Luigi Marino

Matricola 0124002257

RELAZIONE PROGETTO ALGORITMI E STRUTTURE DATI

- **DESCRIZIONE DEL PROBLEMA**
- **DESCRIZIONE STRUTTURE DATI**
- **FORMATO DATI IN INPUT/OUTPUT**
- **DESCRIZIONE ALGORITMO**
- **CLASS DIAGRAM**
- **STUDIO COMPLESSITA'**
- **TEST/RISULTATI**

DESCRIZIONE DEL PROBLEMA

- ***QUESITO 1***

Problema (traccia)

"Si vuole realizzare la struttura dati HashGraph che consenta di memorizzare un grafo orientato in una hash table. Ogni nodo del grafo viene memorizzato in una cella della hash table insieme alla lista di adiacenza. Progettare ed implementare una struttura dati che, dato un file di input contenente il grafo, costruisca HashGraph corrispondente e consenta di effettuare le seguenti operazioni: AddEdge(i,j), RemoveEdge(i,j), FindEdge(i,j) e DFS(s). Il file di input contiene nel primo rigo due numeri interi, 0<=N<=1000 e 0<=M<=1000, separati da uno spazio che rappresentano rispettivamente il numero di nodi ed il numero di archi. I successivi M righi contengono due numeri interi separati da uno spazio che rappresentano il nodo sorgente ed il nodo destinazione. Dotare il programma di un menu da cui sia possibile richiamare le suddette operazioni."

Analisi del problema

Il quesito che ci viene proposto vede come primo e principale obiettivo quello di immagazzinare all'interno di una struttura dati Hash Table un ulteriore struttura dati, ovvero un grafo orientato. Il problema specifica anche che vanno memorizzate anche le liste di adiacenza di ogni vertice del grafo, sottintendendo quindi di dover utilizzare come metodo di rappresentazione del grafo, quello che vede impegnate le liste di adiacenza, un ulteriore struttura dati. La creazione della struttura dati principale, la Hashgraph, vede coinvolta la creazione di ulteriori strutture dati. Vengono anche menzionate le operazioni che deve poter effettuare la nostra nuova struttura dati, operazioni base che riguardano il grafo orientato. Il fulcro del problema è quindi la corretta implementazione delle nostre strutture dati, in modo da poter effettuare le operazioni desiderate senza alcun problema. In conclusione il tutto dovrà essere "dichiarato" all'interno del nostro programma in modo automatico, dando in input un file.

1. Creazione di una struttura dati linked list.
2. Creazione di una struttura dati grafo orientato.
3. Creazione di una struttura dati hash table.
4. Creazione della struttura dati HashGraph.
5. Immagazzinamento del grafo nell'HashGraph.
6. Implementazione delle operazioni da effettuare.
7. Creazione del menu.

DESCRIZIONE DEL PROBLEMA

- ***QUESITO 2***

Problema (traccia)

*"*Dopo diversi anni ed ingenti investimenti economici, finalmente lo stato di GraphaNui ha la sua diga in grado di produrre energia elettrica e fornire acqua potabile a tutti gli abitanti. È necessario però terminare la rete idrica in modo che tutte le città ricevano l'acqua. A tal fine viene convocato un famoso informatico a cui viene fornita la piantina delle città con l'indicazione delle condotte attualmente presenti, con il compito di determinare il minimo numero di condotte da costruire"

Analisi del problema

Il seguente problema rappresenta delle città come dei nodi di un grafo e le condotte idriche di queste città come gli archi che collegano i vertici. Il nostro compito è quello di portare acqua a tutte le città, quindi di avere tutti i vertici collegati tra di loro in qualche modo. Il grafo è orientato e quindi anche il verso degli archi influirà sul nostro problema. Ci occuperemo innanzitutto della creazione del grafo e della sua implementazione per il problema, dopodiché andremo ad analizzare ogni vertice del grafo attraverso una visita di quest'ultimo e determineremo il numero di archi da dover creare per permettere a tutti i vertici di essere collegati in qualche modo.

1. Creazione del grafo.
2. Visita del grafo e analisi dei suoi vertici.
3. Determinare il numero di archi da creare.

DESCRIZIONE STRUTTURE DATI

- ***QUESITO 1***

Nodi (Linked List)

I nodi sono utilizzati per implementare la linked list in secondo luogo, sono composti da un dato del tipo scelto, in quanto implementati attraverso dei template e di due puntatori ad ulteriori nodi, uno che indica il precedente nodo, uno che indica il successivo.

Le operazioni che possono essere eseguite sui nodi sono 6:

1. SetData -> operazione che ci permette di inserire il valore del nodo, quello che "immagazzina" il nodo.
2. GetData -> operazione che restituisce il valore impostato al nodo.
3. SetNext -> operazione che ci permette di assegnare il nodo successivo a quello sulla quale viene richiamata.
4. GetNext -> operazione che restituisce il nodo successivo a quello sulla quale viene richiamata.
5. SetPrev -> operazione che ci permette di assegnare il nodo precedente a quello sulla quale viene richiamata.
6. getPrev-> operazione che restituisce il nodo precendete a quello sulla quale viene richiamata.

