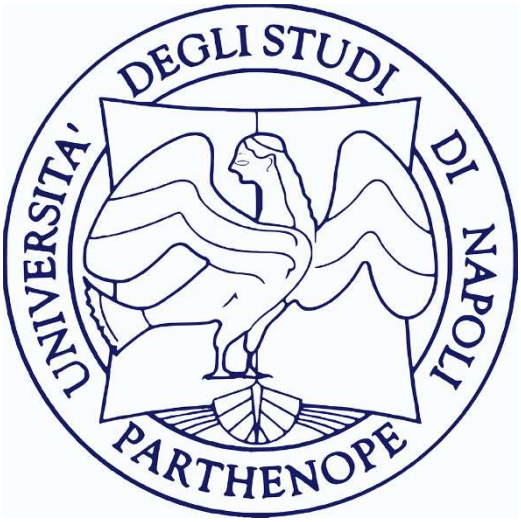


Università degli Studi di Napoli

"Parthenope"

Dipartimento di Scienze e Tecnologie

Corso di Laurea di Informatica

Anno Accademico: 2020/2021

RELAZIONE DEL PROGETTO DI "ALGORITMI E STRUTTURE DATI"

1) Insiemi implementati con Alberi Red Black

2) Percorsi nell'Arcipelago

Studente: **Francesco Calcopietro**

Matricola**: 0124002090**

Professori:

Francesco Camastra

Alessio Ferone

INDICE

Prefazione

In tale relazione verranno trattati:

1)Definizione della struttura Albero Red Black con la conseguente implementazione delle procedure utilizzate per manipolare gli elementi di tale struttura.

2)Definizione di un DAG (Direct Acyclic Graph) con conseguente implementazione di una Breadth First Search modificata per scoprire i cammini massimi partendo da una sorgente singola.

Insiemi implementati con Alberi Red Black

Traccia:

Si vuole realizzare una struttura dati per insiemi generici basata su alberi Red Black. La struttura dati deve consentire di eseguire le operazioni canoniche di unione, intersezione e differenza tra due insiemi. Progettare e implementare una struttura dati basata su alberi Red Black in cui ogni albero rappresenta un insieme e consenta di effettuare le seguenti operazioni: UNION(), INTERSECT() e DIFFERENCE(). Gli elementi degli insiemi (numeri interi) sono memorizzati all'interno di un file di testo. Gli elementi appartenenti ad uno stesso insieme si trovano su una stessa riga separati da uno spazio (righe successive corrispondono ai diversi insiemi). Dotare il programma di un menu da cui sia possibile richiamare le suddette operazioni.

Descrizione del problema:

Come specificato dalla traccia, si deve disporre di un file di testo contenente i valori degli insiemi da considerare. Dalla definizione di questi insiemi numerici, si passa alla costruzione degli Alberi Red Black, i quali conterranno proprio i valori di questi insiemi. E' proprio su questi Alberi che verranno poi effettuate le operazioni richieste: Unione, Intersezione e Differenza. Queste ultime sono le classiche operazioni che vengono effettuate su insiemi numerici:

1. Union: dati due insiemi numerici, e quindi conseguentemente due Alberi Red-Black, si ha l'obiettivo di creare un terzo insieme, e quindi di un Albero Red-Black di unione, che contenga tutti gli elementi distinti del primo e del secondo albero, evitando le duplicazioni dei valori già esistenti.
2. Intersezione: dati due insiemi numerici, e quindi conseguentemente due Alberi Red-Black, si ha l'obiettivo di creare un terzo insieme, e quindi di un Albero Red-Black di intersezione, che contenga tutti gli elementi in comune del primo e del secondo albero.
3. Differenza: dati due insiemi numerici, e quindi conseguentemente due Alberi Red-Black, si ha l'obiettivo di creare un terzo insieme, e quindi di un Albero Red-Black di differenza, che contenga tutti gli elementi del primo albero che non si trovano nel secondo albero.

Prima di parlare dell'implementazione dell'algoritmo, siccome si fa largo utilizzo della struttura Albero Red-Black, è indispensabile effettuare una panoramica sulle

caratteristiche di tale struttura, le sue proprietà e le principali procedure utilizzate per lavorare su di essa.

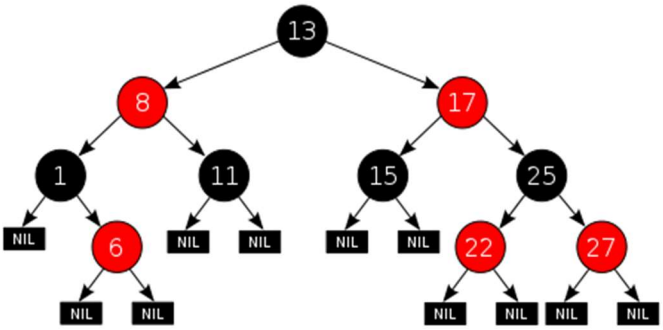
Albero Red-Black

L'albero Red-Black è un particolare Binary Search Tree self balancing, cioè auto bilanciante. Tale struttura, quindi, fa in modo che, a seguito di inserimenti e cancellazioni, l'albero continui ad essere bilanciato a meno di un certo fattore. Siccome l'albero Red-Black eredita da BST, tra le due tipologie di classi c'è un legame **is a.**

Caratteristiche principali

La differenza con i classici BST sta nei nodi. Tali nodi presentano un attributo in più: il **colore**. Oltre al colore, ovviamente sono presenti i principali attributi di un nodo che fa parte di un albero generico:

- puntatore al padre
- puntatore al figlio sinistro
- puntatore al figlio destro
- la chiave
- eventuali dati satelliti associati alla chiave



Proprietà

L'altra differenza con un classico BST sta nel fatto che tale struttura soddisfa 5 proprietà fondamentali:

1. ogni nodo è colorato di nero o di rosso
2. la radice dell'albero è sempre nera
3. i nodi NIL (le foglie) sono neri
4. se un nodo è rosso allora i suoi figli sono neri. Questo significa che non potrà mai avere un nodo rosso come figlio
5. ogni percorso da un nodo interno ad una foglia ha lo stesso numero di nodi neri. Tale quantità prende il nome di **b-altezza**.

Solamente se tutte queste 5 proprietà vengono rispettate, si potrà parlare di Albero Red-Black.

Operazioni su un Albero Red-Black

Le operazioni effettuate su questa struttura si distinguono in:

1. operazioni che possono violare le proprietà dell'intera struttura, come **insert**, **delete, rbTransplant, leftRotate** e **rightRotate**
2. operazioni che raggiungono il loro scopo senza modificare le proprietà della struttura. Esse sono ancor di più classificate in:
    2.1 operazioni di ricerca come **search**, **minimum**, **maximum**, **successor, predecessor**
    2.2 le operazioni di visita **preorder**, **inorder**, **postorder**
    2.3 le operazioni per ripristinare le proprietà della struttura come **fixInsert** e **fixDelete**
    2.4 altre operazioni come la **printTree**

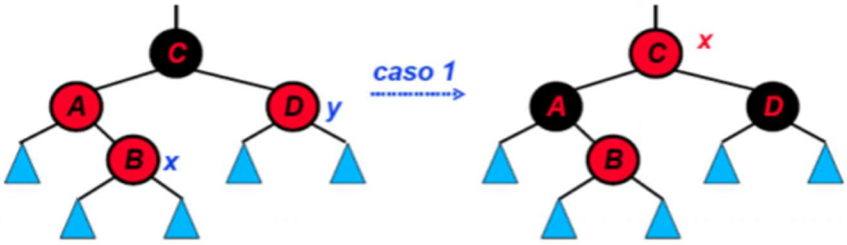
Insert

Prende in input la radice della struttura che si sta costruendo e il nuovo nodo da inserire. Il primo passo è inserire tale nodo in un generico BST, senza tener conto che quest'ultimo è un albero Red-Black. Fatto ciò si colora il nodo di rosso. Si va ad analizzare quale delle 5 proprietà eventualmente si ha violato e si cerca di correggerle. Si dimostra che solo, eventualmente, la proprietà 2 (x è la radice ed è nera) e la proprietà 4 (x e suo padre sono entrambi neri), possono essere state violate. Questo per ogni inserimento che si effettua in tale struttura.
E' proprio per tale motivo che entra in gioco la procedura di fixInsert.

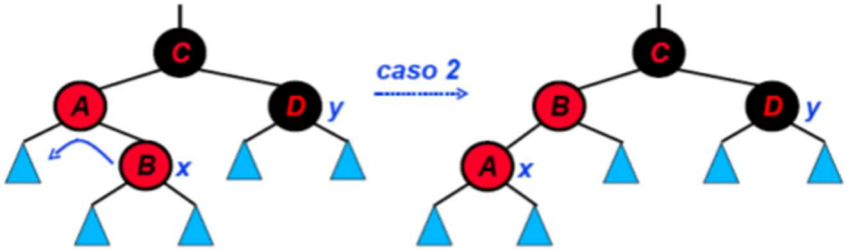
FixInsert

E' la procedura usufruita per ripristinare o, eventualmente, mantenere inviolate, le 5 proprietà della struttura Albero Red-Black, dopo aver effettuato un'operazione di inserimento di un nuovo nodo. Finchè, quindi, sono violate almeno una delle 2 proprietà sopra elencate si esaminano 3 casi:

1. Lo zio di x è rosso.
   Se ci si trova in questo caso la fixInsert colora il padre e lo zio di x di nero, mentre il nonno di x lo colora di rosso e si ripete il ciclo facendo del nonno di x il nuovo x. Si spinge il problema, quindi, verso l'alto.



