Dynamic Disjoint Sets Report

Alessandro Resta

09 Gennaio 2021

Descrizione della struttura dati

Una struttura dati per Dynamic Disjoint Sets (da ora in poi DDS) è una collezione $\Phi = \{S_1, S_2, ..., S_k\}$ di DDS, ovvero insiemi disgiunti contenenti degli oggetti (elementi) che possono variare mediante le operazioni proprie della struttura. Li chiamiamo oggetti perché, come vedremo più avanti, oltre a contenere la chiave, forniscono informazioni come l'elemento della collezione a cui appartengono. Ciascun insieme è identificato da un rappresentante, ovvero un oggetto che appartiene all'insieme; il rappresentante può essere scelto arbitrariamente, l'unico vincolo da soddisfare è che se abbiamo due operazioni che lo richiedono, e tra le due richieste non viene alterato l'insieme che lo contiene, dobbiamo necessariamente ottenere lo stesso risultato. Questa struttura può essere vista come una partizione di un insieme U, il cui generico elemento è:

$$u \in S_1 \cup S_2 \cup ... \cup S_k$$

dunque una possibile applicazione è quella di determinare le componenti connesse di un grafo; si sa, dalla teoria, che quest'ultime rappresentano una partizione dell'insieme dei vertici, dove due nodi sono in un elemento della partizione se sono mutuamente raggiungibili. Sia infatti

$$G < V, E > |V| = n, |E| = m$$

un grafo. Creando n insiemi, ciascuno contenente un vertice del grafo G, ed eseguendo m operazioni di unione tra insiemi che contengono nodi collegati tra di loro con un arco in G è possibile ottenere il risultato cercato.

Per implementare i DDS si utilizzano principalmente:

- Liste concatenate
- Alberi radicati

1 Approccio basato sulle liste

Ogni lista sarà un elemento della collezione e conterrà gli oggetti chiave che, trattandosi di insiemi disgiunti, non potranno avere duplicati né appartententi alla stessa lista né appartenenti ad altre liste della collezione. Di seguito si riporta la descrizione delle procedure Make-Set, Find-Set e Union.

1.1 Operazione Make-Set(x)

Creerà una lista con un solo nodo (passato come parametro) e l'aggiunge alla collezione.

1.2 Operazione Find-Set(x)

Ricercherà il rappresentante della lista in cui è presente il nodo x. Per convenzione il rappresentante si troverà sempre nella head della lista di x.

1.3 Operazione Union(x,y)

Effettuerà l'unione insiemistica tra due liste, i cui rappresentanti sono x e y. L'unione avverrà "prolungando" la lista di y con la lista di x. Nel caso di unione pesata sarà la lista più corta ad attaccarsi a quella più lunga, in modo da ridurre il numero di operazioni da fare per aggiornare il puntatore a lista di ciascun nodo che sta cambiando lista (si veda il diagramma UML di Figura 1).

2 Complessità nel caso di liste concatenate

2.1 Senza l'euristica unione pesata

- Make-Set(x): O(1)
- Find-Set(x): O(1)
- Union(x, y): O(n) con n lunghezza di x

Apparentemente i tempi di esecuzione sembrano soddisfacenti, però è possibile trovare una sequenza di operazioni che renderà la complessità $\Theta(n^2)$. Consideriamo il seguente esempio

Siano

$$x_1, x_2, ..., x_n$$

n oggetti da inserire nella collezione.

Si eseguano n operazioni Make-Set seguite da n-1 operazioni Union del tipo:

- $Union(x_1, x_2)$
- $Union(x_2, x_3)$
- ...
- $Union(x_{n-1}, x_n)$

Abbiamo dunque $m = 2n - 1$ operazioni in totale, n delle quali sono Make-Set (complessità $\Theta(n)$), e n-1 operazioni Union (complessità $\Theta(n^2)$). Dunque il tempo impiegato per eseguire le m operazioni è $\Theta(n^2)$.