Pseudo-Codice metodi

SetData(data)
value <- data

GetData()
return value

SetNext(n)
next<- n

GetNext()
return next

SetPrev(p)
prev <- p

GetPrev()
return prev

Linked List

La Linked List implementata all'interno dell'algoritmo è stata scritta interamente da zero e mediante l'utilizzo di template, in modo da rendere universale il suo utilizzo. La lista si compone di nodi di tipo scelto in precedenza durante la dichiarazione. La lista è composta dalla testa, che quando viene inizializzata è uguale a nullptr e da nient'altro, sarà poi possibile popolare questa lista andando ad utilizzare le varie operazioni.

1. SetHead -> ci permette di assegnare un nodo che sarà la testa della lista.
2. GetHead -> restituisce il nodo che è in testa alla lista.
3. GetTail -> restituisce la coda della lista, l'ultimo elemento.
4. InsertNode -> ci permette di inserire un nodo all'interno della lista.
5. DeleteNode -> permette l'eliminazione di uno specifico nodo nella lista.
6. IsEmpty -> controlla se la lista è vuota e restituisce vero o falso in base alla presenza o meno di nodi.
7. PushFront -> inserisce un nodo in testa alla lista, impostandolo anche come testa.
8. PrintList -> stampa della lista intera.

Pseudo-Codice metodi

*SetHead(Node *h)*
```
head<- h
```

GetHead()
```
return head
```

GetTail()
```
if head->getnext() = nullptr
return head
else
        node temp <- head
        while temp->getnext() != nullptr
temp <- temp->getnext()

return temp
```

InsertNode(data)
```
new node(data)
if isempty = true
 head <- node
else node temp <- head
while temp->getnext() != nullptr
temp <- temp ->getnext()
temp->setnext(node)
node->setprev(temp)
```

DeleteNode(node)

```
if node = head
head<- node->getnext()
else if node->getnext() = nullptr
node->getprev()->setnext(nullptr)
delete node
else
node->getprev()->setnext(node->getnext())
node->getnext()->setprev(node->getprev())
```

IsEmpty()

```
if head = nullptr
        return true
else return false
```

PushFront(data)

```
if isempty() = true
head = data
else
head->setprev(data)
data->setnext(head)
head <- data
```

Vertice Grafo (Grafo)

I vertici del grafo orientato che andremo ad implementare successivamente sono composti dalla lista di adiacenza che rappresenta tutti gli altri vertici ai quali è collegato il nostro vertice, da un valore specifico e dal colore del vertice. Il vertice del grafo viene implementato per la costruzione di un grafo orientato nel nostro caso. Le operazioni che si possono effettuare sui vertici del grafo sono:

1. SetData -> permette di immagazzinare un dato di tipo intero che generalmente rappresenta il numero del vertice.
2. GetData -> restituisce il valore intero del vertice.
3. SetColore -> imposta il colore del vertice, 0, 1, 2 – bianco, grigio, nero.
4. GetColore -> restituisce l'intero che rappresenta il colore del vertice.
5. AggiungiArco -> aggiunge un arco tra l'attuale nodo e il nodo destinazione che diamo in input alla funzione, verificando prima che sia possibile aggiungerlo. La creazione dell'arco viene fatta andando ad inserire nella lista di adiacenza del nostro vertice la destinazione, in quanto grafo orientato.
6. RimuoviArco -> rimuove l'arco tra l'attuale vertice e quello destinazione, se esistente, andando a rimuovere dalla lista di adiacenza del nostro vertice quello dato in input.
7. TrovaArco -> verifica la presenza di un arco tra due vertici, ovvero verifica la presenza all'interno della lista di adiacenza del vertice e restituisce vero o falso.
8. GetAdj -> restituisce l'intera lista di adiacenza del vertice.
9. PrintAdj -> stampa l'intera lista di adiacenza del vertice.

Pseudo-Codice metodi

SetData(valore)
value<-valore

GetData()
return value

SetColore(c)
colore<-c

GetColore()
return colore

AggiungiArco(destinazione)
getadj()->insertnode(destinazione)

RimuoviArco(destinazione)
list<- getadj()
list->deletenode(destinazione)

TrovaArco(destinazione)
list<- getadj()
tmp <- list->gethead()
while tmp != destinazione
tmp->getnext()

GetAdj()
return adj

PrintAdj()
adj->printlist()

Grafo

La struttura dati grafo è stata implementata mediante un vettore, libreria vector di C++, che al suo interno contiene i vertici che compongono il nostro grafo (verticeGrafo), le operazioni che si possono effettuare sul grafo sono 3:

1. AggiungiVertice -> aggiunge un vertice all'interno del grafo, ovvero effettua una push_back sul vector.
2. GetVertice -> restituisce il vertice che si trova nella posizione richiesta, per il modo in cui è stata strutturata l'implementazione, andando ad effettuare una getvertice su 1, restituirà il vertice del grafo che ha valore 1 e così via.
3. PrintGrafo -> stampa l'intero grafo, andando a verificare anche le eventuali connessioni tra vertici.