2. Lo zio di x è nero e x è figlio destro.
   Se ci si trova in questo caso si effettua una rotazione sinistra per trasformare questo caso nel caso 3.



3. Lo zio di x è nero e x è figlio sinistro.
   Se ci si trova in questo caso si devono aggiustare i colori:
   - Il padre di x diventa nero
   - Il nonno di x diventa rosso

   Dopodichè, si effettua una rotazione destra sul nonno di x.

Effettuando questa serie di azioni, il ciclo non verrà più eseguito poiché il padre di x è nero.

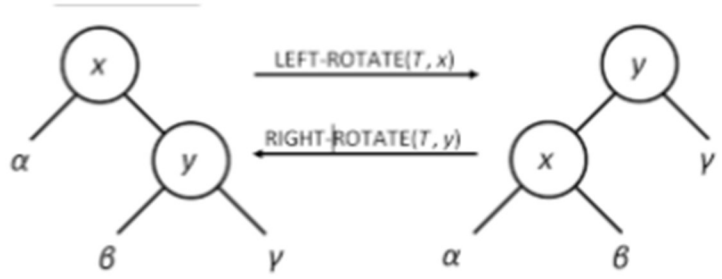
Per quanto riguarda la complessità di tale insieme di operazioni di insert + fixInsert, si ha un O(logn) perché:

1) O(logn) per scendere fino al punto di inserimento
2) O(1) per effettuare l'inserimento
3) O(logn) per risalire e ripristinare le proprietà della struttura. Tale è la complessità della procedura fixInsert

LeftRotate e RightRotate

Dall'analisi che è stata fatta sull'operazione di insert, e, più precisamente, sull'operazione di fixInsert, le procedure essenziali per ripristinare le proprietà della struttura Albero Red-Black sono le rotazioni, rispettivamente a sinistra e a destra. Servono principalmente per ribilanciare l'intero albero.

Per quanto riguarda la leftRotate si passa come parametri di input la radice della struttura e il nodo che vogliamo far ruotare, chiamandolo x. Si fa diventare y il padre di x, il quale di conseguenza alla rotazione diventerà figlio sinistro di y. Dopodichè x avrà come figlio sinistro il sottoalbero ∝, come prima della rotazione e figlio destro il sottoalbero β, il quale era il figlio sinistro di y prima della rotazione. Il nuovo figlio destro di y, il sottoalbero γ, continuerà ad essere il figlio destro di y.
Si nota che, dopo aver fatto questa rotazione, le proprietà dell'albero binario di ricerca continueranno ad essere rispettate.
Per quanto riguarda la procedura rightRotate, essa è l'analogo della leftRotate, ma questa volta la sequenza di azioni è al contrario.
Si avrà il seguente scenario:



Essendo semplicemente un insieme di assegnazioni e confronti, tali procedure presenteranno una complessità pari a O(1).

RbTransplant

Essa è una delle procedure interne a quella di cancellazione di un certo nodo dell'albero Red-Black considerato. Rappresenta una variante della classica Transplant effettuata su un albero BST. L'idea è semplicemente quella di inserire la radice di un sottoalbero al posto della radice di un altro sottoalbero. E' quindi importante andare a controllare se i vari collegamenti tra gli antenati e i nodi correnti siano stati effettuati nel miglior modo possibile. Anche qui, siccome sono presenti solo assegnazioni e confronti, la complessità sarà sempre O(1).

Delete

Essa è la procedura che ci permette di andare ad eliminare un nodo dalla struttura. E' proprio grazie alla presenza di questa procedura che si usufruisce del nodo sentinella TNULL. Si usufruirà, come detto prima, di una variante della procedura di Transplant, la rbTransplant. La procedura delete presenta delle istruzioni in più rispetto alla

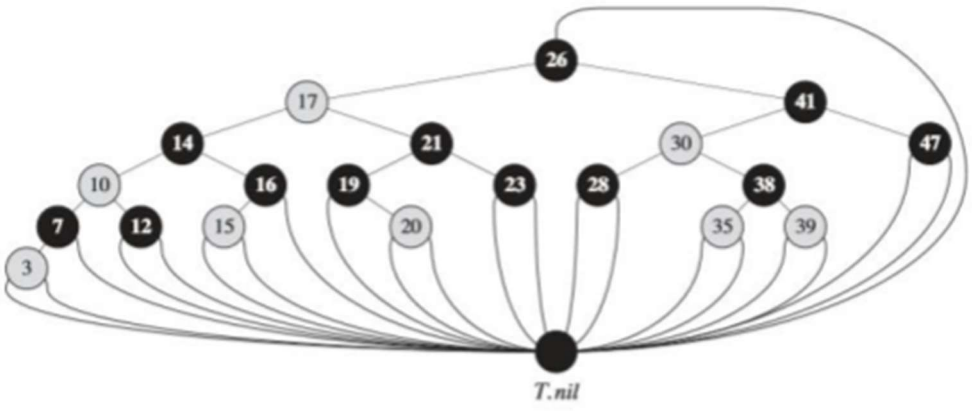
classica treeDelete, applicata su un normale BST, che servono a tenere traccia del colore di un particolare nodo, y. Quando si vuole cancellare un nodo z:

1) Se z ha meno di due figli, y punta a z (che viene rimosso)
2) Se z ha due figli, y punta al successore di z, x, e sostituirà z

In entrambi i casi si deve ricordare il colore di y prima di rimuoverlo o spostarlo e si deve tenere traccia del nodo x che prende il posto di y poichè potrebbe causare delle violazioni. Dopo aver cancellato z, se il nodo y era nero allora è necessario richiamare la procedura **fixDelete** per ripristinare le proprietà dell'albero cambiando i colori di alcuni nodi ed effettuando delle rotazioni.
La procedura di **delete**, insieme alla procedura di **fixDelete** avranno una complessità pari a O(logn).

Utilizzo della struttura e considerazioni

Nell'implementazione dell'algoritmo risolutore dell'esercizio, solamente le operazioni di inserimento e cancellazione saranno effettivamente utilizzate. Le altre operazioni non verranno sfruttate, ma le loro implementazioni sono state lo stesso aggiunte nella definizione della classe RBTree. Inoltre, proprio come è stato spiegato nella sezione riguardante l'operazione di **delete,** le caratteristiche grafiche dell'albero utilizzato saranno così definite:



Sarà, quindi, presente un solo nodo NIL, il quale corrisponderà contemporaneamente al padre della radice e l'unica foglia della struttura. Tale nodo, nell'implementazione dell'esercizio, sarà identificato come nodo TNULL.

Scelte progettuali

Di seguito verrà fatta un'analisi approfondita sulle classi utilizzate, sul loro diagramma e le loro relazioni.

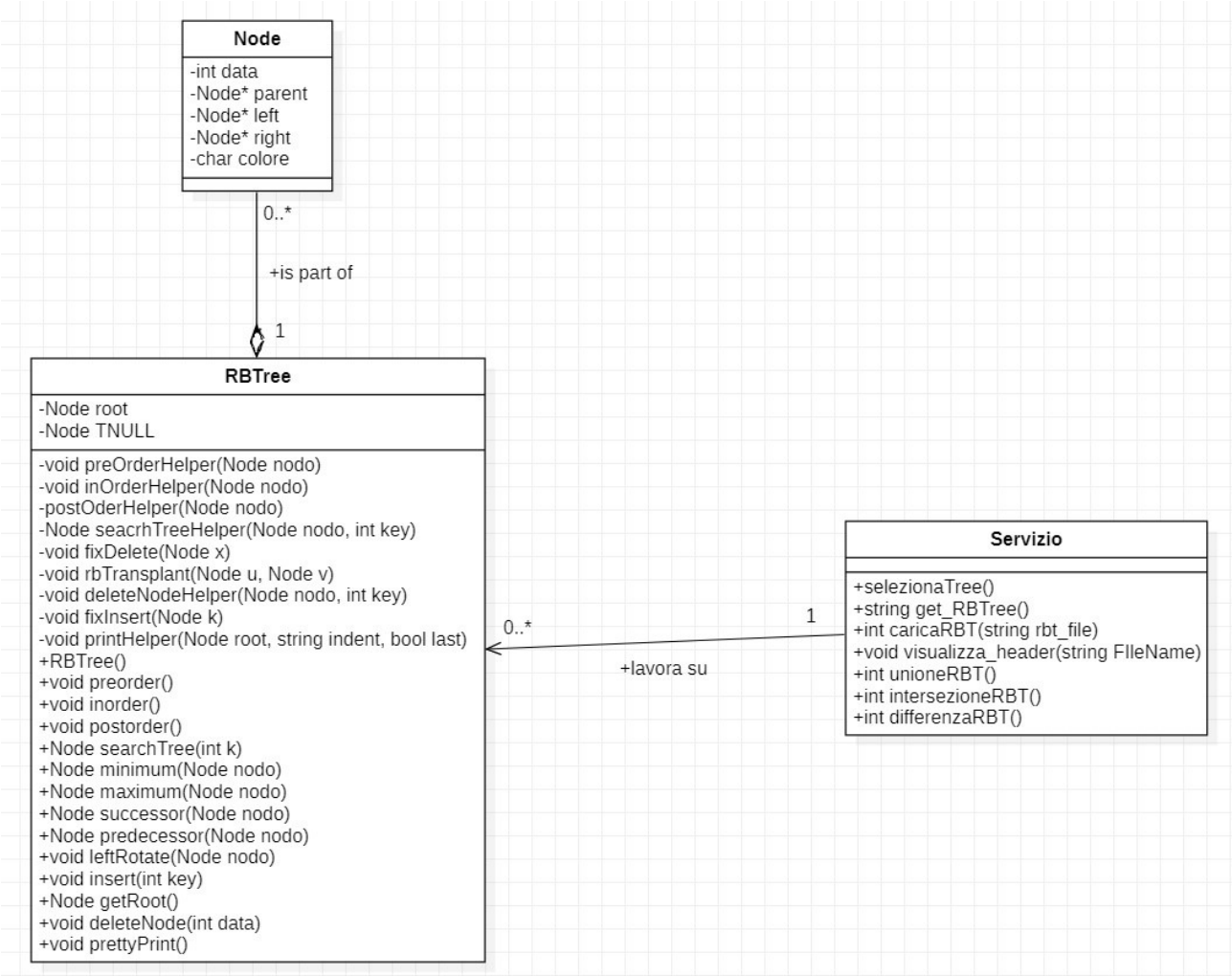
Analisi delle classi

Ci si serve principalmente di 3 classi:

