

Università degli studi di Napoli Parthenope



Dipartimento di Scienze e Tecnologie - Laurea in Informatica

Progetto di Algoritmi e Strutture Dati

Giovanni D'Angelo

0124001369

11/10/2019

Dipartimento di Scienze e Tecnologie - Laurea in Informatica	1
Introduzione della traccia - Numeri Amicabili	3
Strutture dati utilizzate	3
Diagramma delle Classi	4
Implementazione	5
Header.h	5
Function.cpp	6
Main.cpp	9
Output del programma	12
Analisi della complessità computazionale	13
Introduzione della traccia - Il Convegno dell'Amicizia	14
Strutture dati utilizzate	14
Diagramma delle classi	15
Specifiche sull'algoritmo	16
Implementazione	16
Header.h	16
Function.cpp	18
Main.cpp	24
Output del programma	25
Analisi della complessità computazionale	27

Introduzione della traccia - Numeri Amicabili

Traccia:

“Due numeri si dicono amicabili se il primo numero è uguale alla somma dei divisori del secondo ed il secondo numero è uguale alla somma dei divisori del primo. (es.: (220 e 284), (1184 e 1210)). Dato un array di interi trovare le coppie di numeri amicabili, utilizzando una hash table implementata con il metodo dell'indirizzamento aperto con doppio hashing.”

Descrizione:

Per questo primo quesito viene richiesto un algoritmo il quale, dato in input una HashTable ht e un numero x , si verifichi che $x \in ht$ e che ci sia un numero $y \in ht$ tale che la somma dei divisori di x sia uguale a y e che la somma dei divisori di y sia a sua volta uguale ad x , quindi x ed y costituiscono una coppia amicabile. La traccia richiede di trovare tutti i numeri amicabili all'interno di una HashTable.

Strutture dati utilizzate

Come richiesto dalla traccia, viene utilizzata una HashTable, una struttura dati molto utile per l'operazione più frequente che verrà effettuata nell'algoritmo, cioè la ricerca. In particolare, l'HashTable viene utilizzata quando il numero delle chiavi utilizzate K è molto più piccolo dell'universo delle chiavi possibili U poiché richiede spazio di memoria $\Theta(|K|)$ e non $\Theta(|U|)$. È stata implementata con il metodo dell'indirizzamento aperto: Ogni cella all'interno della tabella contiene un elemento oppure NULL.

Per l'inserimento, cancellamento o ricerca di un elemento, viene utilizzata una funzione di hash, in particolare il doppio hashing, il quale prende in input la chiave k e l'indice di partenza dell'ispezione i .

$$\bullet h(k, i) = (h1(k) + i \cdot h2(k)) \bmod m$$

dove $h1$ e $h2$ sono funzioni di hash ausiliarie, m è la grandezza della nostra HashTable.

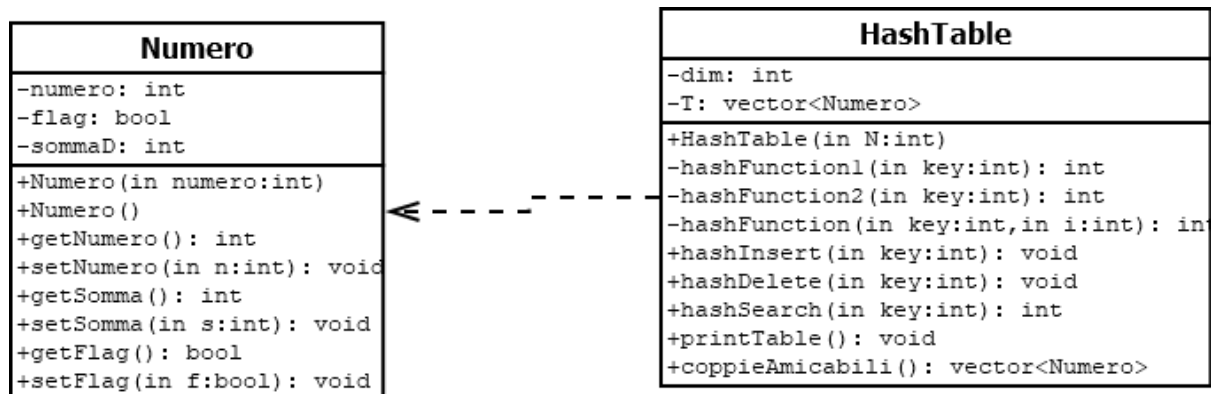
$$\bullet h1(k) = k \bmod m$$

$$\bullet h2(k) = 1 + \bmod{m-1}$$

Il valore $h2(k)$ deve essere relativamente primo con m perché sia ispezionata l'intera hash table.

Diagramma delle Classi

Qui riportiamo il diagramma delle classi realizzato per implementare l'algoritmo di ricerca dei numeri amichevoli, in modo da sfruttare l'approccio Object Oriented offerto dal C++.



La classe **Numero** identifica l'oggetto memorizzato all'interno della **HashTable**. L'oggetto **Numero** ha due costruttori, uno che non richiede input ed uno che richiede un intero. Ogni **Numero** ha come parametri "numero" che è il suo identificativo, "sommaD" che rappresenta la somma dei suoi divisori ed infine flag, un booleano che vale true se l'oggetto è stato istanziato nella **HashTable** e false altrimenti. Il costruttore senza parametri viene utilizzato come costruttore di default.

Quando viene istanziato imposta i valori di numero, flag e sommaD con valori predefiniti e viene utilizzato alla creazione della **HashTable**, la quale sarà completamente riempita da questi numeri in particolare flag sarà impostata a false, in modo da indicare che la cella occupata da questo **Numero** è in realtà libera. Il secondo costruttore prende in input un intero N che sarà direttamente salvato nell'attributo "numero" e rappresenterà il numero salvato.

Quando viene istanziato un oggetto **Numero**, viene anche chiamata la funzione `CalcoloDivisori()`, il quale risultato sarà salvato nella variabile `sommaD`. Inoltre, flag viene settata a true. La classe **HashTable** usa un solo costruttore che prende come input m che corrisponde al size della **HashTable**. Quando viene istanziata una **HashTable**, viene creato anche un vector¹ di Numeri con size m.

¹ Il tipo di dati vector è fornito dalla STL del C++

Implementazione

Qui di seguito viene mostrato il codice per implementare queste classi e relativi metodi, con eventuali commenti dove necessario.