Pseudo-Codice metodi

```
AggiungiVertice(v)
listavertici->push_back(v)

GetVertice(i)
return listavertici->at(i)

PrintGrafo()
for i = 1 to listavertici->size()
tmp<-getvertice(i)
tmp->printadj()
```

HashTable

La HashTable proposta ed implementata è quella che gestisce le collisioni mediante liste di adiacenza, quindi la nostra hashtable è costituita da un array di Linked List di tipo Vertice Grafo, per iniziare ad indirizzarci verso lo scopo del nostro quesito, quello di salvare un grafo all'interno di una hashtable. Le operazioni che si possono effettuare sono:

1. HashFunction -> funzione di Hash che restituisce l'indice nel quale immagazzinare il vertice del grafo.
2. HashInsert -> funzione che inserisce nella table il vertice del grafo, calcolando anche l'indice mediante HashFunction.
3. GetDim -> restituisce la dimensione della tavola.
4. GetTable -> restituisce la linked list che si trova nell'indice richiesto.
5. HashDelete -> elimina un vertice del grafo dalla HashTable.
6. HashSearch -> cerca un vertice all'interno della HashTable, restituisce vero o falso.
7. PrintTable -> stampa l'intera HashTable.

Pseudo-Codice metodi

HashFunction(key)
return key%dim

HashInsert(key, vertex)
table[hashFunction(key)].push_front(vertex)

GetDim()
return dim

GetTable(i)
return table[i]

HashDelete(vertex)
index <- hashfunction(vertex->getdata())
temp <- gettable(index).gethead()->getnext()
while temp != gettable(index).gettail()
if temp = vertex
gettable(index).deletenode(temp)
temp->getnext()

HashSearch(key)
index <- hashfunction(key)
temp <- table(index).gethead()
while temp != table(index).gettail()
if temp->getdata() = key
return true
temp->getnext()
return false

HashGraph

La HashGraph è la struttura principale di questo quesito, in quanto "non esistente" ma da implementare da zero. La HashGraph è composta da una HashTable, che a sua volta è composta da un array di Linked List di vertici del grafo. Viene inizializzata con una dimensione e viene creata la Hash Table illustrata precedentemente. Le operazioni che effettua sono anche quelle richieste dal quesito:

1. AddEdge -> aggiunge un arco tra due vertici, andando a scavare tra le varie strutture dati, prendendo la table nella corretta posizione, estraendo il vertice in quella posizione e andando a lavorare sulla lista di adiacenza di quel vertice, aggiungendone un altro.
2. RemoveEdge -> lo stesso procedimento fatto in precedenza per aggiungere un arco, viene fatto per la rimozione, andando a rimuovere il nodo dalla lista di adiacenza.
3. DFS_init -> inizializza la DFS, in quanto utilizziamo un menu che può richiamare più volte la DFS, abbiamo bisogno di impostare tutti i colori nuovamente come al momento dell'inizializzazione.
4. DFS -> la ricerca in profondità all'interno del grafo.

Pseudo-Codice metodi

addEdge(i,j)
If i<dim and i >= 0 and j<dim and j>= 0
tavola->getTable(i)->getHead()->getData()->getAdj()->InsertNode(tavola->getTable(j)->getHead()->getData())

RemoveEdge(i,j)
If i<dim and i >= 0 and j<dim and j>= 0
tavola->getTable(i)->getHead()->getData()->getAdj()->rimuoviArco(tavola->getTable(j)->getHead())

DFS_INIT()
For i = 1 to dim
tavola->gettable(i)->gethead()->getdata()->setcolore(0)

DFS(sorgente)
For each v € adj[sorgente]
If colore = 0
Dfs(v)

- ***QUESITO 2***

Nodi (Linked List)

I nodi sono utilizzati per implementare la linked list in secondo luogo, sono composti da un dato del tipo scelto, in quanto implementati attraverso dei template e di due puntatori ad ulteriori nodi, uno che indica il precedente nodo, uno che indica il successivo.

Le operazioni che possono essere eseguite sui nodi sono 6:

7. SetData -> operazione che ci permette di inserire il valore del nodo, quello che "immagazzina" il nodo.
8. GetData -> operazione che restituisce il valore impostato al nodo.
9. SetNext -> operazione che ci permette di assegnare il nodo successivo a quello sulla quale viene richiamata.
10. GetNext -> operazione che restituisce il nodo successivo a quello sulla quale viene richiamata.
11. SetPrev -> operazione che ci permette di assegnare il nodo precedente a quello sulla quale viene richiamata.
12. getPrev-> operazione che restituisce il nodo precendete a quello sulla quale viene richiamata.

Pseudo-Codice metodi

SetData(data)
value <- data

GetData()
return value

SetNext(n)
next<- n

GetNext()
return next

SetPrev(p)
prev <- p

GetPrev()
return prev

Linked List

La Linked List implementata all'interno dell'algoritmo è stata scritta interamente da zero e mediante l'utilizzo di template, in modo da rendere universale il suo utilizzo. La lista si compone di nodi di tipo scelto in precedenza durante la dichiarazione. La lista è composta dalla testa, che quando viene inizializzata è uguale a nullptr e da nient'altro, sarà poi possibile popolare questa lista andando ad utilizzare le varie operazioni.