2.2 Con l'euristica unione pesata

Questa strategia sceglie la lista con lunghezza minore per effettuare l'unione

- Make-Set(x): O(1)
- Find-Set(x): O(1)
- Union(x, y): O(n) con n lunghezza di x

Potrebbe sembrare che la complessità non sia cambiata, ma esiste un teorema che afferma: "una sequenza di m operazioni Make-Set, Union, Find-Set, n delle quali sono Make-Set impiega tempo O(m+nlog(n))". Quindi è possibile trovare un limite superiore che, a meno di non soddisfare la disuguaglianza $m < cn^2$ con c costante reale, fa si che la complessità diventi subquadratica rispetto n; questo evidenzia il vantaggio dell' euristica in termini computazionali. E' chiaro che se la disuguaglianza sopra non è rispettata non viene introdotto nessun miglioramento.

3 Approccio basato sugli alberi radicati

Ogni albero sarà un elemento della collezione e conterrà gli oggetti chiave (sottoforma di nodi) che saranno univoci, come per l'approccio basato su liste, nell'insieme di tutti gli oggetti di tutti gli insiemi della collezione. Di seguito viene riportato lo pseudocodice (con l'uso delle euristiche unione per rango e compressione dei cammini) e la descrizione delle procedure Make-Set, Find-Set e Union.

3.1 Operazione Make-Set(x)

Creerà un'albero con un solo nodo x (passato come parametro) e l'aggiunge alla collezione. Il rank di x (indicato con x.rank) viene inizializzato a zero. Il rank di un nodo z è un limite superiore all'altezza del sottoalbero che vede z come radice. Qundi indicando con r il rank di z e con h l'altezza dell'albero di cui z è radice deve valere sempre $h \leq r$. Per indicare il parent di x usiamo la notazione x.p.

MAKE-SET(x)
1. x.p = x
2. x.rank = 0

3.2 Operazione Find-Set(x)

Ricercherà il rappresentante dell'albero in cui è presente il nodo x. In questo caso sarà sempre la radice dell'albero di x.
Con l'euristica della compressione dei cammini tutti i nodi presenti nel cammino che va da x alla radice modificheranno il loro puntatore, che adesso si riferirà

alla radice dell'albero di appartenenza. Questa strategia accorcerà i tempi necessari ad ottenere l'informazione cercata. Si noti che questa procedura utilizza un metodo a doppio passaggio: durante il primo passaggio si risale il cammino dal nodo x fino alla radice e la si restituisce (istruzione 3); durante il secondo passaggio si torna dalla radice al nodo x seguendo lo stesso cammino usato per salire, aggiornando però i puntatori a parent dei vari nodi incontrati (istruzione 2).

```
FIND-SET(x)
1. if x ≠ x.p
2.     x.p = FIND-SET(x.p)
3. return x.p
```

3.3 Operazione Union(x,y)

Effettuerà l'unione insiemistica tra due alberi, i cui rappresentanti sono x e y. Senza uso di euristiche consisterà nel collegare le radici dei due alberi, in modo da formarne uno solo. Usando l'euristica unione per rango la radice che diventerà figlia dell'altra sarà quella con il rank inferiore. In caso di parità di rank si può scegliere arbitrariamente il nodo che diventa figlio tra x e y, e aggiungiamo uno al rank del nodo che ha acquisito il nuovo figlio. L'obiettivo dell'euristica unione per rango è quello di creare un albero che non presenti un altezza (ovvero il numero di archi che vanno dalla radice alla foglia più distante) superiore a quella massima tra i due alberi che si fondono qualora abbiano rank diversi; questo renderà più efficiente la ricerca dei rappresentanti partendo dalle foglie, perché verranno risaliti meno livelli dell'albero.

```
UNION(x, y)
1. x = FIND-SET(x)
2. y = FIND-SET(y)
3. if x.rank > y.rank
4.    y.p = x
5. else x.p = y
6.    if x.rank == y.rank
7.        y.rank = y.rank + 1
```