- Classe Node, che identifica la struttura dei nodi presenti all'interno di un Albero Red-Black. Tale classe ha i classici attributi che fanno riferimento alle caratteristiche di un nodo generico: puntatore al padre, figlio sinistro e al figlio destro, chiave e colore. Tutti questi attributi saranno privati. Tale classe non presenta metodi.
- Classe RBTree, che identifica la struttura generica di un Albero Red-Black. Tale classe presenta due soli attributi di tipo Node: root (radice) e il nodo sentinella TNULL. Saranno presenti tutti i metodi che implementano le operazioni principali che possono essere effettuate su tale struttura: insert, delete, search, minimum, maximum, predecessor, successor, rbTransplant, leftRotate, rightRotate, preorder, inorder, preorder, fixInsert, fixDelete, getRoot, printTree.
- Classe Servizio. Essa è la classe che permette la risoluzione del problema. Non presenta nessun attributo. I metodi saranno: selezionaTree, il quale fa scegliere all'utente su quali alberi far effettuare le operazioni di unione, intersezione o differenza. Get_RBTree preleva il file di input nella quale sono presenti gli insiemi numerici. caricaRBT visualizza a schermo gli insiemi numerici presenti all'interno del file di input. Visualizza_header, utilizzato per visualizzare a video l'intestazione dell'output. Infine sono presenti le tre procedure per implementare le 3 operazioni di unione, intersezione e differenza: unioneRBT, intersectRBT, differenzaRBT.

Si vuole quindi, riassumere graficamente quanto detto attraverso un semplice Class Diagramm.

UML Class Diagram



Oltre a quanto è stato già detto nelle sezioni precedenti, ci si vuole soffermare sulle relazioni presenti tra tali classi. Ad esempio, tra la classe Node e la classe RBTree c'è una relazione di aggregazione. Quella dell'aggregazione è una relazione molto forte in quanto un nodo "è una parte di" un RBTree. Si giustificano le due tipologie di molteplicità in quanto in un albero Red-Black possono essere presenti da 0 a N nodi, mentre 0 o N nodi specifici possono far parte solo di un RBTree; un nodo non può esserlo per due RBTree contemporaneamente. L'altra relazione la si trova tra la classe Servizio e la classe RBTree. Si parla, in questo caso di associazione diretta, in quanto la classe Servizio lavora attraverso alberi Red-Black. Anche qui si ha la stessa tipologia di molteplicità: un Servizio può lavorare su 0 a N alberi, ma 0 o N alberi fanno parte di un solo Servizio.

Descrizione dell'algoritmo

Arrivati a questo punto si vuole definire nel dettaglio l'algoritmo risolutore, mettendo assieme tutto quanto è stato detto fino ad ora.

Formato dati in input/output

Come specificato esplicitamente dalla traccia del problema, in input l'utente deve creare un file di testo, nella quale inserire i valori degli insiemi numerici, i quali poi serviranno per la costruzione degli alberi Red-Black. Costruite le varie strutture, l'utente potrà scegliere, in fase di esecuzione, non solo su quali strutture lavorare, ma anche quale operazione effettuare, tra unione, intersezione e differenza. Fatte queste scelte, l'algoritmo eseguirà le varie operazioni e farà visualizzare:

- Una rappresentazione grafica degli alberi scelti in input, con anche il risultato della visita inorder applicata ad essi
- Il risultato dell'operazione scelta
- La rappresentazione grafica dell'albero di output, risultante dall'operazione scelta, e anche il risultato della visita inorder applicata ad esso

Composizione dell'algoritmo

L'algoritmo risolutore fa uso di 5 file in tutto: 2 file header e 3 file cpp:

Red-Black-Tree.hpp

In tale file saranno presenti le definizioni delle due classi Node e RBTree, con i loro attributi e metodi, secondo tutte le caratteristiche già spiegate e definite nella sezione 4.

Servizio.hpp

In tale file sarà presente la definizione della classe Servizio con i suoi metodi, anche in questo caso, secondo la teoria definita nella sezione 4.

Red-Black-Methods.cpp

In tale file saranno presenti le implementazioni dei metodi classici che si possono effettuare su un generico albero Red-Black. Per maggiori informazioni consultare la sezione 3 e la sezione 4.

Servizio-Methods.cpp

Questo file rappresenta il cuore dell'algoritmo risolutore, in quanto implementa le tre operazioni richieste esplicitamente dalla traccia (unione, intersezione e differenza), oltre ad altre operazioni aggiuntive per migliorare lo svolgimento dell'algoritmo. A questo punto è buona norma definire nel dettaglio proprio le 3 operazioni richieste.

UnioneRBT

Tale procedura non prende in input nessun parametro. Implementa la suguente sequenza di passi:

```
unioneRBT()
a<-selezionaTree()              //si sceglie il primo insieme
b<-selezionaTree()              //si sceglie il secondo insieme
for i<-1 to n_node[a]           //costruzione del primo albero Red-Black
        rbt[1].insert(node[a][j])
        RBTreeUnion.insert(node[a][j])
        nUnion<-nUnion+1
stampa a video della struttura del primo albero
rbt[1].inorder()
stampa della visita inorder sul primo albero
for i<-1 to n_node[b]           //costruzione del secondo albero Red-Black
        rbt[2].insert(node[b][j])
        ricerca dell'elmento node[b][j] nel primo albero
        if(node[b][j] non è presente nel primo albero)
                RBTreeUnion.insert(node[b][j])
                nUnion<-nUnion+1
stampa a video della struttura del secondo albero
rbt[2].inorder()
stampa della visita inorder sul secondo albero
stampa a video dell'albero unione appena costruito
RBTreeUnion.inorder()
Stampa a video della visita inorder sull'albero unione
```

Spiegazione del codice

L'idea è quella di, selezionati i due insiemi di input, costruire 2 alberi Red_Black. Si utilizza, per tanto, un array di tipo RBTree, chiamato rbt, il quale contiene 2 alberi Red Black. RBTreeUnion è l'albero che si costruisce facendo l'operazione di union, che contiene gli elementi distinti delle due strutture di input, secondo la definizione di unione. nUnion è la variabile che conta quanti numeri contiene RBTreeUniuon. n_node è un vector globale che contiene la dimensione degli insiemi numerici che si trovano nel file .txt di input. Tale vector è essenziale per poter costruire gli alberi di input. C'è il vector di vector golabe node che invece contiene gli elementi dei vari insiemi numerici caricati prima di svolgere l'operazione di unione. Da notare che, durante la costruzione del secondo albero, viene fatta una ricerca se l'elemento corrente non è presente nel primo albero e quindi se non è già stato inserito nella struttura di unione.

Complessità

Si hanno 2 cicli for con operazioni di insert all'interno. Sapendo che tale operazione ha una complessità O(logn), e viene ripetuta n volte, si ha O(2*n*logn) e quindi O(n*logn). A questa si aggiungono: l'operazione di ricerca, effettuata n volte poiché è in un ciclo, le 3 stampe e le 3 operazioni di inorder. Si ha quindi O(n*logn)+O(3*n) +O(3*n), e quindi O(n*logn). In totale si ha: O(n*logn+n*logn) e quindi O(n*logn).

IntersezioneRBT

Tale procedura non prende in input nessun parametro. Implementa la suguente sequenza di passi:

```
intersezioneRBT()
a<-selezionaTree()              //si sceglie il primo insieme
b<-selezionaTree()              //si sceglie il secondo insieme
for i<-1 to n_node[a]           //costruzione del primo albero Red-Black
        rbt[1].insert(node[a][j])
stampa a video della struttura del primo albero
rbt[1].inorder()
stampa della visita inorder sul primo albero
for i<-1 to n_node[b]           //costruzione del secondo albero Red-Black
        rbt[2].insert(node[b][j])
stampa a video della struttura del secondo albero
rbt[2].inorder()
```

```
stampa della visita inorder sul secondo albero
if(n_node[a]>n_node[b]      //ricerca della grandezza minore tra i 2 insiemi
        c<-b
        d<-0
else
        c<-a
        d<-1
for i<-1 to n_node[c]               //costruzione albero di intersezione
ricerca l'elemento dell'insieme con dimensione più piccola in quello con dimensione più
grande
if(l'elemento ricercato è presente)
        RBTreeInsersect.insert(node[c][j])
        nIntersect<-nIntersect+1
stampa a video dell'albero intersezione appena costruito
RBTreeIntersect.inorder()
Stampa a video della visita inorder sull'albero intersezione
```

Spiegazione del codice

Si costruiscono i 2 alberi Red_Black selezionati. Si utilizza un array di tipo RBTree, chiamato rbt, il quale contiene 2 alberi Red- Black. RBTreeIntersect è l'albero di intersezione che contiene gli elementi in comune delle due strutture di input, secondo la definizione di intersezione. nIntersect conta quanti numeri contiene RBTreeIntersect. n_node è un vector globale che contiene la dimensione degli insiemi numerici che si trovano nel file .txt di input. C'è lil vector di vector node che invece contiene gli elementi dei vari insiemi numerici caricatati prima di svolgere l'operazione di intersezione. Per costruire l'albero di intersezione, inizialmente si vede chi, tra le due strutture di input, ha la dimensione più piccola, in modo tale da definire la grandezza della struttura di intersezione. Scoperto ciò si va ad indicare con la variabile 'd' l'indice di tale struttura nell'array rbt. Dopodichè, si cerca ogni elemento dell'albero con dimensione più piccola nell'albero in posizione 'd' dell'array rbt, in modo tale da trovare tutti gli elementi in comune e inserirli nella terza struttura.

