Algoritmi e strutture dati

(12 CFU)

Docenti:

Lab.: prof. Francesco Camastra

Teoria: prof. Giuseppe Salvi

Giuseppe Cianci

Anno 2015/2016

Relazione d'esame: Traccia #2

Università degli Studi di Napoli "Parthenope"

ALGORITMI E STRUTTURE DATI

Studente: Giuseppe Cianci Matr.: 0124001064

a. 2015/2016

TRACCIA 2: QUESITO 1

Costruire un encoder (codificatore) ed un decoder (decodificatore) che utilizzi la codifica di Huffman. L' implementazione deve far uso di una coda di priorità, realizzata mediante un MIN-HEAP. Definire il formato (od i formati) dei dati accettati in input e definire il formato del file codificato. Calcolare il tasso di compressione ottenuto. Il codificatore deve essere almeno in grado di codificare qualunque file di testo (.txt) e, facoltativamente, anche altri tipi di formati (bmp, wav, ...).

Analisi tecniche del problema.

Il quesito richiede di realizzare tramite la codifica di Huffman un algoritmo capace di comprimere e poi decomprimere un file.

Descrizione

La codifica di Huffman è un algoritmo usato per la compressione di dati e file di tipo *loseless*, cioè senza perdita di informazioni durante la compressione. L'idea di fondo di questa codifica è quella di sfruttare le frequenze di occorrenze dei simboli all'interno del file. Ad ogni simbolo viene associata una parola binaria più o meno lunga a seconda della frequenza dello stesso. Ai simboli più frequenti vene logicamente associata una parola più corta, mentre per quelli meno frequenti si associa una parola più lunga. Una volta definito come codificare i singoli simboli, si potrà ottenere la codifica semplicemente concatenando le codifiche individuali dei simboli. L'algoritmo di Huffman si basa quindi su una codifica a lunghezza variabile.

Ovviamente una volta codificati dati e informazioni è necessario essere in grado di poterle decodificare in qualsiasi momento, e la sola associazione simbolo → parola binaria non basta per tale proposito, potrebbe infatti capitare che il prefisso di una parola più lunga equivalga a una parola più corta portando quindi a delle ambiguità inaccettabili durante la decodifica. Per ovviare a tale problema, la codifica di Huffman usa un metodo specifico per scegliere la rappresentazione di ciascun simbolo, risultando in un codice senza prefissi (cioè in cui nessuna stringa binaria di nessun simbolo è il prefisso della stringa binaria di nessun altro simbolo) che esprime il carattere più frequente nella maniera più breve possibile. È stato dimostrato che la codifica di Huffman è il più efficiente sistema di compressione di questo tipo: nessun'altra mappatura di simboli in stringhe binarie può produrre un risultato più breve nel caso in cui le frequenze di simboli effettive corrispondono a quelle usate per creare il codice.

Per quanto riguarda i file di testo, i metodi di codifica dati standard, come il sistema ASCII o Unicode, usano stringhe binarie di lunghezza fissa (blocchi) per codificare caratteri (stringhe di lunghezza 8 nel sistema ASCII e di lunghezza 16 nel sistema Unicode). Ad esempio, nel codice ASCII il carattere **A** viene codificato con **`01000001`**, **B** con **`01000010`**, **C** con **`01000011`**, e così via…

In questi casi, procedendo, con la compressione, spesso si ha un risparmio in termini di spazio fino circa 70% inferiore rispetto al file originale.

Un esempio concreto di come opera la codifica di Huffman è il seguente:
Supponiamo di avere la seguente stringa $X = "AEBCDAEAAAECCDAABACA"$ si tratta di una stringa essenzialmente su un alfabeto di soli 5 simboli di lunghezza 20.
Tale stringa se codificata ad esempio con il sistema ASCII verrebbe memorizzata nella seguente forma:

Simbolo	A	B	C	D	E
Codifica	1000001	1000010	1000011	1000100	1000101

Ogni simbolo è rappresentato con $8bit$ per un totale di: $8 \times 20 = 160bit$

```
01000001 01000101 01000010 01000011 01000100 01000001 01000101 01000001 01000001 01000001 01000101 01000011 01000011 01000100 01000001 01000001 01000010 01000001
```

Tuttavia come si vede chiaramente utilizzare $8bit$ per rappresentare un alfabeto di soli 5 simboli risulta inefficiente in quanto con gli stessi bit sarebbe possibile rappresentare ben 2^8 simboli diversi. Il numero di bit più piccolo necessario alla codifica è dato da: $\lceil \log_2 5 \rceil = 3$.
Consideriamo quindi, fra le possibili, la seguente codifica:

Simbolo	A	B	C	D	E
Codifica	000	001	010	011	100

Ogni simbolo è rappresentato con $3bit$ per un totale di: $3 \times 20 = 60bit$

```
000 100 001 010 011 000 100 000 000 000 100 010 010 011 000 000 001 000 010 000
```

Abbiamo ottenuto quindi un risparmio di 100 bit rispetto ai 160 iniziali. Ma si può fare di meglio, applicando cioè la codifica di Huffman. Come si può notare alcuni simboli sono più frequenti di altri, cioè sono più probabili nella stringa, ha senso quindi cercare di associare una parola binaria più corta o lunga a seconda della probabilità del simbolo, usando quindi una codifica a lunghezza variabile. Ovviamente con la restrizione che sia una codifica prefisso.

Simbolo	A	B	C	D	E
Codifica	1	0001	01	0000	001

Sommando la lunghezza di ogni parola otteniamo un totale di: $43bit$

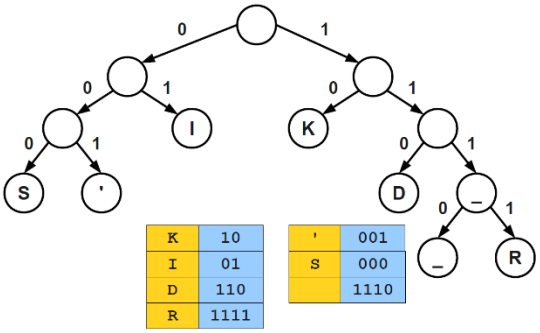
```
1 001 0001 01 0000 1 001 1 1 1 1 001 01 01 0000 1 1 0001 1 01 1
```

Utilizzando la codifica di Huffman abbiamo ottenuto un risparmio di $\sim 28\%$ rispetto alla codifica a $3bit$ e ben $\sim 73.1\%$ rispetto a quella ASCII.

Rappresentazione di una codifica prefisso

Le codifiche associate ad ogni simbolo possono essere rappresentate da alberi binari in cui le corrispondono ai caratteri da codificare, ed i relativi percorsi radice-foglia corrispondono alle codifiche dei caratteri.

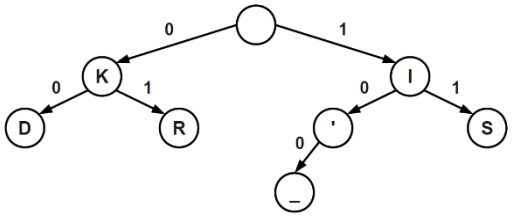
1. Ogni foglia rappresenta un carattere da codificare.
2. L'arco da un nodo al suo figlio sinistro è etichettato con **0**, l'arco da un nodo al suo figlio destro è etichettato con **1**.
3. La sequenza di etichette che si leggono su di un percorso radice-foglia, corrisponde alla codifica del relativo carattere associato alla foglia.
4. La lunghezza della parola codice associata ad un generico carattere x è uguale alla lunghezza del cammino dalla radice dell'albero alla foglia associata ad x cioè è uguale alla profondità della foglia.



K	10
I	01
D	110
R	1111

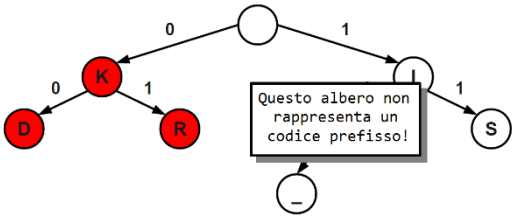
'	001
S	000
_	1110

Per evitare ambiguità durante la decodifica del file, è necessario come già detto in precedenza trovare una codifica prefisso. Questo tipo di codifica in oltre hanno il vantaggio di avere semplici algoritmi per codifica/decodifica e di non essere una limitazione per l'ottenimento di prestazioni massime, in termini di compressione.



K	0
I	1
D	00
R	01

'	10
S	11
_	100



K	0
I	1
D	00
R	01

'	10
S	11
_	100

Compressione dei dati, scelta del codice migliore

Sia X una stringa definita su un generico alfabeto $C = \{c_1, c_2, \dots, c_n\}$, vogliamo codificare questa stringa in una stringa Y in un alfabeto binario. A tal fine supponiamo di utilizzare una data codifica rappresentata dall'albero binario T. Ci chiediamo:

- Quanto sarà lunga la nostra stringa codificata? ($|Y|$)

$$|Y| = \sum_{c \in C} f(c) \cdot d_T(c) \; = B(T)$$

Dove:
- $f(c)$ è una legge che $\forall\, c \in C$ associa la sua frequenza in X.
-$d_T(c)$ è una legge che $\forall\, c \in C$ associa la profondità della foglia associata al carattere c nell'albero T (corrispondente alla lunghezza della codifica di c).
- $B(T)$ è il costo dell'albero, ovvero la codifica rappresentata mediante T.

- In che modo scegliere T affinche $B(T)$ sia minimizzata?
  Per rispondere a questa domanda è necessario un'analisi più approfondita…
  È ovvio però che tale albero deve essere completo (ogni nodo finale ha tutte le foglie) in modo da ridurre al minimo i path.

Il problema di cercare un albero T tale che $B(T)$ sia il più piccolo possibile, è risolvibile in maniera efficiente con un algoritmo greedy.

> Sia X una stringa definita su un generico alfabeto $C = \{c_1, c_2, \dots, c_n\}$, un alfabeto di caratteri la cui frequenza di apparizione in X è $f(c_1), f(c_2), \dots f(c_n)$.
>
> 1. Costruiamo il nostro albero di codifica T un passo alla volta in maniera "Bottom Up", partendo da n foglie ciascuna contenente un carattere c_i e la sua frequenza $f(c_i)$.
> 2. Ad ogni passo prendiamo due nodi che indicheremo con p, q che hanno le frequenze minime e creiamo un nuovo nodo che sarà il loro padre (p e q diventeranno quindi fratelli) e prenderà il loro posto. La frequenza del nuovo nodo sarà pari alla somma delle frequenze di p e q.
> 3. Iteriamo il passo 2 fin che l'albero non sarà costruito, cioè fin che non rimarrà un unico nodo che sarà quindi la radice.

Così facendo, le foglie con caratteri associati di "alta frequenza" verranno presi nel passo 2 il più tardi possibile, e quindi avranno una "piccola" profondità nell'albero (codifica "corta"), mentre quelli di "bassa frequenza" verranno presi nel passo 2 il più presto possibile, e quindi avranno una "alta" profondità nell'albero (codifica "lunga").

Più in specifico essenzialmente un'implementazione di tale codifica potrebbe essere posta nel seguente modo:

1. Ordina i simboli in una qualche struttura dati, per esempio una coda di priorità in base al conteggio delle loro occorrenze.
2. Ripete i seguenti passi finché la coda non contiene un unico elemento:
    a. Estrai dalla coda i due simboli con la frequenza di conteggio minore. Crea un nodo "padre" che ha come "figli" questi due elementi.
    b. Assegna la somma del conteggio delle frequenze dei figli al padre ed inserirlo nella coda di priorità.
3. Assegna una parola codice ad ogni elemento basandosi sul path a partire dalla radice.

Spesso la scelta di un algoritmo efficiente in questa fase non è necessariamente importante, infatti il numero di simboli presenti un alfabeto è generalmente un numero molto piccolo (comparato con la lunghezza del messaggio da codificare), quindi si può considerare questa complessità di tempo al più costante.

Decompressione dei dati

La decompressione dei dati avviene in maniera molto semplice, leggendo bit a bit la stringa codificata e scendendo man mano a destra o a sinistra in base al valore del bit, l'albero di Huffman fino ad arrivare ad una foglia, essa corrisponde al simbolo codificato dai bit appena letti. Raggiunta una foglia e decodificato quindi il simbolo si riparte dalla radice dell'albero continuando a leggere i bit della stringa codificata.

Dettagli importanti

La codifica presenta però alcuni "problemi" che necessitano di particolare attenzione.

- Affinché sia possibile decodificare una stringa o un file codificato è necessario avere informazioni sull'albero di Huffman usato per la compressione, quindi c'è bisogno di conoscere la codifica usata ed i simboli dell'alfabeto originale.
  Per ovviare a questo problema si può far precedere la codifica da un *header* contenente abbastanza informazioni per ricostruire la codifica. Ad esempio si possono memorizzare tutti i simboli con le relative frequenze o ancora meglio l'albero stesso.

- Quando comprimiamo una stringa X su un qualche alfabeto in una stringa binaria Y la sua lunghezza $|Y|$ sarà sicuramente un numero $\in \mathbb{N}$. Tutti i computer però memorizzano i file in multipli di bit, i Byte. E poiché $1Byte = 8bit$ avremo che se $|Y|$ non è un multiplo di 8 la memorizzazione prevedrà l'aggiunta di bit "sporchi" per completare l'ultimo Byte.
  Quindi durante la decodifica, non sapendo quando fermarsi, questi bit potrebbero essere interpretati come altri simboli da decodificare portando quindi ad un errore.
  Possibili soluzioni:
  - Si può utilizzare un simbolo particolare, che si sa non presente nella stringa, che concatenato alla fine della stessa funzionerà da terminatore, in modo da fermare la decodifica evitando così i bit aggiunti.
  - Si può memorizzare $|X|$ insieme all'header, in modo tale che una volta decodificati tutti i simboli di X l'algoritmo si interrompa.

Compressione ed entropia

Entropia e compressione sono due argomenti strettamente collegati tra di loro, l'entropia, nella teoria dell'informazione può essere vista come la casualità contenuta in una stringa, ed è perciò strettamente collegata al numero minimo di bit necessari per memorizzarla.
Sia X una stringa definita su un alfabeto $C = \{c_1, c_2, \dots, c_n\}$ e sia $\mathbb{P}(c_i)$ la probabilità di occorrenza di un dato simbolo c_i, l'entropia di Shannon è definita come:

$$\mathbb{H}[X] = -\sum_{i=1}^{n} \mathbb{P}(c_i) \cdot \log_2 \mathbb{P}(c_i)$$

Essa rappresenta essenzialmente il numero medio minimo di bit necessari a rappresentare X. Il valore dell'entropia varia fra $[0, \log_2 |C|]$

- Nel caso di una stringa molto casuale, l'entropia della stringa risulta alta, i simboli si presentano con circa la stessa probabilità rendendo la compressione inefficace. Ad esempio nella codifica di Huffman su un alfabeto a 2^8 simboli, nel quale si presentano tutti con frequenze molto vicine, avrò che $\mathbb{H}[X] \cong 8$ e pertanto la compressine risulta inefficace se non inutile.
- Nel caso di stringhe poco casuali, con probabilità molto diverse, avremo un'entropia bassa e quindi la compressione risulterà più efficace.

Lunghezza massima di una codifica

Sia X una stringa definita su un generico alfabeto C, e sia Y la stringa binaria ottenuta dalla codifica, allora la lunghezza massima delle codifica associata ad un singolo simbolo è pari a $|C| - 1$, quando l'albero di Huffman è completamente sbilanciato da un lato. Ciò avviene nel caso degenere in cui le frequenze/probabilità dei simboli siano pari a quelle della sequenza di Fibonacci.

Implementazione, funzionamento ed esecuzione

Come funziona

Il programma permette di comprimere o decomprimere un file in due modi: Aprendo l'eseguibile seguendo le istruzioni di digitare manualmente la path assoluta o relativa in cui si trova in nostro file. Oppure in maniera molto più semplice e diretta con un *drag and drop*, trascinando il file direttamente sull'eseguibile.
In entrambi i casi, una volta terminata l'elaborazione, il risultato verrà posto nella stessa path del file, sostituendo quindi l'originale e verranno mostrate alcune informazioni riguardo l'esecuzione.

Tipi di file ammessi

Compressione: Il programma lavora leggendo e scrivendo il file in maniera "binaria" sfruttando come alfabeto di partenza i 2^8 possibili simboli contenuti in un byte, pertanto è in grado di comprimere qualsiasi tipo di file indipendentemente dal formato. L'unica limitazione potrebbe essere la dimensione massima del file, $4\ GiB$ per le macchine che rappresentano il tipo *long int* su $32bit$, e praticamente nessuna limitazione nel caso il tipo sia rappresentato con $64bit$.

Decompressione: Possono essere decompressi solo i file che hanno la giusta estensione, di default "*.cnc*",questo per ovvi motivi legati alla necessità di memorizzazione dell'header insieme alla codifica.

Esempio #1

```
Giuseppe Cianci Pio
0124001064

Compressione del file: C:\Users\CncGp\workspace\Huffman_Compression\Release\HuffUML.psd in corso...

Dimensioni del file prima dell'elaborazione:     91901666 Byte.
Dimensioni del file dopo dell'elaborazione:      14295535 Byte.
Rapporto di compressione: 84.444749%

L'entropia del file e': 0.484806
Operazione completata, tempo trascorso: 3.374931 sec
```

Si sta comprimendo un file di tipo "*PhotoShop Document*" essenzialmente un'immagine RAW. Il file era grande $87,6MB$ che dopo la compressione sono scesi a $13,6MB$, un risparmio di circa 84.4%. Tale risultato è stato possibile in quanto come si può vedere l'entropia del file era molto bassa e ha premesso così migliore efficacia.

```
Giuseppe Cianci Pio
0124001064

Decompressione del file: C:\Users\CncGp\workspace\Huffman_Compression\Release\HuffUML.cnc in corso...

Dimensioni del file prima dell'elaborazione:     14295535 Byte.
Dimensioni del file dopo dell'elaborazione:      91901666 Byte.
Rapporto di compressione: -542.869721%

Operazione completata, tempo trascorso: 2.446150 sec
```

La decodifica del file avviene in modo analogo, una volta decompresso il file ritorna alle sue dimensioni originali, cioè ben 542.8% volte più grande della codifica.

Esempio #2

```
Giuseppe Cianci Pio
0124001064

Compressione del file: C:\Users\CncGp\workspace\Huffman_Compression\Release\Immagine.png in corso...

Dimensioni del file prima dell'elaborazione:     116036 Byte.
Dimensioni del file dopo dell'elaborazione:      116278 Byte.
Rapporto di compressione: -0.208556%

L'entropia del file e': 7.970658
Operazione completata, tempo trascorso: 0.024997 sec
```

Si è scelto di comprimere un file di tipo "*png*", si tratta di un formato per immagini che utilizza già di per sé una codifica *lossy*. Ed infatti come ci si aspetterebbe l'entropia di questo file risulta altissima, molto vicina al valore massimo 8, e come conseguenza effettuare la codifica di Huffman non porta a risultati sperati, anzi il file compresso risulta perfino più grande.

Esempio #3

```
Giuseppe Cianci Pio
0124001064

Compressione del file: C:\Users\CncGp\workspace\Huffman_Compression\Release\testo breve.txt in corso...

Dimensioni del file prima dell'elaborazione:     46 Byte.
Dimensioni del file dopo dell'elaborazione:      53 Byte.
Rapporto di compressione: -15.217391%

L'entropia del file e': 3.737932
Operazione completata, tempo trascorso: 0.008373 sec
```

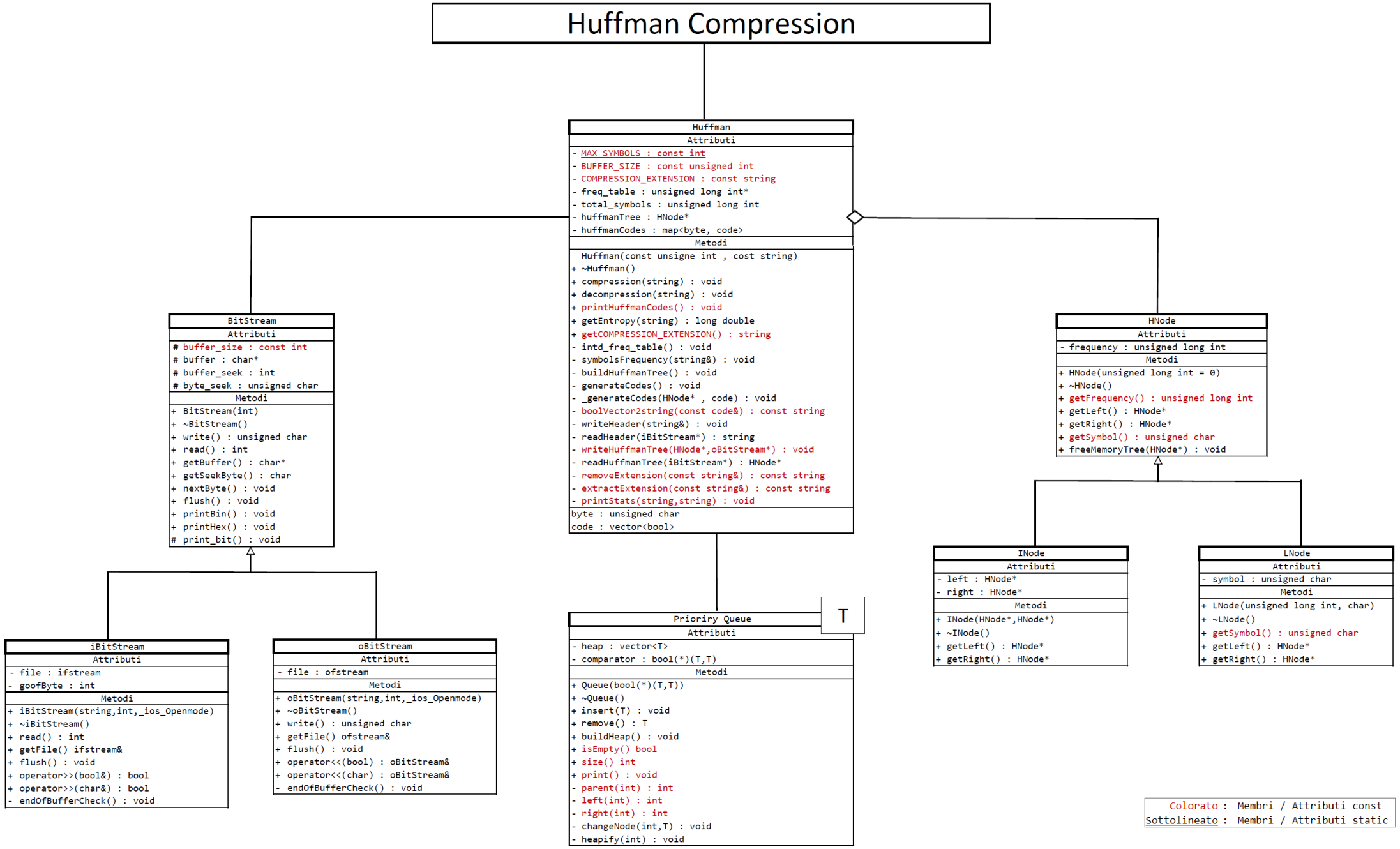
In questo esempio si sta codificando un file di testo molto piccolo, soli $46Byte$. Ma come si vede, il file compresso è più grande del 15%. Questo perché va ricordato che oltre alla codifica viene incluso anche l'header, la cui dimensione in questi casi influisce significativamente sul peso finale del file.