9. SetHead -> ci permette di assegnare un nodo che sarà la testa della lista.
10. GetHead -> restituisce il nodo che è in testa alla lista.
11. GetTail -> restituisce la coda della lista, l'ultimo elemento.
12. InsertNode -> ci permette di inserire un nodo all'interno della lista.
13. DeleteNode -> permette l'eliminazione di uno specifico nodo nella lista.
14. IsEmpty -> controlla se la lista è vuota e restituisce vero o falso in base alla presenza o meno di nodi.
15. PushFront -> inserisce un nodo in testa alla lista, impostandolo anche come testa.
16. PrintList -> stampa della lista intera.

Pseudo-Codice metodi

*SetHead(Node *h)*
```
head<- h
```

GetHead()
```
return head
```

GetTail()
```
if head->getnext() = nullptr
return head
else
        node temp <- head
        while temp->getnext() != nullptr
temp <- temp->getnext()

return temp
```

InsertNode(data)
```
new node(data)
if isempty = true
 head <- node
else node temp <- head
while temp->getnext() != nullptr
temp <- temp ->getnext()
temp->setnext(node)
node->setprev(temp)
```

DeleteNode(node)

```
if node = head
head<- node->getnext()
else if node->getnext() = nullptr
node->getprev()->setnext(nullptr)
delete node
else
node->getprev()->setnext(node->getnext())
node->getnext()->setprev(node->getprev())
```

IsEmpty()

```
if head = nullptr
        return true
else return false
```

PushFront(data)

```
if isempty() = true
head = data
else
head->setprev(data)
data->setnext(head)
head <- data
```

Vertice Grafo (Grafo)

I vertici del grafo orientato che andremo ad implementare successivamente sono composti dalla lista di adiacenza che rappresenta tutti gli altri vertici ai quali è collegato il nostro vertice, da un valore specifico e dal colore del vertice. Il vertice del grafo viene implementato per la costruzione di un grafo orientato nel nostro caso. Le operazioni che si possono effettuare sui vertici del grafo sono:

10. SetData -> permette di immagazzinare un dato di tipo intero che generalmente rappresenta il numero del vertice.
11. GetData -> restituisce il valore intero del vertice.
12. SetColore -> imposta il colore del vertice, 0, 1, 2 – bianco, grigio, nero.
13. GetColore -> restituisce l'intero che rappresenta il colore del vertice.
14. AggiungiArco -> aggiunge un arco tra l'attuale nodo e il nodo destinazione che diamo in input alla funzione, verificando prima che sia possibile aggiungerlo. La creazione dell'arco viene fatta andando ad inserire nella lista di adiacenza del nostro vertice la destinazione, in quanto grafo orientato.
15. RimuoviArco -> rimuove l'arco tra l'attuale vertice e quello destinazione, se esistente, andando a rimuovere dalla lista di adiacenza del nostro vertice quello dato in input.
16. TrovaArco -> verifica la presenza di un arco tra due vertici, ovvero verifica la presenza all'interno della lista di adiacenza del vertice e restituisce vero o falso.
17. GetAdj -> restituisce l'intera lista di adiacenza del vertice.
18. PrintAdj -> stampa l'intera lista di adiacenza del vertice.

Pseudo-Codice metodi

SetData(valore)
value<-valore

GetData()
return value

SetColore(c)
colore<-c

GetColore()
return colore

AggiungiArco(destinazione)
getadj()->insertnode(destinazione)

RimuoviArco(destinazione)
list<- getadj()
list->deletenode(destinazione)

TrovaArco(destinazione)
list<- getadj()
tmp <- list->gethead()
while tmp != destinazione
tmp->getnext()

GetAdj()
return adj

PrintAdj()
adj->printlist()

SetPadre(v)
padre<-v

GetPadre()
return padre

Grafo

La struttura dati grafo è stata implementata mediante un vettore, libreria vector di C++, che al suo interno contiene i vertici che compongono il nostro grafo (verticeGrafo), le operazioni che si possono effettuare sul grafo sono 3:

4. AggiungiVertice -> aggiunge un vertice all'interno del grafo, ovvero effettua una push_back sul vector.
5. GetVertice -> restituisce il vertice che si trova nella posizione richiesta, per il modo in cui è stata strutturata l'implementazione, andando ad effettuare una getvertice su 1, restituirà il vertice del grafo che ha valore 1 e così via.
6. PrintGrafo -> stampa l'intero grafo, andando a verificare anche le eventuali connessioni tra vertici.
7. DFS_VISIT -> effettua la DFS visit su un vertice. (VI E' UNA MODIFICA CHE ANALIZZEREMO SUCCESSIVAMENTE)
8. DFS -> effettua la DFS su tutto il grafo.Risolvi() -> questo risolve quello che è il problema proposto, lo andremo ad analizzare in seguito.