Complessità

Si hanno due cicli for con solo operazioni di insert. Tale operazione ha complessità pari a O(logn), ripetuta per n volte diventa O(n*logn). I due cicli for insieme daranno fuori un O(2*n*logn), ma questo diventa sempre O(n*logn). Il terzo ciclo for presenta

un'operazione di ricerca (O(logn)) e l'operazione di insert (O(logn)), effettuate, nel caso peggiore, n volte. In tutto si ha: O(n*logn+n*logn) => O(2*n*logn) => O(n*logn). A questa si aggiungono le 3 stampe e le 3 procedure di inorder: O(3*n)+O(3*n) => O(n). In totale tale procedura presenterà una complessità pari a: O(n*logn+n) => O(n*logn).

DifferenzaRBT

Tale procedura non prende in input nessun parametro. Implementa la suguente sequenza di passi:

```
differenzaRBT()
a<-selezionaTree()        //si sceglie il primo insieme
b<-selezionaTree()        //si sceglie il secondo insieme
for i<-1 to n_node[b]     //costruzione dell'albero contenente gli elementi del 2° insieme
        rbt[1].insert(node[b][j])
stampa a video del primo albero
rbt[1].inorder()
stampa della visita inorder sul primo albero
for i<-1 to n_node[a]     //costruzione dell'albero contenente gli elementi del 1° insieme
      rbt[2].insert(node[a][j])
      ricerca dell'elemento corrente del primo insieme nel secondo insieme
      if(l'elmento non è stato trovato)
            RBTDifference.insert(node[a][j])
            nDifference<-nDifference+1
stampa a video della struttura del secondo albero
rbt[2].inorder()
stampa della visita inorder sul secondo albero
stampa a video della struttura di differenza
RBTDifference.inorder()
stampa della visita inorder sull'albero di differenza.
```

Spiegazione del codice

L'idea è quella di, selezionati i due insiemi di input, costruire 2 alberi Red_Black. Si utilizza, pertanto, un array di tipo RBTree, chiamato rbt, il quale contiene 2 alberi Red-Black. RBTreeDifference è l'albero che si costruisce facendo l'operazione di differenza, che contiene gli elementi del primo insieme che non si trovano nel secondo insieme.

nDifference è la variabile che conta quanti numeri contiene RBTreeDifference. n_node è un vector globale che contiene la dimensione degli insiemi numerici che si trovano nel file .txt di input. Tale vector è essenziale per poter costruire gli alberi di input. C'è vector di vector globale node che invece contiene gli elementi dei vari insiemi numerici caricatati prima di svolgere l'operazione di differenza. Per costruire l'albero di differenza, si fa la ricerca, all'interno del primo albero, che contiene gli elementi del secondo insieme, di ogni elemento del secondo albero, che invece contiene gli elementi del primo insieme. Se gli elementi non vengono trovati, li si aggiungono all'interno dell'albero di differenza.

Complessità

Si hanno 2 cicli for. Il primo effettua solo l'operazione di insert O(logn), eseguito per n volte diventa O(n*logn). Il secondo effettua un'operazione di ricerca per ogni ciclo, quindi O(n*logn), alla quale poi, si aggiunge un'operazione di insert (O(logn)). Si ha quindi: O(n*logn)+O(n*(logn+logn)) => O(n*logn)+O(2*n*logn) => O(n*logn)+O(n*logn) => O(n*logn). A questa complessità si aggiungono le 3 stampe e le visite inorder effettuate sulle strutture: O(n*logn)+O(3*n)+O(3*n). In totale si ha O(n*logn).

Main.cpp

Tale file contiene la definizione di un semplice menù nella quale, a scelta dell'utente, vengono effettuate una delle 3 operazioni richieste dalla traccia, fino a quando non si sceglie di concludere il programma.

Test/risultati

In questa ultima sezione, si definisce cosa verrà visualizzato all'atto di esecuzione del programma. Per prima cosa, verrà visualizzato il contenuto del file MainTest.txt, cioè il file di input contenente i valori numerici degli insiemi.

```
Digitare la parola 'MainTest' per visualizzarne il contenuto =MainTest

Il file selezionato e' composto dai seguenti insiemi:
Insieme n.1 avente 20 elementi:
74 49 57 24 45 92 52 99 22 96 98 61 35 32 85 4 75 76 66 8
Insieme n.2 avente 12 elementi:
34 68 14 70 11 63 47 73 66 83 81 30
Insieme n.3 avente 17 elementi:
20 4 73 49 47 86 58 94 80 74 52 17 22 78 3 42 15
```

Fatto ciò, si sceglie quali strutture utilizzare e quali operazioni effettuare.

```
--------------Implementazione--------------------
Digitare:
1 per eseguire l'operazione di union
2 per eseguire l'operazione di intersezione
3 per eseguire l'operazione di differenza
4 per uscire e terminare il programma
1

E' stata scelta l'operazione di union
Digitare 1 per selezionare il primo insieme, 2 per selezionare il secondo insieme, 3 per selezionare il terzo insieme

Seleziona il numero del primo insieme:1
Seleziona il numero del secondo insieme:2
```

Fatte le scelte, si visualizzeranno i risultati già descritti nella sezione 5.1.

Di seguito l'implementazione della Union:

```
Albero Red-Black 1 composto da 20 nodi
R----57(BLACK)
     L----45(BLACK)
     |    L----24(RED)
     |    |    L----8(BLACK)
     |    |    |    L----4(RED)
     |    |    |    R----22(RED)
     |    |    R----35(BLACK)
     |    |         L----32(RED)
     |    R----49(BLACK)
     |         R----52(RED)
     R----92(BLACK)
          L----74(RED)
          |    L----61(BLACK)
          |    |    R----66(RED)
          |    R----76(BLACK)
          |         L----75(RED)
          |         R----85(RED)
          R----98(BLACK)
               L----96(RED)
               R----99(RED)
Visita inorder del primo albero:
4 8 22 24 32 35 45 49 52 57 61 66 74 75 76 85 92 96 98 99
Albero Red-Black 2 composto da 12 nodi
R----68(BLACK)
     L----34(RED)
     |    L----14(BLACK)
     |    |    L----11(RED)
     |    |    R----30(RED)
     |    R----63(BLACK)
     |         L----47(RED)
     |         R----66(RED)
     R----73(RED)
          L----70(BLACK)
          R----83(BLACK)
               L----81(RED)
Visita inorder del secondo albero:
11 14 30 34 47 63 66 68 70 73 81 83
ALbero Red-Black di unione composto da 31 elementi
R----57(BLACK)
     L----24(RED)
     |    L----8(BLACK)
     |    |    L----4(BLACK)
     |    |    R----14(BLACK)
     |    |         L----11(RED)
     |    |         R----22(RED)
     |    R----45(BLACK)
     |         L----34(RED)
     |         |    L----32(BLACK)
     |         |    |    L----30(RED)
     |         |    R----35(BLACK)
     |         R----49(BLACK)
     |              L----47(RED)
     |              R----52(RED)
     R----74(RED)
          L----66(BLACK)
          |    L----61(BLACK)
          |    |    R----63(RED)
          |    R----70(BLACK)
          |         L----68(RED)
          |         R----73(RED)
          R----92(BLACK)
               L----76(RED)
               |    L----75(BLACK)
               |    R----83(BLACK)
               |         L----81(RED)
               |         R----85(RED)
               R----98(BLACK)
                    L----96(RED)
                    R----99(RED)
Visita inorder dell'albero di unione:
4 8 11 14 22 24 30 32 34 35 45 47 49 52 57 61 63 66 68 70 73 74 75 76 81 83 85 92 96 98 99
```