Esempio #4

```
Giuseppe Cianci Pio
0124001064

Compressione del file: C:\Users\CncGp\workspace\Huffman_Compression\Release\testo lungo.txt in corso...

Dimensioni del file prima dell'elaborazione:     19596 Byte.
Dimensioni del file dopo dell'elaborazione:      11147 Byte.
Rapporto di compressione: 43.115942%

L'entropia del file e': 4.470814
Operazione completata, tempo trascorso: 0.018986 sec
```

Come ultimo esempio è stato codificato un testo più lungo che come mostrato, compresso, è più piccolo di circa 43.1% rispetto all'originale.

IMPLEMENTAZIONE DELL'ALGORITMO DI HUFFMAN



DESCRIZIONE DETTAGLIATA DELL'IMPLEMENTAZIONE

Dal diagramma UML si può vedere chiaramente che l'implementazione si basa sulle seguenti classi:

a. ***Huffman:*** Classe principale del progetto, combina le restanti in modo da mettere in pratica l'algoritmo.

b. ***HNode:*** Definisce la struttura generica di un nodo dell'albero di Huffman.
INode: Classe derivata da HNode, definisce la struttura di un nodo interno all'albero di Huffman.
LNode: Classe derivata da HNode, definisce la struttura di un nodo foglia dell'albero di Huffman.

c. ***BitStream:*** Classe che implementa la struttura generica di un bit-stream per la scrittura o lettura sul file.
iBitStream: Classe derivata da BitStream, definisce un "input bit-stream" per la lettura da un file.
oBitStream: Classe derivata da BitStream, definisce un "output bit-stream" per la scrittura su un file.

d. ***PriorityQueue:*** Una coda di priorità generica realizzata con un template.

HUFFMAN:

Huffman	
Attributi	
- MAX_SYMBOLS : const int	Numero dei simboli contenuti in un byte, su computer tradizionali corrisponde a 256.
- BUFFER_SIZE : const unsigned int	Dimensioni scelte in Byte per il buffer del Bit-Stream in lettura ed in scrittura. [Default = 1024Byte]
- COMPRESSION_EXTENSION : const string	Estenzione scelta per il file codificato in modo da poterlo riconoscere al momento della decodifica. [Default = ".cnc"]
- freq_table : unsigned long int*	Puntatore ad un array dinamico di dimensione MAX_SYMBOLS contenente le frequenze di occorrenza per ogni simbolo del file.
- total_symbols : unsigned long int	Numero totale dei simboli presenti nel file da comprimere, corrisponde alla dimensione del file in byte.
- huffmanTree : HNode*	Puntatore all'albero di Huffman generato a partire dalle frequenze dei simboli.
- huffmanCodes : map<byte, code>	Map che memorizza il codice di huffman calcolato per ogni simbolo prensente nel file.
Metodi	
+ Huffman(const unsigne int , cost string)	Costruttore con parametri, riceve in input le dimensione per il buffer e l'estensione del file compresso, altrimenti usa quelle di default.
+ ~Huffman()	Distruttore, dealloca la memori utilizzata durante il funzionamento.
+ compression(string) : void	Funzione per la codifica del file: Comprime il file dato in input mediante la sua path.
+ decompression(string) : void	Funzione per la decodifica del file: Decomprime il file dato in input mediante la sua path.
+ printHuffmanCodes() : void	Stampa le parole codice binarie assegnate ad ogni simbolo.
+ getEntropy(string) : long double	Ritorna l'entropia del file dato in input mediante la sua path.
+ getCOMPRESSION_EXTENSION() : string	Ritorna l'estensione scelta per il file codificato.
- intd_freq_table() : void	Inizializza freq_table e total_symbols.
- symbolsFrequency(string&) : void	Calcola total_symbols e la frequenza delle occorrenze per ogni simbolo memorizzandole in freq_table.
- buildHuffmanTree() : void	Costruisce l'albero di Huffman partendo dalla frequenza di occorrenza dei simboli.
- generateCodes() : void	Assegna ad ogni simbolo una parola binaria, percorrendo l'albero di Huffman.
- _generateCodes(HNode* , code) : void	
- boolVector2string(const code&) : const string	Semplice funzione che converte un vettore di bool in una stringa per poterla stampare con più facilità.
- writeHeader(string&) : void	Scrive l'header sul file codificato dato in input mediante la sua path.
- readHeader(iBitStream*) : string	Legge l'header sfruttando l'input bit-stream (iBitStream) fornito in input.
- writeHuffmanTree(HNode*,oBitStream*) : void	Memorizza l'albero di Huffman linearizzato sull'header del file mediante l'output bit-stream fornito.
- readHuffmanTree(iBitStream*) : HNode*	Legge l'albero di Huffman linearizzato tramite l'input bit-stream (iBitStream) fornito in input.
- removeExtension(const string&) : const string	Rimuove l'estenzione ad una stringa contenente il path di un file.
- extractExtension(const string&) : const string	Estrae l'estenzione da una stringa contenente il path di un file.
- printStats(string,string) : void	Stampa le statistiche e informazioni riguardo la compressione fornendo le path al file prima e dopo l'elaborazione.
byte : unsigned char	Typedef
code : vector<bool>	Typedef

Come già detto rappresenta la classe principale di tutta l'implementazione, e contiene i metodi e funzioni necessari alla codifica e decodifica.

Funzionamento ed uso:

La classe è essenzialmente molto semplice e intuitiva da usare, tra i metodi pubblici abbiamo: `printHuffmanCode()`, `getEntropy()` e `getCOMPRESS_EXTRENSION()` che sono solo metodi "*getters*", o di accesso alle informazioni e che non ci permettono quindi di cambiare lo stato interno del nostro oggetto.
Ben più importanti per l'implementazione della classe sono i metodi: `compress()` e `decompress()`. Essi sono due metodi indipendenti che si descrivono da soli, consentono infatti rispettivamente di comprimere e decomprimere un file.
Il funzionamento di `compress()` è il seguente:

1. Vengono calcolate le frequenze di occorrenza dei simboli all'interno del file tramite `symbolsFrequency()`.
2. Viene costruito l'albero di Huffman e vengono generati i codici associati con `buildHuffmanTree()` e `generateCodes()`.
3. Viene generato l'header e quindi scritto all'inizio del file compresso con `writeHeader()`.
4. Per ogni simbolo letto dal file da comprimere, scrivo sul file compresso il corrispettivo codice di Huffman.

Il funzionamento di `decompress()` è il seguente:

1. Legge l'header del file ricreando così l'albero di Huffman e ottenendo la vecchia estensione tramite `readHeader()`.
2. Legge bit a bit il file tramite i quali percorre l'albero (`1` -> figlio SX, `0` -> figlio DX) fino a che non si raggiunge una foglia, viene quindi scritto sul file il simbolo contenuto e si riparte poi ripercorrendo l'albero dalla radice.

Quindi tramite solo questi due metodi pubblici è possibile usare l'intero programma con facilità.

Listato della classe:

Header: Huffman.h

```
1.  #ifndef HUFFMAN_H_
2.  #define HUFFMAN_H_
3.  #include <iostream>
4.  #include <iomanip>         //PER setw()
5.  #include <climits>         //Per CHAR_BIT, numero di bit in un char
6.  #include <string.h>        //Per memset
7.  #include <vector>
8.  #include <map>
9.  #include <math.h>
10. #include "BitBuffer/BitBuffer.h"
11. #include "HuffmanTree/HuffmanTree.h"
12. #include "PriorityQueue/PriorityQueue.h"
13. using namespace std;
14.
15. class Huffman {
16.
17.     typedef unsigned char byte;
18.     typedef vector<bool> code;
19.
20. public:
21.     //  COSTRUTTORE
22.     Huffman(const unsigned int bufferSize = 1024, const string extension = "cnc");
23.     //  DISTRUTTORE
24.     virtual ~Huffman();
25.
26.     //  METODI DELLA CLASSE
27.     void printHuffmanCodes() const;                    //Stampa i codici di Huffman generati per ogni carattere.
28.     void compress(string path);                        //Comprime il file che si trova alla path di input
29.     void decompress(string path);                      //Decomprime il file che si trova alla path di input
30.     long double getEntropy(string path);
31.     string getCOMPRESS_EXTRENSION() const {return COMPRESS_EXTRENSION;}
32.
33.
34. private:
35.     //  ATTRIBUTI PRIVATI
36.     static const int MAX_SYMBOLS = 1<<CHAR_BIT;        //Numero dei possibili caratteri diversi in un byte - 2^(CHAR_BIT)
37.     const unsigned int BUFFER_SIZE;                    //Dimensione del buffer usato per la lettura e scrittura dei file.
38.     const string COMPRESS_EXTRENSION;                  //Estensione del file compresso.
39.     unsigned long int* freq_table;                     //Tavola delle frequenze relative ad ogni carattere [mediante array dinamico].
40.     unsigned long int total_symbols;                   //Numero totale dei simboli da codificare, equivale alla grandezza del file
41.     HNode* huffmanTree;                                //Puntatore all'albero di Huffman generato per la compressione/decompressione.
42.     map<byte,code> huffmanCodes;                       //Map di tipo <char, vector<bool>> contenente il carattere con relativo codice
43.
44.     //  METODI PRIVATI
45.     inline void intd_freq_table();                     //Inizializza a tutti 0 le frequenze
46.     void symbolsFrequency(string& path);               //Calcola le frequenze per ogni simbolo del file che si trova alla path.
47.     void buildHuffmanTree();                           //Costruisce l'albero di Huffman basandosi sulle frequenze.
48.     void generateCodes();                              //Genera i codici relativi ad i singoli caratteri e li inserisce nella map.
49.     void _generateCodes(HNode* root, code c);
50.     const string boolVector2String(const code&) const;  //Funzione per la stampa, converte i vettori di bool in una stringa
51.
52.     void writeHeader(string& path);                    //Scrive l'header sul file che si trova alla path di input.
53.     string readHeader(iBitStream* reader);             //Legge l'header sul file che si trova alla path di input.
54.     void writeHuffmanTree(HNode* root, oBitStream* writer) const; //Funzione che linearizza l'albero di huffman e lo scrive su un file
55.     HNode* readHuffmanTree(iBitStream* reader);        //Funzione legge l'albero di huffman linearizzato da un file di input
56.
57.     const string removeExtension(const string& path) const; //Rimuove, se c'e', l'estensione da una stringa contenente un path.
58.     const string extractExtension(const string& path) const;//Ritorna, se c'e',l'estensione di una stringa contenente un path.
59.     void printStats(string path1, string path2) const;      //Stampa le statistiche dopo la compressione o la decompressione
60. };
61.
62. #endif /* HUFFMAN_H_ */
```

Source: Huffman.cpp

```cpp
#include "Huffman.h"


/** Descrizione:
 *  Inizializza a nullptr, l'albero di huffman, l'array contenente le frequenze dei simboli, la dimensione del buffer
 *  e l'estensione del file una volta compresso.
 *  Parametri:
 *  bufferSize  - Dimensione, in byte, del buffer usato per la lattura e scrittura      [default 1024 byte]
 *  extension   - Estensione che identifica il file una volta compresso                 [default "cnc"]
 * */
Huffman::Huffman(const unsigned int bufferSize, const string extension) : BUFFER_SIZE(bufferSize), COMPRESS_EXTRENSION(extension){
    huffmanTree = nullptr;                                              //Inizialmente l'albero di huffman e' vuoto
    freq_table = nullptr;
    total_symbols = 0;
}

/** Descrizione:
 *  libera la memoria allocata per l'array delle frequenze e per l'albero di huffman
 * */
Huffman::~Huffman() {
    delete[] freq_table;                                                //Libero la memoria destinata alla tabella delle frequenze
    huffmanTree->freeTreeMemory(huffmanTree);

}


/** Descrizione:
 *  Inizializza freq_table e total_symbols, utilizza memset in quanto molto piu' veloce rispetto altri metodi
 * */
inline void Huffman::intd_freq_table(){
    if(freq_table!=nullptr)                                             //Mi assicuro che la memoria per freq_table sia stata allocata
    memset(freq_table,0,sizeof(unsigned long int)*MAX_SYMBOLS);
    total_symbols = 0;
}
/** Descrizione:
 *  Legge tramite un buffer un file e ne calcola il numero di occorrenze di ogni simbolo memorizzandolo in freq_table e calcola il
 *  numero totale dei simboli, cioe' le dimensioni in byte del file, salvandolo in total_symbols
 *  Parametri:
 *  path                    - Stringa contenente il percorso del file
 *  Ritorno:
 *  total_symbols           - Il numero totale dei simboli presenti nel file
 **/
void Huffman::symbolsFrequency(string& path){
    freq_table = new unsigned long int[MAX_SYMBOLS];                    //Alloco la memoria per contenere le frequenze
    intd_freq_table();                                                  //Inizializzo la tavola delle frequenze
    iBitStream buff(path, BUFFER_SIZE, ios::binary);                    //Creo un buffer per il file

    char c;
    while(buff >> c)  ++freq_table[ (byte) c ];                         //Per ogni char aumento il valore corrispondente nell'array.

    buff.getFile().clear();
    total_symbols = buff.getFile().tellg();
}

/** Descrizione:
 *  Costruisce un albero di huffman partendo delle frequenze dei simboli calcolata su un file.
 *  NB: Richiede che le frequenze siano state calcolate dalla funzione symbolsFrequency
 *  Ritorno:
 *  total_symbols           - Il numero totale dei simboli presenti nel file
 **/
void Huffman::buildHuffmanTree(){
/*Funzione che una volta calcolate le frequenze costruisce l'albero di Huffman sfruttando una coda di priorita'
 * 1)Si crea per ogni carattere con frequenza>0, un nodo foglia e lo si inserisce in una min-coda di priorità.
 * 2)Si estraggono i due nodi minimi e si crea nodo in cui quest'ultimi sono figli e la cui radice è un nodo contenente la somma
 *   delle frequenze dei figli.
 * 3)Si aggiunge l'albero appena creato nella min-coda
 * 4)Si ripetono i passaggi 2-3 fin che non rimane un unico nodo(albero) nella coda (cioe' l'albero di huffman)*/

    PriorityQueue<HNode*> queue( [](HNode* a, HNode* b){return a->getFrequency() < b->getFrequency();} );

