n]$.

L'operazione può quindi essere ripetuta con lo heap $A[1 \ldots n-1]$, poi con lo heap $A[1 \ldots n-2]$, e così via fino allo heap $A[1 \ldots 2]$; terminata quest'ultima operazione, il minimo si trova in $A[1]$.

La procedura di ordinamento risultante è detta heapsort() ed è dovuta a Williams (1964); si noti che invece di chiamare max() per leggere il massimo e una deleteMax() per rimuoverlo, ad ogni passo il primo e l'ultimo elemento vengono scambiati e maxHeapRestore() viene chiamata direttamente.

heapsort(**item**[] A, **int** n)

```
heapBuild(A, n)
for i = n downto 2 do
    A[1] ↔ A[i]
    maxHeapRestore(A, 1, i − 1)
```

La chiamata iniziale a heapBuild() costa $\Theta(n)$. Poiché lo heap $A[1 \ldots n]$ ha n elementi, l'altezza della radice (il nodo 1) è $O(\log n)$. Ciascuna chiamata a maxHeapRestore() sulla radice richiede pertanto $O(\log n)$ tempo. Essendoci un numero $O(n)$ di tali chiamate, la complessità di heapsort() è $O(n \log n)$.

1.2 Unione di insiemi disgiunti (Merge-Find)

Un'altra struttura che opera sugli insiemi è l'*unione di insiemi disgiunti* (in inglese *merge-find set* o *union-find set*). In tale struttura, un insieme generativo S viene partizionato in una collezione di sottoinsiemi (detti anche componenti o parti), su cui sono ammesse un'operazione di ricerca (che permette di stabilire a quale componente appartiene un generico elemento di S) e un'operazione di unione (che unisce due componenti distinte $X \subseteq S$ e $Y \subseteq S$ in una sola nuova componente $X \cup Y$, distruggendo X e Y e lasciando inalterate tutte le altre componenti).

La diversità rispetto all'usuale tipo di dato insieme consiste nel fatto che non sono previste operazioni di inserimento o di cancellazione; tutti gli elementi vengono inseriti all'inizio, suddivisi in una componente disgiunta per elemento, e le successive operazioni di unione fanno sì che le componenti rimangano comunque disgiunte.

Nel seguito, assumeremo che l'insieme generativo sia costituito dagli interi compresi fra 1 e n; in gran parte delle applicazioni pratiche, questa scelta non riduce la generalità della struttura, in quanto insiemi di dimensione costante n possono essere mappati su vettori della stessa dimensione, e i valori $1 \ldots n$ rappresentano gli indici di tale vettore. Si ottiene quindi la seguente specifica:

MFSET

```
% Crea n componenti {1}, ..., {n}
MFSET Mfset(int n)

% Restituisce il rappresentante della componente contenente x
int find(int x)

% Unisce le componenti che contengono x e y
merge(int x, int y)
```

Mfset(n) inizializza la struttura di dati, creando n componenti, ognuna delle quali contiene uno degli interi fra 1 e n. La funzione find(x) restituisce l'identificatore della componente contenente x; tale funzione, applicata a due elementi della stessa componente, deve restituire lo stesso identificatore; applicata a due elementi contenuti in componenti disgiunte, deve restituire due identificatori diversi. Nel seguito, l'identificatore di una componente sarà dato da uno dei suoi elementi, che prenderà il nome di rappresentante. La procedura merge(x, y) unisce le componenti contenenti gli elementi x e y, se sono disgiunte; non fa nulla, se x e y appartengono già alla stessa componente.

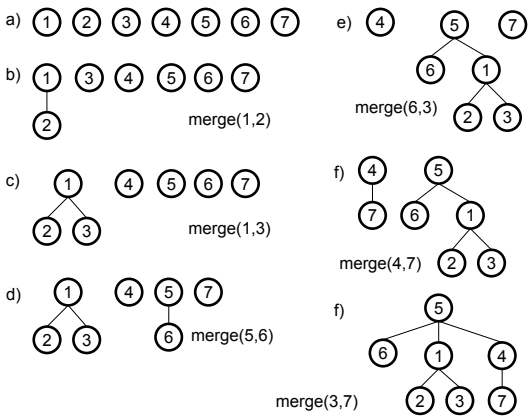

Figura 1.5: Sequenza di fusioni in un mfset.

