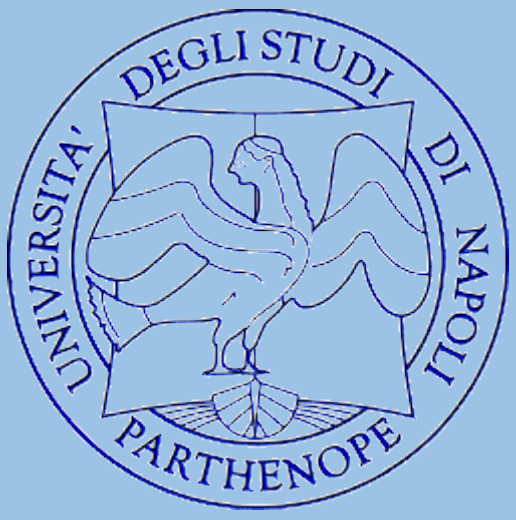
LUCA PEZZOLLA

Matricola 0124002411

RELAZIONE ALGORITMI E STRUTTURE DATI PARTE II



Traccia 2:

Quesito numero 1, “HashGraph”.

• Descrizione problema

Il seguente problema richiede l’implementazione di un Grafo orientato all’interno di una struttura dati Hash Table, in modo tale che ogni nodo del Grafo venga memorizzato in una delle celle della tabella. Lo scopo è quindi quello di unire le due strutture dati, cercando da una parte di mantenere il vantaggio in termini di complessità di tempo dei metodi di una Hash Table, dall’altra quella di sfruttare tali metodi per effettuare operazioni tipiche di un Grafo sui nodi da cui esso è composto.

•Descrizione strutture dati

Le strutture dati utilizzate in questo problema sono rispettivamente il Grafo e la Hash Table. Nello specifico, per quanto concerne la struttura dati Grafo la scelta è stata quella di evitare la creazione di una classe Grafo apposita e si è preferito utilizzare esclusivamente la classe Nodo. Quest’ultima ha al suo interno, come membro privato, un puntatore ad una lista di puntatori a nodi che sta ad indicare la lista di adiacenza del Nodo. La scelta di implementare una lista di adiacenza piuttosto che una matrice di adiacenza è dovuta al risparmio che si ottiene in termini di spazio. I nodi del Grafo sono identificati da un campo informazione di tipo intero, per facilitare il riconoscimento di questi ultimi nel momento in cui vengono allocati e nel momento in cui si vuole operare su di essi. Di conseguenza, il primo nodo allocato sarà identificato dal numero 0 e occuperà la posizione 0 nel vector di nodi, il secondo nodo sarà identificato dal numero 1 e così via. La struttura dati Grafo vera e propria viene quindi rappresentata come un insieme di nodi (contenuti nel vector) e viene assegnata come attributo privato alla classe HashGraph. La struttura dati Hash Table, implementata mediante un’apposita classe, è stata rappresentata come un puntatore ad una lista di Nodi. Tale lista sta ad indicare le singole celle dell’Hash Table, all’interno delle quali verranno memorizzati i Nodi del Grafo mediante funzione di hash, per la quale è stato scelto di utilizzare il metodo della divisione. Il problema delle collisioni è stato conseguentemente risolto mediante il concatenamento.