```

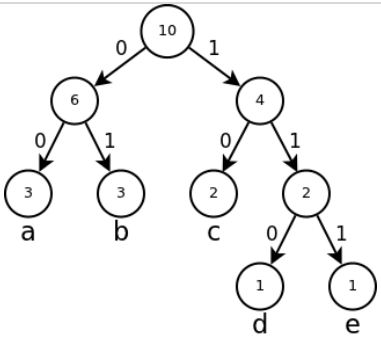
```
71.
72.     //Scorro i caratteri e per tutti quelli con frequenze positive, alloco e poi inserisco nella coda un nodo foglia 'LNode'
73.     for(int i = 0; i<MAX_SYMBOLS; i++)
74.         if(freq_table[i]!=0)
75.             queue.insert(new LNode(freq_table[i], (byte)i ));
76.
77.     while(queue.size()>1){                          //Fin quando nella coda non resta un solo elemento...
78.         HNode* a = queue.remove();                  //Estraggo i due puntatori ai nodi con frequenza minima
79.         HNode* b = queue.remove();
80.         INode* c = new INode(a,b);                  //Creo un nuovo nodo 'INode' interno avente come figli i nodi puntati da a e b
81.         queue.insert(c);                            //Inserisco tale nodo nella coda di priorita'
82.     }
83.
84.
85.     huffmanTree = queue.insert();                  //Salvo l'albero risultante dall'estrazione dell'ultimo elemento
86. }
87.
88.
89. /** Descrizione:
90.  *  Funzione ricorsiva che una volta creato l'albero di Huffman, genera i codici relativi ad ogni carattere e li memorizza nella map
91.  *  huffmanCodes semplicemente percorrendo il cammino radice-foglia effettuato per raggiungere il carattere.
92.  *  - Il codice e' una sequenza binaria di 0 e 1
93.  *  - lungo tanto quanto il percorso radice-foglia
94.  *  - Composto da 1 se si va a sinistra e 0 se si va a destra lungo il percorso per raggiungere il carattere
95.  *
96.  *  NB: Richiede che sia stato costruito l'albero di huffman tramite la funzione buildHuffmanTree
97.  **/
98. void Huffman::generateCodes(){
99.     //CASO  #1 L'albero di huffman e' vuoto, non devo eseguire alcuna operazione
100.    if(huffmanTree == 0) return;
101.
102.    //CASO  #2 L'albero di huffman ha un solo nodo (la radice)
103.    if(huffmanTree->getLeft() == 0 and huffmanTree->getRight()==0){
104.        code c(1,true); //Vettore di bool con un solo elemento '1'
105.        huffmanCodes.insert(make_pair(huffmanTree->getSymbol(), c));            //Lo inserisco nella map
106.    }
107.
108.    //CASO  #3 L'albero ha piu' di un nodo, chiamo la procedura ricorsiva.
109.    else {
110.        code c; //vettore di bool vuoto.
111.        _generateCodes(huffmanTree,c);
112.    }
113. }
114. void Huffman::_generateCodes(HNode* root, code c){
115.    // CASO BASE Mi trovo ad una foglia?
116.    if(root->getLeft() == 0 and root->getRight()==0){
117.        huffmanCodes.insert(make_pair(root->getSymbol(), c));
118.        return;
119.    }
120.    //Altrimenti se non mi trovo alla foglia chiamo ricorsivamente a destra e a sinistra
121.    code c_sx = c;
122.    c_sx.push_back(true);                       //vettore di bool con aggiunto alla fine '1'
123.    _generateCodes(root->getLeft(), c_sx);       //Chiamo ricorsivamente nel sottoalbero sinistro
124.    code c_dx = c;
125.    c_dx.push_back(false);                      //vettore di bool con aggiunto alla fine '0'
126.    _generateCodes(root->getRight(), c_dx);      //Chiamo ricorsivamente nel sottoalbero destro
127. }
128.
129. /** Descrizione:
130.  *  Converte un vettore di bool in una stringa in modo da poterla stampare con facilita'.
131.  *  Parametri:
132.  *  v                       - vettore di bool da convertire in stringa
133.  **/
134. const string Huffman::boolVector2String(const code& v) const {
135.    string s = "";
136.    for(unsigned int i=0; i<v.size(); i++){
137.        if(v[i])
138.            s+="1";
139.        else
140.            s+="0";
141.    }
142.    return s;
143. }
```

```
144. /** Descrizione:
145.  *  Stampa i codici di huffman relativi ad ogni simbolo memorizzati nella map huffmanCodes.
146.  **/
147. void Huffman::printHuffmanCodes() const{
148.     if(huffmanCodes.empty()) return;                            //Se la map e' vuota non stampo nulla
149. 
150.     map<byte,code>::const_iterator it = huffmanCodes.begin(); //Iteratore alla map in modo da poterla scorrere con facilita'
151. 
152.     // CALCOLO IL CODICE PIU' LUNGO
153.     unsigned int max_len=0;
154.     for(;it!=huffmanCodes.end();it++)                           //Calcolo la lunghezza massima fra tutti i codici, in modo da
155.         if(it->second.size()>max_len)                           //adattare la stampa ai codici
156.             max_len = it->second.size();
157. 
158.     // GENERO IL SEPARATORE
159.     string separator = "+---+";                                 //Genero il separatore utilizzato durante la stampa
160.     for(unsigned int i=0; i<max_len; i++)                       //La sua lunghezza varia in funzione di  max_len
161.         separator+="-";
162.     separator+="+";
163. 
164.     // STAMPO TUTTI I CODICI                                    //Stampo tutti i codici.
165.     cout<<separator<<endl;
166. 
167.     for(it=huffmanCodes.begin();it!=huffmanCodes.end();it++){
168.         cout<<"|'"<<it->first<<"'|" <<setw(max_len)<<boolVector2String(it->second) << "|" <<endl;
169.         cout<<separator<<endl;
170.     }
171. }
172. 
173. 
174. /** Descrizione:
175.  *  Scrive l'albero di huffman su un file, tramite un buffer oBitStream fornito in input, la scrittura avviene tramite
176.  *  una visita pre-order dell'albero.
177.  *  Parametri:
178.  *  root        - puntatore alla radice dell'albero di huffman da scrivere
179.  *  writer                  - puntatore al bit-stream che scrive sul file
180.  **/
181. void Huffman::writeHuffmanTree(HNode* root, oBitStream* writer) const {
182.     if (root->getLeft() == nullptr and root->getRight() == nullptr){  //CASO BASE: root e' una foglia?
183.         *writer << true;                                              //Scrivi 1 + la rappresentazione binaria del simbolo
184.         char c = root->getSymbol();
185.         *writer << c;
186.     }else {                                                           //ALTRIMENTI:
187.         *writer<<false;                                               //Scrivi 0 ed effettua le chiamate ricorsive ai figli
188.         writeHuffmanTree(root->getLeft(), writer);
189.         writeHuffmanTree(root->getRight(), writer);
190.     }
191. }
192. 
193. /** Descrizione:
194.  *  Legge l'albero di huffman da un file, tramite un buffer iBitStream fornito in input, la lettura avviene tramite
195.  *  una visita pre-order dell'albero.
196.  *  Parametri:
197.  *  reader                  - puntatore al bit-stream che legge dal file
198.  *  Ritorno:
199.  *  Puntatore all'albro di huffman ricostruito leggendo il file
200.  **/
201. HNode* Huffman::readHuffmanTree(iBitStream* reader){
202.     bool b;
203.     *reader >> b;                                                 //Leggo un bit dallo stream
204.     if (b==1) {                                                   //Se questo bit e' 1, vuol dire che mi trovo ad una foglia e che
205.         char c;                                                   //dopo ho memorizzato un simbolo
206.         *reader >> c;                                             //Pertanto lo leggo, creo un nuovo nodo foglia 'LNode' e lo ritorno
207.         return new LNode(0, c);
208.     }else {                                                       //Altrimenti effettuo le chiamate ricorsive ai figli
209.         HNode* leftChild = readHuffmanTree(reader);
210.         HNode* rightChild = readHuffmanTree(reader);
211.         return new INode(leftChild, rightChild);                  /Creando un nuovo nodo Interno 'INode'
212.     }
213. }
214. 
```

```
/** Descrizione:
 *  Scrive sul file fornito da una path l'header necessario alla decodifica del file. L'header si presenta nella seguente forma:
 *  |      PARTIZIONE A      |    PARTIZIONE B   |           PARTIZIONE C           |
 *  +----+----+----+----+----+----+----+----+----+----------------------------+----+
 *  |char|char|....|char|'\0'| unsigned long int |     Albero di huffman ..... |'\0'|
 *  +----+----+----+----+----+----+----+----+----+----------------------------+----+
 *  A:= Stringa contenente la vecchia estensione del file, puo' essere lunga a piacere.
 *  B:= Valore contenente il numero totale di simboli codificati, cioe' la dimensione in byte del file prima della compressione.
 *  C:= linearizzazione dell'abero di huffman scritto mediante l'apposita funzione
 *  Parametri:
 *  path            - Stringa contenente il percorso del file
 **/
void Huffman::writeHeader(string& path){

    string new_path = removeExtension(path) + '.' + COMPRESS_EXTRENSION;
    oBitStream writer(new_path,BUFFER_SIZE, ios::binary);                    //Creo un bit-stream di output sul nuovo file

    for(auto c : extractExtension(path))                                     //Scrivo l'estenzione corrente nel file
        writer << c;
    writer << '\0';

    for(unsigned int i = 0; i < sizeof(unsigned long int); i++)              //Scrivo total_symbols nel file, diviendolo byte a byte
        writer << ( (char*)&total_symbols )[i];

    writeHuffmanTree(huffmanTree,&writer);                                   //Scrivo l'albero di huffman linearizzato
    writer << '\0';
}

/** Descrizione:
 *  Legge dal file fornito da un bit-stream l'header necessario alla decodifica del file. L'header si presenta nella seguente forma:
 *  |      PARTIZIONE A      |    PARTIZIONE B   |           PARTIZIONE C           |
 *  +----+----+----+----+----+----+----+----+----+----------------------------+----+
 *  |char|char|....|char|'\0'| unsigned long int |     Albero di huffman ..... |'\0'|
 *  +----+----+----+----+----+----+----+----+----+----------------------------+----+
 *  A:= Stringa contenente la vecchia estensione del file, puo' essere lunga a piacere.
 *  B:= Valore contenente il numero totale di simboli codificati, cioe' la dimensione in byte del file prima della compressione.
 *  C:= linearizzazione dell'abero di huffman scritto mediante l'apposita funzione
 *  Parametri:
 *  reader                    - Puntatore al bit-stream che legge dal file
 *  Ritorno:
 *  Ritorna una stringa contenente la vecchia estensione letta dell'header
 **/
string Huffman::readHeader(iBitStream* reader){
    char c;
    string old_extension;

    do{                                                                  //Leggo l'estensione vecchia del file
        *reader >> c;
        old_extension += c;
    }while(c!='\0');

    for(unsigned int i = 0; i < sizeof(unsigned long int); i++){         //Leggo total_symbols
        *reader >> ( (char*)&total_symbols )[i];
    }

    huffmanTree = readHuffmanTree(reader);                               //Leggo l'albero di huffman linearizzato.

    if(reader->getSeekByte() !='\0')                                     //Se dopo la lettura dell'albero, il seek del buffer
        reader>nextByte();                                               //Non si trova sul terminatore allora passo al prossimo byte

    Reader->nextByte();                                                  //Sposto avanti il seek del buffer in modo da spostarlo
    *reader >> c;                                                        //sul blocco di byte da leggere.
    return old_extension;
}
```

```
280. /** Descrizione:
281.  *  Comprime il file fornito in input tramite path effettuando le seguenti operazioni:
282.  *  a) Calcola la frequenza dei simboli con symbolsFrequency().
283.  *  b) Costruisco l'albero di huffman e genero i codici con buildHuffmanTree() e generateCodes().
284.  *  c) Genero e scrivo l'header sul file con writeHeader().
285.  *  d) Per ogni simbolo letto dal file da comprimere, scrivo sul file compresso il corrispettivo codice di huffman
286.  *  Parametri:
287.  *  path                    - Stringa contenente il percorso del file
288.  **/
289. void Huffman::compress(string path){
290.     symbolsFrequency(path);                                                 //Calcolo la frequenza dei simboli
291.     buildHuffmanTree();                                                     //Costruisco l'albero e genero i codici
292.     generateCodes();
293.     writeHeader(path);                                                      //Scrivo l'header
294.     string new_path = removeExtension(path) + '.' + COMPRESS_EXTRENSION;
295. 
296.     {   //Nuovo scope per sfruttare i distruttori degli stream, in modo da stampare le statistiche.
297.         iBitStream reader(path,BUFFER_SIZE,ios::binary);                    //Stream per la lettura
298.         oBitStream writer(new_path, BUFFER_SIZE, ios::app | ios::binary);   //Stream per la scrittura, app
299.                                                                             //per scrivere dopo l'header
300.         char c;
301.         while(reader >>  c){                                                //Per ogni simbolo letto dal buffer
302.             code& codice = huffmanCodes[ (byte)c ];                         //Associo il relativo codice e lo scrivo sul file.
303.             for(bool bit : codice)  writer << bit;
304.         }
305.     }
306. 
307.     printStats(path,new_path);
308. }
309. 
310. /** Descrizione:
311.  *  Decomprime il file fornito in input tramite path effettuando le seguenti operazioni:
312.  *  a) Legge l'header del file ricreando cosi' l'albero di huffman e ottenendo la vecchia estensione.
313.  *  b) Legge bit a bit il file tramite i quali percorre l'albero (1 -> figlio SX, 0 -> figlio DX) fino a che non si raggiunge
314.  *     una foglia, viene quindi scritto sul file il suo simbolo contenuto e si riparte poi a percorrere l'albero dalla radice.
315.  *  Parametri:
316.  *  path                - Stringa contenente il percorso del file
317.  **/
318. void Huffman::decompress(string path){
319.     string old_extension;
320. 
321.     {   //Nuovo scope per sfruttare i distruttori degli stream, in modo da stampare le statistiche.
322.         iBitStream reader(path,BUFFER_SIZE, ios::binary);               //BitStream di input
323.         old_extension = readHeader(&reader);                            //Leggo l'header e salvo la vecchia estensione
324.         oBitStream writer(removeExtension(path) + '.' + old_extension,BUFFER_SIZE, ios::binary); //BitStream di output
325. 
326.         HNode* pnt = huffmanTree;                                       //Si parte dalla radice
327.         bool bit;
328.         while(reader >> bit){                                          //Per ogni bit letto dallo stream...
329. 
330.             if(bit)                                                    //Percorro l'albero, se il bit e' 1 -> figlio SX
331.                 pnt = pnt->getLeft();
332.             else                                                       //altrimenti se e' 1 -> figlio DX.
333.                 pnt = pnt->getRight();
334. 
335.             if((pnt->getLeft() == 0 and pnt->getRight() == 0)){   //Se sono arrivato ad una foglia, scrivo il suo simbolo contenuto
336.                 char c = (pnt->getSymbol());                      //E ricomincio l'esplorazione dell'albero dalla sua radice
337.                 if(total_symbols-- > 0)                           //Se ho scritto tutti i simboli originari, smetto di scrivere
338.                     writer << c;                                  //L'ultimo byte infatti potrebbe contenere dei bit sporchi
339.                     pnt = huffmanTree;
340.             }
341. 
342.         }
343. 
344.     }
345.     printStats(path,removeExtension(path) + '.' + old_extension);
346. }
347. 
```

```
348. /** Descrizione:
349.  *  Calcola l'entropia di shannon del file calcolata fra [0,1]
350.  *  Parametri:
351.  *  path            - Stringa contenente il percorso del file
352.  *  Ritorno:
353.  *  Un valore compreso tra [0,1] rappresentante l'entropia del file.
354.  * */
355. long double Huffman::getEntropy(string path){
356.     if(freq_table == nullptr)                                   //Se la tavola delle frequenze è ancora vuota...
357.         symbolsFrequency(path);
358.
359.     long double p = 0, h = 0;                                   //p è la probabilità frequentista di un certo simbolo i nel file.
360.                                                                 //h e' dove viene memorizzata l'entropia
361.     for(int i = 0; i< MAX_SYMBOLS; i++){
362.         p=(long double)freq_table[i]/(long double)total_symbols ;
363.         if(p != 0)
364.         h+=p*log2(p);
365.     }
366.     return -h;
367. }
368.
369. /** Descrizione:
370.  *  Rimuove l'estensione da una stringa path fornita in input
371.  *  Parametri:
372.  *  path            - Stringa contenente il percorso del file
373.  *  Ritorno:
374.  *  Una stringa contenente il path senza estenzione
375.  **/
376. const string Huffman::removeExtension(const string& path) const{
377. /**Funzione che dato il path di un file, restituisce una stringa con estensione rimossa.*/
378.     size_t lastdot = path.find_last_of(".");        //Il tipo di ritorno di find_last_of e' size_t
379.     if (lastdot == string::npos)  return path;      //npos sta ad indicare "nessuna occorrenza" ed e' uguale al
380.                                                     //massimo valore possibile per size_t (2^32-1) oppure (2^64-1)
381.
382.       //Ritorna la sottostringa che va da 0 all'ultimo punto escluso    [0,lastdot[
383.       return path.substr(0, lastdot);
384. }
385.
386. /** Descrizione:
387.  *  Estrae l'estensione da una stringa path fornita in input
388.  *  Parametri:
389.  *  path            - Stringa contenente il percorso del file
390.  *  Ritorno:
391.  *  Una stringa contenente l'estenzione del path fornito in input
392.  **/
393. const string Huffman::extractExtension(const string& path) const{
394. /**Funzione che dato il path di un file, restituisce l'estensione.*/
395.     size_t lastdot = path.find_last_of(".");        //Il tipo di ritorno di find_last_of e' size_t
396.     if (lastdot == string::npos) return "";         //npos sta ad indicare "nessuna occorrenza" ed e' uguale al
397.                                                     //massimo valore possibile per size_t (2^32-1) oppure (2^64-1)
398.
399.     //DA DOPO IL PUNTO FINO ALLA FINE
400.     return path.substr(lastdot+1);
401. }
402.
403. /** Descrizione:
404.  * Stampa sul terminale informazioni riguardanti la compressione come la dimensione dei file e
405.  * il rapporto di compressione ottenuto.
406.  * */
407. void Huffman::printStats(string path1, string path2) const{
408.     fstream file1(path1.c_str());
409.     fstream file2(path2.c_str());
410.
411.     file1.seekg(0,file1.end);
412.     file2.seekg(0,file2.end);
413.
414.     unsigned long int size1 =   file1.tellg();
415.     unsigned long int size2 =   file2.tellg();
416.     cout << endl;
417.     cout << std::fixed << "Dimensioni del file prima dell'elaborazione:\t " << size1 << " Byte." << endl;
418.     cout << std::fixed << "Dimensioni del file dopo dell'elaborazione:\t " << size2 << " Byte." << endl;
419.     cout << std::fixed << "Rapporto di compressione: " << ((double)size1-(double)size2)/size1*100<< "%"<<endl;
420.     cout << endl;
421. }
```



Albero di Huffman

Per generari i codici di huffman affinchè l'efficienza della compressione sia massimizzata, si ricorre ad una tecnica greedy, i simboli e le loro relative codifice vengono rappresentati come un albero binario che viene costruito iterativamente in base alle frequenze di occorrenza. Tale albero ha essenzialmente le seguenti caratteristice:

- Ogni nodo foglia contiene il simbolo e la sua frequenza.
- Ogni nodo interno/padre contiene solo la somma delle frequenze dei nodi figli.

Sfruttando queste proprietà è stato creato attraverso l'uso delle classi e dell'eriditarieta una rappresentazione dell'albero binario che permette il minor spreco di spazio.

HNODE:

HNode			
Attributi			
-	`frequency : unsigned long int`	Contiene la frequenza di occorrenza del simbolo memorizzata nel nodo.	
Metodi			
+	`HNode(unsigned long int = 0)`	Costruttore con parametri, riceve in input la frequenza di occorrenza.	[Default = 0]
+	`~HNode()`	Distruttore della classe.	
+	`getFrequency() : unsigned long int`	Metodo getter, ritorna la frequenza memorizzata nel nodo.	
+	`getLeft() : HNode*`	Metodo getter, ritorna un puntatore al figlio sinistro del nodo.	
+	`getRight() : HNode*`	Metodo getter, ritorna un puntatore al figlio destro del nodo.	
+	`getSymbol() : unsigned char`	Metodo getter, ritorna il simbolo memorizzato nel nodo.	
+	`freeMemoryTree(HNode*) : void`	Funzione ricorsiva per cancellare l'albero liberandone la memoria allocata.	

`HNode`, `(Huffman Node)` definisce la classe base e più generale dal quale sono derivate `INode` e `LNode`, ha un unico attributo che è `frequency`, derivato ad entrambe e tutti i metodi *getter* sono metodi virtuali. Tramite ereditarietà e polimorfismo è possibile creare un albero binario di `INode` per i nodi interni e `LNode` per le foglie ma comunque avere ancora algoritmi generici che lavorano sulla classe base.

INODE:

INode		
Attributi		
-	`left : HNode*`	Contiene un puntatore al figlio sinistro del nodo.
-	`right : HNode*`	Contiene un puntatore al figlio destro del nodo.
Metodi		
+	`INode(HNode*,HNode*)`	Costruttore con parametri, riceve due puntatori ad Hnode* e crea un nodo padre ad entrambi.
+	`~INode()`	Distruttore di classe.
+	`getLeft() : HNode*`	Metodo getter, ritorna un puntatore al figlio sinistro del nodo.
+	`getRight() : HNode*`	Metodo getter, ritorna un puntatore al figlio destro del nodo.

`INode`, `(Internal Node)` definisce un generico nodo interno dell'albero di Huffman, in quanto tale i suoi unici attributi sono `left`, `right` rispettivamente puntatori al figlio sinistro e destro, e la frequenza ereditata dalla classe base. Sono disponibili i metodi *getter* per ottenere i figli e ovviamente non è presente un metodo per il simbolo.
Il costruttore di `INode` richiede i puntatori a due `HNode`, esso crea così un nodo padre avente quest'ultimi come figli e come frequenza la somma delle loro frequenze (a tal proposito per inizializzare la frequenza del nodo viene chiamato il costruttore della classe base). Grazie al polimorfismo non importa che siano `INode` o `LNode`, saranno trattati entrambi allo stesso modo.

LNODE:

LNode			
Attributi			
-	`symbol : unsigned char`	Simbolo memorizzato nel nodo foglia.	
Metodi			
+	`LNode(unsigned long int, char)`	Costruttore con parametri, riceve in input la frequenza di occorrenza del simbolo ed il simbolo stesso.	
+	`~LNode()`	Distruttore di classe.	
+	`getSymbol() : unsigned char`	Metodo getter, ritorna il simbolo memorizzato nel nodo.	
+	`getLeft() : HNode*`	Metodo getter, ritorna un puntatore al figlio sinistro del nodo.	Poiché LNode è un nodo foglia ritorna nullptr
+	`getRight() : HNode*`	Metodo getter, ritorna un puntatore al figlio destro del nodo.	Poiché LNode è un nodo foglia ritorna nullptr

`LNode`, `(Leaf Node)` rappresenta un nodo foglia dell'albero di Huffman. Possiede quindi oltre a `frequency` derivate dalla classe base, un solo attributo `symbol` contenente il simbolo da codificare. Essendo una foglia le funzioni *getter* per ottenere i figli ritorneranno un `nullptr`. Il costruttore con parametri richiede la frequenza e un simbolo, anche in questo caso durante la creazione viene richiamato il costruttore di `HNode` per inizializzare la parte di oggetto derivata dalla classe base.

Come si può vedere nessuna di queste tre classi utilizza metodi *setter*, questa scelta è stata fatta poiché non sono necessari durante la creazione dell'albero, una volta creato il nodo il suo contenuto resterà costante fino alla fine. Aggiungerli avrebbe creato solamente una possibilità di alterare la struttura dell'albero portando quindi ad errori durante la creazione dei codici binari per la codifica.

Listato delle classi:

Header: HNode.h

```
#ifndef HNODE_H_
#define HNODE_H_

/* Classe 'HNode' -> Huffman Node, definisce in modo generico la struttura di un nodo dell'albero di huffman*/
class HNode {
public:
    /**COSTRUTTORI*/
    HNode(unsigned long int frequency=0) : frequency(frequency){}         //Costruttore con parametri.

    /**DISTRUTTORE*/
    virtual ~HNode(){}                                                    //Distruttore.
    void freeTreeMemory(HNode* root);                                     //Funzione ricorsiva, libera la memoria di un albero di nodi

    /**METODI GETTER & SETTER*/
    unsigned long int getFrequency() const {return frequency;}            //Metodo getter, restituisce la frequenza del nodo.
    virtual HNode* getLeft() {return nullptr;}                            //Metodo getter, restituisce un puntatore al figlio sinistro
    virtual HNode* getRight() {return nullptr;}                           //Metodo getter, restituisce un puntatore al figlio destro.
    virtual unsigned char getSymbol() const {return 0;}                   //Metodo getter, restituisce un il simbolo memorizzato.

private:
    /**ATTRIBUTI PRIVATI*/
    unsigned long int frequency;                                          //Frequenza del simbolo

};
#endif /* NODE_H_ */
```

Source: HNode.cpp

```
#include "HNode.h"

void HNode::freeTreeMemory(HNode* root){
    if(root==nullptr)
        return;
    freeTreeMemory(root->getLeft());
    freeTreeMemory(root->getRight());

    delete root;
}
```

Header: INode.h

```
#ifndef INODE_H_
#define INODE_H_

#include "HNode.h"

/* Classe 'INode' -> Internal Node, derivata dalla classe HNode, definisce appunto un nodo interno dell'albero di huffman
 * dotato di puntatori ai nodi figli e di un campo frequenza il cui valore e' pari alla somma delle frequenze nei nodi figli*/
class INode: public HNode {
public:
    /**COSTRUTTORE*/ //Creo un nodo che ha come figli a e b e come frequenza la somma delle frequenze.
    INode(HNode* a, HNode* b) : HNode(a->getFrequency()+b->getFrequency()), left(a), right(b) {}; //Costruttore con parametri.
    virtual ~INode(){};                                                                           //Distruttore.

    /**METODI GETTER & SETTER*/
    HNode* getLeft() {return left;}                                 //Metodo getter, restituisce un puntatore al figlio sinistro.
    HNode* getRight() {return right;}                               //Metodo getter, restituisce un puntatore al figlio destro.

private:
    HNode* left;                                                    //Puntatore al figlio sinistro.
    HNode* right;                                                   //Puntatore al figlio destro.
};

#endif /* INODE_H_ */
```

Header: LNode.h

```
#ifndef LNODE_H_
#define LNODE_H_

#include "HNode.h"

/* Classe 'LNode' -> Leaf Node, definisce un nodo foglia dell'albero di huffman, dotato cioe' oltre che di frequenza anche
 * del campo contenente il simbolo.*/
class LNode: public HNode {
public:
    /**COSTRUTTORI*/
    LNode(unsigned long int f, unsigned char s) : HNode(f), symbol(s){} //Costruttore con parametri.
    /**DISTRUTTORE*/
    virtual ~LNode(){};                                                 //Distruttore.
    /**METODI GETTER & SETTER*/
    unsigned char getSymbol()const{return symbol;}                      //Metodo getter, restituisce un il simbolo memorizzato.
    HNode* getLeft() {return nullptr;}                                  //Metodo getter, restituisce un puntatore al figlio sinistro.
    HNode* getRight() {return nullptr;}                                 //Metodo getter, restituisce un puntatore al figlio destro.
                                                                        //Poiché LNode è una foglia i metodi restituiscono nullptr.

private:
    unsigned char symbol;                                               //Simbolo memorizzato nel nodo foglia.
};

#endif /* LNODE_H_ */
```

BitStream

Quando memorizziamo un file, la più piccola unità di informazione con cui si può rappresentare è il byte, pertanto esso verrà memorizzato mediante suoi multipli e dei bit aggiuntivi saranno concatenati se necessario. In oltre anche nelle operazioni base di lettura e scrittura la più piccola unità con cui si può operare è il byte.
Ciò tuttavia porta a dei problemi significativi durante lo sviluppo e l'implementazione della codifica di Huffman, essa infatti richiede di scrivere o leggere i file mediante singoli bit oppure di memorizzarne un numero arbitrario non necessariamente multiplo di un byte, ad esempio quando al simbolo viene sostituita la relativa codifica.