Esempio (MFSET) Sia $A = \{1,2,3,4\}$. Mfset(4) restituisce $S = \{\ \{1\}, \{2\}, \{3\}, \{4\}\}$. Poiché find$(1) \neq$ find(3), eseguendo merge$(1,3)$ si ottiene $S = \{\{1,3\},\{2\},\{4\}\}$. Adesso, find$(4) \neq$ find(2) ed eseguendo merge$(4,2)$, si ha $S = \{\{1,3\},\{2,4\}\}$. Essendo find$(3) \neq$ find(4), l'esecuzione di merge$(3,4)$ dà per risultato $S = \{\{1,3,2,4\}\}$. D'ora in poi, non sono più possibili ulteriori fusioni. ■

1.2.1 Realizzazione basata su liste

Una realizzazione semplice della struttura MFSET consiste nel creare una lista per ognuno degli insiemi. L'elemento in testa alla lista viene scelto come rappresentante. La procedura Mfset(n) crea n liste, una per ogni elemento. Utilizzando puntatori alla testa e alla coda, merge() può concatenare due liste in tempo $O(1)$. Sfortunatamente, la funzione find() richiede tempo $O(n)$ nel caso peggiore, quindi l'intera lista deve essere scandita per individuare il rappresentante. Questo può essere evitato includendo in ogni cella della lista un puntatore alla testa della lista. A questo punto, tuttavia, è l'operazione merge() a richiedere tempo $O(n)$, in quanto tutti i puntatori della lista che viene concatenata devono essere aggiornati. Sebbene siano possibili alcune ottimizzazioni, è più semplice cambiare approccio e utilizzare una realizzazione basata su foresta.

1.2.2 Realizzazione basata su foresta

Una realizzazione efficiente per la struttura MFSET prevede che gli elementi di ciascuna componente siano ospitati nei nodi di un albero radicato (non necessariamente ordinato) così che la partizione in componenti sia rappresentata da una foresta di alberi radicati. Ciascuna componente è identificata dalla radice dell'albero corrispondente. In questo modo, la funzione find() si riduce a restituire la radice dell'albero, "salendo" attraverso un percorso da un generico nodo fino alla radice; la procedura merge() lavora in modo analogo, risalendo gli alberi fino ad arrivare alla radice e imponendo che una delle due radici divenga un nuovo figlio dell'altra.

Esempio (Realizzazione basata su foresta) Sia dato un MFSET S di 7 componenti, ciascuna contenente il solo intero i, $1 \leq i \leq 7$. S è rappresentato con la foresta di Figura 1.5(a). L'effetto su S dell'esecuzione della sequenza di operazioni merge$(1,2)$, merge$(1,3)$, merge$(5,6)$, merge$(6,3)$, merge$(4,7)$ e merge$(3,7)$ è mostrato nelle Figg.1.5(b)-(g). ■

La foresta sui nodi $1,\ldots,n$ è implementata come un vettore di padri p; quando $p[x] = x$, significa

che x non ha padre e quindi è la radice del sottoalbero. La procedura Mfset( ) inizializza ogni intero in $1,\ldots,n$ come radice del proprio sottoalbero. La funzione find(), dato un intero x, risale da esso fino alla radice che viene restituita come identificativo della componente. La merge() richiama due volte la find(), per identificare i rappresentanti r_x e r_y degli interi x e y; se i rappresentanti sono diversi, procede alla fusione degli insiemi, scrivendo nel campo $p[r_y]$ l'indice r_x e rendendo quindi la (ex) radice r_y figlia della radice r_x.