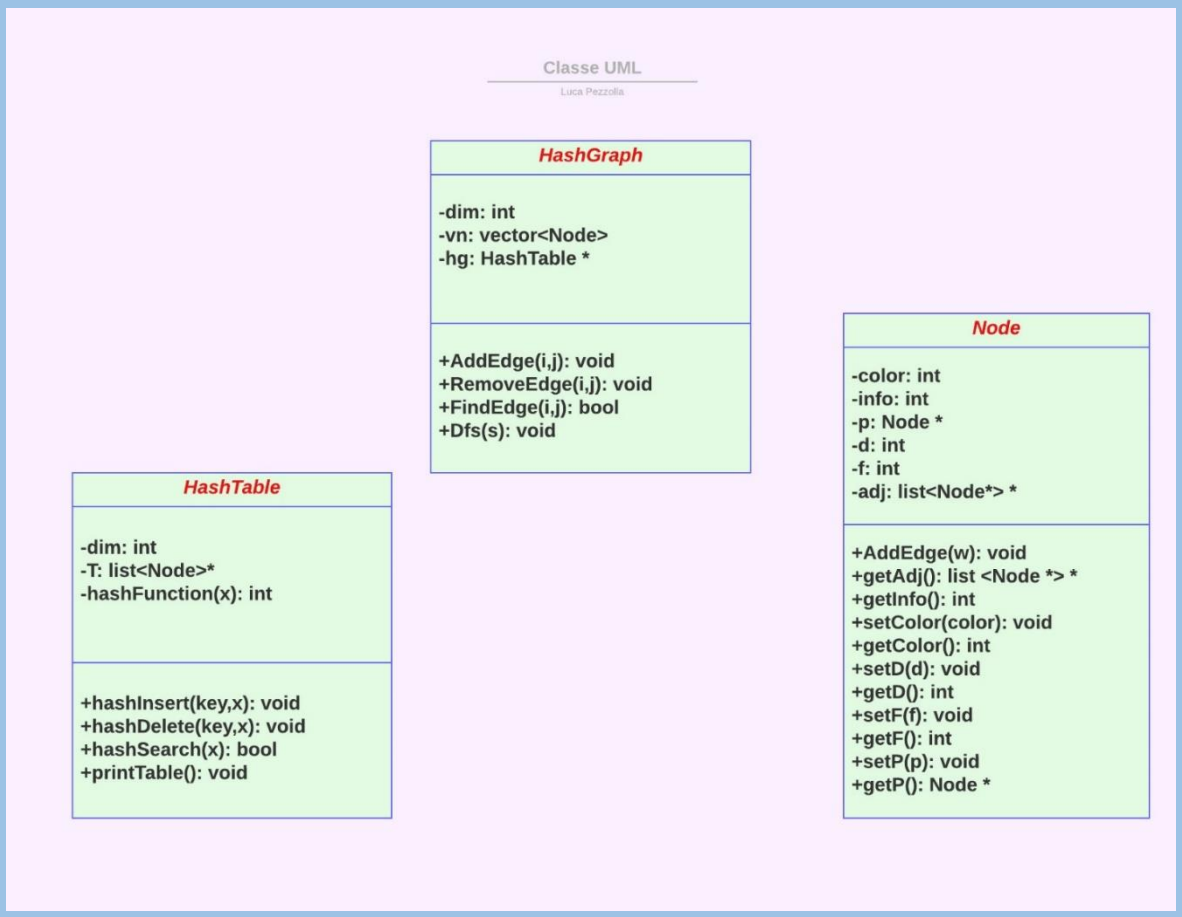
•Formato dati in input/output

In input l'algoritmo prende un file di testo, contenente nella prima riga il numero di nodi del Grafo da allocare ed il numero di archi. Il nodo sorgente e il nodo destinazione sono indicati nelle righe successive. Tali numeri sono sempre separati da uno spazio. In output l'algoritmo genera la corrispondente HashGraph e consente all'utente di effettuare le seguenti operazioni: AddEdge(i,j), RemoveEdge(i,j), FindEdge(i,j) e DFS(s) tramite un apposito menu.

• Descrizione algoritmo

Il primo passo dell'algoritmo è quello di leggere il numero dei nodi e degli archi del Grafo, contenuti nella prima riga del file. Fatto ciò, verrà allocato un vector di nodi di dimensioni pari al numero letto dal file. Il secondo passo è quello di allocare un HashGraph con 701 celle e di salvare, tramite Hash Insert, i nodi del Grafo nelle celle della Hash Table. La Hash Insert sfrutta il metodo della divisione (k mod m) per effettuare i vari inserimenti. Il terzo passo è quello di leggere le successive righe del file per collegare i nodi del Grafo salvati nella Hash Table. Fin quando ci sono nodi da collegare, a patto che siano stati effettivamente allocati e mappati, viene sfruttata la funzione AddEdge(i,j) contenuta in HashGraph. Terminate queste operazioni, l'utente ha a disposizione un menu attraverso il quale può richiamare i metodi sopraccitati della struttura dati HashGraph.

• Class Diagram



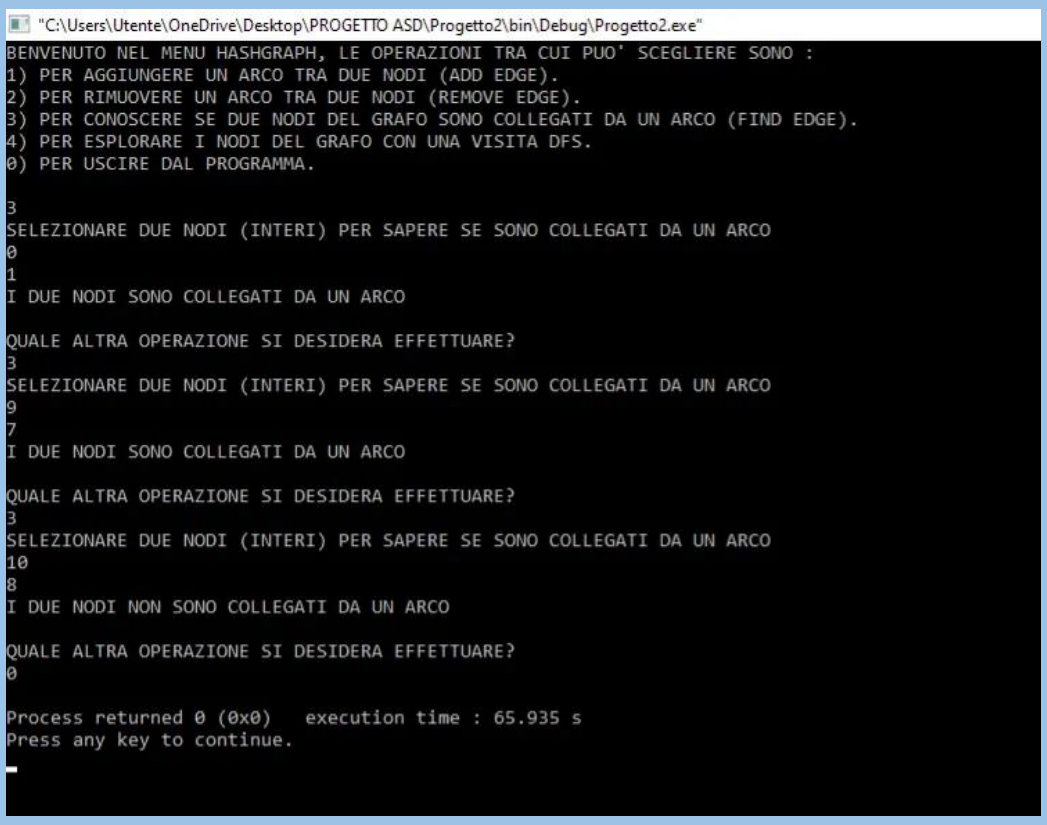
• Studio complessità

L'aspetto vantaggioso dell'utilizzare una struttura dati Hash Table risiede nell'efficienza, in termini di complessità di tempo, dei suoi metodi. In questo specifico caso la scelta è stata quella di utilizzare una Hash Table che sfrutta il concatenamento per risolvere il problema delle collisioni. L'universo delle possibili chiavi, cioè i nodi del Grafo, è al massimo pari a 1000 ed è stato scelto il valore 701 come numero di celle da assegnare alla Hash Table. Tale valore non è ovviamente casuale. Si tratta infatti di un numero primo non troppo vicino ad una potenza del 2, che in questo caso ammette in media 2 collisioni per ogni cella. Per quanto concerne la classe Hash Table, la complessità dei metodi rispecchia quella che ci si aspetterebbe da una struttura dati di questo tipo. Il metodo hashInsert(key,x) sfrutta la funzione di Hash per inserire il nodo del Grafo in tempo O(1). I metodi hashDelete(key,x) e hashSearch(x) invece,