Di seguito l'implementazione della Intersezione:

```
Digitare:
1 per eseguire l'operazione di union
2 per eseguire l'operazione di intersezione
3 per eseguire l'operazione di differenza
4 per uscire e terminare il programma
2

E' stata scelta l'operazione di intersezione
Digitare 1 per selezionare il primo insieme, 2 per selezionare il secondo insieme, 3 per selezionare il terzo insieme

Seleziona il numero del primo insieme:1
Seleziona il numero del secondo insieme:2

Albero Red-Black 1 composto da 20 nodi
R----57(BLACK)
     L----45(BLACK)
     |    L----24(RED)
     |    |    L----8(BLACK)
     |    |    |    L----4(RED)
     |    |    |    R----22(RED)
     |    |    R----35(BLACK)
     |    |         L----32(RED)
     |    R----49(BLACK)
     |         R----52(RED)
     R----92(BLACK)
          L----74(RED)
          |    L----61(BLACK)
          |    |    R----66(RED)
          |    R----76(BLACK)
          |         L----75(RED)
          |         R----85(RED)
          R----98(BLACK)
               L----96(RED)
               R----99(RED)
Visita inorder del primo albero:
4 8 22 24 32 35 45 49 52 57 61 66 74 75 76 85 92 96 98 99
Albero Red-Black 2 composto da 12 nodi
R----68(BLACK)
     L----34(RED)
     |    L----14(BLACK)
     |    |    L----11(RED)
     |    |    R----30(RED)
     |    R----63(BLACK)
     |         L----47(RED)
     |         R----66(RED)
     R----73(RED)
          L----70(BLACK)
          R----83(BLACK)
               L----81(RED)
Visita inorder del secondo albero:
11 14 30 34 47 63 66 68 70 73 81 83
Albero Red-Black di intersezione composto da 1 elementi
R----66(BLACK)
Visita inorder dell'albero di intersezione:
66
```

Di seguito l'implementazione della differenza:

```
Digitare:
1 per eseguire l'operazione di union
2 per eseguire l'operazione di intersezione
3 per eseguire l'operazione di differenza
4 per uscire e terminare il programma
3

E' stata scelta l'operazione di differenza
Digitare 1 per selezionare il primo insieme, 2 per selezionare il secondo insieme, 3 per selezionare il terzo insieme

Seleziona il numero del primo insieme:1
Seleziona il numero del secondo insieme:2

Albero Red-Black 1 composto da 20 nodi
R----57(BLACK)
     L----45(BLACK)
     |    L----24(RED)
     |    |    L----8(BLACK)
     |    |    |    L----4(RED)
     |    |    |    R----22(RED)
     |    |    R----35(BLACK)
     |    |         L----32(RED)
     |    R----49(BLACK)
     |         R----52(RED)
     R----92(BLACK)
          L----74(RED)
          |    L----61(BLACK)
          |    |    R----66(RED)
          |    R----76(BLACK)
          |         L----75(RED)
          |         R----85(RED)
          R----98(BLACK)
               L----96(RED)
               R----99(RED)
Visita inorder del primo albero:
4 8 22 24 32 35 45 49 52 57 61 66 74 75 76 85 92 96 98 99
Albero Red-Black 2 composto da 12 nodi
R----68(BLACK)
     L----34(RED)
     |    L----14(BLACK)
     |    |    L----11(RED)
     |    |    R----30(RED)
     |    R----63(BLACK)
     |         L----47(RED)
     |         R----66(RED)
     R----73(RED)
          L----70(BLACK)
          R----83(BLACK)
               L----81(RED)
Visita inorder del secondo albero:
11 14 30 34 47 63 66 68 70 73 81 83
Albero Red-Black di differenza composto da 11 elementi
R----68(BLACK)
     L----34(RED)
     |    L----14(BLACK)
     |    |    L----11(RED)
     |    |    R----30(RED)
     |    R----63(BLACK)
     |         L----47(RED)
     R----73(RED)
          L----70(BLACK)
          R----83(BLACK)
               L----81(RED)
Visita inorder dell'albero di differenza:
11 14 30 34 47 63 68 70 73 81 83
```

Percorsi nell'Arcipelago

Traccia:

Dopo il terribile terremoto del Dicembre 2019, a seguito di ingenti investimenti economici, la regina dell'arcipelago di Grapha-Nui e riuscita a ripristinare i collegamenti tra le isole. E ora necessario incentivare il turismo nell'arcipelago per rimpinguare le casse del governo ed a tal fine il Primo Ministro vuole sapere quali collegamenti tra le isole pubblicizzare maggiormente. Viene nuovamente convocata la famosa consulente informatica Ros Walker con il compito di calcolare i percorsi che massimizzano la soddisfazione dei turisti, partendo da una generica isola verso tutte le altre isole. Ros ha a disposizione la piantina dell'arcipelago con la rete di collegamenti tra le isole. Per ogni collegamento la piantina specifica la direzione ed un valore (anche negativo) che misura la qualità del collegamento. I collegamenti tra le isole non danno origine a cicli.

Descrizione del problema

Leggendo la traccia e tralasciando la sua parte metaforica, le isole che vengono menzionate rappresentano i nodi di un unico grande grafo. Tale grafo è diretto (lo si capisce dal fatto che, nel momento in cui si parla della piantina viene specificato che c'è una direzione nel collegamento tra le isole) e pesato (lo si capisce dal fatto che ogni collegamento ha una qualità). Infine tale grafo è anche aciclico, proprio come dice l'ultima frase della traccia. Si sta quindi trattando di un DAG (Direct Acyclic Graph). Lo scopo della traccia è quello di trovare i cammini massimi partendo da sorgente singola, metaforicamente parlando quindi "calcolare i percorsi che massimizzano la soddisfazione dei turisti, partendo da una generica isola verso tutte le altre isole". Come è stato fatto per l'esercizio precedente, siccome si lavora ampiamente su una struttura dati, prima di parlare dell'algoritmo risolutore, si vuole effettuare una panoramica sulle caratteristiche, prima di un grafo generico, e poi in particolare sulla struttura dati DAG.

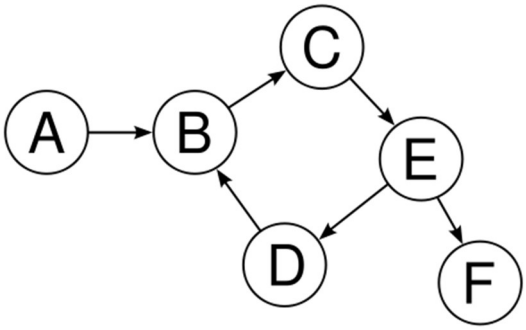
La struttura dati Graph

Esso è una struttura dati composta da una coppia di insiemi: V che indica un insieme di vertici o nodi, E che indica un insieme di archi o collegamenti tra tali nodi. E' rappresentato proprio in questo modo: G=(V,E).

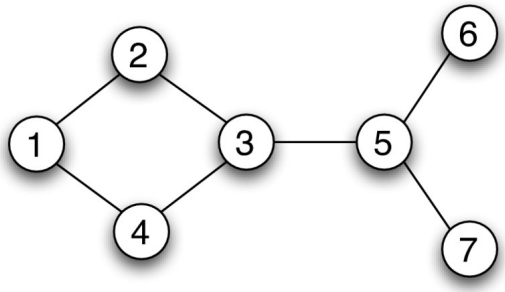
Classificazione

E' proprio sulle caratteristiche degli archi che si distinguono:

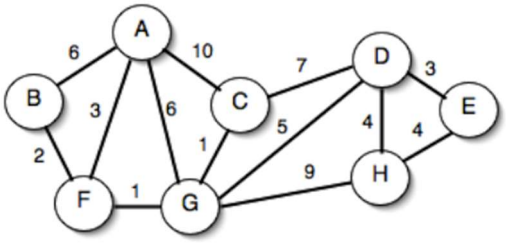
- Grafi diretti o orientati: insieme di vertici e archi nella quale gli archi sono una coppia ordinata (v,w) di nodi. Questo fa capire che il collegamento tra due nodi è effettuato tramite freccia perché c'è un'orientazione



- Grafi indiretti o non orientati: insieme di nodi o vertici e archi nella quale gli archi sono delle coppie non ordinate (v,w) di nodi, cioè non c'è un'orientazione. La differenza, quindi, è che qui si può passare dal nodo v al nodo w e viceversa; nei Directed Graph bisogna sottostare alla direzione dell'arco



- Grafi pesati: tipologia di grafo che può essere sia diretto che indiretto. La differenza sta nel fatto che esiste una funzione peso w:E->R, la quale associa ad ogni arco un valore, chiamato proprio peso. Tale grafo, come anche quelli precedenti, possono generare dei cicli.



Nomenclatura

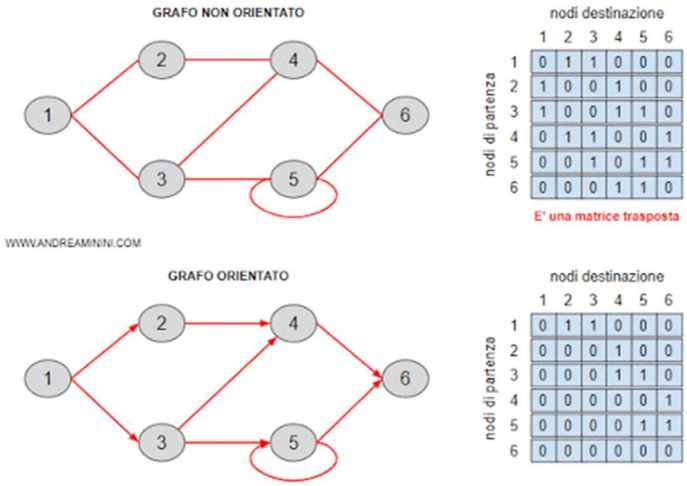
Definite le varie tipologie di grafo si hanno le seguenti definizioni:

1. Se v e w sono nodi che appartengono a V e sono collegati da un arco (v,w) che appartiene a E allora si dice che i nodi sono **adiacenti**
2. **Cammino di un grafo**: sequenza di nodi v0, v1, v2, ..., vn-1 tali che per ogni coppia di tali nodi (v0, v1), (v1, v2) ... c'è un arco appartenente a E.
3. **Lunghezza di un cammino**: è intesa o come il numero di archi del cammino oppure il numero di nodi del cammino -1.
4. **Cammino semplice**: cammino in cui tutti i nodi, eccetto eventualmente il primo e l'ultimo, sono distinti.