Header.h

Questo file contiene lo scheletro delle classi e l'intestazione delle funzioni.

```
1. #ifndef HEADER_H
2. #define HEADER_H
3. #include <iostream>
4. #include <vector>
5. using namespace std;
6.
7. //Scheletro classi
8. class Numero {
9. private:
10.     int numero;
11.     int sommaD; //Somma dei divisori del numero
12.     bool flag; //Flag identifica se il numero è stato istanziato o meno nella
        Hash table.
13.     //Un oggetto eliminato ha comunque flag positiva in modo da poter proseguire
        nella ricerca dopo un eventuale cancellazione.
14. public:
15.     Numero(int num);
16.     Numero();
17.     ~Numero(){};
18.     int getNumero();
19.     int getSomma();
20.     bool getFlag();
21.     void setFlag(bool f);
22.     void setNum (int n);
23.     void setSomma(int s);
24. };
25.
26. //////////////////////////////////////
27.
28. class HashTable{
29.     private:
30.         int dim;
31.         vector<Numero> T;
32.         int hashFunction1(int key); //Prima funzione di hash
33.         int hashFunction2(int key); //Seconda funzione di hash (Per iterare)
34.         int hashFunction (int key, int i); //Funzione di hash utilizzata
        effettivamente
35.     public:
36.         HashTable(int m);
37.         ~HashTable(){};
38.         void hashInsert(int key);
39.         void hashDelete(int key);
40.         int hashSearch(int key);
41.         void printTable();
42.         vector<Numero> coppieAmicabili(); //Trovo i numeri amicabili all'interno
        della Hashtable
43. };
44.
45. int CalcoloDivisori(int num); //Dato un numero, ne calcola la somma dei divisori
46.
47. #endif
```

Function.cpp

In questo file sono implementate i metodi delle classi ed in particolare l'algoritmo richiesto dalla traccia.

```
1. #include "Header.h"
2. Numero::Numero(int num){
3.
4.     numero = num;
5.     sommaD = CalcoloDivisori(num);
6.     flag = true; // Il numero è istanziato nella Hashtable
7. };
8.
9. Numero::Numero()
10. {
11.     numero = -1; //Valori fittizzi usati per gli elementi eliminati
12.     sommaD = -1;
13.     flag=false; //il numero non è istanziato, la locazione è libera
14. };
15.
16. int Numero::getNumero()
17. {
18.     return numero;
19. }
20.
21. int Numero::getSomma()
22. {
23.     return sommaD;
24. }
25.
26. bool Numero::getFlag()
27. {
28.     return flag;
29. }
30.
31. void Numero::setFlag(bool f)
32. {
33.     flag = f;
34. }
35.
36. void Numero::setNum (int n)
37. {
38.     numero = n;
39. }
40.
41. void Numero::setSomma(int s)
42. {
43.     sommaD = s;
44. }
45.
46. //////////////////////////////////////
47.
48. int HashTable::hashFunction1(int key)
49. {
50.     return (key % dim);
51. }
52.
53. int HashTable::hashFunction2(int key)
54. {
55.     return (1+ key%dim-1);
56. }
57.
58. int HashTable::hashFunction (int key, int i)
```

```

59. {
60.     return ((hashFunction1(key)+ i*hashFunction2(key))%dim);
61. }
62.
63. HashTable::HashTable(int m)
64. {
65.     dim=m; T = vector<Numero>(dim);
66. }
67.
68. vector<Numero> HashTable::coppieAmicabili()
69. {
70.     int index,i;
71.     vector<Numero> risultato;
72.     for (i=0; i<dim;i++) //Controllo tutta l'hash Table
73.     {
74.         if (T[i].getNumero() > 0) //Salto gli elementi vuoti
75.         {
76.             index = hashSearch(T[i].getSomma());
77.             if (index != -1 && index != i) //Se la search ha dato risultato e la
somma dei divisori è diversa dal numero stesso
78.             {
79.                 if(T[index].getSomma() == T[i].getNumero() ) //Se la somma del
numero precedentemente trovato è uguale al numero di indice i
80.                 {
81.                     risultato.push_back(T[i]); //Salva la coppia nel vettore
risultato
82.                 }
83.             }
84.         }
85.     }
86.     return risultato;
87. }
88. }
89.
90. void HashTable::hashInsert(int key)
91. {
92.     int index = hashSearch(key); //Prima di inserire controllo se l'elemento è
inserito
93.     if (index != -1)
94.     {
95.         return; //In caso di duplicati, non inserisco
96.     }
97.     int i = 0;
98.     int j;
99.     Numero n(key);
100.    while (i <= dim)
101.    {
102.        j = hashFunction(key,i);
103.        if (T[j].getNumero() < 1) //Se la cella è vuota (o è stata
cancellata)
104.        {
105.            T[j] = n; //Salva il numero nella Hashtable
106.            i=dim+1; //Esco dal while
107.        }
108.        else
109.            i=i+1;
110.    }
111. }
112. }
113.
114. void HashTable::hashDelete(int key)
115. {
116.     int index = hashSearch(key); //Cerco l'indice nella mia Hashtable

```

```

117.     if (index != -1) //Se la search ha dato risultato
118.     {
119.         T[index].setSomma(-1); //Setto valori fittizzi in modo da non
            considerarli in eventuali ricerche/stampe
120.         T[index].setNum(-1);
121.         T[index].setFlag(true);
122.         cout << "elemento eliminato!" << endl;
123.     }
124.     else
125.     {
126.         cout << "l'elemento non e' stato trovato" << endl;
127.     }
128. }
129.
130. int HashTable::hashSearch(int key)
131. {
132.     int i = 0;
133.     int j= hashFunction1(key);
134.     while (T[j].getFlag() == true || i <= dim)
135.     {
136.         j= hashFunction(key,i);
137.         if (T[j].getNumero() == key && T[j].getFlag() != false ) //Se il numero
            c'è e non è stato rimosso
138.             return j; //ritorno l'indice di posizione
139.         i=i+1;
140.     }
141.     return -1; //l'elemento cercato non c'è
142. }
143.
144. void HashTable::printTable()
145. {
146.     int i =0;
147.
148.     for (i =0; i<T.size();i++)
149.     {
150.         if (T[i].getFlag() == true && T[i].getNumero() > 0) //Stampa solo
            elementi presenti e non cancellati
151.         {
152.             cout << T[i].getNumero() << "--> Somma = " << T[i].getSomma();
153.             cout << endl;
154.         }
155.     }
156. }
157.
158. //////////////////////////////////////
159.
160. int CalcoloDivisori(int num)
161. {
162.     int resto,somma=1;
163.     for (int i=2;i<=num/2;i++) //Per tutti i numeri da 2 a num/2 (la somma è
        inizializzata ad 1, non è necessario sommare 1)
164.     {
165.         resto=num%i; //calcolo il resto
166.         if(resto==0) //se il resto e' 0 allora
167.             {somma=somma+i;} // "i" è un divisore e va aggiunto
168.     }
169.     return somma;
170. }

```

Diamo uno sguardo ravvicinato a due delle funzioni più importanti, `CalcoloDivisori()` e `CoppieAmicabili()`.

CalcoloDivisori() viene richiamato ogni volta che viene istanziato un Numero. Questo algoritmo prende in input un numero “num” e ne calcola la somma dei divisori. L’idea è semplicemente di iterare tutti i numeri “i” da 2 a num/2. Se l’operazione num modulo i restituisce 0 allora questo è un divisore e deve essere sommato al risultato in output.