La risoluzione di questo problema è però abbastanza semplice, occorre infatti scrivere infatti basta utilizzare un byte di buffer:

- Scrittura: Invece di scrivere sul file i bit si riempie il buffer usando gli operatori bitwise, una volta che il buffer è pieno finalmente si è in grado di scriverlo sul file.
- Lettura: Viene letto un byte dal file, tramite sempre gli operatori bitwise ne vengono estratti i vari bit che lo compongono e sono quindi usati nell'algoritmo, una volta estratti tutti i bit si legge un altro byte e si ricomincia il processo.

Inizialmente questo modo di procedere con un buffer byte a byte era stato incluso direttamente nelle funzioni `compress()` e `decompress()` tuttavia mi sono reso subito conto che questa implementazione non solo poteva essere confusionaria ma in oltre poiché il buffer era piccolo il numero di accessi alla memoria di massa era notevole a tutto svantaggio ovviamente delle prestazioni. Perciò ho deciso di creare una mia classe che si occupasse della scrittura dei bit, la `BitStream`.

I primi vantaggi sono senza dubbio quelli legati al codice, molto più semplice da capire, si è passato infatti da questo:

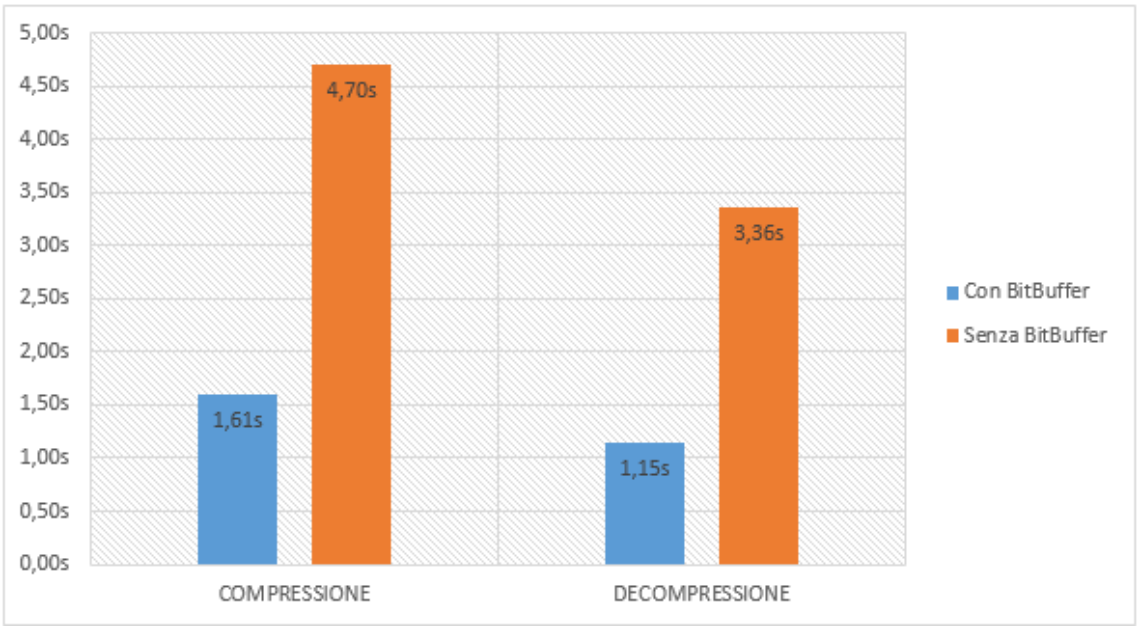
```
        frequency(path);    //Calcolo la frequenza dei caratteri nel file
        buildTree();        //Costruisco l'albero di Huffman
        generateCodes();    //Genero i codici di Huffman

        //Ifstream richiede un const char* e non una stringa!
        ifstream file(path.c_str(), ios::in|ios::binary);  //Apro il mio file in modalitÃ  lettura e binario.
        ofstream output((removeExtension(path)+COMPRESS_EXTRENSION).c_str(), ios::out|ios::binary); //Creo il file.

        cout << "Effettuo la compressione..." << endl;
        /**SCRIVO L'HEADER DEL FILE COMPRESSO*/
        writeHeader(output, path);                      //Scrivo l'header in cima al file.

        /**Leggo i singoli byte per poi sostituirli con il codice di Huffman*/
        byte in;                                        //Buffer dove viene memorizzato il byte letto
        byte out=0;                                     //Buffer dove viene scritto un byte compresso
        int bit=0;                                      //Contatore del numero di bit già scritti nel buffer out
```

```
17.         while(file >> std::noskipws >> in){         //noskipws permette di leggere TUTTI i caratteri, spazi compresi.
18.             code& c = huffmanCodes[in];             //reference al codice di Huffman relativo al carattere letto
19.             for(unsigned int j=0; j<c.size(); j++){ //Scrivo bit a bit il codice nel buffer
20.                 if(bit<8){                          //Se ho spazio nel buffer scrivo
21.                     out|=(c[j]<<(8-bit-1));         //Scrivo il j-esimo bit del codice nella 1° posizione disp del buffer, partendo da sx
22.                     bit++;                          //tale informazione mi è data dal valore della variabile 'bit'
23.                 }
24.
25.                 if(bit==8){                         //Se ho riempito il buffer, lo scrivo e riparto a scrivere dalla posizione 0
26.                     output<<out;                    //Scrivo sul file
27.                     bit=0;                          //Azzero il contatore dei bit scritti (parto da 0)
28.                     out = 0;                        //Azzero il buffer out
29.                 }
30.             }
31.         }
32.         /*Alla fine dei cicli se restano dei bit nel buffer non ancora scritti, cioè che non hanno 'completato' il byte, li scrivo*/
33.         if(bit>0)                                   //Scrivo i restanti bit nel buffer
34.             output<<out
```

a questo:

```
1.      symbolsFrequency(path);                                         //Calcolo la frequenza dei simboli
2.      buildHuffmanTree();                                             //Costruisco l'albero e genero i codici
3.      generateCodes();
4.      writeHeader(path);                                              //Scrivo l'header
5.      string new_path = removeExtension(path) + '.' + COMPRESS_EXTRENSION;
6.
7.
8.          iBitStream reader(path,BUFFER_SIZE,ios::binary);            //Stream per la lettura
9.          oBitStream writer(new_path, BUFFER_SIZE, ios::app | ios::binary);   //Stream per la scrittura, app
10.                                                                     //per scrivere dopo l'header
11.         char c;
12.         while(reader >>  c){                                        //Per ogni simbolo letto dal buffer
13.             code& codice = huffmanCodes[ (byte)c ];                 //Associo il relativo codice di huffman e lo scrivo
14.             for(bool bit : codice)  writer << bit;
15.         }
16.
```

Un altro vantaggio sono le prestazioni qui di seguito vi è un confronto:
Il file utilizzato per il test è un'immagine bitmap di circa 50MB utilizzando un buffer di 8KB, come si può vedere dal grafico le differenze sono notevoli si passa da $4{,}70\ sec.$ a $1{,}61\ sec.$ per la compressione e da $3{,}36\ sec.$ a $1{,}15\ sec.$ per la decompressione con un incredibile aumento delle prestazioni del $\sim 219{,}9\%$!



In seguito verrà illustrato più in dettaglio l'implementazione della classe.

BITSTREAM:

BitStream	
Attributi	
# buffer_size : const int	Contiene la dimensione in byte del buffer.
# buffer : char*	Il buffer stesso, rappresentato come un array dinamico di char.
# buffer_seek : int	Contiene il seek del buffer, in modo da conoscere sempre la posizione in lettura o scrittura sul buffer stesso
# byte_seek : unsigned char	Contiene il seek del byte puntato dal buffer_seek, consente di conoscere quanti bit del byte sono stati scritti/letti.
Metodi	
+ BitStream(int)	Costruttore di classe, ha come parametro le dimensioni del buffer.
+ ~BitStream()	Distruttore della classe, dealloca la memoria ed eventualmente scrive i bit restanti nel buffer.
+ write() : unsigned char	Scrive il contenuto del buffer sul file, ritorna il numero di trash bit aggiunti per completare l'ultimo byte.
+ read() : int	Legge memorizzando nel buffer il file, ritorna il numero di byte letti.
+ getBuffer() : char*	Metodo getter, ritorna un puntatore al buffer.
+ getSeekByte() : char	Metodo getter, ritorna il byte del buffer indicizzato dalla posizione del byte_seek.
+ nextByte() : void	Consente di avanzare di un byte sul buffer.
+ flush() : void	Svuota il buffer e resetta a zero le posizioni dei seek.
+ printBin() : void	Stampa su console in formato binario il contenuto del buffer in questo momento.
+ printHex() : void	Stampa su console in formato hex il contenuto del buffer in questo momento.
# print_bit() : void	Metodo privato, converte un char in binario è una funzione di appoggio per printBin().

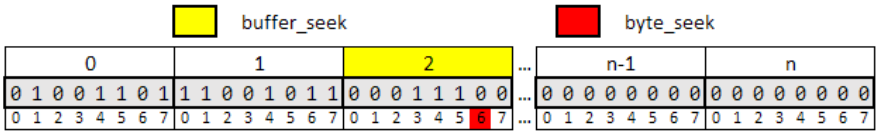
`BitStream` è la classe principale che definisce lo stream di bit dal quale derivano `oBitStream` e `iBitStream` rispettivamente stream di output e input. Lo stream è internamente rappresentata con un array dinamico di char che per la scrittura, viene riempito byte per byte usando gli operatori bitwise sui bit che vengono mandati allo stream, quando il buffer è pieno questo viene infine scritto sul file, per la lettura il processo è analogo ma in direzione opposta prima leggo il file con il buffer e poi ne manipolo e uso i bit.
Ciò permette di ridurre al minimo le operazioni di I/O con le memorie di massa estremamente più lente della memoria RAM e quindi consente un grande aumento delle prestazioni ovviamente a discapito dello spazio utilizzato per memorizzare il buffer.

La classe base presenta i suoi attributi con visibilità *protected*, in modo tale da potervi accedere liberamente dalle classi derivate.

OBITSTREAM:

oBitStream	
Attributi	
- file : ofstream	Output stream al file che si sta scrivendo con il buffer.
Metodi	
+ oBitStream(string,int,_ios_Openmode)	Costruttore con parametri, riceve in input la path del file, la dimensione del buffer, e l'open mode del file.
+ ~oBitStream()	Distruttore della classe, dealloca la memoria ed eventualmente scrive i bit restanti nel buffer.
+ write() : unsigned char	Scrive il contenuto del buffer sul file, ritorna il numero di trash bit aggiunti per completare l'ultimo byte.
+ getFile() ofstream&	Metodo getter, ritorna un reference allo stream del file che si sta scrivendo
+ flush() : void	Svuota il buffer e resetta a zero le posizioni dei seek.
+ operator<<(bool) : oBitStream&	Overload dell'operatore <<, scrive il bit in input nel buffer.
+ operator<<(char) : oBitStream&	Overload dell'operatore <<, scrive il char in input nel buffer.
- endOfBufferCheck() : void	Controlla se il buffer e' terminato e se necessario lo scrive ripartendo da capo.

`oBitStream`, `(output BitStream)` eredita dalla classe `BitStream` la struttura interna del bit-stream e fornisce essenzialmente solo le funzioni base per la scrittura. L'overload dell'operatore `<<` rende in oltre l'uso di questa classe molto intuitivo e simile a quello degli stream base definiti dal linguaggio stesso.



Possiamo idealmente immaginare in questo modo il nostro stream. Man mano che si inseriscono i bit alla posizione indicizzata dai due seek il buffer si riempie ed in fine quando è pieno viene automaticamente scritto sul file.
La classe permette come si vede dai metodi di operare anche tramite char, questa scelta è stata fatta in quanto spesso è necessario inserire un ottetto di bit come ad esempio quando si memorizza un carattere, si evitano così cicli per scorrere con operatori bitwise il char per inviarli allo stream. In oltre ciò permette anche un'ulteriore ottimizzazione:

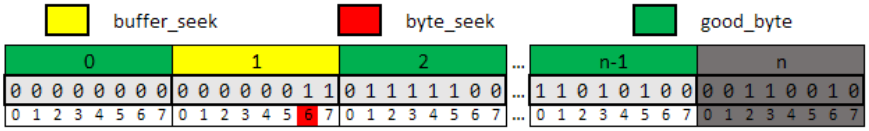
- Se inviando il char allo straem il `byte_seek` si trova alla posizione `0`, cioè all'inizio del byte basta semplicemente copiare il contenuto del char nella cella del buffer ed aumentare il `buffer_seek.`
- Negli altri casi è necessario inviare allo stream ogni singolo bit del char.

Quindi per come è definita la classe si può comportare anche come un semplice buffer di byte.

IBITSTREAM:

iBitStream	
Attributi	
- file : ifstream	Input stream al file che si sta leggendo con il buffer.
- goofByte : int	Numero di byte effettivamente letti e usabili nel buffer.
Metodi	
+ iBitStream(string,int,_ios_Openmode)	Costruttore con parametri, riceve in input la path del file, la dimensione del buffer, e l'open mode del file.
+ ~iBitStream()	Distruttore della classe, dealloca la memoria usata dal buffer.
+ read() : int	Legge memorizzando nel buffer il file, ritorna il numero di byte letti.
+ getFile() ifstream&	Metodo getter, ritorna uno stream al file che si sta leggendo
+ flush() : void	Svuota il buffer e resetta a zero le posizioni dei seek.
+ operator>>(bool&) : bool	verload dell'operatore >>, legge un bit dal buffer.
+ operator>>(char&) : bool	verload dell'operatore >>, legge un byte dal buffer.
- endOfBufferCheck() : void	Controlla se il buffer e' terminato e se necessario lo rilegge.

`iBitStream`, `(input BitStream)` è la classe complementare a quella vista in precedenza e fornisce essenzialmente solo le funzioni base per la lettura. Anche in questo caso, l'overload dell'operatore `>>` permette un facile uso dello stream.



Lo stream è essenzialmente identico al precedente ma in aggiunta abbiamo l'attributo `good_byte`. Esso ci dà informazioni riguardanti il numero effettivo di byte letti dal buffer, in modo da evitare di leggere bit che non fanno parte del file.
Si consideri il seguente esempio: Sto leggendo un file grande 1000 byte tramite un buffer di 256 byte, le prime tre letture avvengono senza problemi il buffer si riempie totalmente durante la lettura e quindi `good_byte = 256`. Alla quarta lettura però nel file restano solo 232 byte che non sono necessari a riempire il buffer pertanto tenendo conto di ciò `good_byte = 232` so che del mio buffer solo i primi 232 fanno parte del file.

Man mano che si leggono i bit alla posizione indicizzata dai due seek il buffer si svuota ed in fine quando vuoto viene automaticamente riletto dal file. La classe permette di operare sia sui bit che sui char con gli stessi vantaggi descritti in precedenza.

Listato delle classi:

Header: BitStream.h

```
1.  #ifndef BITSTREAM_H_
2.  #define BITSTREAM_H_
3.  #include <iostream>
4.  #include <string.h>
5.  #include <fstream>
6.  using namespace std;
7.
8.  /**Classe che simula un bit stream per la scrittura/lettura su file, permettendo di scrivere o leggere bit a bit un file
9.   * la classe per velocizzare le operazioni utilizza un buffer che viene scritto solamente quando e' pieno*/
10. class BitStream {
11. public:
12.     BitStream(int bufferSize);                    //Costruttore della classe, ha come parametro la dimensione del buffer.
13.     virtual ~BitStream();                         //Distruttore della classe, dealloca la memoria e scrive i bit restanti nel buffer
14.
15.     unsigned char write();                        //Scrive il contenuto del buffer sul file, ritorna il numero di trash bit scritti.
16.     int read();                                   //Legge memorizzando nel buffer il file, ritorna il numero di byte letti.
17.     char* getBuffer() {return buffer;}            //Metodo getter, ritorna un puntatore al buffer.
18.     char getSeekByte(){return buffer[buffer_seek];} //Metodo getter, ritorna il byte del buffer indicizzato dal byte_seek.
19.     void nextByte(){                              //Consente di avanzare di un byte sul buffer.
20.         buffer_seek++;
21.         byte_seek = 0;
22.     }
23.     virtual inline void flush(){}                 //Svuota il buffer e resetta a zero le posizioni dei seek.
24.     void printBin();                              //Stampa su console in formato binario il contenuto del buffer in questo momento.
25.     void printHex();                              //Stampa su console in formato hex il contenuto del buffer in questo momento.
26.
27. protected:
28.     char* buffer;                                 //Buffer, rappresentato come Array di char
29.     const int buffer_size;                        //Dimensione del buffer, fornita in input
30.     int buffer_seek;                              //Seek del buffer, indica la posizione per la lettura/scrittura sul buffer
31.     unsigned char byte_seek;                      //Seek del byte, indica la posizione di un bit sul byte del buffer_seek
32.
33.     void print_bit(char c);                       //Metodo privato, converte un char in binario per la stampa.
34. };
35. #endif /* BITSTREAM_H_ */
```

Source: BitStream.cpp

```cpp
#include "BitStream.h"

/**Descrizione:
 * Costruttore della classe, verifica la bonta' dell'input ed inizializza i vari attributi della classe
 * Parametri:
 * bufferSize               - Dimensione del buffer da utilizzare per la lettura/scrittura
 * */
BitStream::BitStream(int bufferSize) : buffer_size(bufferSize) {
    if(bufferSize <= 0){
        cerr << "ERRORE: IL BUFFER NON PUO' AVERE SIZE NEGATIVO O NULLO." << endl;
        return;
    }

    buffer = new char[bufferSize]();
    buffer_seek = 0;
    byte_seek = 0;
}

/**Descrizione:
 * Distruttore della classe, libera la memoria allocata per il buffer.
 * */
BitStream::~BitStream() {
    delete[] buffer;
}

/**Descrizione:
 * Stampa il contenuto del buffer, byte a byte, in formato binario, utilizza print_bit().
 * */
void BitStream::printBin(){
    for(int i = 0; i<buffer_size; i++){
        print_bit(buffer[i]);
        std::cout << " ";
    }
}

/**Descrizione:
 * Stampa il contenuto del buffer, byte a byte, in formato esadecimale.
 * */
void BitStream::printHex(){
    for(int i = 0; i<buffer_size; i++){
        std::cout << std::hex << (int)buffer[i] << " ";
    }
}

/**Descrizione:
 * Stampa il char fornito in input in formato binario.
 * Parametri:
 * c                        - char da stampare in binario.
 * */
void BitStream::print_bit(char c){
    for(int i = 0; i<CHAR_BIT; i++){
        std::cout << (c<0);                     //Stampo il bit più significativo
        c = c << 1;                             //e shift a destra di un bit.
    }

}
```