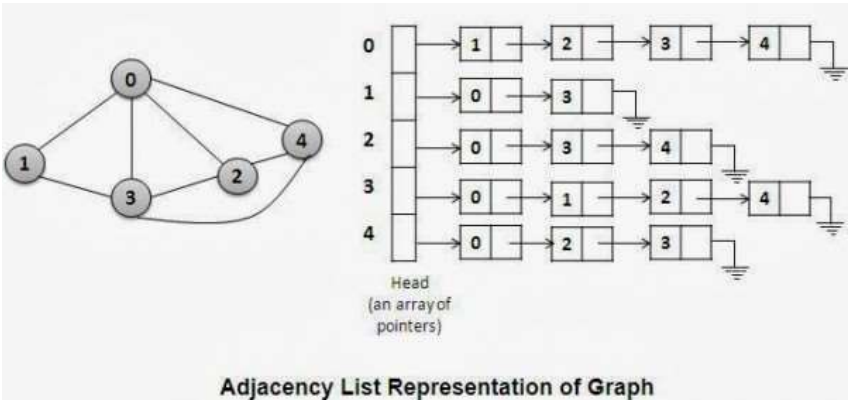
Metodi di rappresentazione

Esistono due tipologie di rappresentazione di un grafo generico:

1. Matrice di adiacenze: si sfrutta una matrice A di dimensione n x n, nella quale n indica il numero totale di nodi appartenenti al grafo. In ogni posizione Aij si troveranno o valori booleani o semplicemente 0 o 1, a seconda se per ogni coppia di nodi (i,j) esiste un arco o no. Si osserva che la matrice di un grafo non orientato è simmetrica o trasposta.



2. Lista di adiacenze: in generale si ha una struttura dati lista che dice quali nodi sono adiacenti al nodo corrente.



Adjacency List Representation of Graph

Operazioni su un grafo

Come ogni struttura dati che ha lo scopo di immagazzinare una quantità definita di dati, anche per i grafi le operazioni principali sono: inserimento, cancellazione, ricerca e algoritmi secondari che si sfruttano per migliorare le perfomance di un generico algoritmo. Tra le operazioni su cui però, ci si vuole soffermare di più, sono gli **algoritmi di visita.**

Si dice visita di un grafo un insieme di cammini con origine in uno stesso vertice. Si intende quindi l'attraversamento di un grafo per intero sfruttando un criterio specifico. Si distinguono due tipologie di visita.

BFS – Breadth First Search

Sta per algoritmo di visita in ampiezza. Dato un grafo G=(V,E) e un vertice sorgente s, tale algoritmo visita in ampiezza sistematicamente tutti i nodi che sono raggiungibili da s.

- Viene calcolata la **distanza** (ossia il minimo numero di archi) da s a ciascun vertice raggiungibile
- Viene generato un **albero BF** con radice s che contiene tutti i vertici raggiungibili

L'osservazione che viene fatta è che per ogni vertice raggiungibile dalla sorgente s, **il cammino nell'albero BF** corrisponde a un **cammino minimo** da s a v nel grafo considerato.

Tale algoritmo lavora sia sui grafi orientati che non.

Procedimento

La visita in ampiezza costruisce un albero BF che inizialmente è costituito dalla sorgente s che è la radice.
Se un nodo v viene scoperto, durante l'ispezione della lista di adiacenza del nodo u (già scoperto), vengono aggiunti a BF il nodo v e l'arco (u,v). Il vertice u è il predecessore (o padre) di v nell'albero BF.
Un vertice può avere un solo padre. Se u è lungo il cammino che va dalla sorgente s al nodo v, allora u è un antenato di v e v è un discendente di u.
Tale algoritmo scopre tutti i vertici che si trovano a distanza k dalla sorgente s prima di quelli a distanza k+1.
L'algoritmo colora ogni vertice di bianco, di grigio o di nero.
All'inizio tutti i vertici sono colorati di **bianco**.
Un vertice viene scoperto quando viene incontrato per la prima volta e cessa di essere bianco.
Un vertice è **grigio** se tra i suoi nodi adiacenti ci sono vertici bianchi (vertici non ancora scoperti).
Un vertice è **nero** se tutti i vertici adiacenti sono stati scoperti.
La complessità di tale algoritmo, in generale, è O(V+E), perché:

1. O(V) per l'operazione di enqueue di ogni nodo all'interna della coda utilizzata dall'algoritmo
2. θ(E) per esaminare tutte le liste di adiacenze

Nel caso peggiore, cioè quando il grafo non è sparso, ma full connected, si ha:
|E|=V x V => O(V+V^2) =>O(V^2)

DFS – Depth First Search

Chiamata anche visita in profondità. Tale strategia consiste nell'esplorare il grafo andando ad ogni istante il più possibile in profondità. Nella DFS gli archi vengono scoperti a partire dall'ultimo vertice scoperto che abbia ancora vertici non scoperti uscenti da esso.
Quando tutti gli archi uscenti da v sono scoperti, la visita torna indietro per esplorare gli archi uscenti dal vertice dal quale v era stato scoperto. Il processo continua fino a scoprire tutti i vertici che sono raggiungibili dal vertice sorgente originario.

Se rimane qualche vertice non ancora scoperto, viene selezionato uno di essi e diventa la nuova sorgente e la ricerca riparte da essa. L'intero processo viene ripetuto sino a quando tutti i vertici del grafo non sono stati scoperti.
Come nella BFS quando un vertice v viene scoperto, durante l'ispezione della lista di adiacenza di un vertice u già scoperto, viene fatta l'assegnazione p[v] = u.
A differenza della BFS dove il sottografo dei vertici scoperti è un albero, nel caso della DFS può essere una **foresta DFS**, perché la visita può essere ripetuta da più sorgenti.
Il sottografo dei predecessori in una visita di profondità è Gp = (V, Ep) dove Ep ={(u,v) : v ∈ V and p[v] !=NULL}.
Il sottografo forma una **foresta DF** composta da vari **alberi DF**

Procedimento

Inizialmente tutti i nodi sono colorati di bianco.
I nodi sono bianchi quando non sono stati scoperti, grigi quando vengono scoperti, neri quando la visita è finita (ossia la lista di adiacenza è stata completamente ispezionata).
Questa tecnica assicura che ogni vertice vada a finire in un solo albero DF e quindi che tutti gli alberi DF siano disgiunti.
La DFS associa ad ogni vertice v due etichette d[v] e f[v]:

- d[v] indica il tempo in cui è stato scoperto
- f[v] quando è terminata la visita

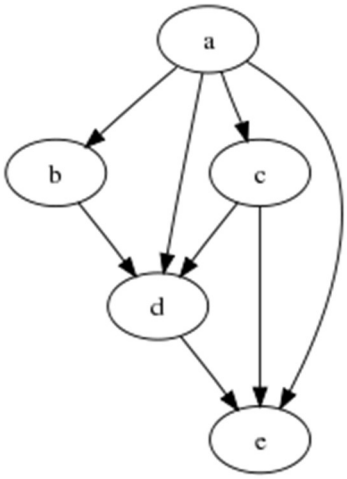
Il vertice è bianco prima del tempo d[v], grigio quando ha un tempo compreso tra d[v] e f[v] e nero per un termine superiore a f[v].
Anche qui la complessità è O(V+E) perché:

1. O(V) per il primo ciclo for di inizializzazione che viene effettuato sul grafo
2. θ(E) per esaminare tutte le liste di adiacenze

Nel caso peggiore, cioè quando il grafo non è sparso, ma full connected, si ha:
|E|=V x V => O(V+V^2) =>O(V^2).

La struttura dati DAG (Direct Acyclic Graph)

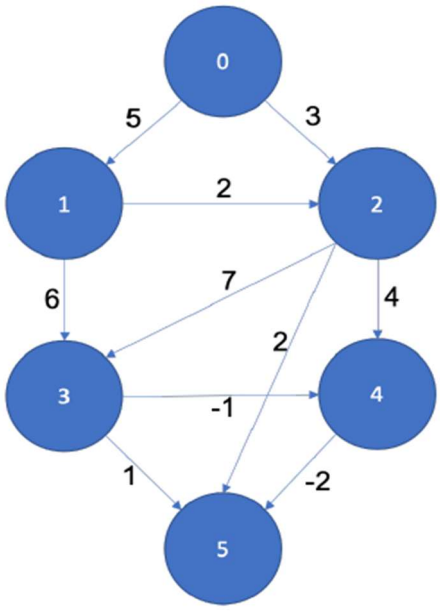
Avendo definito nelle sezioni precedenti le principali tipologie di grafo, esso rappresenta una tipologia di grafo derivante dal classico grafo diretto (Directed Graph).
Ha la caratteristica ulteriore di essere aciclico, cioè di non presentare archi all'indietro.



Su tale grafo valgono tutte le proprietà e operazioni che si sono definite in questa documentazione nel parlare di un grafo generico.

Utilizzo della struttura e considerazioni

Nella risoluzione di tale esercizio, com'è stato già detto nella prefazione, si utilizza un generico DAG che abbia però le caratteristiche di essere anche pesato, cioè, ci sarà implicitamente una funzione w, chiamata funzione peso, che associa ad ogni arco un peso, che rappresenterà la qualità tra le varie isole dell'arcipelago. Si avrà una situazione del genere:



Inoltre, per calcolare i cammini massimi, verrà utilizzata la BFS ma con una profonda modifica rispetto a come è stata spiegata nelle sezioni precedenti. La motivazione di tale modifica è, siccome essa è utilizzata per trovare i cammini minimi, in questo esercizio si vuole fare il netto contrario. Per questo motivo si vuole tenere traccia, per

ogni generico nodo, la sua chiave, la qualità del cammino massimo corrente e la lista di nodi che fanno parte di questo cammino massimo, facendo la massima attenzione a massimizzare la qualità e quindi il peso dell'intero cammino.