scorrono la lista dei concatenamenti, se quest'ultima è presente, arrivando ad una complessità nel caso peggiore (comunque improbabile) pari ad O(l), dove l è proprio la lunghezza della lista. Il discorso è affine per quanto riguarda i metodi della struttura dati finale HashGraph. In tutti i metodi dell'HashGraph la prima operazione è sempre quella di richiamare la funzione hashSearch(x) per verificare che i nodi su cui si vuole effettuare un'operazione siano effettivamente stati salvati nella Hash Table. I metodi AddEdge(i,j) e RemoveEdge(i,j), ad esempio, sfruttano tale funzione per verificare che i nodi tra cui si vuole aggiungere o rimuovere un arco siano presenti, raggiungendo così una complessità al più lineare. La complessità è lineare anche nel caso del metodo FindEdge(i,j) che non fa altro che scorrere la lista di adiacenza del nodo i e controllare che il nodo j sia presente o meno. La complessità della DFS(s) rimane O(V+E) in quanto non è stata necessaria alcuna modifica rispetto all'algoritmo di partenza.

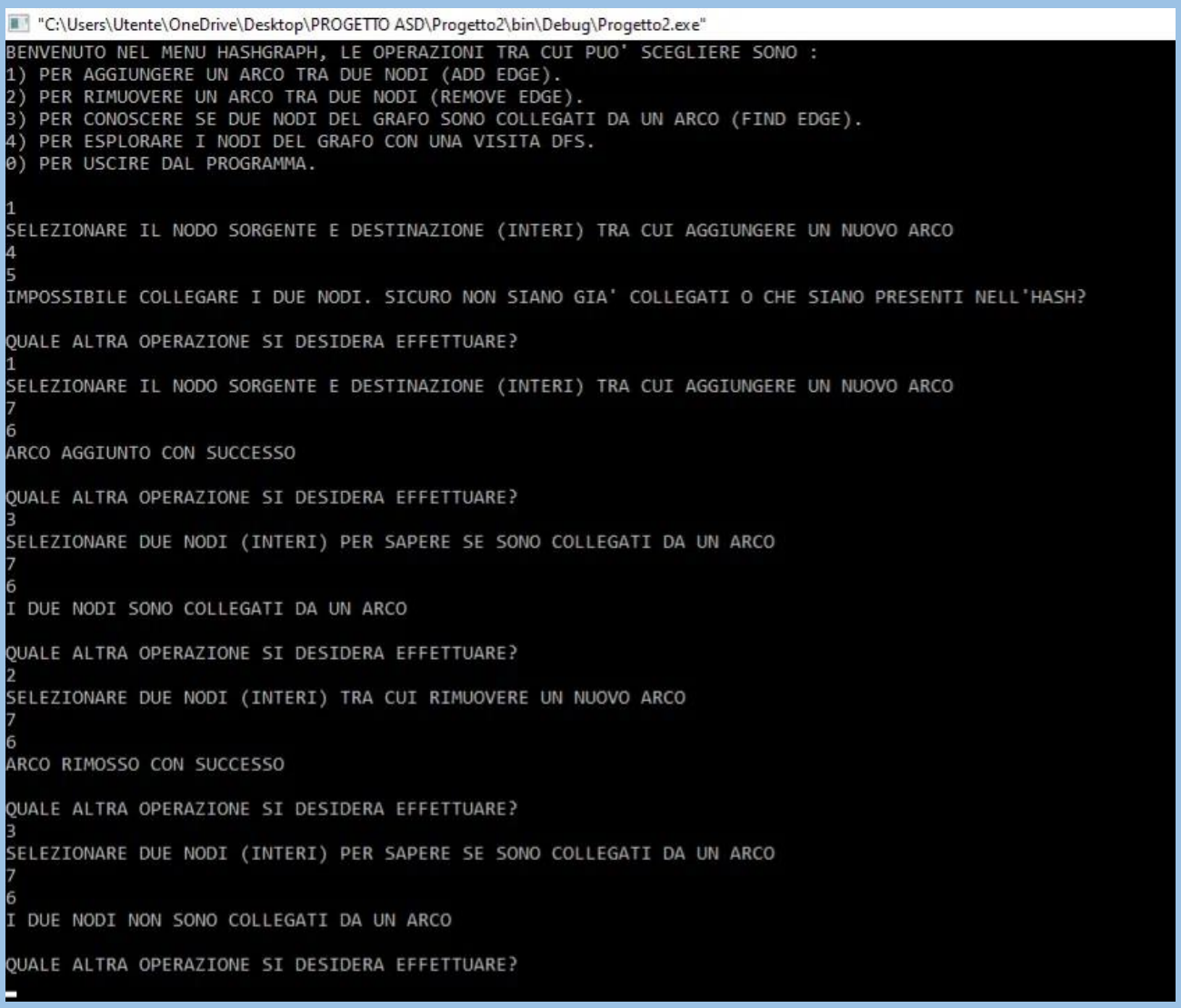
• Test/Risultati

Il seguente test è stato effettuato sul file di input condiviso sulla piattaforma e-learning:

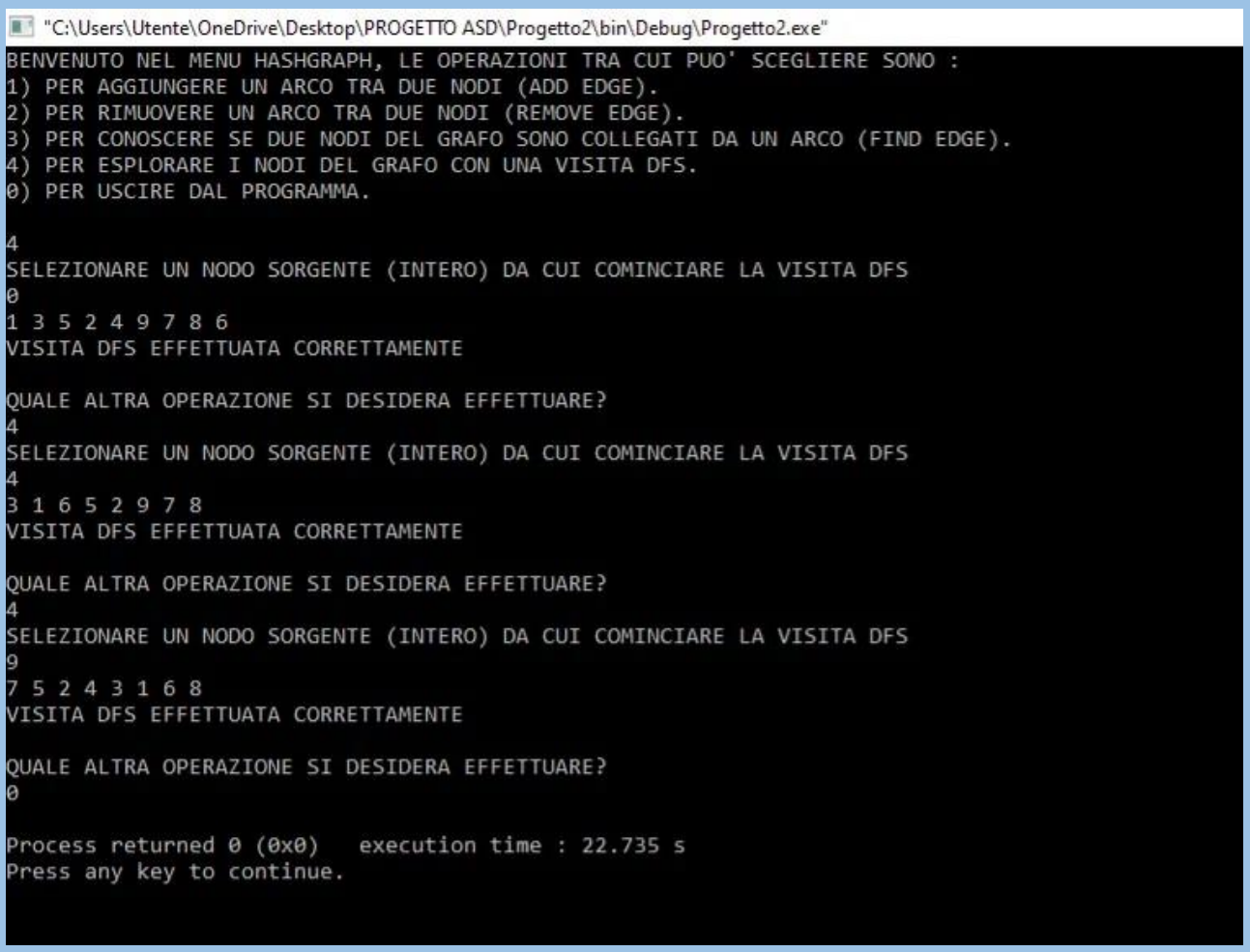


In questo primo screenshot viene effettuato un controllo sulla presenza o meno di un arco, su tre coppie di nodi differenti. Le tre coppie di nodi sono presenti nel

file di input e da come si può osservare, nei primi due casi l'arco viene effettivamente trovato, ma nell'ultimo no. Il motivo è molto semplice: il file chiedeva di allocare 10 nodi del Grafo (da 0 a 9) e di conseguenza il nodo numero 10 non esiste e non può essere effettuata alcuna operazione su quest'ultimo.



In questo secondo screenshot vengono effettuate diverse operazioni. La prima è quella di aggiungere un arco tra i nodi 4 e 5. L'operazione non viene però consentita in quanto i due nodi erano presenti nel file di input e quindi erano già stati collegati. In seguito viene richiamata la AddEdge(i,j) sui nodi 7 e 6 e si verifica che tale arco sia stato effettivamente aggiunto. Dopodiché questo arco viene rimosso e viene richiamata nuovamente la FindEdge(i,j) per verificare che la rimozione sia stata eseguita correttamente.



In questo ultimo screenshot viene testata la DFS(s) su diversi nodi del Grafo. Ciò che il programma stampa sono ovviamente i nodi visitati partendo dal nodo sorgente selezionato. Al termine di tale operazione, il colore di ogni nodo viene riportato nuovamente a 0 (cioè bianco) e il padre è nuovamente nullo. In questo modo l'utente può comunque effettuare una nuova visita DFS, sia selezionando la stessa sorgente di prima, sia con una nuova, a patto che essa sia effettivamente presente nel Grafo e quindi nell'HashGraph.

Quesito numero 2, “Condotte Idriche”.