Pseudo-Codice metodi

AggiungiVertice(v)
listavertici->push_back(v)

GetVertice(i)
return listavertici->at(i)

PrintGrafo()
for i = 1 to listavertici->size()
tmp<-getvertice(i)
tmp->printadj()

DFS()
For i = 1 to listavertici.size()
If colore[i] = bianco
DFS_VISIT(i)

DFS-VISIT(u)
colore[u] <- grigio
For each v € adj[u]
If colore[v] = nero
padre[v] <-u
If colore[v] = bianco
padre[v] <-u
DFS-VISIT(v)
colore[u] <- nero

FORMATO DATI IN INPUT/OUTPUT

- ***QUESITO 1***

Input

Per quanto concerne il primo quesito, riceviamo in input un file di testo che contiene nel primo rigo due numeri interi:

0<=N<=1000
0<=M<=1000

I successivi M ed N righi contengono due numeri separati da uno spazio che indicano il nodo sorgente e quello destinazione.
In input quindi riceviamo un file che viene poi letto e vengono convertiti in interi i valori.

Output

In output abbiamo una visualizzazione a schermo di un menu, che offre la possibilità di effettuare cinque scelte:

```
******************************
 1 - Aggiungi un arco tra due vertici.
 2 - Rimuovi un arco tra due vertici.
 3 - Trova un arco tra due vertici.
 4 - Effettua la DFS.
 5 - Esci.
 Scegli un numero dal menu e premi invio.
```

In base alla scelta dell'utente si avrà un diverso output per ogni caso, nel primo caso:

```
Inserisci il primo vertice 1
Inserisci il secondo vertice 2

Arco aggiunto con successo.
```

Secondo caso:

```
Inserisci il primo vertice 3
Inserisci il secondo vertice 49

Non e stato possibile rimuovere l'arco. (NODI INESISTENTI O ARCO INESISTENTE)
```

Terzo caso:

```
Inserisci il primo vertice 1
Inserisci il secondo vertice 2

Arco non esistente
```

Quarto caso:

```
Inserisci il nodo sorgente dal quale far partire la DFS 1
{1} {3} {5} {2} {4} {9} {7} {8} {10} {6}
```

- ***QUESITO 2***

Input

Nel secondo quesito per quanto riguarda i dati in input non notiamo una grossa differenza con il primo quesito, in quanto abbiamo due interi N e P, N indica le città e P le condotte idriche esistenti.

1<=N<=1000
0<=P<=10000

I successivi P righi contengono due numeri separati da uno spazio che indicano le città collegate tra di loro. In input quindi riceviamo un file che viene poi letto e vengono convertiti in interi i valori.

Output

In output abbiamo la visualizzazione a schermo del numero minimo di condotte idriche costruire per collegare tutte le città tra di loro.

DESCRIZIONE ALGORITMO

- ***QUESITO 1***

L'algoritmo e la risoluzione del problema si concentra tutto all'interno delle strutture dati che abbiamo creato o modificato appositamente per permettere il salvataggio di un grafo all'interno di una HashTable, analizzeremo infatti le strutture dati e le varie modifiche.

Partendo dalla struttura HashTable, al suo interno andiamo a creare un array di LinkedList, contenenti i vertici del grafo, questa è stata l'unica modifica effettuata a quella che è una classica HashTable.

Passiamo adesso alla struttura HashGraph, quella che riesce finalmente a risolvere il nostro quesito,(la struttura è composta dalla sua dimensione e da una HashTable, quello che però ci interessa analizzare sono le operazioni richieste:

1. AddEdge(i,j)

La funzione che ci permette di aggiungere un arco tra due vertici del grafo, il modo in cui lo fa è attraverso i metodi delle varie strutture dati che compongono la HashGraph. Andiamo a richiamare diverse funzioni fino ad ottenere la lista di adiacenza del vertice contenuto in posizione i nella table, tramite la funzione poi del vertice, andiamo ad inserire nella lista di adiacenza del vertice il vertice j, ottenendolo come abbiamo ottenuto i in precedenza.

2. RemoveEdge(i,j)

La rimozione di un arco viene effettuata allo stesso modo con il quale viene creato un arco tra due vertici, la differenza sta nella funzione che viene richiamata una volta che otteniamo il vertice del grafo, richiamiamo la DeleteNode, eliminando j dalla lista di adiacenza del vertice i, se esistente.

3. FindEdge(i,j)

Allo stesso modo delle funzioni precedenti, essendo un grafo orientato, andiamo ad ispezionare la lista di adiacenza di i, nel caso in cui troviamo j al suo interno, allora diremo che abbiamo trovato l'arco che collega i due vertici, nel caso opposto, diremo che non esiste.

4. DFS(s)

La DFS permette di fare una visita in profondità del grafo a partire dal vertice s, viene ispezionata la sua lista di adiacenza e le liste di adiacenza varie dei vertici contenuti in quest'ultima, in modo ricorsivo, nel caso in cui i loro colori siano impostati a 0, restituendo tutti i vertici nell'ordine in cui vengono scoperti a partire dal nodo sorgente.