CoppieAmicabili() viene considerato come metodo della HashTable. L’idea è quella di scorrere una sola volta l’HashTable e, per ogni elemento “i” diverso da NIL², si effettua una ricerca nella HashTable inserendo come parametro di input la somma dei divisori di “i”, salvandolo nella variabile “index”.

Index avrà valore -1 se la ricerca non sarà andata a buon fine oppure avrà il valore dell’indice della HashTable contenente un potenziale numero amicabile³. Ora non resta che verificare che la somma dei divisori di “index” sia uguale al numero in “i”. se ciò è verificato, sia il numero conservato in index che in i saranno numeri amicabili ed “i”⁴ verrà messo in un vettore che conserva tutte le coppie trovate fino a quel momento.

Main.cpp

```
1. #include "Functions.cpp"
2. #define M 10007
3.
4. int main()
5. {
6.     int i, scelta,n,numero,temp;
7.     vector<Numero> risultato;
8.
9.     HashTable ht(M);
10.    cout << "Seleziona la modalità:" << endl;
11.    cout << "1) Visualizza i numeri amicabili da 1 ad n" << endl;
12.    cout << "2) Inserisci n numeri da tastiera"<<endl;
13.    cin >> scelta;
14.    switch(scelta) {
15.
16.    case 1 :
17.
18.        cout << "Inserire fino a che numero calcolare i numeri amicabili (Max
10000)"<< endl;
19.        cin >> n;
20.        cout << "Calcolo amicabili da 1 a " << n<< "..." << endl;
21.        for (i=0;i<=n;i++)
22.        {
23.            ht.hashInsert(i);
24.        }
25.        cout << "coppie :" << endl;
26.        risultato=ht.coppieAmicabili();
27.        for (i=0;i<risultato.size();i++)
28.        {
```

² In realtà una cella di HashTable non istanziata corrisponde qui ad un oggetto Numero con “numero” e “SommaD” pari a -1.

³ Index potrebbe effettivamente essere uguale ad “i”, come per esempio nel numero 6, il quale è effettivamente uguale alla sua somma dei divisori. Di conseguenza, viene implicato che index debba essere diverso da i.

⁴ Facendo così si incontreranno inevitabilmente un doppione per ogni numero amicabile. In questo progetto si è scelto di non considerarlo come un vero problema e di conservarne entrambe le coppie, es: “220 e 284”, “284 e 220” sono considerate come due coppie .

```

29.         cout << risultato[i].getNumero() << "-->
" << risultato[i].getSomma()<<endl;
30.     }
31.     cout << "Programma terminato" << endl;
32.     break;
33.
34.     case 2 :
35.         cout << "Quanti numeri vuoi inserire?" << endl;
36.         cin >> n;
37.         cout << "Inserire " << n << " numeri" << endl;
38.         for (i=0;i<n;i++)
39.         {
40.             cin >> temp;
41.             ht.hashInsert(temp);
42.         }
43.         while (1)
44.         {
45.             cout << "Cosa vuoi fare?" << endl;
46.             cout << "1)Vedi coppie amicabili" << endl;
47.             cout << "2)Rimuovi numeri dalla tabella" << endl;
48.             cout << "3)Inserisci numero nella tabella" << endl;
49.             cout << "4)Visualizza la tabella" << endl;
50.             cout << "5)Esci" << endl;
51.             cin >> scelta;
52.             switch(scelta) {
53.
54.                 case 1 :
55.                     risultato=ht.coppieAmicabili();
56.                     for (i=0;i<risultato.size();i++)
57.                     {
58.                         cout << risultato[i].getNumero() << "-->
" << risultato[i].getSomma()<<endl;
59.                     }
60.                     break;
61.
62.                 case 2 :
63.                     cout << "Che numero vuoi eliminare?" <<endl;
64.                     cin >> numero;
65.                     ht.hashDelete(numero);
66.                     break;
67.                 case 3 :
68.                     cout << "Inserisci nuovo numero" << endl;
69.                     cin >> numero;
70.                     ht.hashInsert(numero);
71.                     break;
72.                 case 4 :
73.                     cout << "Stampo la tabella:"<< endl;
74.                     ht.printTable();
75.                     break;
76.                 case 5:
77.                     return 0;
78.                 default :
79.                     cout << "Scelta non valida"<< endl;
80.             }
81.         }
82.         break;
83.
84.     default :
85.         cout << "Scelta non valida, programma in chiusura" << endl;
86.         return 0;
87.     }
88. }

```

Al fine di testare il corretto funzionamento dell'algoritmo, si è scelto di ipotizzare che l'array di interi presi in input fossero i primi n numeri naturali⁵

Il main è composto da un semplice menù che consente di fare due cose.

- 1) Visualizzare i primi N numeri amicabili⁶
- 2) Gestire la hashtable

La prima operazione è abbastanza semplice, una volta inserito N verranno inseriti nella HashTable tutti i numeri naturali da 1 a N , per poi eseguire `CoppieAmicabili()` sulla tabella di Hash. Una volta fatto, verranno stampate tutte le coppie amicabili trovate, per poi terminare il programma.

La seconda operazione viene utilizzata nel caso si vogliano inserire determinati numeri per verificare se sono o meno amicabili. Una volta selezionata la seconda opzione infatti si potrà:

- 1) Vedere le coppie amicabili dei numeri inseriti fino a quel momento.
- 2) Eliminare un numero nella hashtable.
- 3) Inserire un nuovo numero.
- 4) Stampare i numeri registrati fino a quel momento
- 5) Uscire dal programma.

⁵ La scelta è stata puramente arbitraria ed al fine di rendere il testing più veloce. Ciò non preclude la possibilità di utilizzare l'algoritmo per input diversi scelti dall'utente attraverso la gestione della hashtable.

⁶ Su un massimo di $M=10007$ elementi. Questa scelta è arbitraria, è stato scelto di creare una HashTable con questo size per comodità nella ricerca di un intero abbastanza grande e primo.