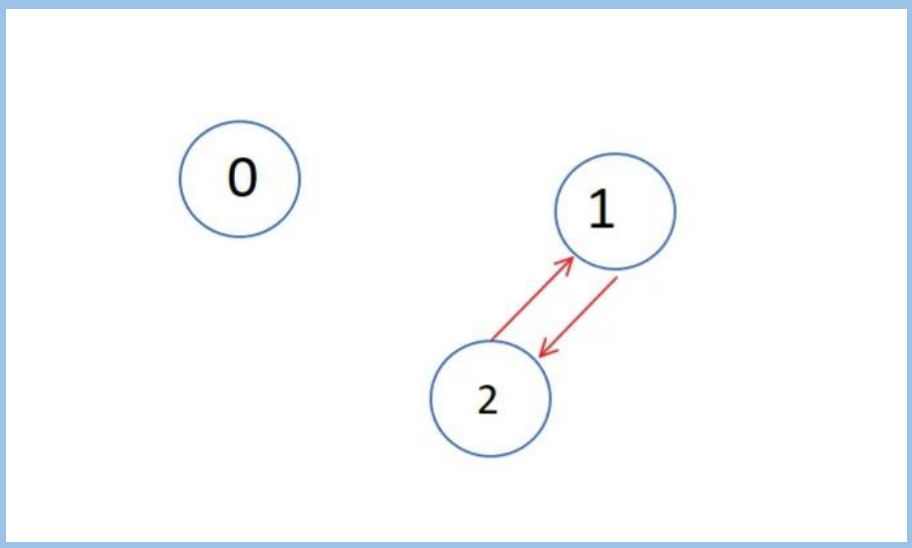
• Descrizione problema

Il seguente problema richiede che tutte le città dello stato di Grapha Nui abbiano la loro condotta idrica in modo da ricevere l’acqua. E’ possibile vedere le singole città come i nodi di un Grafo e le condotte idriche come degli archi. L’obiettivo è quello di costruire il minor numero di condotte in modo tale che ogni città abbia la propria e possa quindi ricevere l’acqua. L’idea di base per risolvere tale problema è stata quella di sfruttare l’algoritmo di visita DFS. Tale idea viene fuori da un quesito molto semplice: Partendo dal nodo zero (che rappresenta il bacino della diga), quanti nodi riesco a visitare portando loro l’acqua? D’altronde, la condizione necessaria affinché un nodo riceva l’acqua è che esso abbia collegato a sé un arco entrante, che in termini di visita DFS si traduce con “tale nodo ha un padre oppure no?”. Di conseguenza, se il nodo avrà padre dopo la visita DFS, non sarà collegato alla sorgente zero, viceversa sì. Tuttavia, l’algoritmo di visita DFS così come lo conosciamo non basta per risolvere questo problema e sono state apportate delle modifiche precise. Gli screen che seguono non rappresentano la soluzione definitiva, bensì solamente il ragionamento che ha poi portato alla soluzione finale. L’algoritmo risolutivo si può trovare nella sezione “Descrizione algoritmo”.

```
void Graph::DFS(Node *s) {
    s->setColor(1);
    for(auto v : *s->getAdj()) {
        v->setP(s);
        if(v->getColor() == 0) {
        DFS_visit(v);
    }
    }
    s->setColor(2);
}
```

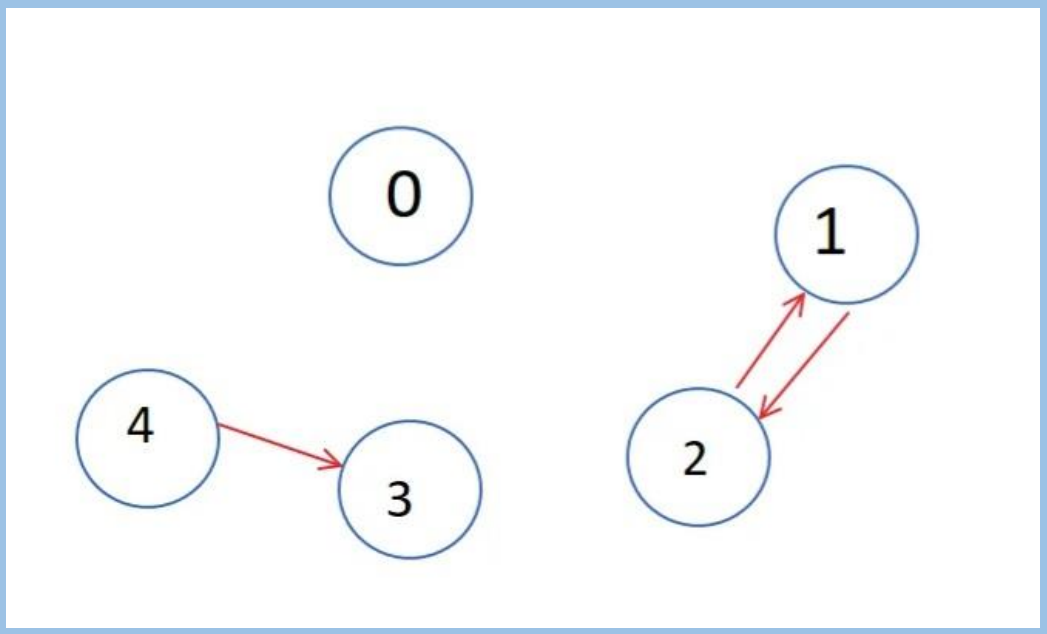
In prima istanza la scelta era stata quella di effettuare l’istruzione “v->setP(s)” al di fuori dell’if. In questo modo ad ogni nodo che avesse un arco entrante veniva assegnato il padre, a prescindere se questo fosse già stato visitato o

meno. Tale soluzione però non può dirsi univocamente corretta, il problema sorge nel caso in cui nel nostro Grafo siano presenti dei cicli, quindi degli archi all'indietro.



In un caso simile a quello mostrato in figura, se venisse fatta una visita DFS mostrata in precedenza partendo dal nodo 0, i nodi 1 e 2 sarebbero padri l'uno dell'altro e nessuno dei due verrebbe quindi collegato al nodo 0. La soluzione precedente non può quindi dirsi corretta in presenza di cicli che non sono raggiungibili dalla sorgente 0.

Una seconda modifica è stata quindi quella di creare una nuova funzione chiamata FindCycle(s) di tipo booleano. Tale funzione veniva richiamata sul nostro Grafo prima di effettuare la visita DFS. Nel caso in cui ritornasse un valore di verità pari a 1 (true) allora veniva effettuata una visita DFS senza alcuna modifica (quindi con assegnazione del padre solo se quel nodo non fosse già stato visitato). Nel caso in cui, invece, la funzione restituisse false, si sarebbe effettuata la visita DFS modificata come mostrato in precedenza, la quale funziona correttamente in assenza di cicli che non sono raggiungibili dalla sorgente. Tale soluzione, però, non teneva conto di un ulteriore problema: gli archi trasversali. Nel caso in cui avessimo avuto un Grafo con archi all'indietro e archi trasversali l'algoritmo di visita DFS non avrebbe funzionato.