Il modo in cui andiamo a ricevere i valori interi dal file di testo è illustrato nel main, andiamo a lavorare sul file attraverso la libreria fstream e sui caratteri attraverso la libreria string. I valori vengono convertiti in interi attraverso la funzione stoi, effettuata su una stringa che viene ricavata attraverso la getline. Sfruttiamo il modo in cui è composto il file di testo e quindi lo spazio che divide i numeri ci permette di effettuare la stoi sul numero "di destra" attraverso la substr sul numero di "sinistra", di seguito il codice:

```
int main()
{
    ifstream file;
    file.open("file1.txt");
    string str1;
    getline(file, str1);
    int var1, var2, N, M, count = 0;
    size_t pos;
    N = stoi(str1, &pos);
    M = stoi(str1.substr(pos));
    hashGraph hg(N + 1);
    while (getline(file, str1) && count < M)
    {
        count++;
        var1 = stoi(str1, &pos);
        var2 = stoi(str1.substr(pos));
        hg.addEdge(var1, var2);
    }
```

In questo modo andiamo ad aggiungere gli archi poi effettuando questa operazione in un while, affinchè riesce ad estrarre una line continueremo ad aggiungere archi e ovviamente un secondo controllo è fatto su un contatore, che non superi M, in modo che se ci siano ulteriori caratteri superiori ad M, non continui ad estrarli.

Ci viene proposto un menu quando avviamo il programma:

```
**********************************
1 - Aggiungi un arco tra due vertici.
2 - Rimuovi un arco tra due vertici.
3 - Trova un arco tra due vertici.
4 - Effettua la DFS.
5 - Esci.
Scegli un numero dal menu e premi invio.
```

Il menu è costruito attraverso uno switch statement che prende in input un numero da 1 a 5 e prosegue con altre interfacce, quelle illustrate in precedenza, in base alla scelta dell'utente.

- ***QUESITO 2***

Per la risoluzione del secondo quesito la visita del grafo è stato il punto centrale, attraverso la visita del grafo in profondità (la DFS), andremo ad effettuare la visita del grafo nella sua interezza, con una piccola modifica all'algoritmo che conosciamo. Effettuiamo la DFS su tutti i vertici del grafo, andando a richiamare la DFS_VISIT, che effettuerà l'operazione di controllo in questo modo:

Oltre ad effettuare la classica visita sui vertici di colore bianco, andremo a verificare anche i vertici di colore nero contenuti all'interno della lista di adiacenza di ogni vertice, impostando anche per loro il padre (la sorgente da cui parte la visita).

In questo modo, alla fine della DFS potremmo andare a verificare tutti i vertici che fanno parte del grafo ma che non hanno un padre, saranno tutti quelli che non sono collegati tra di loro, quindi avranno bisogno di un arco. Da ricordare però, che la fonte principale della condotta idrica non è da considerare, quindi avremo sempre 1 arco in più di quanti realmente ne sono richiesti, infatti andremo a restituire il contatore – 1.

Di seguito l'algoritmo nello specifico:

```
void DFS_VISIT(verticeGrafo *sorgente)
{
    sorgente->setColor(1);
    Node<verticeGrafo *> *tmpvertex = sorgente->getAdj()->getHead();
    linkedList<verticeGrafo *> *tmplist = sorgente->getAdj();

    while (tmpvertex != tmplist->getTail()->getNext() && tmpvertex!= nullptr)
    {
        if (tmpvertex->getData()->getColore() == 2)
        {
            tmpvertex->getData()->setPadre(sorgente);
        }

        if (tmpvertex->getData()->getColore() == 0)
        {
            tmpvertex->getData()->setPadre(sorgente);
            DFS_VISIT(tmpvertex->getData());
        }

        tmpvertex = tmpvertex->getNext();
    }
    sorgente->setColor(2);
}
```

Il primo IF è quello che riguarda la nostra modifica, che ci permette di impostare i padri anche per i vertici di colore nero.

Mentre questo è l'algoritmo che ci andrà a restituire in output il numero di archi da costruire al fine della risoluzione:

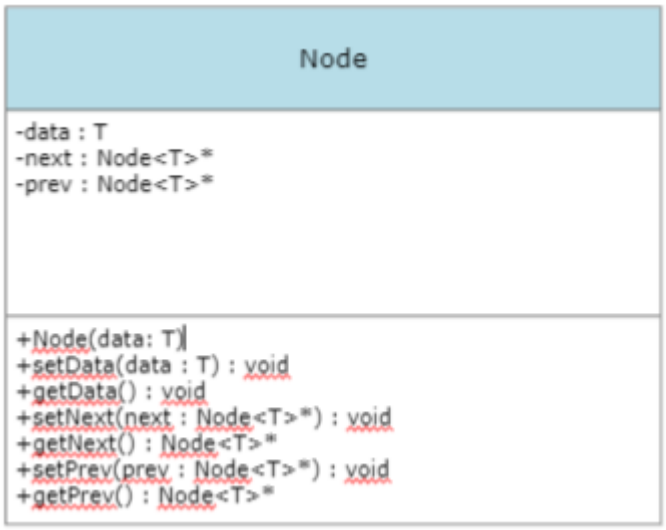
```
int risolvi(){
    for (int i = 0; i<listaVertici->size(); i++){
        if (getVertice(i)->getPadre() == nullptr)
        {
            count++;
        }
    }
    return count-1;
}
```