Output del programma

caso 1, visualizzazione dei primi 5000 numeri naturali:

```
giovanni@giovanni-VirtualBox: ~/Scrivania/asd/NumeriAmicabili
File Modifica Visualizza Cerca Terminale Aiuto
giovanni@giovanni-VirtualBox:~/Scrivania/asd/NumeriAmicabili$ ./main
Seleziona la modalit :
1) Visualizza i numeri amicabili da 1 ad n
2) Inserisci n numeri da tastiera
1
Inserire fino a che numero calcolare i numeri amicabili (Max 10000)
```

```
giovanni@giovanni-VirtualBox: ~/Scrivania/asd/NumeriAmicabili
File Modifica Visualizza Cerca Terminale Aiuto
giovanni@giovanni-VirtualBox:~/Scrivania/asd/NumeriAmicabili$ ./main
Seleziona la modalit :
1) Visualizza i numeri amicabili da 1 ad n
2) Inserisci n numeri da tastiera
1
Inserire fino a che numero calcolare i numeri amicabili (Max 10000)
5000
Calcolo amicabili da 1 a 5000...
coppie :
220--> 284
284--> 220
1184--> 1210
1210--> 1184
2620--> 2924
2924--> 2620
Programma terminato
giovanni@giovanni-VirtualBox:~/Scrivania/asd/NumeriAmicabili$
```

caso 2, verifica di n numeri inseriti da tastiera e gestione della HashTable:

```
giovanni@giovanni-VirtualBox: ~/Scrivania/asd/NumeriAmicabili
File Modifica Visualizza Cerca Terminale Aiuto
giovanni@giovanni-VirtualBox:~/Scrivania/asd/NumeriAmicabili$ ./main
Seleziona la modalit :
1) Visualizza i numeri amicabili da 1 ad n
2) Inserisci n numeri da tastiera
2
Quanti numeri vuoi inserire?
3
Inserire 3 numeri
220
286
284
Cosa vuoi fare?
1)Vedi coppie amicabili
2)Rimuovi numeri dalla tabella
3)Inserisci numero nella tabella
4)Visualizza la tabella
5)Esci
1
220--> 284
284--> 220
Cosa vuoi fare?
1)Vedi coppie amicabili
2)Rimuovi numeri dalla tabella
3)Inserisci numero nella tabella
4)Visualizza la tabella
5)Esci
```

Analisi della complessità computazionale

Prendiamo in considerazione per il calcolo della complessità solo le operazioni che rientrano in gioco nella funzione `CoppieAmicabili()` ed il calcolo della somma dei divisori.

Ogni volta che si vuole inserire un Numero n si calcolerà la somma dei suoi divisori, effettuando un ciclo `for` che farà esattamente $n/2$ confronti, rendendo il costo per l'inserimento pari a $O(n/2) = O(n)$ + costo inserimento effettivo di n .

Fortunatamente, dovendo inserire i numeri fino ad un massimo di $N < m$, l'inserimento e la ricerca all'interno della tabella di Hash con indirizzamento aperto avranno complessità pari ad $O(1)$, poiché non incontreranno collisioni⁷. In particolare, la prima operazione che verrà effettuata sarà una ricerca con la medesima chiave (con complessità pari ad $O(1)$). Se la ricerca andrà a buon fine, allora si tratterebbe di un duplicato da non inserire, altrimenti si può effettuare l'inserimento, sempre con costo $O(1)$. Il costo quindi per inserimento è quindi costante, ed essendo eseguito per ogni elemento sarà uguale ad $O(N)$.

Quindi il costo per il calcolo dei divisori e l'inserimento è $O(N^2)$.

Ora passiamo al calcolo delle complessità per la funzione `CoppieAmicabili`.

Avendo a disposizione una HashTable con le somme dei divisori già calcolate, l'algoritmo si limiterà ad effettuare una ricerca con costo $O(1)$ per ogni elemento all'interno della HashTable facendo inoltre due addizionali confronti: $O(1) + O(1) + O(1) = O(1)$, il quale verrà eseguito m volte⁸, quindi la complessità è $O(m)$.

Il costo totale dell'algoritmo sarà uguale ad $O(N^2 + m)$.

⁷ Stiamo considerando l'ipotesi per cui l'hash Table sia molto più grande dei numeri da inserire. Se così non fosse, ci sarebbero collisioni, rendendo il costo della ricerca non più costante.

⁸ Grandezza della hash table

Introduzione della traccia - Il Convegno dell'Amicizia

Traccia:

“La cerimoniera Rosanna deve organizzare l'annuale incontro dei ministri dell'Amicizia e, per ridurre gli eventuali contrasti, decide di invitare solo nazioni che hanno relazioni diplomatiche con almeno altre tre nazioni presenti all'incontro. Per il suo lavoro, dispone di un elenco con le relazioni bilaterali tra i vari paesi.

-Dati di input:

È assegnato un file di testo contenente nel primo rigo due interi separati da uno spazio: il numero N delle nazioni e il numero R delle relazioni bilaterali; gli R righi successivi contengono ciascuno una coppia di interi separati da spazio, le due nazioni diplomaticamente collegate tra loro, numerate da 1 a N .

-Dati di output:

Determinare qual è il numero MASSIMO di ministri che è possibile invitare (anche nessuno!) in modo che ciascuno incontri almeno altri tre ministri con cui è in relazione.”

Descrizione:

Si è scelto di implementare il problema con la struttura dati grafo non orientato, con i ministri considerati come vertici e le relazioni come archi. L'idea è di verificare se un vertice V ha un numero di adiacenze pari o superiori a 3, in caso contrario sarà necessario rimuovere il vertice e controllare se i vertici adiacenti rispettino il requisito. Ciò viene risolto attraverso una rivisitazione dell'algoritmo di Dijkstra.

Strutture dati utilizzate

Le strutture dati utilizzate sono Grafo e Coda di priorità minima basata su min-heap.