MFSET

```
int[] p
int[] rank

MFSET Mfset(int n)
    MFSET t = new MFSET
    t.p = int[1...n]
    t.rank = int[1...n]
    for i = 1 to n do
        t.p[i] = i
        t.rank[i] = 0
    return t

int find(int x)
    if p[x] = x then
        return x
    else
        return find(p[x])

merge(int x, int y)
    r_x = find(x)
    r_y = find(y)
    if r_x ≠ r_y then
        p[r_y] = r_x
```

1.2.3 Euristica del rango

Nella realizzazione basata su foresta, la creazione di un MFSET ha costo $O(n)$, che è ovviamente ottimale; mentre il costo di find() e merge() dipende dall'altezza degli alberi. Purtroppo, è possibile scegliere una sequenza particolarmente sfortunata di unioni, che renda l'albero una lista di lunghezza n; in questo caso, le operazioni find() and merge() hanno costo $O(n)$.

Esempio (Alberi di altezza massima) Nella merge(x,y), l'albero contenente y diviene sempre figlio dell'albero contenente x; considerando il MFSET di Figura 1.5(a), la sequenza di operazioni merge$(5,6)$, merge$(4,5)$, merge$(3,4)$, merge$(2,3)$, merge$(1,2)$ produce un albero di altezza massima. ■

La complessità può essere abbassata mantenendo il più possibile ridotto il livello massimo delle foglie di ciascun albero. A tal fine, durante la merge() è opportuno scegliere come radice della nuova componente quella relativa alla componente di altezza maggiore; questa tecnica prende il nome di *euristica sul rango*.

Questa informazione viene mantenuta in un vettore ausiliario *rank*; se r_x è la radice dell'albero contenente x, $rank[r_x]$ contiene l'altezza di tale albero. Nella creazione della struttura, tutte le altezze vengono inizializzate a zero. Nella funzione merge(), se $rank[r_x] > rank[r_y]$, si appende la componente radicata in r_y alla componente radicata in r_x, e l'altezza non cambia; se $rank[r_y] > rank[r_x]$, si opera in maniera simmetrica; infine, se $rank[r_y] = rank[r_x]$, si sceglie arbitrariamente quale albero rendere figlio dell'altro e si aggiunge 1 alla relativa posizione in *rank*.

```
merge(int x, int y)
r_x = find(x)
r_y = find(y)
if r_x ≠ r_y then
    if rank[r_x] > rank[r_y] then
        p[r_y] = r_x
    else if rank[r_y] > rank[r_x] then
        p[r_x] = r_y
    else
        p[r_x] = r_y
        rank[r_y] = rank[r_y] + 1
```

Per giustificare la bontà di questa scelta, cerchiamo di calcolare una limitazione superiore all'altezza dell'albero.

Teorema 1.1 Un albero MFSET con radice r ottenuto tramite euristica sul rango ha almeno $2^{rank[r]}$ nodi.

Dimostrazione. Procediamo per induzione sul numero di operazioni merge() effettuate. All'inizio (nessuna operazione merge()), tutti gli alberi hanno *rango* 0 ed esattamente 1 nodo; poiché $2^0 = 1$, il teorema è dimostrato nel caso base. Consideriamo l'$(n+1)$-esima operazione merge(x,y), e supponiamo che la proprietà sia vera per tutte le operazioni precedenti. Siano X,Y gli insiemi contenenti x e y, con $X \neq Y$; siano $rank[r_x]$ e $rank[r_y]$ il rango delle radici r_x e r_y, rispettivamente; sia $rank[r]$ il rango della radice risultante dalla $(n+1)$-esima operazione merge(). Sono dati tre casi:

- $rank[r_y] < rank[r_x]$: abbiamo che $rank[r] = rank[r_x]$; inoltre, per ipotesi induttiva:

$$|X \cup Y| = |X| + |Y| \geq 2^{rank[r_x]} + 2^{rank[r_y]} > 2^{rank[r_x]} = 2^{rank[r]}$$

  (in quanto $2^{rank[r_y]} > 0$)
- $rank[r_x] < rank[r_y]$: simmetrico del precedente;
- $rank[r_x] = rank[r_y]$: la radice r_y prende r_x come nuovo figlio, e il suo rango viene incrementato di uno: $rank[r] = rank[r_y] + 1$. Dall'ipotesi induttiva, abbiamo che:

$$|X \cup Y| = |X| + |Y| \geq 2^{rank[r_y]} + 2^{rank[r_y]} = 2^{rank[r_y]+1} = 2^{rank[r]}$$

■

Sia quindi r la radice di un albero contenente k nodi; dal teorema, sappiamo che $k \geq 2^{rank[r]}$; applicando il logaritmo ad entrambi i lati della disequazione, otteniamo $rank[r] \leq \log k$, e quindi in generale possiamo dire che la complessità di find() e merge() è $O(\log n)$.