4 Complessità nel caso di alberi radicati

4.1 Senza l'euristiche unione per rango e compressione dei cammini

- Make-Set(x): O(1)
- Find-Set(x): O(n)

- Union(x, y): O(n) con n dimensione di x

Se ho n operazioni Union e/o Find-Set la complessità diventa O(n^2)

4.2 Con l'euristiche unione per rango e compressione dei cammini

Abbiamo un teorema che afferma: "in una sequenza di m operazioni, n delle quali sono Make-Set il tempo è O(m+α(n))". α è una funzione che cresce molto lentamente, ed è definita in modo che $\alpha(n) \leq 4$ per $0 \leq n \leq A_4(1)$ con $A_4(1)$ molto maggiore di 10^{80}; si può affermare che per qualsiasi applicazione pratica la funzione α vada bene. Il tempo di esecuzione è dunque lineare nel numero di operazioni e il costo ammortizzato di ciascuna operazione è costante, poiché è ottenuto divedendo il costo totale delle operazioni (in questo caso O(m)) per il numero di operazioni (che sono m). Quanto appena detto si può dimostrare con il metodo del potenziale dell'analisi ammortizzata. Maggiori informazioni sono reperibili dalla pagina 477 del libro *Introduzione agli algoritmi e strutture dati di Thomas H. Cormen*.

Diagramma UML delle classi

In Figura 1 è riportato il diagramma UML delle classi, che descrive graficamente la struttura del progetto nella sua parte implementativa. Di seguito si riporta brevemente la spiegazione delle varie classi

4.3 RootedTree

Rappresenta gli alberi radicati. Gli attributi principali sono:

- keyCounter: intero che tiene il conteggio dei nodi dell'albero
- root: puntatore, di tipo TreeNode, alla radice dell'albero
- pos: intero che contiene un indice, valido nell'array di puntatori collection della classe DisjoinSetsForest, in corrispondenza del quale è possibile trovare la posizione dell'albero all'interno della collezione. Serve a velocizzare la ricerca dei vari alberi per gestire efficientemente la loro cancellazione ogni volta che viene eseguita una union che li coinvolge.

I metodi principali sono:

- ins: permette l'inserimento del primo nodo (che diverrà radice). Questo perché i successivi ed eventuali "inserimenti" avverranno tramite la procedura union e non sarà possibile inserire direttamente elementi nell'albero senza passare per le procedure della struttura DDS.
- getKeyCounter, updateKeyCounter: servono ad ottenere e aggiornare il conteggio delle chiavi dell'albero rispettivamente.

- getRoot: permette di ottenere il puntatore alla radice dell'albero.
- setPos, getPos: servono ad aggiornare e ottenere l'attributo pos rispettivamente.

4.4 CircularList

Rappresenta le liste circolari. Di seguito gli attributi principali:

- head, tail: puntatori, di tipo ListNode, alla testa e alla coda della lista circolare rispettivamente.
- pos: come per gli alberi serve a mantenere la posizione della lista all'interno della collezione, presente nella classe DisjointSetsCircularList. Il fine è identico al caso degli alberi, ovvero viene velocizzata l'operazione di cancellazione delle liste che si annettono ad altre mediante operazione union.

I metodi principali sono:

- ins: permette di inserire il primo nodo della lista (che diverrà head). Tutti gli altri, come per gli alberi, vengono introdotti tramite union.
- getHead: serve ad ottenere il valore dell'attributo head.
- getTail, setTail: permettono di ottenere ed aggiornare l'attributo tail.

Tutti gli attributi e metodi non descritti hanno le stesse funzionalità viste nella sez. 4.3, l'unica differenza è che si applicano alle liste.