In una situazione come quella mostrata in figura, cosa sarebbe accaduto? Da un lato abbiamo i nodi 1 e 2 collegati a vicenda, dall'altra i nodi 4 e 3 collegati da un solo arco. Utilizzando l'algoritmo illustrato in precedenza, la funzione FindCycle(s) avrebbe restituito true in quanto vi è effettivamente un ciclo tra i nodi 1 e 2, e quindi sarebbe stata effettuata una visita DFS normale. Visitando i nodi partendo dalla sorgente 0, però, non avrebbe portato ad una soluzione corretta. L'algoritmo avrebbe restituito un numero di collegamenti maggiori rispetto al necessario. Guardando l'immagine è evidente che basti aggiungere due archi per portare l'acqua ad ogni città. Con una visita DFS, invece, l'algoritmo avrebbe aggiunto tre archi invece che due. Nella foresta DFS avremmo infatti avuto il nodo 0, il nodo 1 e il nodo 2 appartenenti allo stesso albero (con 1 padre di 2), il nodo 3 e per ultimo il nodo 4. I nodi senza padre sarebbero stati quindi tre (1,4,3) e non due, questo perché tra i nodi 3 e 4 vi è un arco trasversale e con una normale visita DFS questi due nodi vengono "assegnati " a due alberi diversi. La soluzione finale tiene conto di tutti gli aspetti illustrati in questo paragrafo.

•Descrizione strutture dati

La struttura dati necessaria per risolvere tale problema è ovviamente un Grafo orientato. La scelta è stata quella di implementare due classi: una classe Nodo e una classe Grafo. La classe Nodo contiene i vari metodi get e set e ha come attributo privato, tra gli altri, un puntatore ad una lista di puntatori a nodi che sta ad indicare la sua lista di adiacenza. La classe Grafo, oltre ad avere una DFS_VISIT, ha come attributo privato un puntatore ad un vector di puntatori a nodi che sta ad indicare l'insieme dei nodi del Grafo su cui andremo a lavorare. I metodi implementati non sono altro che quelli necessari alla risoluzione del problema assegnato.

•Formato dati in input/output

In input l'algoritmo legge da un file di testo le informazioni riguardanti il Grafo orientato che andremo a costruire. Nella prima riga del file sono dichiarati il numero dei nodi e degli archi che andranno a comporre il nostro Grafo. Nelle righe successive, invece, troviamo il nodo sorgente e il nodo destinazione tra cui effettuare i vari collegamenti. In ognuna delle righe del file i due numeri sono sempre separati da uno spazio. In output l'algoritmo restituisce la soluzione al problema, cioè il numero minimo di condotte idriche da costruire e quali condotte debbano effettivamente essere costruite.

• Descrizione algoritmo

Il primo passo dell'algoritmo è quello di leggere il numero di nodi e di archi del nostro Grafo orientato semplicemente utilizzando l'operatore ">>". Il secondo passo è quello di leggere le successive righe del file in modo tale da collegare tra loro i nodi sorgente e destinazione. La lettura del file da questo punto in poi avviene riga per riga e di conseguenza termina o quando non ci sono più righe da leggere, oppure quando sono stati aggiunti tutti gli archi necessari. Quest'ultimo passo è realizzato in maniera molto semplice, in quanto dopo aver letto il numero di archi nel primo passo, l'algoritmo continua a leggere dal file un numero di righe pari al numero di archi da aggiungere, decrementando ad ogni nuovo step questa variabile di tipo intero. In questo modo, se nel file sarà indicato un numero di archi superiore rispetto a quelli esistenti, il file si fermerà non appena non ci saranno più righe da leggere, in caso contrario si fermerà una

volta che la nostra variabile avrà raggiunto un valore pari a zero. Il terzo passo è quello di effettuare una visita DFS su tutti i nodi del nostro Grafo. Per evitare che uno stesso nodo venga visitato più di una volta, viene effettuato un controllo sul colore di quest'ultimo. Se esso risulta effettivamente uguale a zero, cioè bianco, la visita verrà effettuata, altrimenti no. La visita DFS risulta essere leggermente modificata rispetto a quella che conosciamo. Essa agisce in base al tipo di arco che stiamo analizzando. Si parte con una visita DFS dal nodo 0 fino ad arrivare a tutti gli altri nodi che costituiscono il nostro Grafo di input. Entrati nella DFS si scorre ovviamente la lista di adiacenza del nodo sorgente passato per argomento nel main. Dobbiamo tener conto di tre casi differenti:

1) Se siamo di fronte ad un nodo bianco e quindi ad un arco dell'albero, l'algoritmo si comporta come una normale visita DFS. Viene fatta l'assegnazione del padre e successivamente si visitano i nodi raggiungibili da quest'ultimo nella DFS_Visit (nella quale viene fatto lo stesso identico controllo).

2) Se siamo di fronte ad un nodo nero appartenente ad un altro albero, quindi un arco trasversale, l'algoritmo effettua esclusivamente l'assegnazione del padre (assegnazione fondamentale in alcuni casi, come spiegato nel paragrafo "Descrizione problema") senza entrare nella DFS_Visit in quanto quel nodo era già stato visitato in precedenza.

3) Se siamo di fronte ad un nodo grigio e quindi ad un arco all'indietro, l'algoritmo non effettua nessuno dei passaggi precedenti. Non viene fatta alcuna assegnazione del padre e il nodo non viene visitato nella DFS_Visit. L'unica operazione che viene fatta è quella di impostare il suo colore pari a 2, cioè nero.