1.2.4 Compressione dei percorsi

Una limitazione superiore $O(\log n)$ potrebbe essere sufficiente, ma è possibile fare ancora meglio. Un'ulteriore euristica, chiamata compressione dei percorsi, consiste nel modificare la find() in modo che alteri l'albero durante la sua esecuzione, così da rendere più vantaggiose le esecuzioni successive di find() e merge(). In pratica, ogni nodo che viene incontrato da find() nel percorso di risalita dal generico nodo x alla radice, viene reso figlio della radice. Per far questo, la chiamata ricorsiva all'interno di find() è modificata introducendo un assegnamento al padre del nodo x, se questo è diverso da x. In tal modo, durante l'esecuzione, restano "sospesi" gli assegnamenti sui padri dei nodi nel percorso di risalita verso la radice. Quando con la chiamata ricorsiva più interna

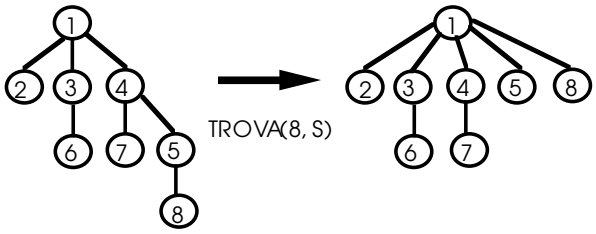

Figura 1.6: Compressione dei percorsi. **TODO**: "trova(8,s)" $\rightarrow$ find(8)

si giunge alla radice, la ricorsione di find() termina restituendo la radice stessa. La radice viene assegnata "a ritroso" ai padri in tutti gli assegnamenti "sospesi", via via che si richiudono tutte le chiamate ricorsive. È importante notare che il numero di operazioni della nuova versione di find() aumenta solo per un fattore costante rispetto alla vecchia versione e che pertanto la sua complessità nel caso pessimo non aumenta.

int find(**int** x)

if $p[x] \neq x$ **then**
  $p[x] = \text{find}(p[x])$
return $p[x]$

Esempio (Compressione dei percorsi) L'effetto dell'esecuzione di find() con la compressione dei percorsi è mostrato nella Figura 1.6. ■

Il vantaggio introdotto dalla compressione dei percorsi si ha quando viene effettuata una sequenza di operazioni formata da m find() e merge(), con $m \geq n$. Senza la compressione dei percorsi, la complessità dovuta all'esecuzione della sequenza sarebbe ovviamente $O(n+m\log n)$, mentre con la compressione dei percorsi è possibile dimostrare che tale costo scende ad $O(n+m\alpha(m,n))$, dove $\alpha(m,n)$ è la funzione inversa della funzione di Ackermann.

$\alpha(m,n)$ è una funzione che cresce lentissimamente, tanto che $\alpha(m,n) \leq 4$ per ogni m ed n rappresentabile con un calcolatore elettronico! Ne segue che ciascuna delle m operazioni find() e merge() della sequenza ha complessità "ammortizzata" $O((n+m\alpha(m,n))/m)$, che è pressoché $O(1)$ per ogni valore "realistico" di n ed m.

1.2.5 Componenti connesse incrementali

Come abbiamo visto nel Capitolo 9.5.5, una visita in profondità è sufficiente per identificare le componenti connesse di un grafo non orientato. Tale algoritmo è ottimale, avendo complessità $O(n+m)$. Supponiamo tuttavia di aggiungere un arco; come cambieranno le componenti? Utilizzando l'algoritmo proposto, l'unica possibilità è far ripartire l'algoritmo, pagando lo stesso costo $O(n+m)$. Supponendo di partire da un grafo senza archi e aggiungendo gli archi uno ad uno, il costo totale sarà $O(m(n+m))$. Questo problema prende il nome di *componenti connesse incrementali*.