4.5 Node, ListNode e TreeNode

La classe Node si specializza in due sottoclassi (TreeNode e ListNode) le cui istanze vengono utilizzate nelle strutture RootedTree e CircularList rispettivamente. Dunque conterrà soltanto un attributo key, che contiene il dato del nodo, e i metodi getKey e setKey per ottenere e aggiornare questo dato. Per quanto riguarda ListNode gli attributi principali sono:

- myList: puntatore, di tipo CircularList, alla lista di appartenenza del nodo
- next: puntatore al nodo successivo della lista circolare

I metodi principali sono:

- setMyList, getMyList: vengono usati per aggiornare e ritornare il puntatore myList. L'aggiornamento avviene prevalentemente nelle operazioni di union, quando i nodi di una lista "migrano" nell'altra.
- setNext, getNext: aggiornano e restituiscono l'attributo next.

Nella classe TreeNode gli attributi sono:

- rank: mantiene il rank del nodo (si veda sez. 3.3 per ulteriori info)

- parent: puntatore al nodo parent

I metodi invece sono:

- setParent, getParent: aggiornano e ritornano l'attributo parent.
- setRank, getRank: aggiornano e ritornano il rank del nodo. L'aggiornamento potrà avvenire soltanto nelle operazioni di union, dove l'altezza di un albero può aumentare.

Nel diagramma UML sono presenti ulteriori metodi/attributi per le classi TreeNode e RootedTree. Questi sono parte di un sistema di visita degli alberi radicati n-ari, che consentono di visualizzare la collezione anche nell'implementazione basata sui RootedTree. Queste aggiunte non sono indispensabili per il funzionamento della struttura dati DDS, pertanto non saranno presenti spiegazioni in questa relazione. E' possibile reperire maggiori dettagli nel libro di testo *Introduzione agli algoritmi e strutture dati di Thomas H. Cormen* a partire dalla pagina 203.

4.6 DisjointSetsCircularList e DisjointSetsForest

Queste due classi sono i modelli che fanno riferimento agli approcci analizzati in questa trattazione della struttura DDS, che l'utilizzatore finale potrà usare per creare nuove istanze. Le classi DisjointSetsForest e DisjointSetsCircularList differiscono soltanto per il tipo dell'attributo collection, che è intuitivo pensare faccia riferimento ad una collezione di alberi e liste rispettivamente, e per un metodo aggiuntivo chiamato link presente in DisjointSetsForest, usato dalla union per collegare due radici di due alberi diversi. Tutti i restanti metodi in comune sono l'implementazione di ciò che abbiamo già descritto nelle sezioni precedenti.