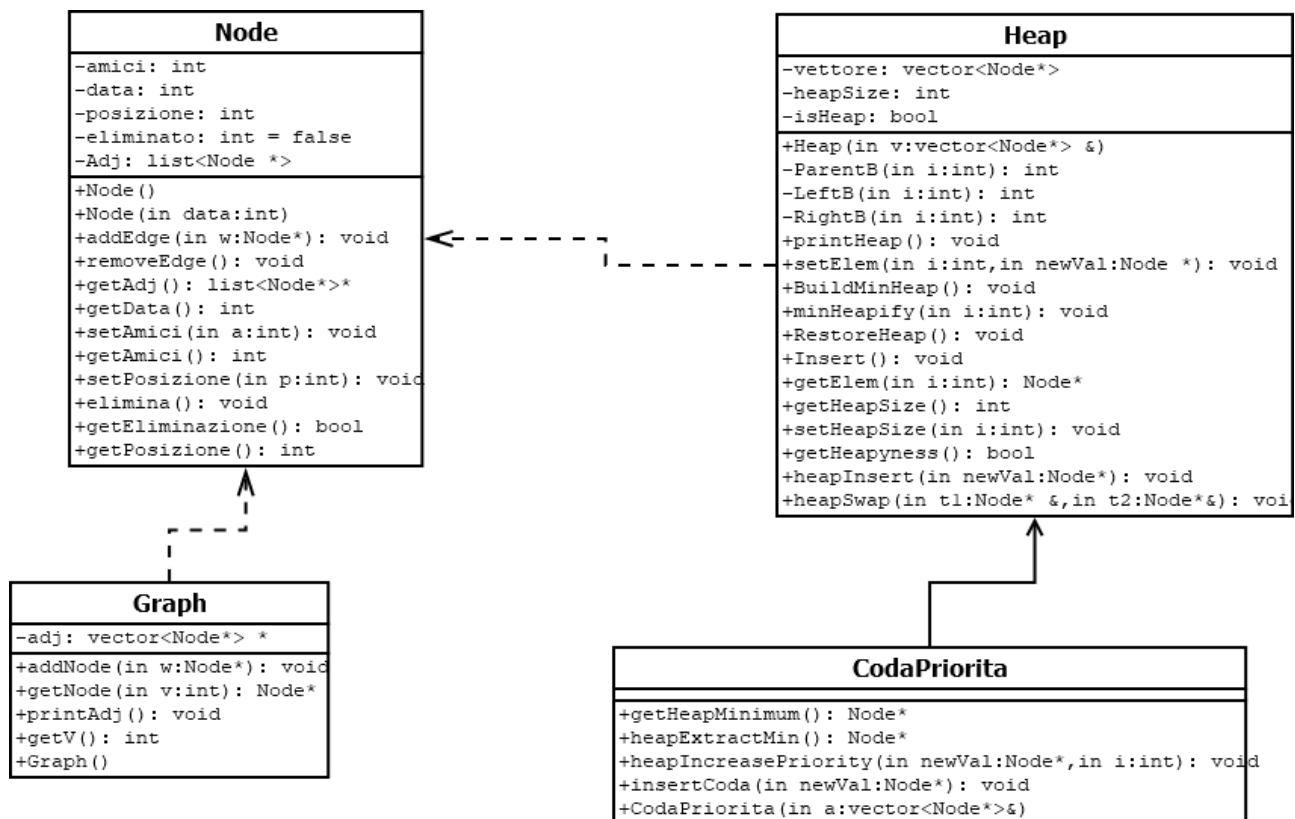
Un Grafo è un insieme di elementi detti vertici uniti da linee chiamate archi. Si dice Grafo una coppia ordinata $G=(V,E)$ di insiemi con V insieme dei nodi e E insieme degli archi tali che gli elementi di E siano coppie di elementi di V . Un grafo può essere orientato se un arco ha una direzione o pesato se un arco ha un peso specifico di attraversamento. Può essere rappresentato tramite matrice di adiacenza o lista di adiacenza, dove ogni vertice ha a sua volta una lista degli indirizzi dei vertici adiacenti). Nel nostro caso, useremo lo implementeremo attraverso lista di adiacenza.

Un Heap è una struttura ad albero binario che soddisfa la proprietà di heap: se A è genitore di B allora la chiave di A è **minore** di B nel caso si stia parlando di min heap, oppure **maggiore** nel caso si stia parlando di max heap. La chiave del valore massimo è sempre appartenente alla radice. Tutte le foglie hanno profondità h o $h-1$ ed i nodi interni hanno grado 2 o al più 1.

Una Coda di priorità è un insieme dinamico⁹ che mantiene gli elementi sempre ordinati a seconda del loro campo chiave che ne determina la priorità. Useremo un min heap per implementarla.

⁹ Insieme la cui composizione cambia nel tempo a seguito di operazioni di inserimento o cancellazione, dotati di campo chiave per la sua identificazione.

Diagramma delle classi



La classe Node identifica l'elemento vertice del grafo ed ha questi attributi:

“amici” indica il numero di archi adiacenti, “data” è il numero che identifica il nodo, “posizione” indica la sua posizione all'interno del vettore della coda di priorità, “eliminato” indica se è stato rimosso dalla coda di priorità, “Adj” è la sua lista di adiacenza¹⁰, che può essere riempita attraverso il metodo addEdge().

Il costruttore che prende in input “data” crea anche lo spazio di memoria per poter istanziare la lista di adiacenza.

La classe Grafo gestisce i vertici attraverso un vettore di Node. Il costruttore non prende nulla in input e crea lo spazio di memoria per istanziare il vettore di nodi che possono essere aggiunti attraverso addNode().

La classe Heap è un min-heap che viene usato per poter implementare la Coda di Priorità. Utilizza un vettore di Node di cui gli “amici” rappresentano la chiave per ordinarli.

La Coda di Priorità è una classe ereditata dall'Heap e di cui ne conserva il costruttore. Il suo obiettivo è di tenere costantemente come testa il Node che ha meno “amici”.

¹⁰ Viene usato il tipo di dati list fornito dalla STL di C++

Specifiche sull'algoritmo

Un modo per risolvere il problema sarebbe stato quello di ciclare i vertici adiacenti di ogni nodo fin quando non ci sarebbero più nodi con “amici” < 3 . Questo approccio, per quanto semplice, ha un costo computazionale quadratico $O(V^2)$.

Al fine di rendere più efficiente l'algoritmo si è pensato di utilizzare l'algoritmo di Dijkstra, il cui scopo è quello di risolvere il problema dei cammini minimi da sorgente unica, in modo da ciclare sui vertici del grafo al più una volta.

Per quanto riguarda l'algoritmo di Dijkstra discuteremo solo dei passi principali che verranno rivisitati nella soluzione finale. Dijkstra utilizza una coda di priorità minima per estrarre ad ogni iterazione il vertice con la stima minima del cammino, visita il vertice e dopo attua una procedura di rilassamento sui nodi adiacenti. L'algoritmo si conclude quando vengono visitati tutti i nodi. Nella soluzione vedremo come la testa della coda di priorità corrisponde al nodo con minor numero di archi. Nel caso in cui questo vertice non rispetti il requisito dei 3 “amici” allora sarà rimosso dalla coda di priorità e si continua con la procedura di rilassamento dei nodi adiacenti, che nel nostro caso, significa decrementare il numero degli “amici” del nodo che è appena stato rimosso.