Header: oBitStream.h

```
#ifndef OBITSTREAM_H_
#define OBITSTREAM_H_
#include "BitStream.h"

/** Classe che simula un output bit-stream, eredita dalla classe BitStream la struttura interna del bit-stream
 *  e fornisce essenzialmente solo le funzioni base per la scrittura.
 *  */
class oBitStream : public BitStream {
public:
    oBitStream(string path, int bufferSize, _Ios_Openmode openMode = ios::out); //Costruttore di classe con parametri.
    virtual ~oBitStream();                                                      //Distruttore di classe.

    ofstream& getFile() {return file;}                              //Metodo getter, ritorna uno stream al file che si sta scrivendo
    unsigned char write();                                          //Scrive il buffer nel file, ritorna il numero di trash bit.
    inline void flush();                                            //Svuota il buffer e resetta a zero le posizioni dei seek.
    oBitStream& operator<< (bool b);                                //Overload dell'operatore <<, scrive il bit in input nel buffer
    oBitStream& operator<< (char c);                                //Overload dell'operatore <<, scrive il char in input nel buffer

private:
    ofstream file;                                                  //Output stream al file che si sta scrivendo con il buffer
    void endOfBufferCheck();                                        //Controlla se il buffer e' terminato e se necessario lo rilegge.

};
#endif /* OBITSTREAM_H_ */
```

Source: oBitStream.cpp

```
#include "oBitStream.h"

/** Descrizione:
 *  Costruttore della classe, inizializza l'output bit-stream. Viene chiamato il costruttore della classe base.
 *  Parametri:
 *  path            - Stringa contenente il percorso al file che si vuole scrivere con lo stream.
 *  bufferSize      - Dimensione del buffer da utilizzare per la scrittura.
 *  openMode        - Spacifica in che modo aprire il file per la scrittura            [Default = ios::out]
 * */
oBitStream::oBitStream(string path, int bufferSize, _Ios_Openmode openMode) : BitStream(bufferSize){
    if(path == "") {
        cerr << "ERRORE: IMPOSSIBILE APRIRE IL FILE IN SCRITTURA PER IL BITSTREA, PATH NON VALIDA.\n";
        return;
    }

    file.open(path.c_str(), openMode);

    if(!file.is_open()) {
        cerr << "ERRORE: IMPOSSIBILE APRIRE IL FILE IN SCRITTURA PER IL BITSTREAM.\n";
        return;
    }
}

/** Descrizione:
 *  Distruttore della classe, scrive cio' che resta sul buffer nel file ed infine lo chiude.
 *  NB: la memoria viene liberata dal distruttore della classe base.
 * */
oBitStream::~oBitStream() {
    write();
    file.close();
}

/** Descrizione:
 *  Funzione che svuota il buffer setta a zero tutti i suoi byte e resetta a zero il byte_seek ed il buffer_seek.
 * */
inline void oBitStream::flush(){
    memset(buffer, 0 , buffer_size);                        //Uso memset perche molto piu' veloce di altri metodi.
    byte_seek = 0;
    buffer_seek = 0;
}

```

```
42. /** Descrizione:
43.  *  Scrive il buffer sul file.
44.  *  Ritorno:
45.  *  Ritorna il numero di trash bit, cioe' i bit che non facevano parte del buffer ma necessari per scrivere il byte.
46.  * */
47. unsigned char oBitStream::write(){
48.     if(!byte_seek and !buffer_seek)                         //Se il buffer e' vuoto, non ho nulla da scrivere.
49.     return 0;
50.
51.     if(byte_seek == 0)                                      //Se il byte_seek e' zero, scrivo fino al buffer_seek...
52.     file.write(buffer,buffer_seek);
53.     else                                                    //Altrimenti devo scrivere fino buffer_seek poiche' ci sono
54.     file.write(buffer,buffer_seek+1);                       //bit nel buffer che non hanno completato un byte.
55.
56.     return 8-byte_seek;
57. }
58.
59. /** Descrizione:
60.  *  Una volta effettuate operazioni di scrittura, si controlla se il buffer e' pieno, in tal caso avviene una sua scrittura
61.  *  sul file e pertanto sono reinizializzati byte_seek e buffer_seek.
62.  * */
63. void oBitStream::endOfBufferCheck(){
64.     if(byte_seek >= 8){                                     //Se dopo aver scritto un bit, il byte corrente e' terminato...
65.         byte_seek = 0;                                      //Resetto byte_seek ed incremento buffer_seek per scrivere il prossimo
66.         buffer_seek++;
67.     }
68.
69.     if(buffer_seek >= buffer_size){                         //Se ho riempito tutto il buffer...
70.         file.write(buffer,buffer_size);                     //Scrivo il buffer sul file e ricomincio da capo.
71.         flush();                                            //Pulisco il buffer per prepararlo ad una nuova lettura
72.     }
73. }
74.
75. /** Descrizione:
76.  *  Overload dell'operatore <<, la funzione scrive un singolo bit in input nel buffer. Dopo aver scritto il bit
77.  *  si aggiorna il byte_seek e si chiama la funzione endOfBufferCheck(), per verificare se il buffer/byte e' terminato.
78.  *  Parametri:
79.  *  b               - booleano da scrivere nel buffer.
80.  *  Ritorno:
81.  *  la funzione ritorna un reference all'oggetto stesso in modo da effettuare chiamate sequenziali.
82.  *  */
83. oBitStream& oBitStream::operator<< (bool b){
84.     if(b){                                                  //Se il bit da inserire e' 1, lo insersco alla sua posizione
85.         unsigned char mask = 1 << (7-byte_seek);            //tramite una maschera, altrimenti e' necessario solo aumentare
86.         buffer[buffer_seek] |= mask;                        //il byte_seek.
87.     }
88.         byte_seek++;
89.
90.         endOfBufferCheck();                                 //Effettuo i controlli per verificare se il byte/buffer e' terminato.
91.
92.     return *this;
93. }
94. /** Descrizione:
95.  *  Overload dell'operatore <<, la funzione scrive un byte in input nel buffer. Dopo aver scritto il byte
96.  *  si aggiornano i seek e si chiama la funzione endOfBufferCheck(), per verificare se il buffer/byte e' terminato.
97.  *  Parametri:
98.  *  c               - char da scrivere nel buffer.
99.  *  Ritorno:
100. *  la funzione ritorna un reference all'oggetto stesso in modo da effettuare chiamate sequenziali.
101. *  */
102. oBitStream& oBitStream::operator<< (char c){
103.     if(byte_seek == 0){                                    //Se il byte_seek e' 0, mi trovo all'inizio del byte e quindi posso
104.         buffer[buffer_seek++] = c;                         //scrivere c direttamente nel buffer.
105.         endOfBufferCheck();
106.         return *this;
107.     }
108.
109.     for(int i = 0; i < CHAR_BIT; i++){                     //In caso contrario effettuo CHAR_BIT immissioni di bit dal buffer
110.         *this << (c<0);                                    //in modo tale che possa scrivere il char.
111.         c <<=1;
112.     }
113.     return *this;
114. }
```

Header: iBuffStream.h

```
#ifndef IBITSTREAM_H_
#define IBITSTREAM_H_
#include "BitStream.h"

/** Classe che simula un input bit-stream, eredita dalla classe BitStream la struttura interna del bit-stream
 *  e fornisce essenzialmente solo le funzioni base per la lettura.
 *  */
class iBitStream : public BitStream {
public:
    iBitStream(string path, int bufferSize, _Ios_Openmode openMode = ios::in);  //Costruttore di classe con parametri.
    virtual ~iBitStream();                                                      //Distruttore di classe.

    ifstream& getFile() {return file;}                                  //Metodo getter, ritorna uno stream al file che si sta leggendo
    int read();                                                         //Legge nel buffer il file, ritorna il numero di byte letti.
    inline void flush();                                                //Svuota il buffer e resetta a zero le posizioni dei seek.
    bool operator>> (bool& b);                                          //Overload dell'operatore >>, legge un bit dal buffer in ‘b’
    bool operator>> (char& c);                                          //Overload dell'operatore >>, legge un byte dal buffer e in ‘c’

private:
    ifstream file;                                                      //Input stream al file che si sta leggendo con il buffer
    int goodByte;                                                       //Numero di byte effettivamente letti e usabili nel buffer
    void endOfBufferCheck();                                            //Controlla se il buffer e' terminato e se necessario lo rilegge.

};
#endif /* IBITSTREAM_H_ */
```

Source: iBuffStream.cpp

```
#include "iBitStream.h"

/** Descrizione:
 *  Costruttore della classe, inizializza l'input bit-stream. Viene chiamato il costruttore della classe base.
 *  Parametri:
 *  path            - Stringa contenente il percorso al file che si vuole leggere con lo stream.
 *  bufferSize      - Dimensione del buffer da utilizzare per la lettura.
 *  openMode        - Spacifica in che modo aprire il file per la lettura              [Default = ios::in]
 * */
iBitStream::iBitStream(string path, int bufferSize, _Ios_Openmode openMode) : BitStream(bufferSize){
    if(path == "") {
        cerr << "ERRORE: IMPOSSIBILE APRIRE IL FILE IN SCRITTURA PER IL BITSTREAM, PATH NON VALIDA.\n";
        return;
    }

    file.open(path.c_str(), openMode);
    if(!file.is_open()) {
        cerr << "ERRORE: IMPOSSIBILE APRIRE IL FILE IN SCRITTURA PER IL BITSTREAM.\n";
        return;
    }
    goodByte = read();
}

/** Descrizione:
 *  Distruttore della classe, chiude il file aperto per lo stream.
 *  NB: la memoria viene liberata dal distruttore della classe base.
 * */
iBitStream::~iBitStream() {
    file.close();
}

/** Descrizione:
 *  Funzione che svuota il buffer setta a zero tutti i suoi byte e resetta a zero il byte_seek, il buffer_seek e goodByte.
 * */
inline void iBitStream::flush(){
    memset(buffer, 0 , buffer_size);                                //Uso memset perche molto piu' veloce di altri metodi.
    byte_seek = 0;
    buffer_seek = 0;
    goodByte = 0;
}
```

```
41. /** Descrizione:
42.  *  Legge una porzione di file memorizzandolo nel buffer.
43.  *  Ritorno:
44.  *  Ritorna il numero di byte effettivamente letti dal file.
45.  * */
46. int iBitStream::read(){
47.      file.read(buffer, buffer_size );
48.      return file.gcount();                                      //Torna il numero di bytes letti
49. }
50.
51. /** Descrizione:
52.  *  Overload dell'operatore >>, la funzione legge un singolo bit dal buffer e lo memorizza in b. Dopo aver letto il bit
53.  *  si aggiorna il byte_seek e si chiama la funzione endOfBufferCheck(), per verificare se il buffer/byte e' terminato.
54.  *  Parametri:
55.  *  b               - booleano dove viene memorizzato il bit letto, pertanto e' passato con un reference.
56.  *  Ritorno:
57.  *  la funzione ritorna true in caso di successo nella lettura e false in caso di insuccesso (es. file e buffer terminato).
58.  *  */
59. bool iBitStream::operator>> (bool& b){
60.
61.     if(buffer_seek >= goodByte) return false;           //Se non ho piu' bit da leggere ritorno false.
62.     b = buffer[buffer_seek] < 0;                        //Altrimenti estraggo il bit più significativo dal char puntato da buffer_seek
63.     buffer[buffer_seek] <<= 1;
64.     byte_seek++;
65.
66.     endOfBufferCheck();                                 //Effettuo i controlli per verificare se il byte/buffer e' terminato.
67.
68.     return true;
69. }
70. /** Descrizione:
71.  *  Overload dell'operatore >>, la funzione legge un byte dal buffer e lo memorizza in c. Dopo aver letto il byte
72.  *  si aggiornano i seek e si chiama la funzione endOfBufferCheck(), per verificare se il buffer/byte e' terminato.
73.  *  Parametri:
74.  *  c               - char dove viene memorizzato il bit letto, pertanto e' passato con un reference.
75.  *  Ritorno:
76.  *  la funzione ritorna true in caso di successo nella lettura e false in caso di insuccesso (es. file e buffer terminato).
77.  *  */
78. bool iBitStream::operator>> (char& c){
79.     if(buffer_seek >= goodByte) return false;           //Se non ho piu' byte da leggere ritorno false.
80.
81.     if(byte_seek == 0){                                 //Se il byte_seek e' 0, mi trovo all'inizio del byte e quindi posso
82.         c = buffer[buffer_seek++];                      //leggerlo direttamente dal buffer e memorizzarlo in c.
83.         endOfBufferCheck();
84.         return true;
85.     }
86.
87.     for(int i = 0; i < CHAR_BIT; i++){                  //In caso contrario effettuo CHAR_BIT estrazioni di bit dal buffer
88.         bool b=0;                                       //in modo tale che possa formare un char.
89.         *this >> b;
90.         c <<= 1;
91.         c |= b;
92.     }
93.     return true;
94. }
95.
96. /** Descrizione:
97.  *  Una volta effettuate operazioni di lettura, si controlla se il buffer e' stato letto tutto, in tal caso avviene una nuova lettura
98.  *  dal file e pertanto sono reinizializzati byte_seek, buffer_seek e goodByte.
99.  * */
100. void iBitStream::endOfBufferCheck(){
101.     if(byte_seek >= 8){                                    //Se dopo aver letto un bit, il byte corrente e' terminato...
102.         byte_seek = 0;                                     //Resetto byte_seek ed incremento buffer_seek per leggere il prossimo
103.         buffer_seek++;
104.     }
105.
106.     if(buffer_seek >= buffer_size){                        //Se ho letto tutto il buffer...
107.         goodByte=read();                                   //Leggo una nuova porzione di file e ricomincio da capo.
108.         buffer_seek= 0;
109.     }
110. }
```

PRIORITYQUEUE:

PriorityQueue T	
Attributi	
- heap : vector<T>	La coda e' rappresentata internamente con un heap memorizzato in un vector.
- comparator : bool(*)(T,T)	Puntatore ad una funzione per il confronto degli elementi nella coda.
Metodi	
+ Queue(bool(*)(T,T))	Costruttore con parametri, riceve in input la funzione da utilizzare per ordinare la coda.
+ ~Queue()	Distruttore di classe.
+ insert(T) : void	Inserisce un elemento nella coda.
+ remove() : T	Estrae un elemento dalla coda.
+ buildHeap() : void	Consente di costruire l'heap interno alla coda.
+ isEmpty() bool	Restituisce true se la coda e' vuota, false altrimenti.
+ size() int	Ritorna il numero di elementi presenti nella coda.
+ print() : void	Stampa il contenuto del vector con il quale e' rappresentata la coda.
- parent(int) : int	Restituisce l'indice nel vector del padre dell'elemento di indice a.
- left(int) : int	Restituisce l'indice nel vector del figlio sx dell'elemento di indice a.
- right(int) : int	Restituisce l'indice nel vector del figlio dx dell'elemento di indice a.
- changeNode(int,T) : void	Cambia in valore dell'elemento di indice in input.
- heapify(int) : void	Ripristina la proprieta' dell'heap partendo dal nodo di indice in input.

Si tratta dell'implementazione di una semplice coda di priorità basata su un heap. Il vantaggio di quest'implementazione è quello di essere basata su un template, e di ricevere la funzione necessaria al confronto direttamente con il costruttore al momento della creazione. In questo modo è possibile creare una coda di qualsiasi oggetto ed usando qualsiasi funzione di confronto, potendo passare da una coda di min-priorità ad una di max-priorità semplicemente cambiando il segno della funzione di confronto. In oltre, non è necessario più definire una funzione separata infatti il C++11 mette a disposizione le espressioni lambda, che costituiscono un modo efficace per definire una funzione anonima nella posizione in cui viene richiamata o passata come argomento a una funzione.

Come già detto la coda è basata su un heap che è rappresentato internamente attraverso un vector, pertanto tutte le funzioni che operano sull'heap utilizzando gli indici del suddetto per effettuare scambi e confronti.

Listato della classe:

Header: PriorityQueue.h

```
1.  #ifndef QUEUE_H_
2.  #define QUEUE_H_
3.  #include <iostream>
4.  #include <vector>
5.  using namespace std;
6.
7.  template<class T> class PriorityQueue {
8.  public:
9.      /**COSTRUTTORI E DISTRUTTORI*/
10.     PriorityQueue(bool (*compare)(T a,T b) );          //Costruttore con parametri.
11.     virtual ~PriorityQueue();                          //Distruttore.
12.
13.     /**FUNZIONI MEMBRO*/
14.     void print() const;                                //Stampa il contenuto del vector con il quale e' rappresentata la coda
15.     void insert(T obj);                                //Inserisce un elemento nella coda.
16.     T remove();                                        //Estrae un elemento dalla coda.
17.
18.     inline bool isEmpty() const;                       //Restituisce true se la coda e' vuota, false altrimenti.
19.     int size() const {return heap.size();}             //Ritorna il numero di elementi presenti nella coda.
20.     void buildHeap();                                  //Consente di costruire l'heap interno alla coda.
21.
22. private:
23.     vector<T> heap;                                     //La coda e' rappresentata da un heap memorizzato in un vector.
24.     bool (*comparator)(T a,T b);                        //Puntatore ad una funzione per il confronto degli elementi nella coda
25.
26.     /**FUNZIONI MEMBRO PRIVATE*/
27.     inline int parent(int a) const;                     //Restituisce l'indice nel vector del padre dell'elemento di indice a.
28.     inline int left(int a) const;                       //Restituisce l'indice nel vector del figlio sx dell'elemento di indice a.
29.     inline int right(int a) const;                      //Restituisce l'indice nel vector del figlio dx dell'elemento di indice a.
30.     void changeNode(int n, T obj);                      //Cambia in valore dell'elemento di indice n.
31.     void heapify(int n);                                //Ripristina la proprieta' dell'heap.
32.
33. };
```

```
#endif /* PRIORITYQUEUE_H_ */



//         ____        __ _       _     _                        _      _ _          __                 _             _
//        |  _ \  ___ / _(_)_ __ (_)___(_) ___  _ __   ___    __| | ___| | | ___    / _|_   _ _ __  ___(_) ___  _ __ (_)
//        | | | |/ _ \ |_| | '_ \| |_  / |/ _ \| '_ \ / _ \  / _` |/ _ \ | |/ _ \  | |_| | | | '_ \|_  / |/ _ \| '_ \| |
//        | |_| |  __/  _| | | | | |/ /| | (_) | | | |  __/ | (_| |  __/ | |  __/  |  _| |_| | | | |/ /| | (_) | | | | |
//        |____/ \___|_| |_|_| |_|_/___|_|\___/|_| |_|\___|  \__,_|\___|_|_|\___|  |_|  \__,_|_| |_/___|_|\___/|_| |_|_|
//

template <class T> PriorityQueue<T>::Queue(bool (*compare)(T a,T b) ) {     //COSTRUTTORE: Riceve in input un puntatore a funzione da
    if(compare != nullptr)                                                  //             utilizzare per confrontare internamente
        comparator = compare;                                               //             gli elementi della coda.
}
template <class T> PriorityQueue<T>::~Queue() {}                            //DISTRUTTORE

template <class T> inline int PriorityQueue<T>::parent(int a) const{   //PARENT: Restituisce l'indice, nel vector, del padre del nodo
    if(a<=0) return -1;                                                //        memorizzato all'indice a.
        else return (a-1)/2;
}
template <class T> inline int PriorityQueue<T>::left(int a) const{     //LEFT: Restituisce l'indice, nel vector, del figlio SX del
    if(a<0 or 2*a+1 > (int)heap.size() -1) return -1;                  //      nodo memorizzato all'indice a.
        else return 2*a+1;
}
template <class T> inline int PriorityQueue<T>::right(int a) const{    //RIGHT: Restituisce l'indice, nel vector, del figlio DX del
    if(a<0 or 2*a+2 > (int)heap.size() -1) return -1;                  //       nodo memorizzato all'indice a.
        else return 2*a+2;
}

template <class T> void PriorityQueue<T>::heapify(int n){              //HEAPIFY: Ripristina la proprieta' dell'heap partendo dal nodo
    int sx = left(n);                                                  //         dell’albero rappresentato con vector n.
    int dx = right(n);
    int max = n;
    if(sx != -1 and comparator(heap[sx], heap[max]) )
        max = sx;
    if(dx != -1 and comparator(heap[dx], heap[max]) )
        max = dx;

    if(max!=n)
    {
        swap(heap[n],heap[max]);
        heapify(max);
    }
}
template <class T> void PriorityQueue<T>::buildHeap(){              //BUILD HEAP: Costruisce un heap, partendo da un vector contenente
    for(int i= parent(heap.size()-1); i>=0; i--)                    //            elementi che non rispettano la proprieta' dell'heap.
        heapify(i);
}

template <class T> void PriorityQueue<T>::insert(T obj){            //INSERT: Inserisce un nuovo elemento all'interno della coda.
    heap.push_back(obj);                                            //L'elemento viene inserito alla fine del vector ed in seguito
    changeNode(heap.size()-1,obj);                                  //chiamo la funzione changeNode che lo posizionera' correttamente.
}
template <class T> T PriorityQueue<T>::remove(){                    //REMOVE: Estrae un elemento dalla coda.
    unsigned int size = heap.size();
    swap(heap[0],heap[size-1]);                                     //L'elemento da estrarre viene posizionato alla fine del vector,
    T temp = heap.back();                                           //scambiandolo con l'ultimo, quindi viene estratto dal vector
    heap.pop_back();
    heapify(0);                                                     //e viene ripristinata la l'heap dalla cima con heapify.

    return temp;
}

template <class T> void PriorityQueue<T>::changeNode(int nodo, T obj){  //CHANGE NODE: Cambia il valore di un elemento ed in seguito
    if(nodo < 0) return;                                                //             ripristina la proprietà dell'heap.

    heap[nodo] = obj;
    int padre = parent(nodo);

    while(padre>=0 and comparator(heap[nodo], heap[padre]))         //Risale l'heap dal basso verso l'alto e scambia se necessario
    {
        swap(heap[padre],heap[nodo]);
```

```
107.         nodo=padre;
108.         padre = parent(nodo);
109.     }
110. }
111.
112. template <class T> inline bool PriorityQueue<T>::isEmpty() const{   //IS_EMPTY: restituisce true se la coda è vuota, false altrimenti.
113.         return heap.empty();
114. }
115.
116. template <class T> void PriorityQueue<T>::print() const{           //PRINT: Stampa in maniera molto semplice gli elementi del vector.
117.     for(auto x : heap)
118.         cout << " " << x;
119. }
```

TRACCIA 2: QUESITO 2

Si implementi l'algoritmo di Prim per l'albero di copertura minimo in un grafo non orientato. L' implementazione deve far uso di una coda di priorità, realizzata mediante un MIN-HEAP. Si verifichi la correttezza del programma su un problema reale.