Possiamo fare meglio? La struttura di dati MFSET può aiutarci. Si parte da n insiemi disgiunti, uno per nodo. Ogni volta che si aggiunge un arco $[u,v]$, si chiama merge(u,v): se le due componenti associate ad u e v sono disgiunte, vengono unite; altrimenti non succede nulla. Poiché utilizzando la compressione dei percorsi l'operazione di unione ha costo ammortizzato costante e viene ripetuta m volte, e la creazione della struttura di dati ha costo $O(n)$, il tempo totale di questo algoritmo è $O(m+n)$, non diverso dalla visita in profondità. Tuttavia, la struttura MFSET ci permette di gestire

facilmente l'aggiunta di ulteriori archi; per tale motivo, in questa particolare versione di cc() la struttura MFSET viene restituita (come risultato) al chiamante.

```
MFSET cc(GRAPH G)
    MFSET M = Mfset(|G.V|)
    foreach u ∈ G.V() do
        foreach v ∈ G.adj(u) do
            M.merge(u, v)
    return M
```

1.3 Reality check

Code con priorità

La *simulazione* è una branca dell'informatica che cerca di imitare, il più fedelmente possibile, il comportamento dinamico di un sistema o processo reale; esempi di sistemi che possono essere simulati con successo includono gli aerei (nei simulatori di volo), le reti peer-to-peer, ma anche liquidi in un sistema fluidodinamico. Lo scopo della simulazione è generare una storia artificiale del sistema, per permettere lo studio e la valutazione "in vitro" delle caratteristiche del sistema stesso, e rispondere a domande del tipo "cosa succede se"...

Una delle principali tecniche di simulazione, detta *ad eventi* (*event-driven*), richiede che tutte le azioni eseguite nel sistema vengano associate a particolari istanti temporali, detti eventi. Il simulatore non fa altro che eseguire un'azione dopo l'altra, generando nuovi eventi in risposta ad eventi precedenti. Ad esempio, la ricezione di un messaggio in una rete peer-to-peer all'istante t può causare la spedizione di una risposta che verrà ricevuta all'istante $t+\delta$, dove δ è il ritardo di comunicazione.

Una delle principali problematiche per realizzare una simulazione ad eventi è gestire l'insieme di eventi futuri; nuovi eventi possono essere generati (aggiunti) in ogni istante, per poi essere eseguiti nel corretto ordine temporale. È facile vedere che una coda con priorità si presta perfettamente allo scopo, utilizzando l'istante temporale come priorità. Quindi ad ogni passo il simulatore chiama min() e deleteMin() per leggere e rimuovere l'elemento con priorità minore (il prossimo evento da eseguire), e nuovi eventi vengono inseriti con insert().

Insiemi disgiunti

La struttura di dati MFSET è implementata nella Boost Graph Library per risolvere il problema delle componenti connesse incrementali. A sua volta, questa soluzione è utilizzata per realizzare l'algoritmo di Kruskal [cfr. Capitolo 14] per trovare il minimo albero di copertura di un grafo. L'implementazione proposta in questa libreria utilizza un approccio diverso per la creazione dell'insieme disgiunto e per la sua memorizzazione. Invece di utilizzare un vettore di dimensione fissa, permette la creazione di nuovi insiemi con una funzione makeset(a), che crea un nuovo insieme contenente l'oggetto a. Le informazioni relative al padre di un elemento e al suo rango sono memorizzate in dizionari invece che vettori, permettendo quindi la crescita dinamica del numero di elementi.

1.4 Esercizi

Esercizio 1.1 (Heap e ABR)**.** Qual è la differenza fra la proprietà degli ABR e la proprietà *min-heap*? E' possibile utilizzare la seconda per elencare in modo ordinato le chiavi di un albero di n nodi in tempo $O(n)$?