Implementazione

Header.h

Questo file contiene lo scheletro delle classi e l'intestazione delle funzioni.

```
1. #ifndef HEADER_H
2. #define HEADER_H
3. #include <iostream>
4. #include <list>
5. #include <vector>
6. #include <fstream>
7. #include <string>
8. #include <stdio.h>
9. #include <stdlib.h>
10. using namespace std;
11.
12. //Scheletro classi
13. class Node{
14.     int amici; // numero degli amici del nodo
15.     int data; //numero che identifica il nodo
16.     int posizione; //la sua posizione all'interno del vettore della coda di
        priorit 
17.     bool eliminato=false; //identifica se e' stato rimosso dalla coda di
        priorit 
18.     list<Node *> *adj; //lista dei nodi adiacenti
19. public:
20.     Node(int data);
21.     Node(){};
22.     ~Node();
23.     void addEdge(Node *w); //aggiunge un nodo alla lista di adiacenza
24.     void removeEdge(); // decrementa il valore di "amici"
25.     list<Node *> * getAdj(); //ritorna la lista di adiacenza del nodo
26.     int getData(); //ritorna l'identificativo del nodo
27.     int getAmici();
```



```

28.     void setPosizione(int p);
29.     void elimina(); // imposta "eliminato" a true
30.     bool getEliminazione();
31.     int getPosizione();
32. };
33.
34. //////////////////////////////////////
35.
36. class Heap
37. {
38.     private:
39.         vector<Node*> vettore;
40.         int heapSize;
41.         bool isHeap;
42.         int ParentB(int i);
43.         int LeftB(int i);
44.         int RightB(int i);
45.     public:
46.         Heap(vector<Node*> &v);
47.         ~Heap(){};
48.         void printHeap();
49.         void setElem (int i, Node * newVal);
50.         void BuildMinHeap();
51.         void minHeapify(int i); // Questa routine ha il compito di assicurare
            la proprietà dell'heap.
52.         void RestoreHeap(); //Ha la stessa funzione del buildMinHeap ma non
            modifica l'heapSize
53.         void Insert();
54.         Node* getElem (int i);
55.         int getHeapSize();
56.         void setHeapSize(int i);
57.         bool getHeapyness(); //Ci dice se è rispetta la proprietà di heap o
            meno
58.         void heapInsert(Node* newVal);
59.         void heapSwap (Node * &t1, Node* &t2); //Inverte i puntatori di due
            Node
60. };
61.
62. //////////////////////////////////////
63.
64. class CodaPriorita : public Heap
65. {
66.     private:
67.
68.     public:
69.         CodaPriorita(vector<Node*> &a);
70.         ~CodaPriorita(){};
71.         Node* getHeapMinimum();
72.         Node* heapExtractMin();
73.         void heapIncreasePriority(Node *newVal, int i);
74.         void insertCoda(Node * newVal);
75.
76. };
77.

```

```

78. //////////////////////////////////////
79.
80. class Graph
81. {
82.     vector<Node *> *adj;
83. public:
84.     Graph();
85.     ~Graph();
86.     void addNode(Node *w);
87.     Node *getNode(int v);
88.     void printAdj();
89.     int getV(); //necessario per sapere il numero di Vertici
90. };
91.
92. //////////////////////////////////////
93. int Convegno(Graph g);
94. #endif

```

Function.cpp

In questo file sono implementate i metodi delle classi ed in particolare l'algoritmo richiesto dalla traccia.

```

1. #include "Header.h"
2.
3. Node::Node(int data)
4. {
5.     this->data=data;
6.     amici=0;
7.     adj=new list<Node *>;
8. }
9.
10. Node::~~Node()
11. {
12.     adj->clear();
13.     delete adj;
14. }
15.
16. void Node::addEdge(Node *w)
17. {
18.     adj->push_back(w);amici++;
19. }
20.
21. void Node::removeEdge()
22. {
23.     amici--;
24. }
25.
26. list<Node *> * Node::getAdj()
27. {
28.     return adj;
29. }
30.
31. int Node::getData()
32. {
33.     return data;
34. }
35.
36. int Node::getAmici()
37. {

```

```

38.     return amici;
39. }
40.
41. void Node::setPosizione(int p)
42. {
43.     posizione = p;
44. }
45.
46. void Node::elimina()
47. {
48.     eliminato = true;
49. }
50.
51. bool Node::getEliminazione()
52. {
53.     return eliminato;
54. }
55.
56. int Node::getPosizione()
57. {
58.     return posizione;
59. }
60.
61.
62. //////////////////////////////////////
63.
64.
65. int Heap::ParentB(int i)
66. {
67.     return (i-1)>>1;
68. }
69.
70. int Heap::LeftB(int i)
71. {
72.     return (i<<1)|0x1;
73. }
74.
75. int Heap::RightB(int i)
76. {
77.     return (i+1)<<1;
78. }
79.
80.
81. Heap::Heap(vector<Node*> &v)
82. {
83.     vettore = v; heapSize=0; isHeap=false;
84. }
85.
86. void Heap::printHeap()
87. {
88.     for(int i=0; i<heapSize;i++)
89.     {
90.         {
91.             cout << vettore.at(i)->getData();
92.         }
93.     }
94.     cout << endl;
95. }
96.
97. void Heap::setElem (int i, Node * newVal)
98. {
99.     vettore[i] = newVal;
100. }

```

```

101.
102.
103. void Heap::BuildMinHeap()
104. {
105.     heapSize = vettore.size();
106.     for (int j=(vettore.size()/2); j>=0; j--)
107.     {
108.         minHeapify(j);
109.     }
110.     isHeap = true;
111. }
112.
113. void Heap::RestoreHeap()
114. {
115.     for (int j=(vettore.size()/2); j>=0; j--)
116.     {
117.         minHeapify(j);
118.     }
119.     isHeap = true;
120. }
121.
122. void Heap::minHeapify(int i)
123. {
124.     int min;
125.     int l=LeftB(i), r=RightB(i);
126.     if (l<heapSize && vettore.at(i)->getAmici()>vettore.at(l)->getAmici())
127.     {
128.         min=l;
129.     }
130.     else
131.     {
132.         min=i;
133.     }
134.     if (r < heapSize && vettore.at(min)->getAmici()>vettore.at(r)->getAmici())
135.     {
136.         min=r;
137.     }
138.     if (min!=i)
139.     {
140.         heapSwap(vettore.at(i),vettore.at(min));
141.         vettore.at(min)->setPosizione(min);
142.         vettore.at(i)->setPosizione(i);
143.         minHeapify(min);
144.     }
145.
146.
147. };
148.
149.
150.
151. Node* Heap::getElem (int i)
152. {
153.     return vettore[i];
154. }
155.
156. int Heap::getHeapSize()
157. {
158.     return heapSize;
159. }
160.
161. void Heap::setHeapSize(int i)
162. {
163.     heapSize=i;

```

```

164.}
165.
166.bool Heap::getHeapyness()
167.{
168.    return isHeap;
169.}
170.
171.void Heap::heapInsert(Node * newVal)
172.{
173.    vettore.push_back(newVal);
174.    newVal->setPosizione(vettore.size()-1);
175.    BuildMinHeap();
176.    isHeap=true;
177.}
178.
179.void Heap::heapSwap (Node * &t1, Node* &t2)
180.{
181.    Node * temp;
182.    temp=t1;
183.    t1=t2;
184.    t2=temp;
185.}
186.
187.//////////////////////////////////////////////////
188.
189.CodaPriorita::CodaPriorita(vector<Node*> &a):Heap(a){};
190.
191.Node* CodaPriorita::getHeapMinimum()
192.{
193.    return getElem(0);
194.}
195.
196.
197.void CodaPriorita::heapIncreasePriority(Node *newVal, int i)
198.    {
199.        setElem(i, newVal);
200.        while (i>0 && getElem((i-1)/2)->getAmici() > getElem(i)->getAmici())
201.            {
202.                //Inizio Swap
203.                Node *elem1 = getElem((i-1)/2);
204.                Node *elem2 = getElem(i);
205.                setElem((i-1)/2,elem2);
206.                setElem(i,elem1);
207.                //Fine Swap
208.                //Riassegno il valore delle posizioni
209.                getElem((i-1)/2)->setPosizione((i-1)/2);
210.                getElem(i)->setPosizione(i);
211.                i=(i-1)/2;
212.            }
213.    }
214.
215.Node* CodaPriorita::heapExtractMin()
216.{
217.    if (getHeapyness() == true)
218.    {
219.        if(getHeapSize() < 0)
220.        {
221.            cout << "Elementi finiti, impossibile estrarre" << endl;
222.            exit (-1);
223.        }
224.        else
225.        {
226.            Node* min=getElem(0);