Fatto ciò, l'ultimo passo dell'algoritmo è quello di scorrere un'ultima volta i nodi del Grafo e controllare quali nodi hanno effettivamente padre o meno (in verità viene fatto anche un banale controllo sulla presenza di un cappio, nonostante un arco del genere sia da considerarsi irreale in un problema di questo tipo). Se un nodo non ha padre viene collegato al nodo 0 e viene incrementato un contatore. Alla fine verranno stampati i nodi che sono stati collegati al bacino insieme al numero di condotte effettivamente aggiunte. Quest'ultimo valore rappresenta ovviamente il minor numero di condotte idriche da costruire per portare l'acqua ad ogni nodo del nostro Grafo di input.

• Class Diagram

• Studio complessità

La complessità di tale algoritmo è dominata dalla complessità della visita DFS, cioè O(V+E) dove V rappresenta il numero di vertici/nodi visitati, mentre E rappresenta il numero di Archi (Edges). Nonostante la visita sia stata leggermente modificata, l'algoritmo opera comunque in modo da visitare ogni nodo un'unica volta. L'ultimo ciclo for, utilizzato per scorrere tutti i nodi del Grafo, ha anch'esso una complessità lineare. Al suo interno viene fatto esclusivamente un controllo sui padri e sulla presenza di cappi, per ogni singolo nodo. In seguito, se tale controllo ha esito positivo (quindi un nodo non ha padre oppure è padre di se stesso), viene richiamata la funzione *getNode(v) e successivamente la funzione addEdge(w), entrambe con complessità pari ad un O(1). La prima funzione restituisce il nodo richiesto che si trova nel vector di nodi. La seconda funzione, invece, aggiunge il nodo destinazione alla fine della lista di adiacenza del nodo sorgente.

• Test/Risultati

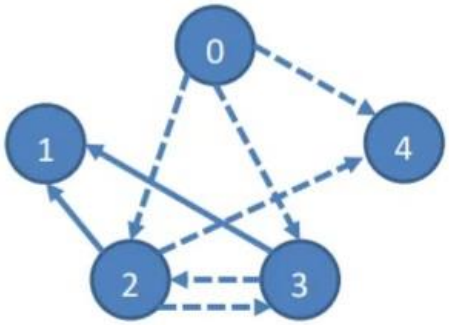
I seguenti test sono stati effettuati rispettivamente sui file presenti su e-learning. I primi due appartengono alla traccia del problema, l'ultimo invece al file di test caricato nella cartella "FileTest".

-Test numero 1:

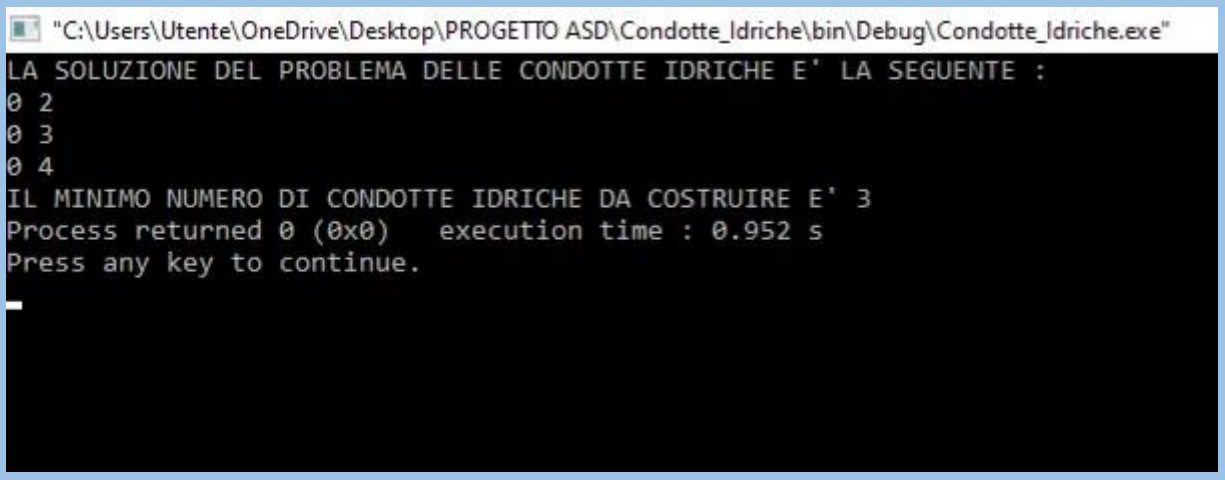
Grafo fornito in input attraverso il file di testo:

Esempio

input.txt	output
4 2 3 1 2 1	3 0-4 0-3 0-2 oppure 0-4 0-3 3-2 oppure 0-4 0-2 2-3 oppure 0-3 3-2 2-4



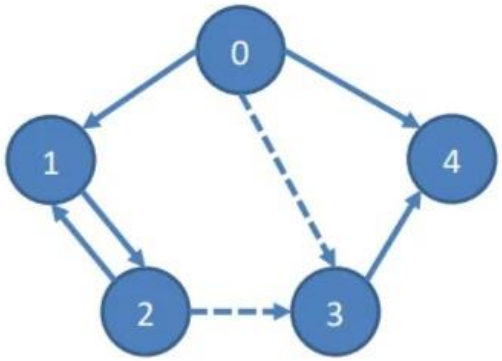
Soluzione dell'algoritmo:



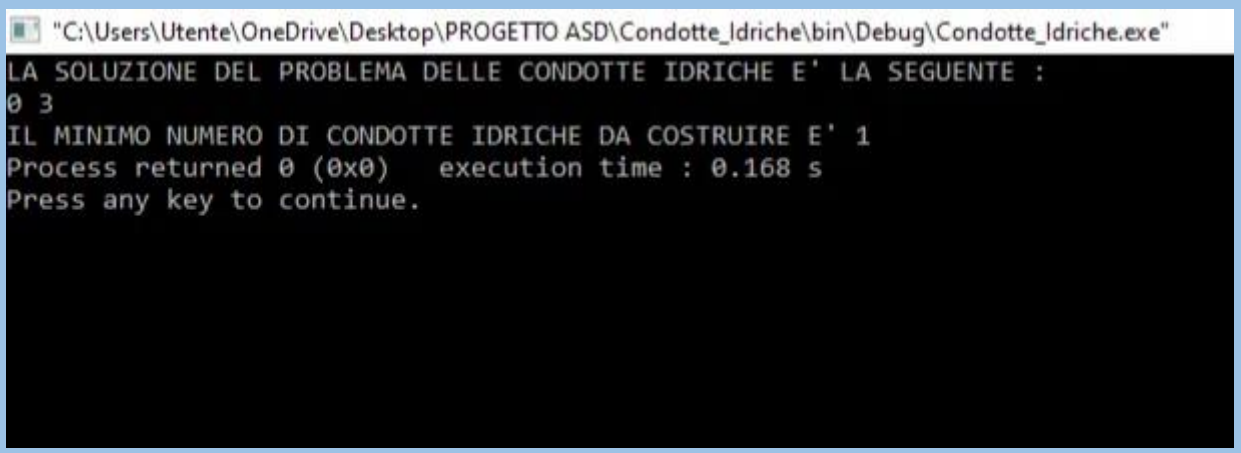
-Test numero 2:

Grafo fornito in input attraverso il file di testo:

input.txt	output
4 5 0 1 1 2 2 1 0 4 3 4	1 2-3 oppure 0-3

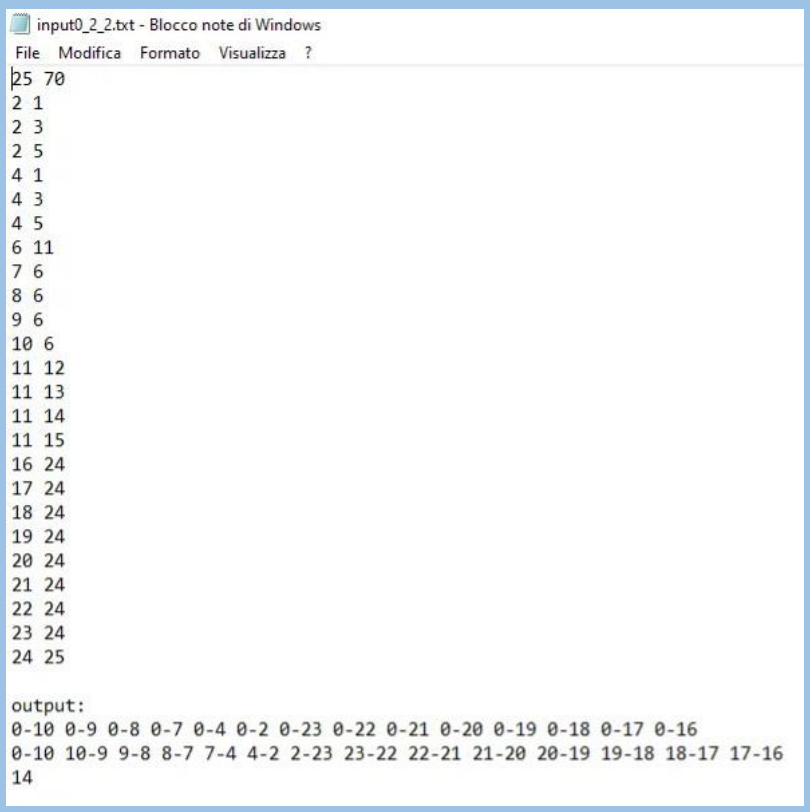


Soluzione dell'algoritmo:



-Test numero 3:

Grafo fornito in input attraverso il file di testo:



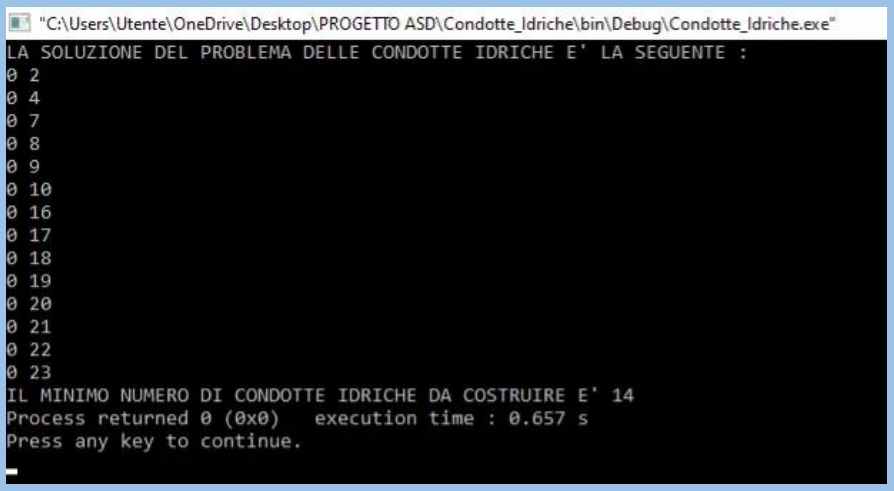
input0_2_2.txt - Blocco note di Windows

File Modifica Formato Visualizza ?

```
25 70
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:
0-10 0-9 0-8 0-7 0-4 0-2 0-23 0-22 0-21 0-20 0-19 0-18 0-17 0-16
0-10 10-9 9-8 8-7 7-4 4-2 2-23 23-22 22-21 21-20 20-19 19-18 18-17 17-16
14
```

Soluzione dell'algoritmo:



"C:\Users\Utente\OneDrive\Desktop\PROGETTO ASD\Condotte_Idriche\bin\Debug\Condotte_Idriche.exe"

```
LA SOLUZIONE DEL PROBLEMA DELLE CONDOTTE IDRICHE E' LA SEGUENTE :
0 2
0 4
0 7
0 8
0 9
0 10
0 16
0 17
0 18
0 19
0 20
0 21
0 22
0 23
IL MINIMO NUMERO DI CONDOTTE IDRICHE DA COSTRUIRE E' 14
Process returned 0 (0x0)   execution time : 0.657 s
Press any key to continue.
```