Scelte progettuali

Di seguito verrà fatta un'analisi approfondita sulle classi utilizzate, sul loro diagramma e le loro relazioni.

Analisi delle classi

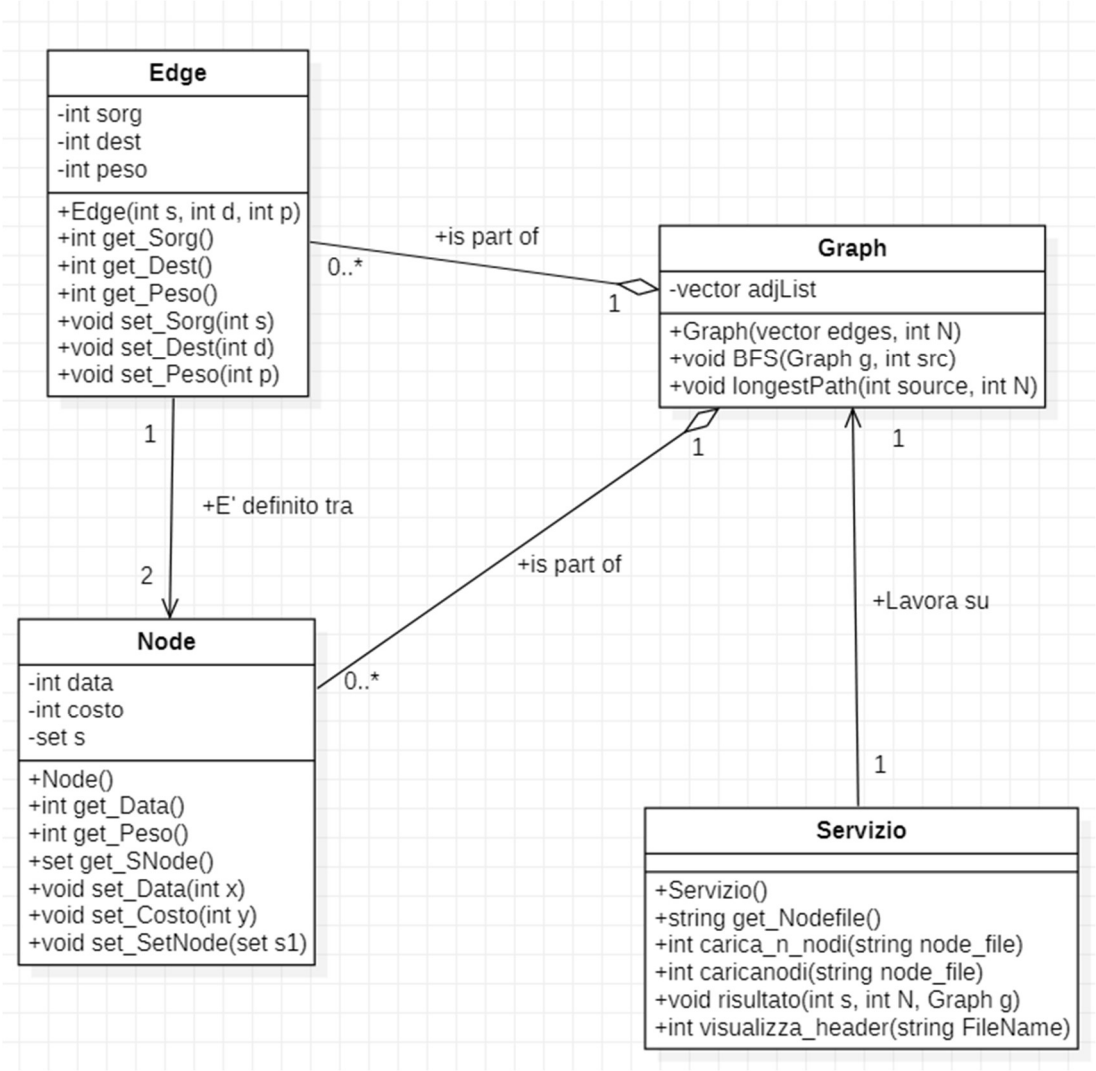
Ci si serve di 4 classi in tutto:

- Classe Edge, che identifica la struttura di un arco presente in un generico grafo. Essa presenta 3 attributi privati: la sorgente, la destinazione e il peso dell'arco. Essendo privati, e venendo ampiamente sfruttati nell'implementazione dell'algoritmo, tale classe presenta 3 metodi get e 3 metodi set per manipolare gli attributi.
- Classe Node, che identifica la struttura di un nodo in un generico grafo. Presenta anch'esso 3 attributi provati: il campo data, campo costo, che identifica la qualità del cammino corrente considerato, e il campo s che identifica il set di nodi che fanno parte del cammino. Ancora una volta, essendo attributi privati, vengono definiti 3 metodi get e 3 metodi set per poter manipolare tali attributi.
- Classe Graph che definisce un grafo generico. Possiede un attributo adjList, privato, che identifica la lista di adiacenze dei vari nodi. Non è di tipo list, ma si sceglie di definirlo come un vector di vector, in quanto possiede tutti gli archi che partono dal nodo corrente e terminano ai vari nodi alla quale il nodo corrente è adiacente. Tale classe possiede i 2 metodi che permettono la risoluzione dell'esercizio: la BFS modificata che trova i cammini massimi e LongestPath che stampa a video, di conseguenza alla BFS, le qualità dei cammini massimi partenti da sorgente singola verso tutti gli altri nodi.
- Classe Servizio. Essa è la classe che svolge i compiti secondari per la risoluzione dell'esercizio. Non presenta nessun attributo. Come metodi implementa:
    1. Get_Nodefile: procedura che acquisisce il nome del file contenente i valori del grafo

2. carica_n_nodi: procedura che acquisisce il numero dei nodi del grafo da analizzare
3. caricanodi: procedura che acquisisce i valori dei nodi del grafo (nodo, nodo adiacente, peso dell'arco)
4. risultato: procedura che richiama, costruito il grafo, la procedura BFS e di conseguenza, la procedura LongestPath
5. visualizza_header: procedura per visualizzare il file di intestazione

A questo punto, si vuole riassumere graficamente quanto detto fino ad ora, attraverso un semplice Class Diagramm.

Class Diagramm



Oltre a quanto è stato già detto nelle sezioni precedenti, ci si vuole soffermare sulle relazioni presenti tra tali classi. Ad esempio, tra la classe Edge e la classe Graph c'è una relazione di aggregazione. Quella dell'aggregazione è una relazione molto forte in

quanto un arco "è una parte di" un Grafo. Si giustificano le due tipologie di molteplicità in quanto in un Grafo possono essere presenti da 0 a N archi, mentre 0 o N archi possono far parte solo di un Grafo; un arco non può esserlo per due Grafi contemporaneamente. L'altra relazione la si trova tra la classe Servizio e la classe Grafo. Si parla, in questo caso di associazione diretta, in quanto la classe Servizio lavora attraverso Grafi. In questo caso le molteplicità sono così definite: un Servizio lavora su un unico Grafo e un Grafo viene analizzato da un solo Servizio. C'è un altro legame di aggregazione tra la classe Grafo e la classe Node. Il nodo, così come un arco è parte di un Grafo. In un grafo possono essere presenti da 0 a N nodi e 0 a N nodi fanno parte di un unico Grafo. Infine abbiamo la relazione che lega la classe Edge con la classe Node. C'è associazione diretta tra tali classi, in quanto un arco è definito tra nodi. Per questo abbiamo che un arco è definito tra 2 nodi e 2 nodi sono collegati da un unico arco.

Descrizione dell'algoritmo

Lo scopo di questa sezione è definire e spiegare nel dettaglio l'algoritmo risolutore.

Formato dei dati in input/output

Anche in quest'esercizio, come esplicitamente definito dalla traccia, l'utente deve definire in un file di testo una serie di elementi: il numero delle isole (il numero di nodi), il numero di ponti (il numero di archi), e i vari collegamenti tra le isole esplicitando la sorgente, la destinazione e la qualità del collegamento.
Utilizzando questi dati l'algoritmo andrà a costruire il Grafo che rappresenterà quindi l'arcipelago. Dopodichè verranno richiamate le dovute procedure per il calcolo dei vari cammini massimi definendo la qualità di questi ultimi.
In output, quindi, verranno visualizzati, per tutti gli N nodi appartenenti al grafo, la qualità di tutti i cammini massimi partenti da questi nodi verso tutti gli altri. La qualità del cammino sarà:

- -infinito: se non esiste un cammino tra i due nodi considerati
- 0: se si sta calcolando la qualità del cammino massimo tra la sorgente e sé stessa.
- Valore intero: negli altri casi generici

Da notare quindi che la qualità di tali cammini massimi può anche essere negativa perché i pesi degli archi possono anche essere negativi.

Composizione dell'algoritmo

L'algoritmo risolutore fa uso di 5 file in tutto: 2 file header e 3 file cpp:

Graph.hpp

Tale file contiene le definizioni delle classi Graph, Edge e Node, con i loro rispettivi attributi e metodi.

Servizio.hpp

Tale file contiene la definizione della classe Servizio con i suoi metodi, secondo quanto detto nelle sezioni precedenti.