```

```

227.         min->elimina(); //Il nodo estratto è stato eliminato, non bisogna piu'
           considerarlo nella coda
228.         Node* elem2 = getElem(getHeapSize()-1);
229.         elem2->setPosizione(0);
230.         setElem(0,elem2);
231.         setElem(getHeapSize(),min);
232.         setHeapSize((getHeapSize()-1));
233.         minHeapify(0);
234.         return min;
235.     }
236. }
237. else
238.     exit(0);
239.}
240.
241.void CodaPriorita::insertCoda(Node * newVal)
242.{
243.    heapInsert(newVal);
244.}
245.
246.//////////
247.
248.Graph::Graph()
249.{
250.    adj = new vector<Node *>;
251.}
252.
253.Graph::~~Graph()
254.{
255.    adj->clear();
256.    delete adj;
257.}
258.
259.void Graph::addNode(Node *w)
260.{
261.    adj->push_back(w);
262.}
263.
264.Node* Graph::getNode(int v)
265.{
266.    return adj->at(v);
267.}
268.
269.void Graph::printAdj(){
270.    for(auto v:*adj){
271.        cout<<"Adj( "<<v->getData()<<"):";
272.        for(auto vv:*v->getAdj())
273.            cout<<" "<<vv->getData();
274.        cout<<endl;
275.    }
276.}
277.
278.int Graph::getV()
279.{
280.    return adj->size();
281.}
282.
283.//////////
284.
285.int Convegno(Graph g)
286.{
287.    int risultato,i;
288.    vector<Node*> a;

```

```

289.     Node* temp_nodo;
290.     CodaPriorita coda(a); //creo una coda vuota
291.     for (i=0;i<g.getV();i++)
292.     {
293.         coda.heapInsert(g.getNode(i)); //inserisco i vertici del grafo nella
        coda di priorit 
294.     }
295.     cout << "Nodi nella coda:" << endl;
296.     coda.printHeap();
297.     Node* u;
298.     while (coda.getHeapSize() > 0 && coda.getHeapMinimum()-
        >getAmici() <= 2) //Fintanto che la coda non si svuota o la testa della coda ha
        pi  di 2 amici
299.     {
300.         temp_nodo = coda.heapExtractMin(); //Estraggo il minimo, ha sicuramente
        meno di 3 amici
301.         for (auto x: *temp_nodo->getAdj()) //Scorro la sua lista di adiacenza
302.         {
303.             if (x->getEliminazione()==false) //Se il nodo trovato nella lista
        di adiacenza non   stato gi  eliminato
304.             {
305.                 x->removeEdge(); //lo rimuovo
306.                 coda.heapIncreasePriority(x,x->getPosizione()); //
307.             }
308.         }
309.     }
310.     cout << "Nodi rimanenti: " << endl;
311.     coda.printHeap();
312.     risultato = coda.getHeapSize();
313.     return risultato;
314. }

```

Analizziamo il metodo pi  importante che risolve il problema richiesto dalla traccia: `Convegno()`.

La funzione prende in input un grafo e restituisce il numero di nodi che rispecchiano il requisito dei 3 “amici”.

Inizialmente viene creata una coda di priorit  vuota nella quale vengono inseriti tutti i nodi del grafo attraverso la funzione `heapInsert()`, inserendo tutti i nodi nell’heap senza ordinarli secondo la propriet  di heap che sar  invece ripristinata con una chiamata a `BuildMinHeap()` successiva.

Ad ogni iterazione del ciclo viene analizzato il nodo alla testa della coda, se questo nodo ha pi  di 2 “amici” allora l’algoritmo   finito. In caso contrario, si estrae il nodo dalla testa attraverso la funzione `heapExtractMin()` e viene “rilassata” la sua lista di adiacenza, decrementando di fatto la loro priorit .

Bisogna adesso aggiornare la priorit  attraverso la funzione `heapIncreasePriority()`, che viene eseguita solo se il nodo che stiamo considerando non sia stato gi  eliminato.

La funzione stampa poi a schermo ci  che rimane all’interno della coda, che corrisponde ai nodi che rispettano il requisito.

Il risultato ritornato sar  poi stampato in un file di output all’interno del main.

Main.cpp

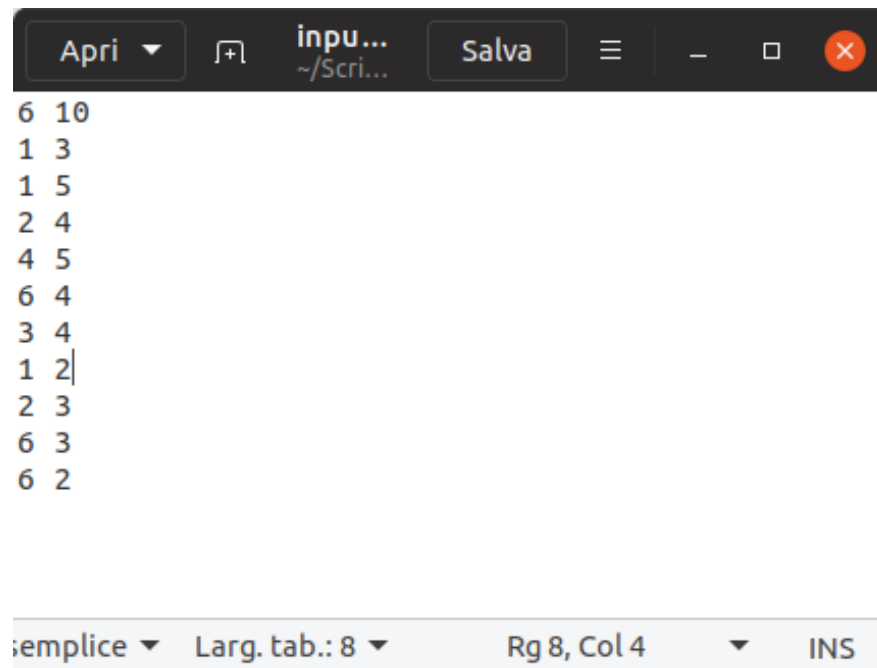
Qui di seguito verrà scritto il codice del main.

```
1. #include "Functions.cpp"
2.
3.
4. int main()
5. {
6.     Graph g; //Creo il grafo
7.     string temp;
8.     int n_nodi,num_archi,x,y,i,risultato;
9.     std::ifstream input("input.txt", ios::in); //Apro il file di input
10.    ofstream output;
11.    output.open("output.txt"); //Apro il file di output
12.    if (input.fail())
13.    {
14.        cout << "Errore nell'apertura del file" << endl;
15.        return -1;
16.    }
17.    getline(input,temp, ' '); //Prendo il numero dei nodi
18.    n_nodi = stoi(temp);
19.    cout << "Numero nodi: " << n_nodi << endl;
20.    for (i=1;i<n_nodi+1;i++)
21.    {
22.        g.addNode(new Node(i)); //Aggiungo i nodi al grafo
23.    }
24.    getline(input,temp); //Prendo il numero degli archi
25.    num_archi = stoi(temp);
26.    cout << "Numero archi: " << num_archi << endl;
27.    for (i=0;i<num_archi;i++)
28.    {
29.        getline(input,temp, ' ');
30.        x=stoi(temp);
31.        getline(input,temp);
32.        y=stoi(temp);
33.        g.getNode(x-1)->addEdge(g.getNode(y-1)); //Essendo un grafo non
orientato, aggiungo sia l'arco X-Y che Y-X.
34.        g.getNode(y-1)->addEdge(g.getNode(x-1));
35.    }
36.
37.    cout <<"Lista di Adiacenza: " << endl;
38.    g.printAdj();
39.
40.    risultato = Convegno(g);
41.
42.    output << risultato << endl;
43.    output.close();
44.    input.close();
45.
46.    return 0;
47. }
```