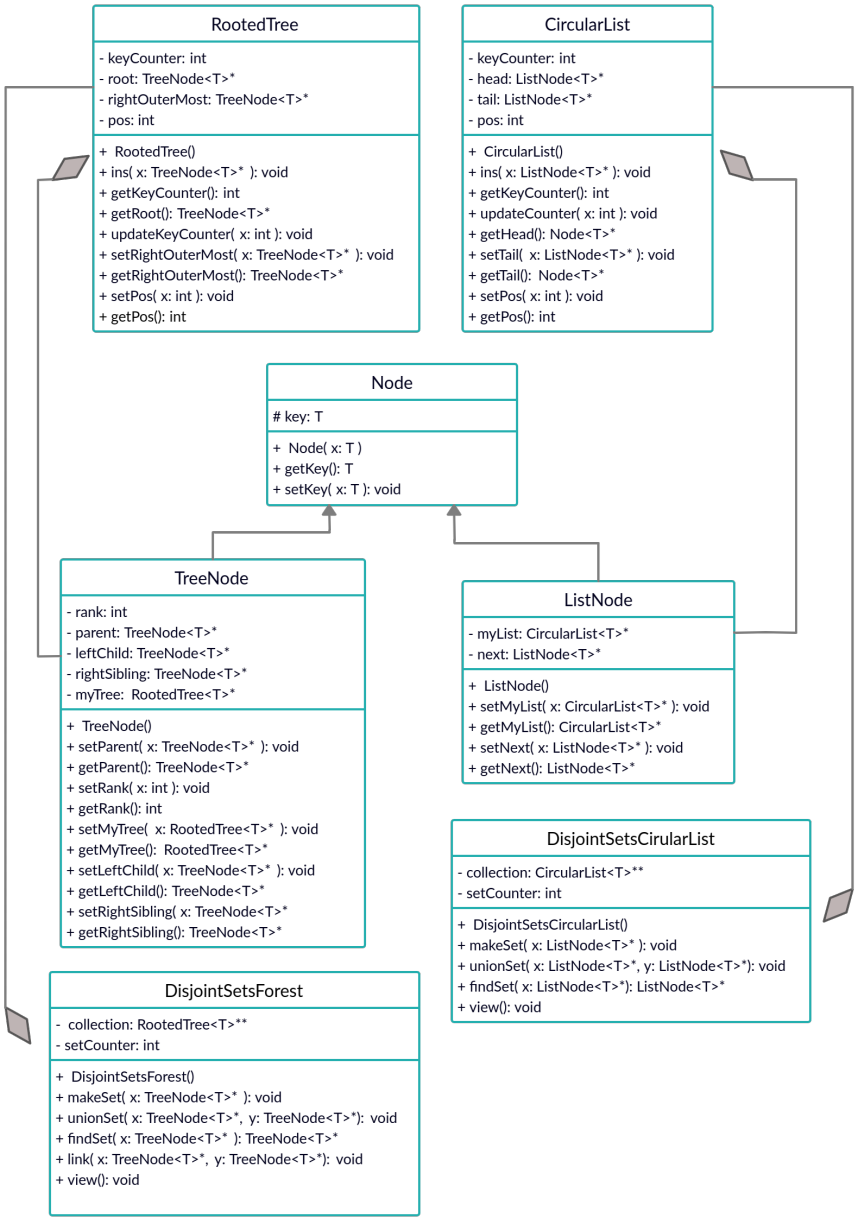

Figure 1: Diagramma UML

Conclusioni

Per concludere questa relazione voglio parlare dell'implementazione utilizzata basata sui nodi. Come si può notare dall'UML i parametri da passare alle funzioni dei DDS sono puntatori a nodi (TreeNode oppure ListNode), questo significa che non sarà possibile lavorare direttamente sulle chiavi memorizzate, perché esse sono inglobate dalla struttura Node. Potrebbe sembrare sconveniente dover usare i nodi ogni volta che si vuol eseguire una qualche operazione sui DDS se, ad esempio, gli elementi degli insiemi della collezione sono semplici numeri o caratteri. Diventa però necessario quando le chiavi sono oggetti, ovvero strutture dati più articolate di semplici tipi primitivi (come int, char, float ecc..); a questo punto soltanto l'utilizzo dei nodi sembra essere la soluzione per far funzionare i DDS anche in questi casi. Risulta dunque una soluzione più complessa dal punto di vista implementativo, ma decisamente più versatile dal punto di vista del loro utilizzo. All'inizio della relazione si è fatto riferimento ad un possibile utilizzo di questa struttura, ovvero trovare le componenti connesse di un grafo; la scelta di usare i nodi permette non solo la popolazione della collezione con le chiavi dei vertici di un grafo, ma addirittura si possono inserire direttamente gli oggetti vertice (se l'insieme dei vertici del grafo è stato pensato come insieme di oggetti).

Miglioramenti futuri

Infine, per rendere ancora più semplice l'uso dei DDS e permettere all'utilizzatore finale di non dover creare i nodi contenitori, è possibile introdurre una HashMap che metta in corrispondeza il contenuto dei nodi con i nodi stessi, e ogni qualvolta viene selezionata una operazione dei DDS la HashMap fornirà il puntatore a Nodo che contiene il valore passato in input al metodo scelto; nel caso medio, dato che le HashMap hanno tempo O(1), non avremo variazioni di complessità.