Analisi tecniche del problema.

Il quesito richiede di realizzare, un algoritmo che dato un grafo non orientato ne restituisca l'albero di copertura minimo, utilizzando per l'implementazione una propria realizzazione di una coda di priorità mediante MIN-HEAP.

Descrizione

L'algoritmo di Prim è un algoritmo greedy che consente di trovare l'albero di copertura minimo per un grafo pesato indiretto. Un albero di copertura o albero di connessione o albero di supporto di un grafo, è un albero che contiene tutti i vertici del grafo, ma degli archi ne contiene soltanto un sottoinsieme, cioè solo quelli necessari per connettere tra loro tutti i vertici con uno e un solo cammino. Infatti ciò che differenzia un grafo da un albero è che in quest'ultimo non sono presenti cammini multipli tra due nodi. Può essere visto anche come un sottografo del grafo di partenza in cui è preso un sottoinsieme di archi in modo tale che esso risulti aciclico, cioè appunto un albero. L'albero ricoprente è anche noto con il termine inglese spanning tree (ST).
Nel caso in cui gli archi siano pesati si può definire l'albero ricoprente minimo, o minimum spanning tree (MST). Un MST non è altro che un albero ricoprente nel quale sommando i pesi degli archi si ottiene un valore minimo.

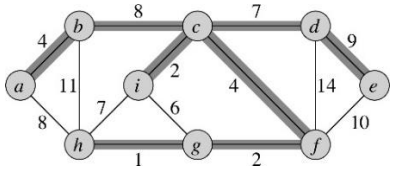
Definizione formale MST

Dato un grafo $G = (V, E)$ indiretto (non orientato), connesso e pesato e assumendo che $\omega: E \to \mathbb{R}$ sia una funzione peso per G, che associa ad ogni arco di E un peso. Un MST per G è un insieme di archi T che forma un albero e tale che:

- $T \subseteq E$, quindi T contiene solo una parte degli archi originali, al limite tutti, ma non di più.
- (V, T) è un sottografo connesso e aciclico.
- La somma dei pesi degli archi scelti è la più piccola possibile, cioè:

$$peso_T = \sum_{e \in T} \omega(e) \quad \text{è il minimo}$$

Dalla definizione, si ha che T forma un albero con un sottoinsieme di archi dai pesi minimi, per questo viene chiamato Minimum Spanning Tree.



NB: L'albero di copertura minimo non è unico, è possibile avere più alberi di copertura con lo stesso peso.

Le definizioni viste finora possono essere estese al caso in cui G non sia connesso: in tal caso si parla di minima foresta ricoprente

Implementazione

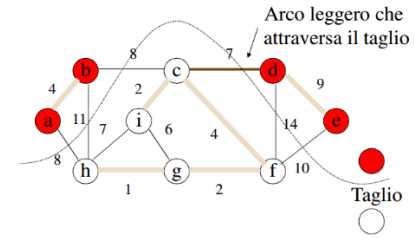
Per risolvere questo problema è possibile utilizzare un approccio di tipo "greedy", che per la soluzione del problema dell'albero di copertura minima coincide con una soluzione globalmente ottima.

L'idea è di accrescere un sottoinsieme A di archi di un albero di copertura aggiungendo un arco alla volta, ad ogni passo si determina un arco che può essere aggiunto ad A mantenendo la proprietà per A di essere un sottoinsieme di archi di un albero di copertura
Un arco di questo tipo è detto arco sicuro.

```
1.  Generic-MST(G, w)
2.   A = "insieme vuoto"
3.   while A non forma un albero di copertura do
4.     trova un arco sicuro (u,v)
5.     A = A unito {(u,v)} //Aggiungi l'arco ad A
6.   return A
```

Per poter implementare tale l'algoritmo abbiamo bisogno di determinare gli archi sicuri, per caratterizzarli sicuri dobbiamo introdurre alcune definizioni:

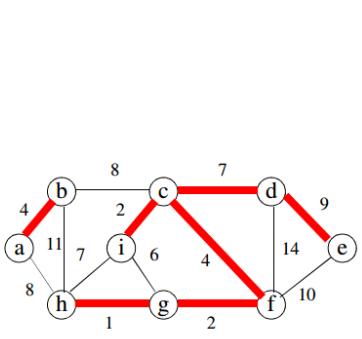
- Un taglio $(S, V-S)$ di un grafo non orientato $G = (V, E)$ è una partizione di V
- Un arco attraversa il taglio se uno dei suoi estremi è in S e l'altro è in $V-S$
- Un taglio rispetta un insieme di archi A se nessun arco di A attraversa il taglio
- Un arco leggero è un arco con peso minimo che attraversa il taglio.

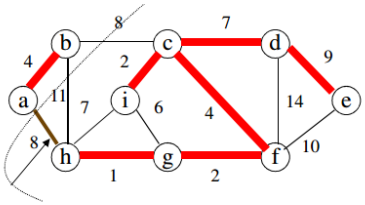


Insieme A: archi in grigio
il taglio rispetta A

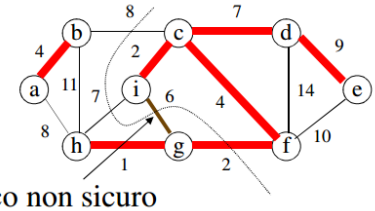
Per riconoscere gli archi sicuri utilizziamo il seguente teorema

> Sia $G = (V, E)$ un grafo non orientato e connesso con una funzione peso w a valori reali definita su E. Sia A un sottoinsieme di E contenuto in un qualche albero di copertura minimo per G. Sia $(S, V-S)$ un qualunque taglio che rispetta A.
> Se (u, v) è un arco leggero che attraversa il taglio. Allora tale arco è sicuro per A.





Arco sicuro



Arco non sicuro

Dimostrazione

Sia $G = (V, E)$ un grafo non orientato e connesso con una funzione peso ω a valori reali definita su E. Sia A un sottoinsieme di E contenuto in un qualche albero T che non contiene l'arco leggero (u, v) di copertura minimo per G. Allora:

"Possiamo costruire un altro albero di copertura minimo T' che include $A \cup \{(u, v)\}$ usando la tecnica del cut-and-paste, mostrando in tal modo che (u, v) è un arco sicuro per A."

- Poiché T è un insieme connesso ed un albero, ci deve essere un qualche percorso unico ρ che connette i vertici u e v, in modo tale che l'arco (u, v) formi un ciclo.
- Pertanto deve esistere almeno un arco (x, y) sul percorso ρ che attraversa un taglio $(S, V - S)$ di (u, v) che rispetta A. Avremo quindi che l'arco $(x, y) \notin A$ (proprio perché il taglio rispetta A).
- Essendo (x, y) sul percorso unico ρ rimuovendolo dividiamo T in due parti che poi possiamo riconnettere di nuovo attraverso (u, v), creiamo così un nuovo albero di copertura $T' = (T - \{(x, y)\}) \cup \{(u, v)\}$.
- Ricordando che per ipotesi (u, v) è un arco leggero che attraversa il taglio $(S, V - S)$, e poiché quest'ultimo è attraversato anche da (x, y), segue che $\omega(u, v) \leq \omega(x, y)$. Dunque si ha: $\omega(T') = \omega(T) - \omega(x, y) + \omega(u, v) \leq \omega(T)$.
- → Ma per ipotesi T era un albero di copertura minimo quindi anche T' lo sarà.

Resta da dimostrare se (u, v) è un arco sicuro per A

- Poiché $A \subseteq T$ e l'arco $(x, y) \notin A \rightarrow A \subseteq T'$ e anche $A \cup \{(u, v)\} \subseteq T'$, di conseguenza poiché T' è un albero di copertura minimo, l'arco (u, v) sarà sicuro per A.

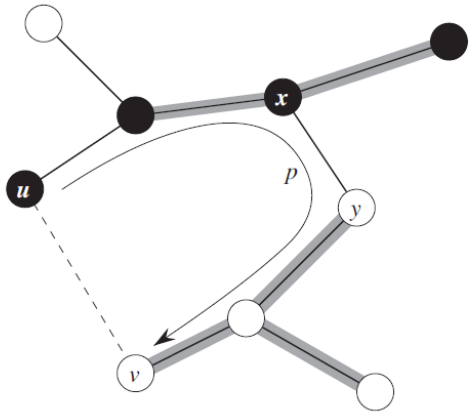


Figure 23.3 The proof of Theorem 23.1. Black vertices are in S, and white vertices are in $V - S$. The edges in the minimum spanning tree T are shown, but the edges in the graph G are not. The edges in A are shaded, and (u, v) is a light edge crossing the cut $(S, V - S)$. The edge (x, y) is an edge on the unique simple path p from u to v in T. To form a minimum spanning tree T' that contains (u, v), remove the edge (x, y) from T and add the edge (u, v).

COROLLARIO

Sia $G = (V, E)$ un grafo non orientato e connesso con una funzione peso w a valori reali definita su E. Sia A un sottoinsieme di E contenuto in un qualche albero di copertura minimo per G. Sia C una componente connessa (un albero) nella foresta $G_A = (V, A)$. Se (u, v) è un arco leggero che connette C a qualche altra componente in G_A allora (u, v) è sicuro per A.

Dimostrazione

Per come è definito l'albero di copertura quest'ultimo è sempre aciclico, inizialmente avremo una foresta $G_A = (V, A)$ composta essenzialmente da $|V|$ alberi di nodi singoli, man mando che si procede con l'algoritmo, il taglio $(C, V - C)$ rispetta A: quindi l'arco leggero (u, v) è un arco sicuro per A per il teorema precedente.

I due algoritmi di copertura più diffusi e conosciuti si basano essenzialmente sull'algoritmo generico visto in precedenza e si differenziano tra di loro solo da come viene scelto l'arco sicuro.

- Nell'algoritmo di Kruskal, l'insieme A è una foresta composta dai vertici del grafo, L'arco sicuro aggiunto ad A è sempre quello con il peso minore nel grafo che connette due distinte componenti di A.
- Nell'algoritmo di Prim l'insieme A forma un singolo albero, L'arco sicuro aggiunto ad A è sempre quello con il peso minore che connette l'albero ad un vertice non appartenente all'albero.

Algoritmo di Prim

L'idea dell'algoritmo di Prim è di partire da un unico insieme connesso rappresentato dall'albero ampliandolo man mano.

1. L'algoritmo procede mantenendo in A un singolo albero, c'è quindi una sola componente connessa. L'albero parte da un vertice arbitrario r (la radice) e cresce fin che non ricopre tutti i vertici del grafo.
2. Ad ogni passo viene aggiunto un arco leggero lungo il taglio $(A, V-A)$ che per il corollario è un arco sicuro in quanto:
   -C'è solo una componente connessa che è l'intero albero di copertura che si sta formando.
   -ovviamente il taglio $(A, V-A)$ rispetta A.

Alla fine dell'algoritmo otterremo il nostro albero di copertura minimo.
Questo algoritmo sfrutta l'approccio greedy in quanto viene scelta di volta in volta la soluzione locale migliore, l'arco leggero.
Per migliorare le prestazioni e implementare efficientemente l'algoritmo abbiamo bisogno di un modo veloce per selezionare l'arco leggero ed inserirlo nel nostro albero.
Questo viene fatto memorizzando tutti i vertici che non sono nell'albero in costruzione in una coda con priorità Q

1. Per ogni nodo la priorità è basata su un campo $key[v]$ che contiene il minimo tra i pesi degli archi che collegano v ad un qualunque vertice dell'albero in costruzione
2. Per ogni nodo si introduce un campo parent $p[x]$ che serve per poter ricostruire l'albero.

L'insieme A è mantenuto implicitamente come

$$A = \{(v, p[v]) : v \; in \; V - \{r\} - Q\}$$

quando l'algoritmo termina Q è vuota e l'albero di copertura in A è dunque:

$$A = \{(v, p[v]) : v \; in \; V - \{r\}\}$$

```
1.  MST-Prim (G, w, r)
2.   for ogni u in Q do
3.    key[u] = infinito
4.   key[r] = 0
5.   p[r] = NIL
6.   Q = V[G]
7.   while Q != vuota do
8.    u = Extract-Min(Q)
9.    for ogni v in Adj[u] do
10.     if v in Q e w(u,v) < key[v] then
11.      p[v]=u
12.      key[v] = w(u,v)
```

Spiegazione codice.

- Le linee 2-6 inizializzano la coda Q con tutti i vertici e pongono a ∞ l'attributo key per ogni vertice ad eccezione del vertice r per il quale $key[r] = 0$ in modo da estrarre r come elemento minimo nella fase iniziale, durante l'algoritmo l'insieme $V - Q$ contiene i vertici dell'albero che si sta costruendo.
- La linea 8 identifica un vertice u incidente su di un arco leggero che attraversa il taglio $(V - Q, Q)$ si estrae u da Q e lo si aggiunge ai vertici dell'albero.
- Le linee 9-12 aggiornano i campi key e p di ogni vertice v adiacente a u che non appartiene ancora all'albero

- Il fatto che il campo $key[v]$ rappresenti il costo minimo tra i pesi degli archi che collegano v ad un qualunque vertice dell'albero in costruzione è preservato perché se si trova un arco che collega v con l'albero (in realtà con il vertice u corrente) che costa meno, si aggiorna key al nuovo valore.
  Al ciclo successivo si esaminerà la coda Q e si troverà che uno dei v esaminati precedentemente è diminuito tanto da essere il vertice con chiave più piccola allora si aggiungerà implicitamente v all'albero, fissando la relazione padre-figlio migliore trovata e si procederà ad espandere la frontiera dei vertici adiacenti a v, stabilendo nuove potenziali relazioni padre-figlio

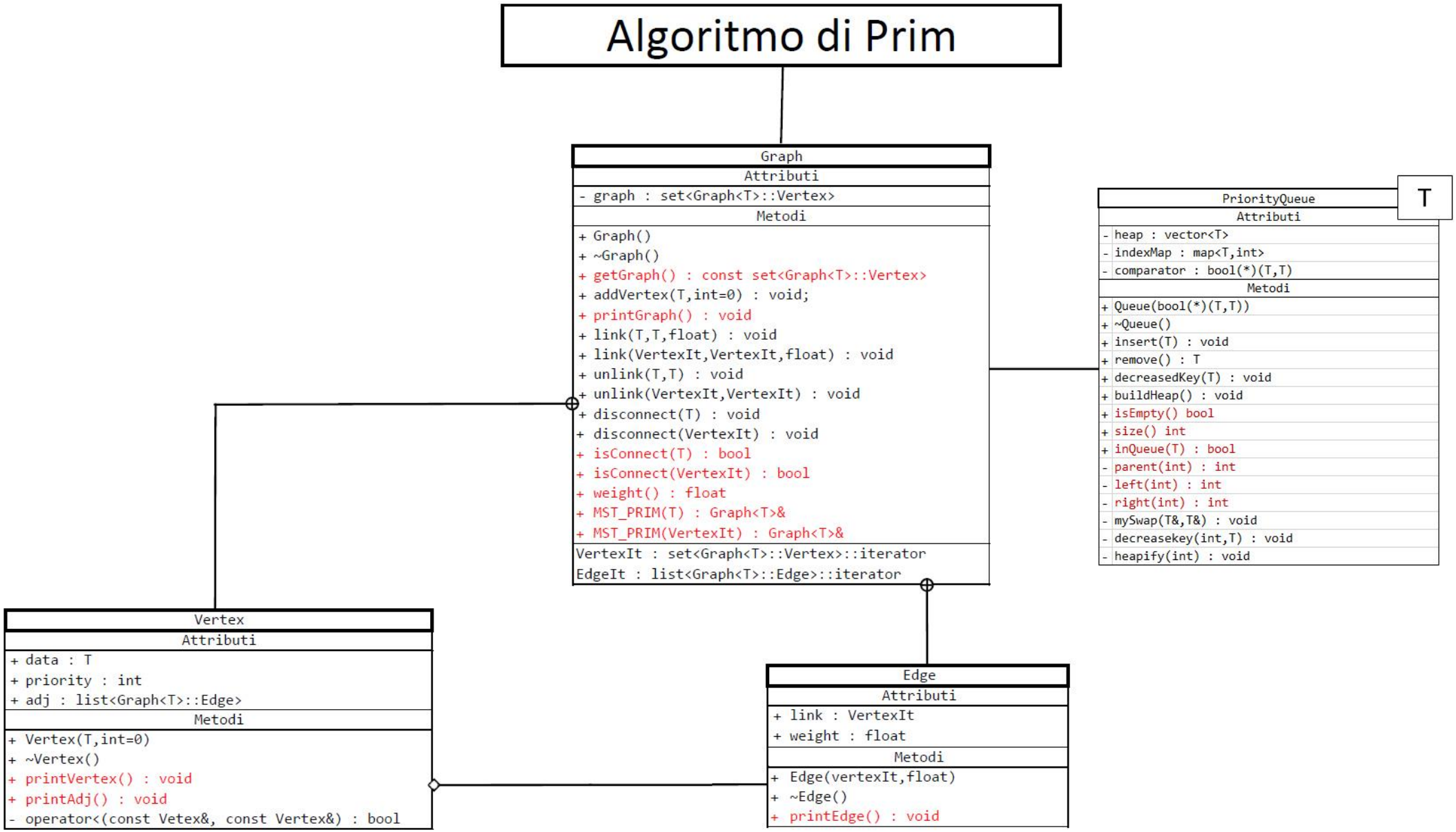
Analisi delle prestazioni

L'efficienza dell'algoritmo di Prim dipende da come viene realizzata la coda con priorità Q
Se Q viene realizzata con uno heap binario:

- Si usa Build-Heap per l'inizializzazione in tempo $O(V)$
- Il ciclo alla linea 7 viene eseguito $|V|$ volte ed ogni operazione Extract-Min è $O(log\ V)$
- Il ciclo for alla linea 9 viene eseguito in tutto $O(E)$ volte
- Il controllo di appartenenza in 10 può essere eseguito in $O(1)$ usando un metodo di indirizzamento diretto
- L'assegnazione in 11 implica una operazione di Decrease-Key sullo heap che costa $O(log\ V)$

Il tempo totale è pertanto un $O(V + V\ log\ V + E\ log\ V) = O(E\ log\ V)$ asintoticamente eguale a quello per l'algoritmo di Kruskal

IMPLEMENTAZIONE DELL'ALGORITMO DI PRIM



DESCRIZIONE DETTAGLIATA DELL'IMPLEMENTAZIONE

Dal diagramma UML si può vedere che l'implementazione si basa essenzialmente sulle seguenti classi:

a. ***Graph:*** Classe principale del progetto, che definisce la struttura del grafo con relativi metodi e attributi.

b. ***Vertex:*** Classe interna a `Graph`, definisce la struttura di un vertice del grafo.

c. ***Edge:*** Classe interna a `Graph`, definisce la struttura di un arco.

d. ***PriorityQueue:*** Una coda di priorità che permette la di verificare l'appartenenza ed effettuare *decrease-key* in $O(logn)$

GRAPH:

Graph	T	
Attributi		
-	graph : set<Graph<T>::Vertex>	Contiene attraverso un std:set tutti i vertici del grafo.
Metodi		
+	Graph()	Costruttore.
+	~Graph()	Distruttore.
+	getGraph() : const set<Graph<T>::Vertex>	Metodo getter, restituisce un reference costante al grafo.
+	addVertex(T,int=0) : void;	Metodo che permette di aggiungere un vertice al grafo.
+	printGraph() : void	Stampa a schermo il grafo, rappresentando ogni suo vertice ed arco.
+	link(T,T,float) : void	Connette i due vertici forniti in input identificati loro campo informazione.
+	link(VertexIt,VertexIt,float) : void	Connette i due vertici forniti in input tramite il loro iteratore.
+	unlink(T,T) : void	Sconnette i due vertici forniti in input identificati loro campo informazione.
+	unlink(VertexIt,VertexIt) : void	Scnnette i due vertici forniti in input tramite il loro iteratore.
+	disconnect(T) : void	Sconnette dal grafo il vertice fornito identificato dal suo campo informazione.
+	disconnect(VertexIt) : void	Sconnette dal grafo il vertice fornito tramite il suo iteratore.
+	isConnect(T) : bool	Restituisce true se il vertice dal suo campo informazione è connesso.
+	isConnect(VertexIt) : bool	Restituisce true se il vertice fornito tramite il suo iteratore è connesso.
+	weight() : float	Restituisce il peso del grafo.
+	MST_PRIM(T) : Graph<T>&	Calcola l'MST partendo dal nodo fornito utilizzando l'algoritmo di Prim.
+	MST_PRIM(VertexIt) : Graph<T>&	Calcola l'MST partendo dal nodo fornito utilizzando l'algoritmo di Prim.
VertexIt : set<Graph<T>::Vertex>::iterator		Typedef
EdgeIt : list<Graph<T>::Edge>::iterator		Typedef

La classe definisce la struttura di un grafo con le varie operazioni effettuabili su di esso. All'interno della classe sono definite anche le classi `Vertex` e `Edge`, esse non sono accessibili all'esterno ma solamente all'interno di `Graph`, favorendo quindi l'incapsulamento. Si è scelto di rappresentare il grafo con un `std::set` poiché essendo rappresentato internamente attraverso alberi `Red&Black` fornisce un tempo di *accesso*, *inserimento*, *cancellazione* dell'ordine di $O(logn)$, permettendo così una bassa complessità di tempo quando vengono eseguite le varie operazioni descritte dai metodi.
Tutti i metodi che lavorano sul grafo consentono di operare sui vertici sia identificandoli attraverso il loro campo informazione `data`, che attraverso un iteratore al `std::set`. Il primo modo, permette un più facile e familiare utilizzo della classe, a patto di pagare il comunque basso costo della ricerca del vertice, l'altro consente, se ne si conosce l'iteratore, di accedervi con un tempo $O(1)$, utile ad esempio all'interno delle funzioni dato che operano direttamente con gli iteratori.