Esercizio 1.2. Si specifichi, passo dopo passo, il contenuto dello *heap* eseguendo la procedura heapsort() per ordinare alfabeticamente gli 11 elementi: O, R, D, I, N, A, M, E, N, T, O. Sorgono difficoltà dalla presenza di elementi con lo stesso valore?

Esercizio 1.3 (k-merging)**.** Si supponga di avere k vettori ordinati, ognuno di m elementi. Si vuole ottenere un vettore ordinato contenente $n = km$ elementi. Per ordinarlo, un possibile algoritmo è il seguente: si fa il merge del primo vettore con il secondo, ottenendo un nuovo vettore di $2m$ elementi; poi si fa il merge del vettore così ottenuto con il terzo, ottenendo $3m$ elementi; e così via.

1. Calcolare la complessità di questo algoritmo
2. Mostrare un algoritmo più efficiente.

Esercizio 1.4. Per chiarezza di esposizione, maxHeapRestore() è una funzione ricorsiva. Nel caso pessimo, l'altezza dell'albero di ricorsione è potenzialmente pari a quello dell'albero heap; lo spazio aggiuntivo richiesto durante l'esecuzione dell'algoritmo Heapsort (oltre cioè a quello richiesto per memorizzare gli n elementi da ordinare) è quindi $O(\log n)$. Scrivere una versione iterativa di maxHeapRestore() che richieda uno spazio aggiuntivo costante.

Esercizio 1.5. Per chiarezza di esposizione, le implementazioni della coda con priorità e di Heapsort viste in questo capitolo sono basate su vettori in cui l'indice del primo elemento è pari ad 1. Ma molti linguaggi di programmazione (Java e C/C++, per esempio) sono basati su vettori il cui primo elemento è 0. Scrivere una coda con priorità e un algoritmo Heapsort per un linguaggio con vettori di questo tipo.

Esercizio 1.6 (Heap d-ario)**.** Si descriva la realizzazione di uno heap d-ario, ovvero uno heap in cui ogni nodo ha al più d figli (uno heap binario ha $d = 2$).

Esercizio 1.7 (Dimensione)**.** Data una radice r, si dimostri che $rank[r]$ contiene l'altezza dell'albero radicato in r, se non si usa la compressione dei percorsi; e una sovrastima di essa, se si usa tale tecnica.

Esercizio 1.8 (Compressione senza ricorsione)**.** Si scriva una versione iterativa della funzione find() con compressione dei percorsi senza usare una pila e senza aumentarne la complessità.

Esercizio 1.9 (Arco minimo della componente)**.** Si progetti un algoritmo che dato un grafo non orientato pesato $G = (V, E)$, restituisca l'insieme delle componenti connesse e l'arco di peso minimo di ogni componente.

1.5 Soluzioni

Soluzione Esercizio 1.1

In un *min-heap*, la chiave di un nodo è più piccola di entrambe le chiavi dei suoi figli. In un albero binario di ricerca, la chiave di un nodo è più grande o uguale alla chiave del suo figlio sinistro, e più piccola o uguale alla chiave del suo figlio destro. Entrambe le proprietà sono ricorsive. E' importante notare che la proprietà di *heap* non ci aiuta a cercare un valore, perché non ci indica che direzione scegliere, visto che i valori sinistro e destro non hanno ordine particolare. Se la proprietà di *heap* ci permettesse di visitare lo *heap* in tempo $O(n)$, visto che la costruzione di uno *heap* richiede tempo $O(n)$ avremmo risolto il problema dell'ordinamento in tempo lineare, cosa contraria alla limitazione inferiore $\Omega(n \log n)$ per l'ordinamento basato su confronti.