Dopo aver aperto il file di input, vengono salvati il numero di nodi e degli archi. Vengono inizialmente salvati solo i nodi, poi, in un ciclo for vengono assegnati gli archi. Viene poi stampata a video la lista di adiacenza, prima di richiamare il metodo Convegno(). A termine del programma, vengono chiusi i descrittori del file di input e output.

Output del programma

Il file di input:

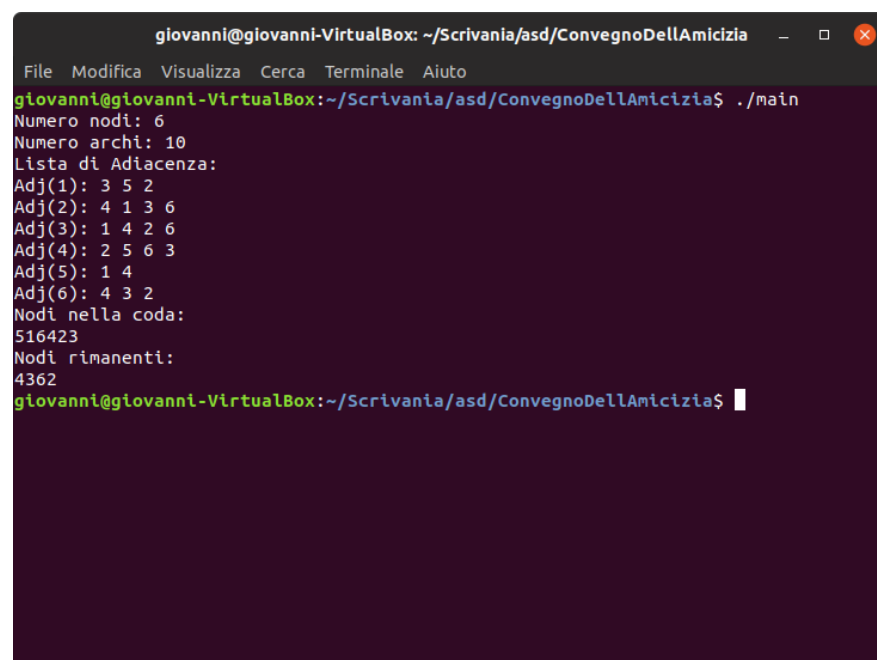


```
Apri  inpu...  Salva  -  x
~/Scri...

6 10
1 3
1 5
2 4
4 5
6 4
3 4
1 2
2 3
6 3
6 2
```

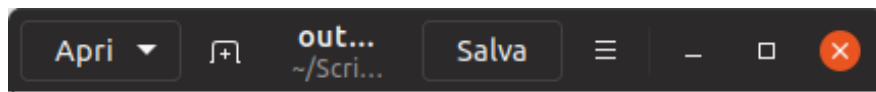
semplice Larg. tab.: 8 Rg 8, Col 4 INS

Esecuzione del programma:

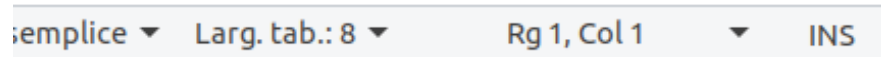


```
giovanni@giovanni-VirtualBox: ~/Scrivania/asd/ConvegnoDellAmicizia
File Modifica Visualizza Cerca Terminale Aiuto
giovanni@giovanni-VirtualBox:~/Scrivania/asd/ConvegnoDellAmicizia$ ./main
Numero nodi: 6
Numero archi: 10
Lista di Adiacenza:
Adj(1): 3 5 2
Adj(2): 4 1 3 6
Adj(3): 1 4 2 6
Adj(4): 2 5 6 3
Adj(5): 1 4
Adj(6): 4 3 2
Nodi nella coda:
516423
Nodi rimanenti:
4362
giovanni@giovanni-VirtualBox:~/Scrivania/asd/ConvegnoDellAmicizia$
```

File di output:



4



Analisi della complessità computazionale

Si tiene conto del calcolo della complessità dei metodi richiamati dalla funzione `Convengo()`.

Inizialmente viene creata una coda in cui vengono inseriti tutti gli elementi non ordinati. Questa operazione ha costo computazionale $O(1)$ e viene eseguita V volte $V * O(1) = O(V)$.

Viene poi fatta una chiamata esterna alla funzione `BuildMinHeap` che ha costo computazionale $O(V)$.
 $O(V) + O(V) = O(V)$.

Il ciclo principale fa una chiamata a `heapExtractMin`, dal costo $O(1)$ al più V volte.

Poiché i nodi eliminati non vengono considerati nel for interno, ogni arco nella lista di adiacenza viene esaminato esattamente una volta e, poiché il numero totale degli archi di adiacenza è $|E|$, ci sono un totale di $|E|$ iterazioni di questo ciclo for, e quindi l'algoritmo chiama `heapIncreasePriority()` al più $|E|$ volte. Ogni operazione di `heapIncreasePriority()` richiede tempo $O(\log_2 V)$, rendendo il costo complessivo $O(V)^{11} + O(V)^{12} + O((V + E) * \log_2 V)^{13} = O((V + E) * \log_2 V)$.

¹¹ Costruzione della coda

¹² Estrazioni del minimo

¹³ Chiamate per incrementare la priorità