Servizio-Methods.cpp

Tale file contiene l'implementazione dei metodi della classe Servizio secondo quanto già detto nella sezione dedicata all'analisi dell'algoritmo.

Graph-Methods.cpp

Tale file contiene l'implementazione dei metodi della classe Graph. Siccome ne fanno parte i due algoritmi che risolvono l'esercizio, si vuole spiegare nel dettaglio cosa fanno.

BFS

Com'è stato già detto, tale procedura è molto diversa rispetto alla BFS standard. L'obiettivo non è solo quello di visitare tutti i nodi, ma di calcolare i cammini massimi, tenendo conto della qualità e tendendo traccia dei nodi che fanno parte di tale cammino. Anche in questa procedura, si fa uso di una coda; tale coda attuerà, come suo solito fare, una politica FIFO (First In First Out). Ci si serve anche di due contenitori Set: 'vertici' e 's1'.
Si ricorda che Set è un contenitore associativo nella quale gli elementi mantengono un ordine preciso ogni qual volta vengono inseriti al suo interno. Inoltre, il contenitore Set ha la proprietà di inserire elementi solo se questi ultimi non sono stati già inseriti in precedenza; quindi gli elementi devono essere unici. Con questa tipologia di esercizio, cioè dovendo fare un'analisi su un grafo aciclico, risulta essere indifferente usufruire di un Set o di un Vector. Se l'esercizio avesse richiesto di fare un'analisi su un grafo non orientato e ciclico, la probabilità di entrare in un ciclo sarebbe stata alta. Allora in quel caso utilizzare un Set per mantenere i nodi visitati nel cammino sarebbe stato

determinante, in quanto non si avrebbe avuto il problema di memorizzare nodi già visitati, proprio perché si sarebbe rispettata la proprietà già definita del contenitore Set. Anche se non si è in questa situazione, si usufruisce lo stesso di contenitori Set in quanto, tra le operazioni che si svolgeranno su di essi, non si farà utilizzo di iteratori, e la complessità per ogni inserimento è $O(log_2 n)$, nella quale n è il numero totale di elementi. Utilizzando un vector, essendo un contenitore sequenziale, inserendo un elemento, si avrà una complessità pari a O(n+m), ove n è il numero totale di elementi nel vector e m il numero di elementi spostati per inserire il nuovo elemento nella sua giusta posizione.
I parametri di ingresso di questa procedura sono: il grafo sulla quale agire e la sorgente da cui partire verso tutti gli altri nodi.
Si inserisce il nodo k, inizialmente la sorgente, nella coda. Ovviamente il campo costo della sorgente è 0, in quanto la qualità del cammino massimo tra la sorgente e sé stessa è 0. Il numero di nodi nel cammino massimo tra la sorgente è sè stesso è 0.
Si entra nel ciclo while e si fa estrazione dalla coda del nodo in testa. Si mantengono le caratteristiche del nodo appena estratto, dopodichè si visita la sua lista di adiacenza. Considerato l'arco corrente di tale lista, si va a vedere se il nodo destinazione fa già parte del cammino massimo calcolato nei passi precedenti, quindi si verifica se fa parte del contenitore 'vertici'. Se l'if dà esito negativo, si aggiorna il cammino massimo definendo il nuovo contenitore 's1', inserendogli il contenuto di 'vertici', più il nuovo nodo scoperto. A questo punto si passa alla qualità. Se la qualità, che è definita nell'array 'quality', è più piccola della qualità del cammino massimo calcolato al passo corrente più il peso del nuovo arco considerato, si aggiorna la qualità di tale cammino. Tale valore viene inserito nella posizione associata al nuovo nodo considerato, nell'array 'quality'. Fatto questo, si passa all'iterazione successiva, considerando tutti gli archi della lista di adiacenza del nodo corrente, aggiornando di volta in volta, i nodi dei nuovi cammini massimi e le qualità dei cammini stessi.
Alla fine, considerati tutti i cammini massimi, l'array 'quality' conterrà tutti i massimi valori dei cammini tra la sorgente e i vari nodi del grafo.
Si definisce, il seguente pseudocodice:

```
BFS(Graph g,int s)
        Creare nodo k con: campo chiave= s, costo=0, set di nodi vuoto
        Enqueue(Q,k)                //inserisco il nodo in coda
        while(Q non è vuoto)
```

```
Node node<-dequeue(Q)                    //estrazione nodo dalla coda
v<-get_Data(node);                       //mantenimento della chiave
cost<-get_Costo(node);  //mantenimento del costo del cammino corrente
vertici<-get_SNode(node); //mantenimento dei nodi del cammino corrente

for(each edge in adjlist[v])
```

//se il nodo destinazione dell'arco esaminato non è stato già inserito in 'vertici' dovrà essere inserito nel contenitore s1 che indica la lista di nodi facente parte del cammino massimo

```
if (la destinazione dell'arco esaminato non si trova in 'vertici')
   set<int>s1<-vertici;          //contenitore set che contiene il
contenuto di 'vertici' più il nuovo nodo aggiunto
   s1.insert(edge.get_Dest());    //aggiungo il nodo al cammino
//verifico la qualità del cammino insieme al nuovo arco
   if(la qualità del nuovo cammino è migliore di quello precedente)
      //aggiorno la qualità del cammino esaminato
      quality[edge.get_Dest()]=edge.get_Peso()+cost;
//inserisco nella coda ogni vertice che era destinazione dell'arco esaminato
   k.set_Data(edge.get_Dest());
   k.set_Costo(cost+edge.get_Peso());
   k.set_SetNode(s1);
   enqueue(Q,k)
```

Complessità

Come operazioni dominanti ci sono:

- Inserimento di un nodo nella coda = O(1), fatta per n vertici => O(V)
- Visita delle liste di adiacenze, pari alla loro somma => O(E)

Quindi, in generale la complessità sarà sempre O(V+E)

LongestPath

Tale procedura, una volta che è stato definito l'array 'quality', va a stampare il suo contenuto, facendo visualizzare all'utente le qualità dei cammini massimi tra i vari nodi.

Complessità

Essendo presente solo un semplice ciclo for, la complessità di tale procedura è O(n).

Main.cpp

In tale file, si caricheranno i valori del file di input e ci sarà un ciclo for che:

- costruirà il grafo secondo tali valori ti tale file
- andrà per ogni nodo a richiamare la BFS per calcolare i cammini massimi
- stamperà a video i risultati tramite la LongestPath delle qualità di tali cammini

Complessità

Tale complessità, determinerà la complessità dell'intero algoritmo.

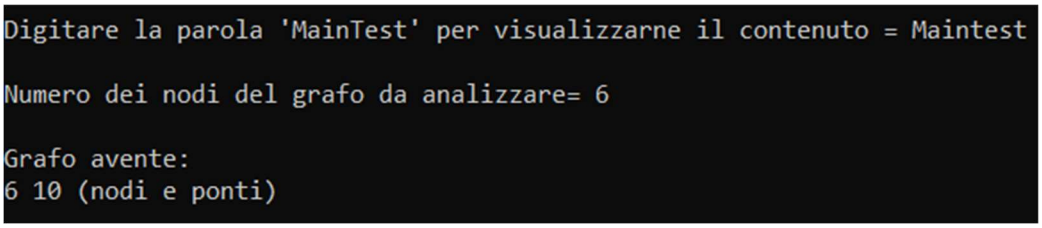
Il ciclo for sarà la parte dominante. Per n volte, quindi, saranno richiamate la BFS e, di conseguenza, la LongestPath. Si avrà quindi:

O(V*(V+E+V)) => O(V*(2V+E)) => O($2V^2$+V*E) => O(V^2+V*E)

Test/risultati

Come è stato fatto per l'esercizio 1, in questa sezione, in conclusione, si vuole vedere all'atto di esecuzione cosa l'utente visualizzerà.

Inizialmente, prelevato il file di input 'MainTest.txt', si visualizzano le caratteristiche del grafo da costruire: il numero di nodi e il numero di archi:



Dopo aver fatto questo, l'algoritmo provvederà a risolvere l'esercizio, e sulla base della costruzione del grafo visualizzerà a schermo, per tutte le sorgenti, qualità dei vari cammini massimi verso tutti gli altri nodi:

```
---------------------------IMPLEMENTAZIONE----------------------------
Qualita' del collegamento dal nodo 0:
Nodo 0= 0
Nodo 1= 5
Nodo 2= 7
Nodo 3= 14
Nodo 4= 13
Nodo 5= 15

Qualita' del collegamento dal nodo 1:
Nodo 0= -INFINITO
Nodo 1= 0
Nodo 2= 2
Nodo 3= 9
Nodo 4= 8
Nodo 5= 10

Qualita' del collegamento dal nodo 2:
Nodo 0= -INFINITO
Nodo 1= -INFINITO
Nodo 2= 0
Nodo 3= 7
Nodo 4= 6
Nodo 5= 8

Qualita' del collegamento dal nodo 3:
Nodo 0= -INFINITO
Nodo 1= -INFINITO
Nodo 2= -INFINITO
Nodo 3= 0
Nodo 4= -1
Nodo 5= 1

Qualita' del collegamento dal nodo 4:
Nodo 0= -INFINITO
Nodo 1= -INFINITO
Nodo 2= -INFINITO
Nodo 3= -INFINITO
Nodo 4= 0
Nodo 5= -2

Qualita' del collegamento dal nodo 5:
Nodo 0= -INFINITO
Nodo 1= -INFINITO
Nodo 2= -INFINITO
Nodo 3= -INFINITO
Nodo 4= -INFINITO
Nodo 5= 0
```