Soluzione Esercizio 1.3

La complessità dell'algoritmo proposto è la seguente:

$$T(n) = 2m + 3m + \ldots + km = \sum_{i=2}^{k} im = m \sum_{i=2}^{k} i = O(mk^2) = O(nk)$$

Un algoritmo alternativo, invece, utilizza un *min-heap* binario di dimensione k, inizializzato con i primi valori di tutti i vettori. Viene estratto il valore minimo dallo *heap*, che viene collocato nella prima posizione libera del vettore finale. Si inserisce quindi nello heap il prossimo valore preso dal vettore cui appartaneva il valore minimo appena rimosso (se ne esistono ancora). Il costo totale è quindi $O(n \log k)$. Si è utilizzata la notazione $\langle i, v \rangle$ per operare su una struttura contenente una coppia di elementi e si sono memorizzati i k vettori in una matrice di dimensione $k \times m$.

```
int [ ] merge(int[][] A, int k, int m)
    PRIORITYQUEUE Q = new PriorityQueue(k)        % Utilizzata per estrarre il minimo
    int[] V = new int[km]                                          % Vettore unito
    int[] p = new int[k]                % Posizione attuale di ognuno dei vettori da unire
    for i = 1 to k do
        p[i] = 2
        Q.insert(⟨i,A[i][1]⟩,A[i][1])
    int c = 1                                                % Posizione nel vettore unito
    while not Q.isEmpty() do
        ⟨i, v⟩ = Q.deleteMin()
        V[c] = v
        c = c + 1
        if p[i] ≤ m then
            Q.insert(⟨i,A[i][p[i]]⟩,A[i][p[i]])
            p[i] = p[i] + 1
    return V
```

Soluzione Esercizio 1.6

La realizzazione è molto semplice: invece di funzioni l () e r (), si utilizza una generica funzione $c(i, j) = (i-1) \cdot d + j + 1$, dove i è l'indice del nodo di cui vogliamo conoscere i figli e j è l'indice del figlio, $1 \leq j \leq d$. La funzione $p()$ è ora definita come $\lceil (i-1)/d \rceil$. L'albero heap è ora meno profondo, in quanto la sua altezza è $O(\log_d n)$; ma l'operazione di cancellazione del minimo, basata su minHeapRestore(), richiede una ricerca del minimo fra $d+1$ valori ad ogni iterazione, e quindi costa $O(d \log_d n)$. Se d è una costante, entrambe le operazioni sono comunque $O(\log n)$.

Soluzione Esercizio 1.7

Nel caso in cui non si attui la compressione dei percorsi, ci sono tre casi: quando un insieme viene creato, è costituito da un solo nodo x e il suo rango è inizializzato a 0, che è uguale all'altezza. Quando due alberi di altezze diverse vengono uniti, il più piccolo viene "agganciato" alla radice del più grande, la cui altezza non varia, così come il rango della radice; quando invece vengono uniti due alberi di altezza uguale, l'albero risultante ha l'altezza incrementata di uno, così come il rango della radice. Nel caso sia presente la compressione dei percorsi, il rango non viene toccato, ma ad ogni operazione find() l'altezza può diminuire; viene quindi rispettata la disequazione.

Soluzione Esercizio 1.8

Una versione iterativa della find() con compressione può essere realizzata in due passate, cercando prima la radice r, e poi ripartendo da x e "comprimendo il cammino" di tutti i nodi che si incontrano fra x ed r.

```
int find(int x)
    int r = x                                          % Radice
    while p[r] ≠ r do r = p[r]
    while p[x] ≠ r do
        int temp = p[x]
        p[x] = r                                       % Comprimi il cammino
        x = temp
    return r
```

Soluzione Esercizio 1.9

Diamo uno schema di soluzione, lasciando al lettore i dettagli. Ovviamente la struttura di dati da utilizzare è basata su uno *heap*, ma questa struttura andrà opportunamente specializzata per ospitare l'informazione sull'arco di peso minimo. Tutte le volte che si analizza un arco $[u,v]$, ci sono due casi:

- Se $[u,v]$ unisce due componenti C_1 e C_2 prima sconnesse in una nuova componente C, il peso dell'arco minimo di C sarà dato dal valore minimo fra i pesi degli archi minimi di C_1 e C_2, e il peso di $[u,v]$.
- Se u e v appartengono alla stessa componente C, si verifica se il peso di $[u,v]$ sia minore dell'attuale peso minimo di C, nel qual caso si aggiorna quest'ultimo.