VERTEX:

Vertex		
Attributi		
+	data : T	Contiene i dati e le informazioni di tipo T che vengono memorizzate nel vertice.
+	priority : int	La priorità del vertice, espressa da un numero intero.
+	adj : list<Graph<T>::Edge>	Contiene una lista di archi ai vertici adiacenti.
Metodi		
+	Vertex(T,int=0)	Costruttore con parametri, richiede in input 'data' e la priorità del nodo.
+	~Vertex()	Distruttore.
+	printVertex() : void	Funzione che stampa il contenuto del vertice.
+	printAdj() : void	Funzione che stampa la lista dei vertici adiacenti, con relativi pesi degli archi.
-	operator<(const Vetex&, const Vertex&) : bool	Overtload dell'operatore <, necessario per confrontare i vertici.

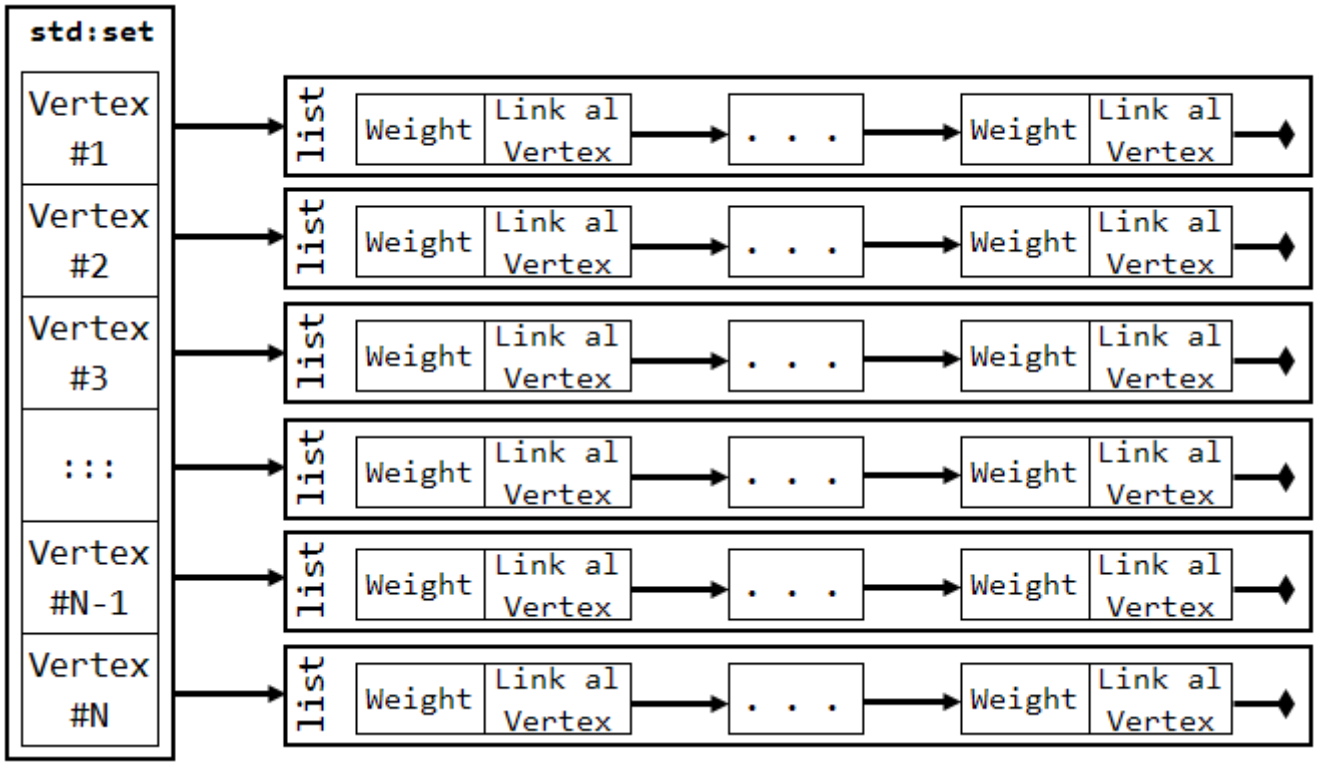
Si tratta di una classe interna a quella principale `Graph`. Definisce la struttura di un vertice del grafo. Tra gli attributi possiamo distinguere il campo informazione del vertice `data`, il dato satellite `priority` ed infine la lista contenente gli archi che puntano ai vertici adiacenti. La classe ha in oltre due metodi di stampa a schermo, uno per il vertice con solo il campo `data`, l'altro che stampa la lista degli adiacenti.
Come si può notare la classe ha tutti membri con visibilità public, ma ciò non è un problema in quanto è definita come privata all'interno di `Graph` e al di fuori non esiste, permettendo l'accesso solo a quest'ultima.

EDGE:

Edge	
Attributi	
+ link : VertexIt	Contiene un iteratore al vertice che l'arco collega.
+ weight : float	Contiene il peso del grafo
Metodi	
+ Edge(vertexIt,float)	Costruttore con parametri, richiede in input l'iteratore al vertice ed il peso per raggiungerlo.
+ ~Edge()	Distruttore.
+ printEdge() : void	Funzione che stampa l'arco.

Si tratta di una classe interna a quella principale `Graph`. Definisce la struttura di un arco, rappresentato tramite il suo peso `weight` e il puntatore al vertice che collega link, ha solo un metodo che serve per stampare a schermo l'arco. Valgono in oltre tutte le affermazioni sulla visibilità fatte prima sulla classe `Vertex`.

La struttura del Grafo definita da `Graph` può essere riassunta dalla seguente immagine:



Ogni vertice contenuto nel `set` quindi associa una lista di archi i quali conducono ai vertici adiacenti.

L'algoritmo di Prim

L'algoritmo è stato implementato come funzione membro pubblica della classe `Graph`. ed il suo funzionamento è ben documentato all'interno della classe stessa e dallo pseudocodice mostrato in precedenza.
L'algoritmo in quanto definito nella classe template è in grado di lavorare con qualsiasi tipo di dato a patto che quest'ultimo abbia definito l'operatore binario di confronto `' < '` necessario ad ordinare gli elementi all'interno del `set`. E l'operatore `' << '` allo stream di output per stampare a video il contenuto del vertice.

Listato della classe

```
1.  #ifndef GRAPH_H_
2.  #define GRAPH_H_
3.  #include <iostream>
4.  #include <climits>
5.  #include <list>
6.  #include <set>
7.  #include <map>
8.  #include "PriorityQueue.h"
9.  using namespace std;
10.
11. template<class T> class Graph {
12.     class Vertex;                                          //Dichiarazione della classe Vertex, definizione in seguito..
13.     class Edge;                                            //Dichiarazione della classe Edge, definizione in seguito..
14.
15.     typedef typename set<Graph<T>::Vertex>::iterator VertexIt;
16.     typedef typename list<Graph<T>::Edge>::iterator EdgeIt;
17.
18.      /*DEFINIZIONE DELLA CLASSE INTERNA VERTEX*/
19.      class Vertex{
20.         friend ostream& operator<<(std::ostream& os, const Vertex& obj){
21.             obj.printVertex();
22.             return os;
23.         }
24.         friend bool operator<(const Vertex& x,const Vertex& y) { return x.data<y.data; }//Funzione di confronto fra i vertici.
25.      public:
26.         Vertex(T data, int pr=0) : data(data), priority(pr), adj() {}                 //Costruttore con parametri.
27.         void printVertex() const;                                                     //Stampa il vertice.
28.         void printAdj() const;                                                        //Stampa gli archi per i vertici adiacenti.
29.
30.         T data;                                                                       //Contiene le "informazioni" del vertice.
31.         mutable int priority;                                                         //Priorita' del vertice.
32.         mutable list<Graph<T>::Edge> adj;                                             //Lista degli archi adiacenti.
33.         //Mutable perche std::set restituisce const_iterator che non permettono la modifica diretta.
34.     };
35.
36.     /*DEFINIZIONE DELLA CLASSE INTERNA EDGE*/
37.     class Edge{
38.     public:
39.         Edge(VertexIt link, float weight) : link(link), weight(weight){}              //Costruttore con parametri.
40.         void printEdge() const;                                                       //Stampa l'arco.
41.         VertexIt link;                                                                //Iteratore al vertice che l'arco connette.
42.         float weight;                                                                 //Peso dell'arco.
43.     };
44.
45. private:
46.     //NB: affinche' sia possibile utilizzare il set e' necessario che gli elementi di tipo 'T' abbiano definita la relazione 'a' < 'b'
47.     set<Graph<T>::Vertex> graph;                                                     //Set di vertici che compongono il grafo.
48.
49.
50. public:
51.     // COSTRUTTORE E DISTRUTTORE
52.     Graph(){}                                                                        //Costruttore di default.
53.     virtual ~Graph(){}                                                               //Distruttore di default.
54.     // GETTER & SETTER
55.     const set<Graph<T>::Vertex>& getGraph() const {return graph;}                    //Metodo getter, ritorna un riferimento costante al grafo.
56.
57.     // METODI DI CLASSE
58.     void addVertex(T data, int priority = 0);                                        //Crea e aggiunge un vertice dati 'data' e 'priority'.
59.     void printGraph() const;                                                         //Stampa il grafo rappresentandolo come liste di adiacenza
60.     void link(T a, T b, float weight);                                               //Collega due vertici 'a' e 'b' con peso 'weight'.
61.     void link(VertexIt a, VertexIt b, float weight);
62.     void unlink(T a, T b);                                                           //Scollega due vertici 'a' e 'b'.
63.     void unlink(VertexIt a, VertexIt b);
64.     void disconnect(T a);                                                            //Disconnette un vertice dal grafo.
65.     void disconnect(VertexIt a);
66.     bool isConnected(T a) const;                                                     //Ritorna true se il vertice e' connesso, false altrimenti
67.     bool isConnected(VertexIt a) const;
68.     float weight() const;                                                            //Calcola il peso del grafo (somma di tutti i pesi).
69.
70.     Graph<T>& MST_PRIM(T start) const;                                               //Calcola e ritorna l'MST del grafo partendo da un vertice
71.     Graph<T>& MST_PRIM(VertexIt start) const;
```

```
};

#endif /* GRAPH_H_ */


//            ____        __ _       _     _                        _      _ _         __                 _               _
//           |  _ \  ___ / _(_)_ __ (_)___(_) ___  _ __   ___    __| | ___| | | ___   / _|_   _ _ __  _______(_) ___  _ __ (_)
//           | | | |/ _ \ |_| | '_ \| |_  / |/ _ \| '_ \ / _ \  / _` |/ _ \ | |/ _ \ | |_| | | | '_ \|_  / |/ _ \| '_ \| |
//           | |_| |  __/  _| | | | | |/ /| | (_) | | | |  __/ | (_| |  __/ | |  __/ |  _| |_| | | | |/ /| | (_) | | | | |
//           |____/ \___|_| |_|_| |_|_/___|_|\___/|_| |_|\___|  \__,_|\___|_|_|\___| |_|  \__,_|_| |_/___|_|\___/|_| |_|_|
//
/*
 +--------------------------------------------------------------------------------------------------------------------------------+
 |                                                 METODI DELLA CLASSE  GRAPH                                                      |
 +--------------------------------------------------------------------------------------------------------------------------------+
*/
template <class T> void Graph<T>::addVertex(T data, int priority){  //ADD_VERTEX: Aggiunge un vertice al grafo.
    Vertex v(data, priority);
    graph.insert(v);
}

template <class T> void Graph<T>::printGraph() const{               //PRINT_GRAPH: Stampa il grafo sottoforma di liste di adiacenza
    for(auto it : graph){   //Per ogni vertice del grafo..
        it.printVertex();   //Stampo prima il vertice e
        it.printAdj();      //poi la lista degli adiacenti.
        cout << endl;
    }
}

template <class T> void Graph<T>::link(T a, T b, float weight) {    //LINK: Connette due vertici 'a' e 'b' in input mediante valore
    VertexIt vertice1 = graph.find(a);                              //      con un arco avente peso pari a 'weight'.
    VertexIt vertice2 = graph.find(b);                              //I vertici sono ricercati nel set tramite il loro valore
                                                                    //e collegati dalla funzione che opera con gli iteratori.
    link(vertice1, vertice2, weight);
}
template <class T> void Graph<T>::link(VertexIt a, VertexIt b, float weight) {
    if(a == graph.end() or b == graph.end())            //Se uno dei due nodi non e' presente nel grafo ritorno
        return;

    (a->adj).push_back(Edge(b,weight));                  //Inserisco 'b' nella lista di adiacenza di 'a'
    (b->adj).push_back(Edge(a,weight));                  //Inserisco 'a' nella lista di adiacenza di 'b'
}

template <class T> void Graph<T>::unlink(T a, T b) {                //UNLINK: Disconnette i vertici 'a' e 'b' in input mediante valore
    VertexIt vertice1 = graph.find(a);
    VertexIt vertice2 = graph.find(b);

    unlink(vertice1, vertice2);                                     //Trovate le occorrenze, chiama la funzione con gli iteratori.
}
template <class T> void Graph<T>::unlink(VertexIt a, VertexIt b) {
    if(a == graph.end() or b == graph.end())                        //Se uno dei due nodi non e' presente nel grafo ritorno
        return;

    for(EdgeIt it = a->adj.begin(); it!=a->adj.end(); it++){  //Ricerco tra gli adiacenti di 'a' l'arco che porta a 'b' e lo elimino.
        if(it->link == b){
            a->adj.erase(it);
            break;
        }
    }

    for(EdgeIt it = b->adj.begin(); it!=b->adj.end(); it++){  //Ricerco tra gli adiacenti di 'b' l'arco che porta a 'a' e lo elimino.
        if(it->link == a){
            b->adj.erase(it);
            break;
        }
    }

}

template <class T> void Graph<T>::disconnect(T a) {                 //DISCONNECT: Disconnette un vertice da tutti i suoi adiacenti.
    VertexIt vertice = graph.find(a);

    disconnect(vertice);
}
```

```
146. template <class T> void Graph<T>::disconnect(VertexIt a) {
147.     if(a == graph.end())                                    //Se il vertice non e' presente nel grafo ritorna.
148.         return;
149. 
150.     for(EdgeIt it = a->adj.begin(); it!=a->adj.end();)      //Poiche' non e' possibile eliminare l'elemento corrente di
151.         unlink((it++)->link, a);                            //un iterator ementre sta scorrendo la lista e' necessario farlo
152.                                                             //puntare all'elemento successivo prima della cancellazione.
153. }
154. 
155. template <class T> bool Graph<T>::isConnected(T a) const{          //IS_CONNECTED: Ritorna true se il vertice in input e' un
156.     VertexIt vertice = graph.find(a);                             //              vertice connesso altrimenti ritorna false.
157. 
158.     return isConnected(vertice);
159. }
160. template <class T> bool Graph<T>::isConnected(VertexIt a) const{
161.     if(a == graph.end())                                    //Se il vertice non e' presente nel grafo ritorna false.
162.         return false;
163.     return !a->adj.empty();                                 //Controlla se la lista di adiacenza di 'a' e' vuota o meno.
164. }
165. 
166. template <class T> Graph<T>& Graph<T>::MST_PRIM(T start) const{     //MTS_PRIM: Ritorna un sottografo che partendo dal vertice
167.     VertexIt vertice = graph.find(start);                           //          'start' fornito, risulta essere un MST.
168. 
169.     return MST_PRIM(vertice);
170. }
171. 
172. 
173. template <class T> Graph<T>& Graph<T>::MST_PRIM(VertexIt start) const{
174.     Graph<T>& MST = *new Graph();                           //Alloco memoria per memorizzare l'MST, inizialmente un grafo vuoto.
175. 
176.     if(start == graph.end())                                //Se il vertice di partenza 'start' non e' presente nel grafo ritorno.
177.         return MST;
178.     //Creo una coda di min-priorita' contenente i vertici.
179.     PriorityQueue<Vertex*> queue( [](Vertex* a, Vertex* b){return a->priority < b->priority;} );
180. 
181.     //INSERISCO I NODI NELL'MST MA SENZA ANCORA CREARE CONNESSIONI, MST=(V,0).
182.     for(auto it : graph) MST.addVertex(it.data);
183. 
184.     //INSERISC NELLA CODA TUTTI I VERTICI DEL GRAFO TRAMITE I LORO PUNTATORI
185.     for(VertexIt it = graph.begin(); it!=graph.end(); it++){
186.         it->priority = INT_MAX;                                 //Setto la priorità del vertice a +infinito (Massimo int rapp)
187.         Vertex* ptr = (Vertex*)&(*it);                          //ottengo il puntatore al vertice del grafo.
188.         queue.insert(ptr);                                      //infine inserisco il puntatore nella coda.
189.     }
190. 
191.     graph.find(start->data)->priority = 0;                      //Setto la priorità del vertice start a 0, in modo da estrarlo
192.     queue.buildHeap();                                          //per primo dalla coda e costruisco l'heap.
193. 
194.     while(!queue.isEmpty()){                                    //Fin che la coda non e' vuota...
195.         Vertex* u = queue.remove();                             //Estraggo il nodo con minore priorita' memorizzandolo in 'u'
196.         for(auto x : u->adj){
197.             Vertex* v = (Vertex*)&(*x.link);                    //Per ogni vertice 'v' adiacente al vertice 'u'
198.             if( queue.inQueue(v) and v->priority > x.weight ){  //Se 'v' si trova nella coda e la sua priorita' e' > del peso
199.                 v->priority = x.weight;                         //dell'arco.. aggiorna la priorita' di 'v'
200.                 queue.decreasedKey(v);
201.                 if(MST.isConnected(v->data))                    //Se il corrispettivo di 'v' nell'MST e' connesso, con
202.                     MST.disconnect(v->data);                    //qualche altro vertice allora lo disconnetto.
203.                 MST.link(v->data,u->data, x.weight);            //E lo riconnetto quindi con il vertice 'u' dell'MST.
204.             }
205.         }
206.     }
207. 
208.     return MST;
209. }
210. 
211. template <class T> float Graph<T>::weight() const{                 //WEIGHT: Ritorna il peso del grafo (somma dei pesi degli archi)
212.     float weight = 0;
213.     for(const auto vertex : graph)                      //Scorro i vertici
214.         for(const auto edge : vertex.adj)               //Scorro gli adiacenti
215.             weight += edge.weight;                      //Sommo i pesi
216.     //Alla fine divido per due in quanto per ogni coppia di vertici collegati ho due archi con stesso peso.
217.     return weight/2;
218. }
```

```cpp
/*
 +-------------------------------------------------------------------------------------------------------------------------------+
 |                                                METODI DELLA CLASSE  VERTEX                                                    |
 +-------------------------------------------------------------------------------------------------------------------------------+
*/
template <class T> void Graph<T>::Vertex::printVertex() const{      //PRINT_VERTEX: Stampa il vertice corrente.
    cout << "[" << data << "]";                                     //NB: Richiede che il tipo 'T' possa essere stampato.
}
template <class T> void Graph<T>::Vertex::printAdj() const{         //PRINT_ADJ: Stampa gli archi ai nodi adiacenti.
    for(auto it : adj)
        it.printEdge();              //Stampa l'arco

    cout << "--x";                   //Stampa simbolo che indica la fine nella lista.
}


/*
 +-------------------------------------------------------------------------------------------------------------------------------+
 |                                                METODI DELLA CLASSE  EDGE                                                      |
 +-------------------------------------------------------------------------------------------------------------------------------+
*/
template <class T> void Graph<T>::Edge::printEdge() const{           //PRINT_EDGE: Stampa l'arco corrente, con peso e vertice.
    cout << "-->(" << weight << ")";     //Stampa il peso
    link->printVertex();                 //E poi il vertice con cui c'e' il collegamento.
}
```

PRIORITYQUEUE:

PriorityQueue	T	
Attributi		
-	heap : vector<T>	La coda e' rappresentata internamente con un heap memorizzato in un vector.
-	indexMap : map<T,int>	Contiene informazioni sulla posizione degli alementi all'interno dell'heap
-	comparator : bool(*)(T,T)	Puntatore ad una funzione per il confronto degli elementi nella coda.
Metodi		
+	Queue(bool(*)(T,T))	Costruttore con parametri, riceve in input la funzione da utilizzare per ordinare la coda.
+	~Queue()	Distruttore di classe.
+	insert(T) : void	Inserisce un elemento nella coda.
+	remove() : T	Estrae un elemento dalla coda.
+	decreasedKey(T) : void	Aggiorna l'heap quando l'elemento in input sia diminuito di valore.
+	buildHeap() : void	Consente di costruire l'heap interno alla coda.
+	isEmpty() bool	Restituisce true se la coda e' vuota, false altrimenti.
+	size() int	Ritorna il numero di elementi presenti nella coda.
+	inQueue(T) : bool	Restituisce true se l'elemento in inputa appartiene alla coda.
-	parent(int) : int	Restituisce l'indice nel vector del padre dell'elemento di indice a.
-	left(int) : int	Restituisce l'indice nel vector del figlio sx dell'elemento di indice a.
-	right(int) : int	Restituisce l'indice nel vector del figlio dx dell'elemento di indice a.
-	mySwap(T&,T&) : void	Scambia due elementi sia nell'heap e nell'indexMap.
-	decreasekey(int,T) : void	Diminuisce in valore dell'elemento di indice in input.
-	heapify(int) : void	Ripristina la proprieta' dell'heap partendo dal nodo di indice in input.

