Dynamic Disjoint Sets Report

Alessandro Resta

09 Gennaio 2021

1 Introduzione ai Dynamic Disjoint Sets

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. Vengono chiamati oggetti perché, come vedremo più avanti, oltre a contenere la chiave forniscono informazioni quali, ad esempio, 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 e può cambiare solo se l'insieme che rappresenta subisce variazioni. 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

La struttura DDS prevede tre operazioni fondamentali:

- Make-Set: inserisce nella collezione un nuovo insieme
- Find-Set: dato un oggetto x restituisce il rappresentante dell'insieme cui x appartiene
- Union: unisce due insiemi della collezione formandone uno solo

2 Implementazione attraverso liste circolari

Le liste concatenate possono essere di vario tipo, di seguito viene illustrata una possibile implementazione dei DDS mediante liste circolari, ovvero liste in cui coda e testa sono collegate tra di loro. Tuttavia altri tipi di liste possono essere scelti al fine di ottenere una diversa realizzazione della struttura.
Ogni lista circolare (da ora in poi semplicemente lista) è un elemento della collezione contenente gli oggetti chiave che, trattandosi di insiemi disgiunti, non potranno avere duplicati né appartententi alla stessa lista né appartenenti ad altre liste della collezione. Le operazioni della struttura DDS in questa implementazione sono riportate di seguito.

Operazione Make-Set(x)

L'operazione Make-Set(x) crea una lista con un solo nodo (passato come parametro) e l'aggiunge alla collezione.

Operazione Find-Set(x)

L'operazione Find-Set(x) ricerca e restituisce il rappresentante della lista in cui è presente il nodo x. Per convenzione poniamo il rappresentante sempre nella head della lista di x.

Operazione Union(x, y)

L'operazione Union(x, y) effettua l'unione insiemistica tra due liste, i cui rappresentanti sono x e y. L'unione avviene "prolungando" la lista di y con la lista di x. Nel caso di utilizzo dell'euristica unione pesata è 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). Quest'ultima strategia migliora il costo computazionale, descritto nel paragrafo 2.2.

2.1 Complessità computazionale in assenza di euristiche

Di seguito si riporta la complessità computazionale delle procedure dei DDS in assenza di euristiche:

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

Esiste una sequenza di operazioni che rende la complessità $\Theta(n^2)$. Si consideri 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:

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

Si hanno 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 Complessità computazionale in presenza dell'euristica unione pesata

La strategia unione pesata, come già anticipato nella descrizione della Union nel caso di liste circolari, consiste nello scegliere la lista con lunghezza minore per effettuare l'unione. La complessità delle singole operazioni rimane inalterata, tuttavia se consideriamo una sequenza di m operazioni, n delle quali sono Make-Set, il tempo di esecuzione è O(m+nlog(n)). La dimostrazione di quanto asserito può essere trovata a [1].

3 Implementazione attraverso alberi radicati

L'implementazione mediante alberi radicati rappresenta la scelta più vantaggiosa in termini di efficienza rispetto a quella basata su liste, perché come si vedrà più avanti il tempo di esecuzione è lineare nel numero di operazioni. Ciò è possibile con l'utilizzo di due euristiche: unione per rango e compressione dei cammini. La prima fa uso del concetto di rank, ovvero un numero associato a ciascun nodo dell'albero che identifica un limite superiore all'altezza (ovvero il numero di archi che vanno dalla radice alla foglia più distante) del sottoalbero radicato in quel nodo. Quindi se chiamo z la radice di un sottoalbero di altezza h e con r indico il rank di z, si avrà sempre $h \leq r$. Le due euristiche unione per rango e compressione dei cammini agiscono sulle operazioni Union e Find-Set rispettivamente, pertanto saranno approfondite successivamente.

Ogni albero è un elemento della collezione contenente gli oggetti chiave (sotto forma di nodi) che sono univoci all'interno della struttura DDS. Di seguito viene riportata la descrizione delle operazioni nel caso di utilizzo degli alberi radicati; in aggiunta è presente lo pseudocodice nel caso di utilizzo delle euristiche sopra menzionate.

Operazione Make-Set(x)

L'operazione Make-Set(x) crea 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 e il parent di x (indicato con x.p) viene posto uguale a x.

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

Operazione Find-Set(x)

L'operazione Find-Set(x) ricerca e restituisce il rappresentante dell'albero in cui è presente il nodo x. In questo caso sarà sempre la radice dell'albero di x.
Utilizzando 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 accorcia i tempi necessari ad ottenere il rappresentante per i nodi coinvolti in una chiamata Find-Set poiché presentano un collegamento diretto con lui. 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 $x \neq x.p$
2. x.p = FIND-SET(x.p)
3. return x.p

Operazione Union(x,y)

L'operazione Union(x,y) effettua 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 diventa figlia dell'altra è quella con il rank inferiore. In caso di parità di rank si può scegliere arbitrariamente il nodo che diventa figlio tra x e y; successivamente si aggiunge uno al rank del nodo che ha acquisito il nuovo figlio per rispettare la proprietà del limite superiore del rank. Per capire come questa strategia introduca un miglioramento si supponga di avere due radici p e q con rank rp e rq rispettivamente tale che $rp < rq$. Collegando la radice q alla radice p nascerà un nuovo albero con altezza $rq + 1$.

Collegando in modo opposto le due radici, ovvero facendo diventare p figlio di q si ha che l'altezza dell'albero ottenuto è al più rq e dunque avendo un albero con altezza inferiore l'operazione Find-Set sarà più efficiente sui nodi foglia,

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
```

3.1 Complessità in assenza di euristiche

- 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)

3.2 Complessità in presenza delle euristiche unione per rango e compressione dei cammini

E' stato dimostrato che se si ha una sequenza di m operazioni, n delle quali sono Make-Set il tempo di esecuzione 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 descritto a [1].

4 Diagramma UML delle classi

Si è scelto di optare per un implementazione che consente all'utente di scegliere quale struttura dati, tra liste circolari e alberi radicati, utilizzare al fine di rappresentare i DDS. Per questo motivo nel diagramma UML di figura 1 è possibile trovare le classi DisjointSetsCircularList e DisjointSetsForest che realizzano la struttura nelle sue due varianti.

Di seguito si riporta brevemente la spiegazione delle altre classi presenti in figura 1:

Node, ListNode e TreeNode

La classe Node si specializza in due sottoclassi (TreeNode e ListNode) che sono di supporto alla struttura DDS e, in particolare, alle due strutture che implementano le liste circolari e gli alberi radicati. Ciascun nodo è un oggetto della collezione, residente in uno specifico insieme della stessa. Dunque contiene un attributo key, che memorizza il dato del nodo, e i metodi getKey e setKey per ottenere e aggiornare quest'ultimo. Per quanto riguarda ListNode gli attributi principali sono:

- myList: è un puntatore, di tipo CircularList (descritta più avanti), alla lista di appartenenza del nodo
- next: è un puntatore al nodo successivo della lista circolare

I metodi principali sono:

- setMyList, getMyList: sono metodi che 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: sono metodi che aggiornano e restituiscono l'attributo next

Nella classe TreeNode gli attributi principali sono:

- rank: è un intero che mantiene il rank del nodo (si veda sez. 3 per ulteriori info)
- parent: è un puntatore al nodo parent

I metodi principali invece sono:

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

RootedTree

La classe RootedTree rappresenta gli alberi radicati. Gli attributi principali sono:

- keyCounter: è un intero che tiene il conteggio dei nodi dell'albero
- root: è un puntatore, di tipo TreeNode, alla radice dell'albero

- pos: è un intero che contiene un indice, valido nell'array di puntatori collection della classe DisjoinSetsForest (descritta in seguito), 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: è un metodo che 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: sono metodi che servono ad ottenere e aggiornare il conteggio delle chiavi dell'albero rispettivamente
- getRoot: è un metodo che permette di ottenere il puntatore alla radice dell'albero
- setPos, getPos: sono metodi che servono ad aggiornare e ottenere l'attributo pos rispettivamente

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 si rimanda il lettore a [2] per ulteriori dettagli

CircularList

La classe CircularList rappresenta le liste circolari. Di seguito gli attributi principali:

- head, tail: sono puntatori, di tipo ListNode, alla testa e alla coda della lista circolare rispettivamente
- pos: è un intero che mantiene la posizione della lista all'interno della collezione, presente nella classe DisjointSetsCircularList (descritta più avanti). Il fine di questo attributo è identico al caso degli alberi radicati, ovvero viene velocizzata l'operazione di cancellazione delle liste che si annettono ad altre mediante operazione Union.

I metodi principali sono:

- ins: è un metodo che permette di inserire il primo nodo della lista (che diverrà head). Tutti gli altri, come per gli alberi, vengono introdotti tramite Union.

- getHead: è un metodo che serve ad ottenere il valore dell'attributo head
- getTail, setTail: è un metodo che permette di ottenere ed aggiornare l'attributo tail

Tutti gli attributi e metodi non descritti hanno le stesse funzionalità viste nella descrizione della classe RootedTree.

DisjointSetsCircularList e DisjointSetsForest

Le classi DisjointSetsCircularList e DisjointSetsForest 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. Queste due classi differiscono soltanto per il tipo dell'attributo collection, che è un puntatore a puntatori a RootedTree per la classe DisjointSetsForest, mentre per la classe DisjointSetsCircularList è un puntatore a puntatori a CircularList. Per i metodi è stata effettuata un'aggiunta alla classe che rappresenta i DDS mediante alberi radicati, ovvero un metodo chiamato link che è di supporto alla procedura Union; questo ha permesso una migliore leggibilità del codice nella parte puramente pratica dell'implementazione dei DDS. Tutti i restanti metodi in comune sono l'implementazione di ciò che abbiamo già descritto nelle sezioni precedenti.

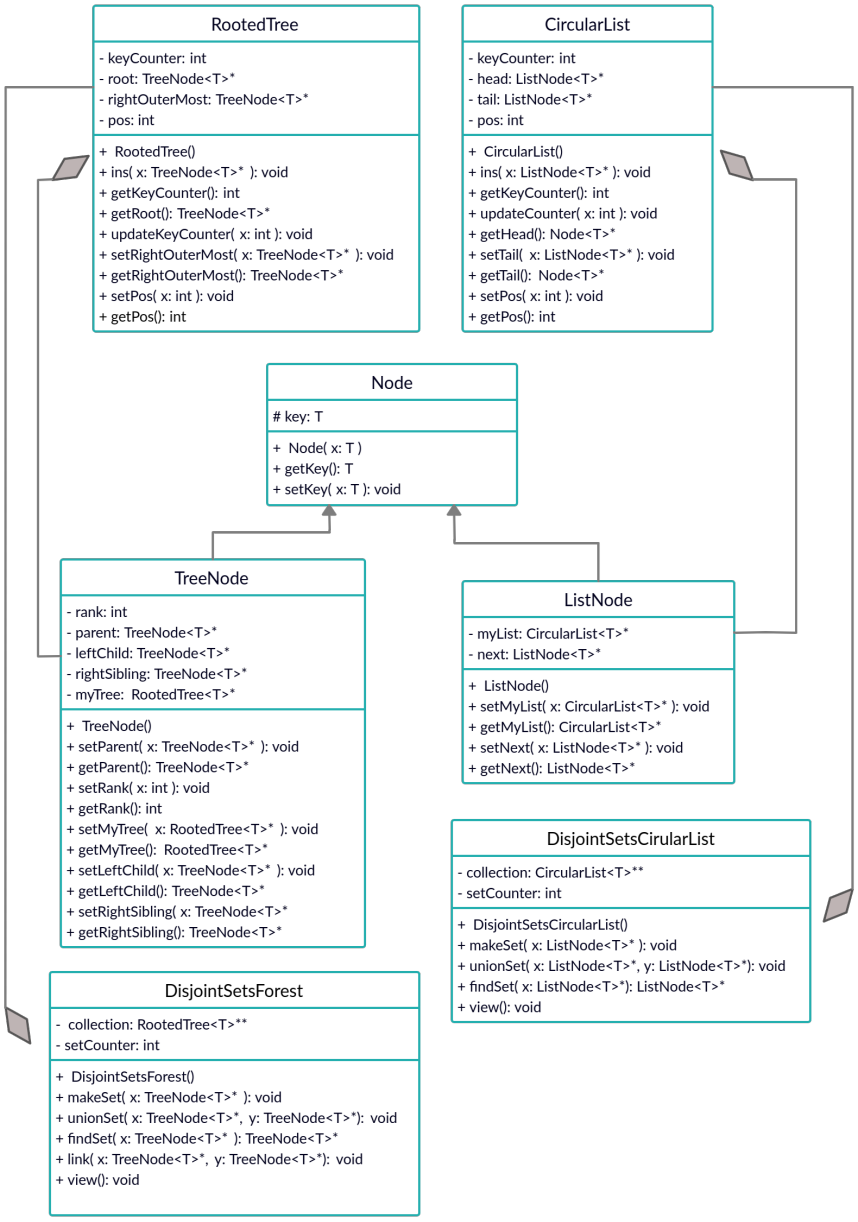

Figure 1: Diagramma UML

5 Conclusioni

Per concludere la trattazione si vuole illustrare l'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 laborioso dover usare i nodi ogni volta 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 anche l'inserimento diretto degli oggetti vertice (se l'insieme dei vertici del grafo è stato pensato come insieme di oggetti).

6 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 si hanno variazioni di complessità. Si è scelto di non introdurre questa struttura di supporto nell'implementazione analizzata perché, come risulta dalla teoria, le HashMap impiegano un tempo costante per la ricerca solo nel caso medio, dunque nel caso peggiore si avrà una variazione della complessità computazionale che rende meno efficiente la struttura DDS.

References

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. Introduction to Algorithms, Third Edition, Chapter 21, pp. 473,477-485

[2] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. Introduction to Algorithms, Third Edition, Chapter 10, pp. 203-204