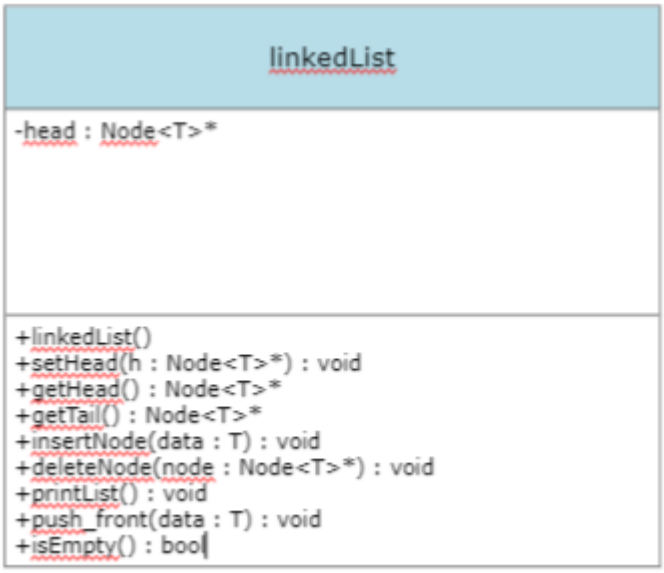
Per quanto riguarda lo stabilire a priori il numero di vertici e di archi, il tutto è fornito nel quesito 1, in quanto il formato in cui ci vengono forniti i file di testo è lo stesso, quindi la costruzione del grafo è uguale, con la sola differenza che viene costruito un grafo vero e proprio e non innestato in una HashTable:

```
Grafo gr(N + 1);
while (getline(file, str1) && count < M)
{
    count++;
    var1 = stoi(str1, &pos);
    var2 = stoi(str1.substr(pos));
    gr.aggiungiArco(var1, var2);
}
```

CLASS DIAGRAM

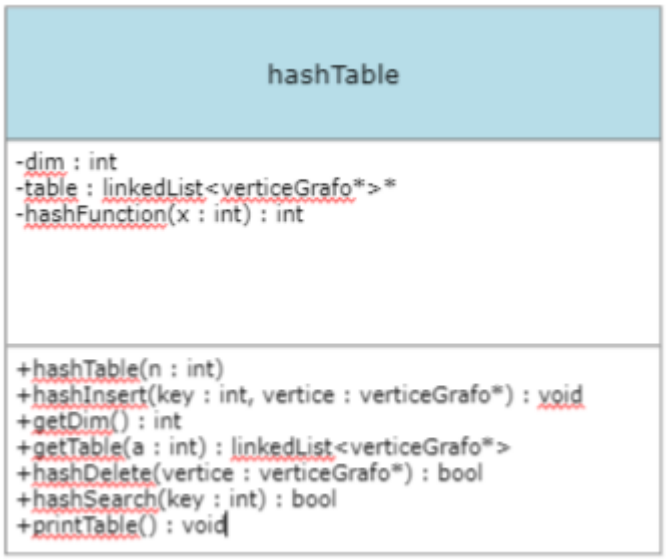
- ***QUESITO 1***






verticeGrafo
-adj : linkedList<verticeGrafo*>*
-data : int
-colore :int
+verticeGrafo(value : int)
+setData(data : int) : void
+getData() : int
+setColor(c : int) : void
+getColore() : bool
+aggiungiArco(destinazione : verticeGrafo*) : bool
+trovaArco(destinazione : verticeGrafo*) : bool
+rimuoviArco(destinazione : verticeGrafo*) : bool
+getAdj() : linkedList<verticeGrafo*>*
+printAdj() : void


Grafo
-listaVertici : vector<verticeGrafo*>
+aggiungiVertice(vertice : verticeGrafo*) : void
+getVertice(numeroVertice : int) : verticeGrafo*
+printGrafo() : void


hashTable
-dim : int
-table : linkedList<verticeGrafo*>*
-hashFunction(x : int) : int
+hashTable(n : int)
+hashInsert(key : int, vertice : verticeGrafo*) : void
+getDim() : int
+getTable(a : int) : linkedList<verticeGrafo*>
+hashDelete(vertice : verticeGrafo*) : bool
+hashSearch(key : int) : bool
+printTable() : void


hashGraph
-dim : int
-tavola : hashTable*
+hashGraph(n : int)
+addEdge(i : int, j : int) : bool
+removeEdge(i : int, j : int) : bool
+findEdge(i : int, j : int) : bool
+DFS_init(sorgente : int) : void
+DFS(sorgente : int) : void

- ***QUESITO 2***


Node
-data : T
-next : Node<T>*
-prev : Node<T>*
+Node(data: T)
+setData(data : T) : void
+getData() : void
+setNext(next : Node<T>*) : void
+getNext() : Node<T>*
+setPrev(prev : Node<T>*) : void
+getPrev() : Node<T>*
linkedList
-head : Node<T>*
+linkedList()
+setHead(h : Node<T>*) : void
+getHead() : Node<T>*
+getTail() : Node<T>*
+insertNode(data : T) : void
+deleteNode(node : Node<T>*) : void
+printList() : void
+push_front(data : T) : void
+isEmpty() : bool


verticeGrafo
-adj : linkedList<verticeGrafo*>*
-data : int
-colore :int
+verticeGrafo(value : int)
+setData(data : int) : void
+getData() : int
+setColor(c : int) : void
+getColore() : bool
+aggiungiArco(destinazione : verticeGrafo*) : bool
+trovaArco(destinazione : verticeGrafo*) : bool
+rimuoviArco(destinazione : verticeGrafo*) : bool
+getAdj() : linkedList<verticeGrafo*>*
+printAdj() : void


Grafo
-listaVertici : vector<verticeGrafo*>*
-numeroVertici : int
-DFS_VISIT(sorgente : verticeGrafo*) : void
+Grafo(n : int)
+aggiungiVertice(vertice : verticeGrafo*) : void
+getVertice(numeroVertice : int) : verticeGrafo*
+aggiungiArco(i : int, j : int) : void
+rimuoviArco(i : int, j : int) : void
+risolvi() : int
+DFS() : void
+printGrafo() : void

STUDIO COMPLESSITA’

- ***QUESITO 1***
  Per analizzare le complessità del quesito 1 ci baseremo sulle operazioni principali che andiamo ad effettuare all’interno del nostro programma e sono:
  1. AddEdge(i,j)
  2. RemoveEdge(i,j)
  3. FindEdge(i,j)
  4. DFS(s)

addEdge(i,j), removeEdge(i,j), findEdge(i,j)

CASO PEGGIORE:

I casi peggiori sono due; Il primo caso peggiore della addEdge è quando andiamo effettivamente ad aggiungere il nodo alla lista di adiacenza di i, scorriamo l’intera lista per verificare che non esista il nodo effettivamente e quindi sarà lineare + il tempo di aggiunta del nodo che consideriamo costante. Il secondo caso peggiore è quando l’arco è già esistente ma è l’ultimo all’interno della lista, quindi scorreremo ancora una volta l’intera lista e avremo una complessità lineare n.

Θ(n)

CASO MIGLIORE:

Il caso migliore è quando abbiamo l’arco esistente come primo elemento della lista di adiacenza di i.

O(1)

Per le funzioni successive che si occupano di operazioni sui nodi, le complessità saranno uguali, in quanto la rimozione di un nodo è anch’essa considerata costante e quindi ignorabile, si riduce tutto ad una ricerca all’interno della lista di adiacenza del vertice i.

DFS(s)

La DFS ha un costo totale di θ(V+E) dove V è il numero di vertici del grafo ed E il numero di archi.

- ***QUESITO 2***

  Per il secondo quesito abbiamo due funzioni da analizzare:

  1. DFS()

     Una DFS modificata che rispetto ad una semplice DFS ha un'assegnazione in più, ovvero nelle righe da 120 a 123 che è costante, quindi il costo sarà Θ(V+E) anche in questo caso.

  2. Risolvi()

     Il metodo risolvi non è altro che una funzione che effettua un controllo sul vettore listaVertici, il quale contiene i vertici del grafo. Viene effettuato un incremento ad un contatore che è costante e quindi il costo totale di Risolvi() è θ(V) dove V è il numero di vertici del grafo.

TEST/RISULTATI

- ***QUESITO 1***

Input:

Il file proposto dal professore all'interno dell'archivio:

```
10 21
0 1
0 4
0 2
2 4
4 3
3 1
1 3
3 5
4 5
5 2
1 6
6 5
5 9
7 5
7 9
6 9
9 7
9 8
9 10
8 10
10 8
```

Output:

1. Verifica l'esistenza di un nodo tra quelli dichiarati all'inizio.

```
Inserisci il primo vertice 1
Inserisci il secondo vertice 3

Arco esistente
```

2. Inserimento di un nuovo arco tra due vertici:

```
Inserisci il primo vertice 5
Inserisci il secondo vertice 10

Arco aggiunto con successo.
```

3. Verifica se l'arco aggiunto esiste:

```
Inserisci il primo vertice 5
Inserisci il secondo vertice 10

Arco esistente
```

4. Rimozione di un arco:

```
Inserisci il primo vertice 1
Inserisci il secondo vertice 3

Arco eliminato con successo.
```

5. Verifica se l'arco rimosso esiste ancora:

```
Inserisci il primo vertice 1
Inserisci il secondo vertice 3

Arco non esistente
```

6. Effettua la DFS da un nodo sorgente:

```
Inserisci il nodo sorgente dal quale far partire la DFS 0
{0} {1} {6} {5} {2} {4} {3} {9} {7} {8} {10}
```

- ***QUESITO 2***

Input:

Il file proposto dal professore all'interno dell'archivio:

```
25 24
2 1
2 3
2 5
4 1
4 3
4 5
6 11
7 6
8 6
9 6
10 6
11 12
11 13
11 14
11 15
16 24
17 24
18 24
19 24
20 24
21 24
22 24
23 24
24 25
```

Output:

```
14
```

Input:

Un grafo orientato qualsiasi:

```
5 4
0 1
1 2
5 4
4 3
```

Output:

```
1
```