Si tratta di una modifica all'implementazione precedente di coda di priorità, per abbassare la complessità di tempo delle operazioni di *decrease-key* e di verifica di appartenenza di un generico elemento dell'heap, passando da $O(n)$ a $O(\log_2 n)$ a discapito però della complessità di spazio richiesta.
Questo miglioramento è ottenuto introducendo una `map<T,int>` che associa ad ogni elemento della coda la sua posizione all'interno dell'heap ciò permette di sapere in ogni momento se un dato elemento è presente o meno nella coda e qual è il suo indice semplicemente effettuando una ricerca su di essa (Prima era necessaria una ricerca sequenziale sul vector che costituisce l'heap). Ovviamente è necessario aggiornare di volta in volta la map tuttavia tali operazioni hanno una complessità al più logaritmica e non aumentando quindi la complessità asintotica.

Listato della classe

```
#ifndef PRIORITYQUEUE_H_
#define PRIORITYQUEUE_H_
#include <iostream>
#include <vector>
#include <map>
using namespace std;

template<class T> class PriorityQueue {
public:
    /**COSTRUTTORI E DISTRUTTORI*/
    PriorityQueue(bool (*compare)(T a,T b));                 //Costruttore con parametri.
    virtual ~PriorityQueue();                                //Distruttore.

    /**FUNZIONI MEMBRO*/
    void insert(T obj);                                     //Inserisce un elemento nella coda.
    T remove();                                             //Estrae un elemento dalla coda.

    inline bool isEmpty() const;                            //Restituisce true se la coda e' vuota, false altrimenti.
    int size() const {return heap.size();}                  //Ritorna il numero di elementi presenti nella coda.
    inline bool inQueue(T obj) const;                       //Restituisce true se l'elemento appartiene alla coda.
    void buildHeap();                                       //Consente di costruire l'heap interno alla coda.
    void decreasedKey(T obj);                               //Ripristina l'heap sapendo che obj è stato diminuito.

private:
    vector<T> heap;                                         //La coda e' rappresentata con un heap memorizzato in un vector.
    map<T, int> indexMap;                                   //map contenente le posizioni degli elementi nel vector.
    bool (*comparator)(T a,T b);                            //Puntatore ad una funzione per il confronto degli elementi nella coda

    /**FUNZIONI MEMBRO PRIVATE*/
    inline int parent(int a) const;                         //Restituisce l'indice nel vector del padre dell'elemento di indice a.
    inline int left(int a) const;                           //Restituisce l'indice nel vector del figlio sx dell'elemento di indice a.
    inline int right(int a) const;                          //Restituisce l'indice nel vector del figlio dx dell'elemento di indice a.

    void decreaseKey(int n, T obj);                        //Diminuisce il valore di un nodo tramite il suo indice.
    void heapify(int n);                                    //Ripristina la proprieta' dell'heap.
    void mySwap(T&,T&);                                     //Funzione di swap, scambia elementi sia nella map che nel vector
};

#endif /* PRIORITYQUEUE_H_ */



//           ____        __ _       _     _                        _      _ _        __                  _           _
//          |  _ \  ___ / _(_)_ __ (_)___(_) ___  _ __   ___    __| | ___| | | ___  / _|_   _ _ __  ___(_) ___  _ __ (_)
//          | | | |/ _ \ |_| | '_ \| |_  / |/ _ \| '_ \ / _ \  / _` |/ _ \ | |/ _ \ | |_| | | | '_ \|_  / |/ _ \| '_ \| |
//          | |_| |  __/  _| | | | | |/ /| | (_) | | | |  __/ | (_| |  __/ | |  __/ |  _| |_| | | | |/ /| | (_) | | | | |
//          |____/ \___|_| |_|_| |_|_/___|_|\___/|_| |_|\___|  \__,_|\___|_|_|\___| |_|  \__,_|_| |_/___|_|\___/|_| |_|_|
//

template <class T> PriorityQueue<T>::PriorityQueue(bool (*compare)(T a,T b) ) {  //COSTRUTTORE: Riceve in input un puntatore a
        comparator = compare;                                                    // funzione da utilizzare confrontare gli elementi.
}
template <class T> PriorityQueue<T>::~ PriorityQueue() {}                        //DISTRUTTORE

template <class T> inline int PriorityQueue<T>::parent(int a) const{   //PADRE DEL NODO: Restituisce l'indice, nel vector, del padre
    if(a<=0) return -1;                                                //                e del nodo memorizzato all'indice a.
        else return (a-1)/2;
}
template <class T> inline int PriorityQueue<T>::left(int a) const{     //FIGLIO SINISTRO: Restituisce l'indice, nel vector, del
    if(a<0 or 2*a+1 > (int)heap.size() -1) return -1;                  //                 figlio sx del nodo memorizzato all'indice a.
        else return 2*a+1;
}
template <class T> inline int PriorityQueue<T>::right(int a) const{   //FIGLIO DESTRO: Restituisce l'indice, nel vector, del figlio DX
    if(a<0 or 2*a+2 > (int)heap.size() -1) return -1;                 //               del nodo memorizzato all'indice a.
        else return 2*a+2;
}

```

```
67. template <class T> void PriorityQueue<T>::heapify(int n){        //HEAPIFY: Ripristina la proprieta' dell'heap partendo dal nodo n
68.     int sx = left(n);                                            //         dell' albero rappresentato con il vector.
69.     int dx = right(n);
70.     int max = n;
71.     if(sx != -1 and comparator(heap[sx], heap[max]) )
72.         max = sx;
73.     if(dx != -1 and comparator(heap[dx], heap[max]) )
74.         max = dx;
75.
76.     if(max!=n)
77.     {
78.         mySwap(heap[n],heap[max]);
79.         heapify(max);
80.     }
81. }
82. template <class T> void PriorityQueue<T>::buildHeap(){           //BUILD HEAP: Costruisce un heap, partendo da un vector contenente
83.     for(int i= parent(heap.size()-1); i>=0; i--)                 //            elementi che non rispettano tale proprietà
84.         heapify(i);
85. }
86.
87. template <class T> void PriorityQueue<T>::insert(T obj){          //INSERT: Inserisce un nuovo elemento all'interno della coda.
88.     if(indexMap.find(obj) != indexMap.end())                     //se l'elemento è gia stato inserito ritorno..
89.         return;
90.
91.     indexMap.insert( make_pair(obj,heap.size()) );
92.     heap.push_back(obj);                                         //L'elemento viene inserito alla fine del vector ed in seguito
93.     decreaseKey(heap.size()-1,obj);                              //chiamo la funzione changeNode che lo posizionera' correttamente.
94. }
95. template <class T> T PriorityQueue<T>::remove(){                 //REMOVE: Estrae un elemento dalla coda.
96.     mySwap(heap[0],heap[heap.size()-1]);                         //L'elemento da estrarre viene posizionato alla fine del vector,
97.     T temp = heap.back();                                        //scambiandolo con l'ultimo, quindi viene estratto dal vector
98.     indexMap.erase(heap[heap.size()-1]);
99.     heap.pop_back();
100.    heapify(0);                                                  //viene ripristinata la proprieta' dell'heap dalla cima con heapify.
101.
102.    return temp;
103. }
104.
105. template <class T> void PriorityQueue<T>::decreaseKey(int nodo, T obj){  //CHANGE NODE: Cambia il valore di un elemento ed in seguito
106.    if(nodo < 0) return;                                                  //             ripristina la proprieta' dell'heap.
107.
108.    heap[nodo] = obj;
109.    int padre = parent(nodo);
110.
111.    while(padre>=0 and comparator(heap[nodo], heap[padre]))      //Risale l'heap dal basso verso l'alto e scambia se necessario
112.    {
113.        mySwap(heap[padre],heap[nodo]);
114.        nodo=padre;
115.        padre = parent(nodo);
116.    }
117. }
118.
119. template <class T> void PriorityQueue<T>::decreasedKey(T obj){   //DECREASED_KEY: Riposiziona un elemento il cui valore e' diminuito
120.    if(indexMap.find(obj) == indexMap.end())                     //L'elemento obj non appartiene alla coda.
121.        return;
122.    decreaseKey(indexMap[obj],obj);
123. }
124.
125. template <class T> inline bool PriorityQueue<T>::isEmpty() const{ //IS_EMPTY: Restituisce true se la coda e' vuota, false altrimenti.
126.        return heap.empty();
127. }
128.
129. template <class T> bool PriorityQueue<T>::inQueue(T obj) const{  //IN_QUEUE: Verifica se un elemento appartiene o meno alla coda
130.        return indexMap.find(obj)!=indexMap.end();
131. }
132.
133. template <class T> void PriorityQueue<T>::mySwap(T& a, T& b) {   //MT_SWAP: Effettua uno swap fra gli elementi dell'heap e della map
134.    swap(indexMap[a], indexMap[b]);
135.    swap(a,b);
136. }
```

Esempi di funzionamento e applicazioni pratiche dell'algoritmo

Pianificazione dei voli di una compagnia aerea.

Supponiamo di dover gestire i voli di una compagnia aerea fra n possibili destinazioni in maniera tale da voler consumare il minor quantitativo di carburante, cioè in modo tale che la distanza totale percorsa fra gli aeri a disposizione lungo i vari aeroporti sia minima.

Tale problema può essere efficacemente risolto utilizzando l'algoritmo di Prim. Si possono considerare infatti i vari aeroporti come vertici di un grafo, e i possibili voli fra la destinazione di partenza e di arrivo come archi con peso pari alla distanza geodetica. Poiché tutti i voli fra le diverse città sono possibili avremo che il grafo in questione è un grafo completo.

Il minimum spanning tree risultante potrà essere considerato come una serie di voli e scali da effettuare per raggiungere la destinazione da una certa città di partenza.

Per l'esempio, è stata creata una classe città molto semplice:

```
1.  const double PI = 3.1415926;
2.  class City{
3.      friend bool operator<(const City& a, const City& b) {return a.nome < b.nome;}
4.      friend ostream& operator<<(std::ostream& os, const City& obj){
5.              cout<< " " << obj.nome << " ";
6.              return os;
7.          }
8.  public:
9.      City(const string Nome, double Lat, double Long) : nome(Nome), latitudine(Lat), longitudine(Long){}
10.     static double distance(const City& a, const City& b) {
11.         //Calcolo della distanza geodetica tra due punti della superficie terrestre
12.         auto a_lat = PI*a.latitudine/180;
13.         auto a_lon = PI*a.longitudine/180;
14.         auto b_lat = PI*b.latitudine/180;
15.         auto b_lon = PI*b.longitudine/180;
16.         return acos( sin(a_lat)*sin(b_lat) + cos(a_lat)*cos(b_lat)*cos(a_lon-b_lon) ) * 6371;
17.     }
18.
19.     string nome;
20.     double latitudine;
21.     double longitudine;
22. };
```

Ogni città è quindi identificata dal nome e dalla sua latitudine e longitudine espresse in gradi. In più la classe definisce un metodo statico in grado di calcolare la distanza geodetica fra due città in input, utilizzando la seguente formula:

$$dist \coloneqq \arccos(sin(lat_1)\, sin(lat_2) + cos(lat_1)\, cos(lat_2)\, cos(lon_1 - lon_2)) \cdot r$$

Dove $(lat_1 lon_1)$ e $(lat_2 lon_2)$ sono rispettivamente latitudine e longitudine delle due città ed r è il raggio della terra.

Per l'esempio sono state inserite le informazioni necessaria dei più importanti aeroporti mondiali.
Creando un grafo di città e effettuando i dovuti collegamenti fra i vertici in modo da ottenere un grafo completo, l'output dell'algoritmo, partendo dall'aeroporto di Roma è il seguente:

```
L'albero di copertura minimo del grafo, partendo dal vertice <  Roma  > e':
[ Atlanta ]-->(1221.91)[ New York ]--x
[ Dubai ]-->(3009.07)[ Istanbul ]--x
[ Istanbul ]-->(1382.63)[ Roma ]-->(3009.07)[ Dubai ]-->(1780.18)[ Mosca ]--x
[ Londra ]-->(327.682)[ Parigi ]-->(5573.43)[ New York ]--x
[ Madrid ]-->(1063.12)[ Parigi ]--x
[ Monaco di Baviera ]-->(729.675)[ Roma ]-->(680.923)[ Parigi ]--x
[ Mosca ]-->(1780.18)[ Istanbul ]-->(5796.28)[ Pechino ]--x
[ New York ]-->(5573.43)[ Londra ]-->(1221.91)[ Atlanta ]--x
[ Parigi ]-->(680.923)[ Monaco di Baviera ]-->(327.682)[ Londra ]-->(1063.12)[ Madrid ]--x
[ Pechino ]-->(5796.28)[ Mosca ]-->(2133.97)[ Tokyo ]--x
[ Roma ]-->(1382.63)[ Istanbul ]-->(729.675)[ Monaco di Baviera ]--x
[ Sidney ]-->(7832.15)[ Tokyo ]--x
[ Tokyo ]-->(2133.97)[ Pechino ]-->(7832.15)[ Sidney ]--x

Il peso dell'MST e': 31531km
```

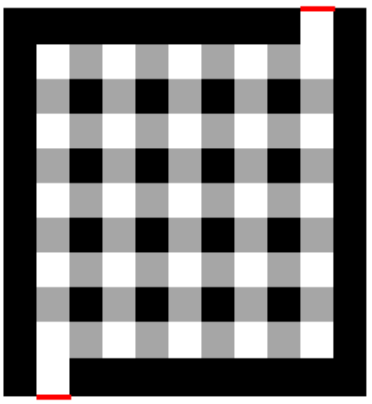
E rappresentandolo graficamente su un planisfero avremo:

Come si vede bene dall'immagine l'MST genera delle rotte aeree con degli scali, da percorrere per collegare tutti gli aeroporti effettuando in volo il minor numero di km.

Generazione di labirinti.

Un altro uso molto interessante dell'algoritmo di Prim è quello di impiegarlo nella generazione casuale di labirinti. Infatti possiamo vedere i percorsi di ogni labirinto come un albero, quindi senza cicli, dove ogni vicolo ceco e l'uscita sono foglie. E pertanto solo uno dei possibili path radice-foglia conduce all'uscita.

Un modo per implementare tale algoritmo potrebbe essere immaginare inizialmente il labirinto come una griglia di vertici, contenuta da un perimetro (le mura del labirinto). Ogni vertice è collegato con i vertici vicini che si trovano alle quattro direzioni cardinali.
Nell'immagine qui di fianco i quadrati bianchi rappresentano i vertici, quelli grigi gli archi che ci sono tra di loro ed in fine i neri rappresentano i muri del labirinto.

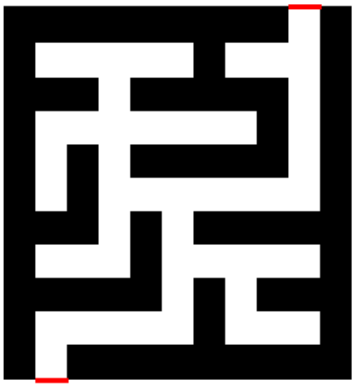


Una volta scelti due vertici qualsiasi uno per l'inizio e l'altro per la fine, l'idea di base potrebbe essere quella di collegare i vertici con archi di peso casuale ed una volta fatto ciò calcolare con l'algoritmo di Prim il minimum spanning tree, colorando di bianco gli archi che fanno parte dell'MST (i possibili percorsi) e di nero quelli non inclusi (i muri del labirinto).

Di seguito vi è un esempio di labirinto di dimensione 5x5 generato nel modo appena descritto:

```
L'albero di copertura minimo del grafo, partendo dal vertice < [0,0] > e':
[[0,0]]-->(41)[[1,0]]--x
[[0,1]]-->(6334)[[1,1]]--x
[[0,2]]-->(15724)[[0,3]]--x
[[0,3]]-->(11478)[[1,3]]-->(15724)[[0,2]]--x
[[0,4]]-->(26962)[[1,4]]--x
[[1,0]]-->(41)[[0,0]]-->(5705)[[2,0]]--x
[[1,1]]-->(16827)[[1,2]]-->(6334)[[0,1]]--x
[[1,2]]-->(9961)[[2,2]]-->(16827)[[1,1]]-->(491)[[1,3]]--x
[[1,3]]-->(491)[[1,2]]-->(11478)[[0,3]]-->(2995)[[2,3]]-->(11942)[[1,4]]--x
[[1,4]]-->(11942)[[1,3]]-->(26962)[[0,4]]-->(4827)[[2,4]]--x
[[2,0]]-->(5705)[[1,0]]-->(14604)[[2,1]]--x
[[2,1]]-->(14604)[[2,0]]-->(3902)[[3,1]]-->(153)[[2,2]]--x
[[2,2]]-->(153)[[2,1]]-->(9961)[[1,2]]-->(292)[[3,2]]--x
[[2,3]]-->(2995)[[1,3]]-->(17421)[[3,3]]--x
[[2,4]]-->(4827)[[1,4]]--x
[[3,0]]-->(21726)[[3,1]]-->(5447)[[4,0]]--x
[[3,1]]-->(3902)[[2,1]]-->(21726)[[3,0]]-->(14771)[[4,1]]--x
[[3,2]]-->(292)[[2,2]]-->(1869)[[4,2]]--x
[[3,3]]-->(17421)[[2,3]]--x
[[3,4]]-->(17035)[[4,4]]--x
[[4,0]]-->(5447)[[3,0]]--x
[[4,1]]-->(14771)[[3,1]]--x
[[4,2]]-->(1869)[[3,2]]-->(4664)[[4,3]]--x
[[4,3]]-->(4664)[[4,2]]-->(7711)[[4,4]]--x
[[4,4]]-->(7711)[[4,3]]-->(17035)[[3,4]]--x
```

Si è deciso di pensare ogni vertice come un punto del piano $\mathbb{N}^2$ appartenente al primo quadrante di un riferimento cartesiano in modo tale da rendere più semplice l'implementazione.
La foto a sinistra mostra attraverso una lista di adiacenza l'MST generato ed i pesi casuali associati ad ogni arco.
Collegando i vari vertici e rappresentando l'albero otteniamo proprio il nostro labirinto generato:



Per rappresentare il labirinto in maniera veloce ho usato una funzione che dal MST genera una matrice di *bool* i cui elementi hanno valore *false* dove si trovano i quadrati neri e valore *true* negli altri. La funzione è la seguente:

```
1.  template <class T> void Graph<T>::printMAZE(int ALTEZZA, int LARGHEZZA) {
2.      bool maze[ALTEZZA*2+1][LARGHEZZA*2+1];
3.      for(int i = 0; i < ALTEZZA*2+1; i++)
4.          for(int j = 0; j < LARGHEZZA*2+1; j++)
5.              maze[i][j]=false;
6.
7.      for(auto vertex : graph){                              //Per ogni vertice del grafo...
8.          int x = vertex.data.x * 2 +1;                      //Ogni vertice è distanziato di 1 spazio dagli altri
9.          int y = vertex.data.y * 2 +1;
10.         maze[x][y] = true;                                 //Metto in tutte le coordinate dei vertici.
11.         for(auto edge : vertex.adj){
12.             int x2 = edge.link->data.x *2 +1;              //Per ogni adiacente
13.             int y2 = edge.link->data.y *2 +1;
14.                                                            //Setto a 1 il collegamento i vertice
15.             if(x < x2 and y == y2) maze[x+1][y] = true;
16.             if(x > x2 and y == y2) maze[x-1][y] = true;
17.             if(x == x2 and y < y2) maze[x][y+1] = true;
18.             if(x == x2 and y > y2) maze[x][y-1] = true;
19.         }
20.     }
21.
22.     for(int i = 0; i <ALTEZZA*2+1; i++){                   //Stampo
23.         cout << endl;
24.         for(int j = LARGHEZZA*2; j >=0; j--)
25.             cout << " " << maze[j][i];}
26. }
```

Fatto ciò tale matrice viene semplicemente stampata a schermo e copiata ed incollata su Excel in modo da rappresentarla automaticamente.

Qui vi è invece di un labirinto 10x10, uno 20x20 ed uno 50x50 generati nello stesso modo: