Corso di Programmazione ad Oggetti

docente: Prof. Francesco Fontanella

Libreria di Strutture Dati

Studente: Nico Fiorini

Anno Accademico 2020/2021

Indice

1 Introduzione **2**

2 Raccolta dei requisiti **2**
2.1 Input / Output per Alberi e Grafi 2
2.2 Alberi 3
2.3 Grafi 3
2.4 D-Heap 3

3 Analisi dei Requisiti **3**
3.1 Alberi 3
3.2 Grafi 4
3.3 D-Heap 4
3.4 Input/Output Alberi e Grafi 5
3.5 Input/ Output D-Heap 5

4 Glossario dei termini **6**

5 Diagramma delle Classi **6**

6 Esempio con Dijkstra **8**
6.1 Tema esercizio 8
6.2 Svolgimento 8

7 Documentazione **9**
7.1 Compilazione 10
7.2 Dipendenze 10
7.3 Output grafico 10

1 Introduzione

Il progetto consiste nell'implementazione di una libreria di strutture dati quali Alberi, Grafi ed D-Heap che prende il nome di Bradipo. Il nome Bradipo deriva dal fatto che l'implementazione non è efficiente, cioè, non soddisfa i costi temporali e spaziali minimi quindi è una libreria relativamente lenta come un Bradipo confrontata con altre librerie, ma può essere funzionale per scopi didattici nello studio di algoritmi che fanno uso delle seguenti strutture dati implementate: Alberi, Grafi, D-Heap.
Come esempio di utilizzo di tale libreria, è stato implementato l'algoritmo di Dijkstra, vedi esempio nel Capitolo: 6

2 Raccolta dei requisiti

La libreria dovrà consentire di immagazzinare dati generici su strutture dati quali alberi, grafi ed heap, quindi si devono poter implementare algoritmi utilizzando l'ausilio dei metodi implementati in tali strutture.

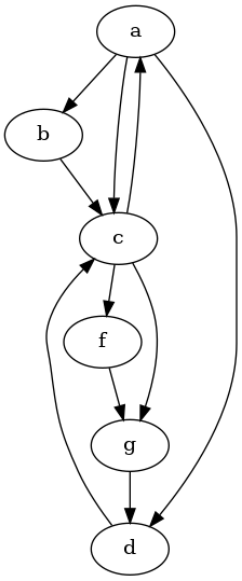
Si potrà usare un tipo generico per istanziare una struttura dati, come esempio di utilizzo si dovrà poter implementare l'algoritmo di Dijkstra per risolvere il problema dei cammini minimi per grafi con archi con peso maggiore di 0.

La libreria ha scopo puramente didattico, quindi potrà essere utile ad uno studente che vuole provare ad implementare una soluzione di un esercizio che fa uso di tali strutture, lo scopo quindi è rendere la libreria semplice e non si dovrà conoscere la struttura interna di tali strutture per poter essere utilizzata.

2.1 Input / Output per Alberi e Grafi

Per quanto riguarda l'Input e Output di tali strutture, si deve aver modo di interagire con tali strutture in modo uniforme con un formato standard per tutte le strutture utilizzando il formato csv come segue:

a,b
a,d
b,c
c,g
c,f
a,c
c,a
d,c
f,g
g,d



Inoltre, per l'output, fornire anche un utility che consenta di scrivere su file in linguaggio Dot in modo da poter generare un immagine che consenta la visualizzazione di una struttura, come nell'immagine precedente generata tramite il software Graphviz.

2.2 Alberi

Si voglio due tipi diversi di Alberi, con la stessa interfaccia Api, un implementazione tramite una lista di puntatori ai figli (rappresentazione collegata) , e l'altra tramite un vettore posizionale (rappresentazione indicizzata).

2.3 Grafi

Anche per i Grafi implementare due tipi, uno tramite la lista di archi, e un'altra tramite una lista di incidenza, anche qui da uniformare l'interfaccia per entrambe le strutture.

2.4 D-Heap

Per quanto riguarda l'Heap da implementare, dovrà consentire oltre al contenuto generico, anche una chiave generica, che permetterà di mantenere l'heap con la priorità basata sulla chiave piuttosto che con il valore. In modo che anche cambiando il contenuto dell'elemento il nodo rimarrà con la stessa priorità. Inoltre, l'heap da implementare dovrà poter essere configurato in modo da dare la priorità alla minima chiave, o alla massima chiave, in base al volere dell'utente. Utilizzare il vettore posizionale (TreePosVector) implementato come struttura dati per l'utilizzo dell'Heap.

3 Analisi dei Requisiti

Dalla nostra analisi preliminare, emergono alcune conclusioni importanti. Le nostre strutture dati sono costituite da nodi e archi. Inoltre, poiché gli alberi sono una forma particolare di grafo non orientato e aciclico, intendiamo soddisfare i requisiti utilizzando un'approccio di programmazione orientata agli oggetti. Nei grafi, i nodi contengono informazioni, e questi nodi sono collegati tra loro attraverso gli archi.

Dai requisiti vogliamo che il contenuto informativo sia generico, tuttavia, ciascun nodo deve essere identificabile in modo univoco. Per questo motivo, richiediamo che il tipo di dati contenuto nei nodi possa essere identificato utilizzando operatori di confronto:(<, <=, >, >=, ==).

Per quanto riguarda l'utilizzo delle nostre strutture dati, lil progetto si baserà sulla programmazione orientata agli oggetti per esporre un'API all'utente. La libreria fornirà interfacce distinte per ciascuna delle nostre strutture dati, lma condividerà i metodi comuni tramite l'ereditarietà. Ad esempio, gli alberi, sebbene siano casi particolari di grafi, avranno un'interfaccia diversa rispetto ai grafi, ma condivideranno le operazioni sui nodi.

3.1 Alberi

Le due tipologie di Alberi sono state implementate una tramite una rappresentazione indicizzata, e una tramite una rappresentazione collegata.

1. **TreePosVector** è la classe che implementa l'albero tramite una rappresentazione indicizzata, ovvero la struttura dell'albero è implicita nell'indice dell'array in cui è posizionato il nodo. In particolare, quest'albero utilizza un vettore posizionale. Un vettore posizionale è un array $P[..]$, di dimensione n tale che $P[\nu]$ contiene l'informazione associata ad un nodo v, e l'informazione associata all'$i-esimo$ figlio di v è in posizione $P[d*v+i]$ Per semplicità di indicizzazione, la posizione 0 dell'array resta inutilizzata. Questa tipologia consente di accedere ai figli di un nodo, o al padre di un nodo in tempo costante. Tuttavia, questa metodologia ha dei vincoli strutturali, in quanto un albero è caratterizzato dal grado massimo del nodo, perciò, un nodo non può avere un numero di figli a piacimento.

2. **TreePtrList** è la classe che implementa l'albero tramite una rappresentazione collegata, questa struttura non ha un vincolo strutturale, perciò più consona da utilizzare in algoritmi di ricerca per i grafi. L'albero contiene un insieme di nodi, e ogni nodo contiene una lista di puntatori ai figli, perciò si è in grado di risalire ai figli in tempo $O(\delta(v))$, dove $\delta(v)$ è il grado del nodo v.

3.2 Grafi

Come richiesto vengono implementati due tipi di grafi, per i grafi viene scelto di non fare distinzione tra grafi orientati e non orientati, e un grafo non orientato si può ottenere duplicando l'informazione, aggiungendo due archi piuttosto che uno. Un esempio di grafo è mostrato in Figura 1

1. **GraphedgeList** è la classe che implementa i Grafi utilizzando un array di archi, questa rappresentazione è molto diffusa, anche se i tempi di esecuzione di alcune operazioni fondamentali non sono molto efficienti rispetto ad altri tipi di rappresentazioni. La struttura del grafo è implicita nella relazione di Arco, che contiene i puntatori ai nodi, il nodo di partenza e il nodo di arrivo. Una visualizzazione grafica della struttura è riportata in Figura 2

2. **GraphIncList** è la classe che implementa i Grafi con la lista di incidenza, ovvero una lista (nel mio caso una map) che contiene tutti i nodi, e una lista che contiene tutti gli archi. I nodi hanno una lista di puntatori agli archi, quindi accedendo ad un nodo, è possibile sapere quali sono gli archi che lo interessano, perciò quali sono le connessioni che lo riguardano. Una visualizzazione grafica della struttura è riportata in Figura 3

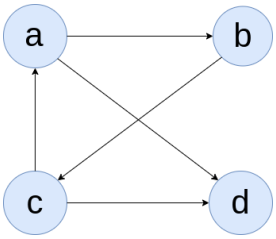


Figura 1: Esempio di un grafo

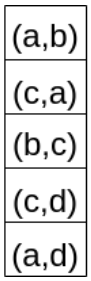


Figura 2: Lista di archi

3.3 D-Heap

L'heap è stato implementato con l'ausilio del vettore posizionale implementato, vedi: 3.1. Un d-heap è un albero radicato d-ario con le seguenti proprietà:

- **Struttura:** un d-heap di altezza h è completo almeno fino a profondita $h-1$.

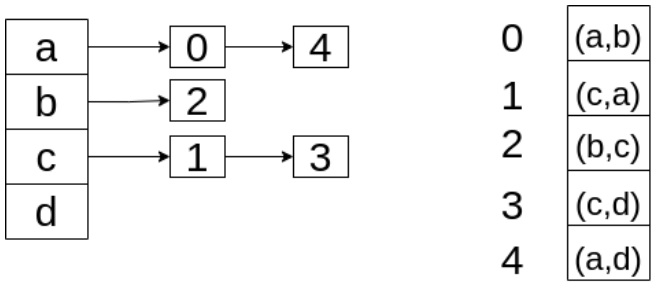


Figura 3: Lista di Incidenza

- **Contenuto informativo:** Ciascun nodo v contiene un elemento $elem(v)$ e una chiave $chiave(v)$ presa da un universo totalmente ordinato.
- **Ordinamento a heap:** Ciascun nodo v diverso dalla radice ha una chiave inferiore a quella del suo genitore: $chiave(v) \geq chiave(padre(v))$

Il vettore posizionale dunque cade a pennello per essere utilizzato nell'implementazione di tale struttura.

3.4 Input/Output Alberi e Grafi

Come richiesto si può riempire un albero o un grafo tramite un file in formato CSV, inoltre, è possibile cambiare il carattere delimitatore nel caso il carattere "," sia scomodo per i nostri scopi. Allo stesso modo l'output sarà simile all'input. Esempio input albero:

Listing 1: tree.txt

```
(padre,figlio)
a
a,l
a,b
l,e
l,r
b,o
```

3.5 Input/ Output D-Heap

Per il DHeap il formato di Input Output è analogo agli alberi e i grafi: Esempio DHeap<int,string>:

Listing 2: dheap.txt

```
(chiave int,valore string)
1,a
2,l
5,b
3,e
8,r
9,o
```

4 Glossario dei termini

- **Node:** Classe non utile allo scopo dell'utente, contiene dati utili all'implementazioni delle strutture dati.
- **Edge:** Classe di utilità all'utente per l'inserimento di nodi connessi tra di loro.
- **BasicGraph:** Classe Base di Tree e Graph, contiene i metodi in comune per l'interazione con i nodi.
- **Graph:** Classe Base astratta che eredità da BasicGraph, contiene l'interfaccia comune per le due implementazioni dei Grafi
- **GraphEdgeList:** Classe che contiene l'implementazione del grafo implementato con la tecnica "Lista di archi", eredita da Graph.
- **GraphIncList:** Classe che contiene l'implementazione del grafo con la tecnica "Lista di incidenza", eredita da Graph
- **Tree:** Classe Base astratta che eredità da BasicGraph, contiene l'interfaccia comune per le due implementazioni di Tree.
- **TreePtrList:** Classe che contiene l'implementazione dell'albero con la tecnica "Lista di puntatori ai figli".
- **TreePosVector:** Classe che contiene l'implementazione dell'albero con la tecnica "Vettore Posizione".
- **DHeap:** Classe che implementa l'heap, è stato implementato con la tecnica DHeap, utilizzando la struttura dati TreePosVector.

5 Diagramma delle Classi

In Figura 4 è rappresentato il diagramma UML delle classi, dove sono mostrati le relazioni tra le classi del progetto.

Si può notare che la Classe base di alberi e grafi è **BasicGraph<T>**, questa classe contiene le operazioni in comune tra Alberi e Grafi ed è una classe astratta in relazione di composizione con la classe di **Node<T>** ed **Edge<T>**. Le classi **Tree<T>** e **Graph<T>** sono anch'esse astratte e contengono l'interfaccia di base delle due tipologie di strutture dati.

Per quanto riguarda le classi che derivano da Tree<T>e Graph<T>, sono le implementazioni delle particolarità strutture dati e sono quelle che l'utente dovrà istanziare per poter usufruire di tale libreria.

Eccezione viene fatta per l'implementazione di **D-heap<K,T>**, in quanto questa struttura ha un interfaccia totalmente differente dalle altre, inoltre, il DHeap implementato fa uso di un albero per la sua implementazione ma non viene trattato dall'utente come un albero, ma come un container con le proprietà descritte in precedenza. Perciò, si è deciso che questa è in relazione di composizione con **TreePosVector<T>**.

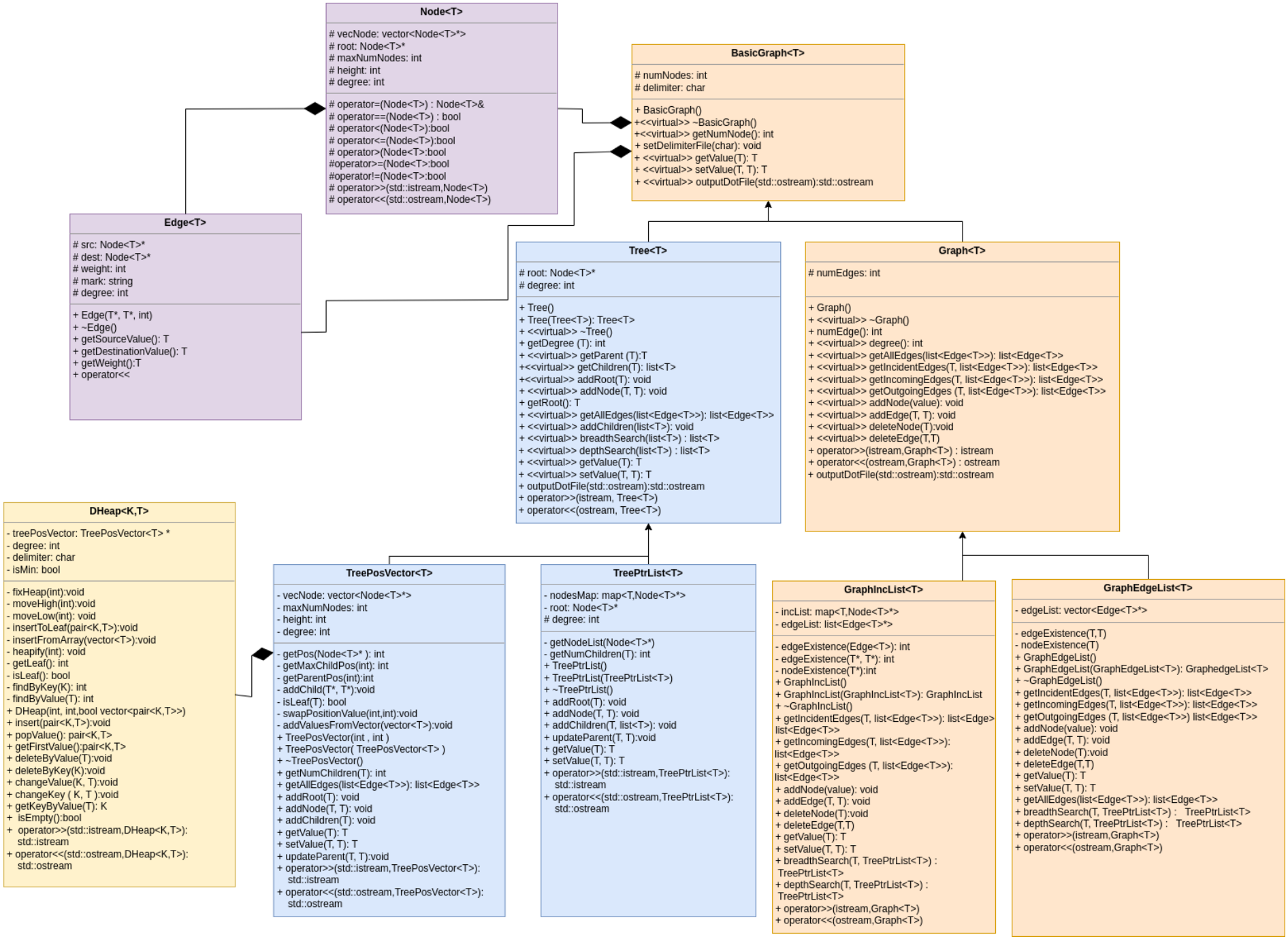


Figura 4: Diagramma delle classi

6 Esempio con Dijkstra

Per utilizzare la libreria si rimanda alla documentazione, qui è riportato un esempio di utilizzo, l'implementazione di Dijkstra in un caso semplicistico e un po' fantasioso.

6.1 Tema esercizio

Immaginiamo che si sviluppi in Italia una linea ferroviaria in grado di attraversare oggetti materiali, e quindi in grado di poter andare da una stazione A, ad una stazione B in linea d'aria. Si prenda in Input da File il grafo non orientato che collega 21 città italiane attraverso questa ferrovia immaginaria, stampare l'albero dei cammini minimi con la radice "Roma".

6.2 Svolgimento

Algorithm 1 Dijkstra's Algorithm

```
1: D_su ← +∞
2: T is a tree consisting of only vertex root
3: Create an empty min-heap S
4: D_ss ← 0
5: S.insert(s, 0)
6: while S is not empty do
7:     u ← S.popValue()
8:     for each (u, v) in G do
9:         if D_sv = +∞ then
10:            S.insert(v, D_su + w(u, v))
11:            D_sv ← D_su + w(u, v)
12:            Make u the parent of v in T
13:        else if D_su + w(u, v) < D_sv then
14:            S.decreaseKey(v, D_sv − (D_su − w(u, v)))
15:            Make u the new parent of v in T
16:        end if
17:    end for
18: end while
```

Facendo riferimento allo pseudo codice dell'algoritmo di Dijkstra 1, notiamo che abbiamo bisogno di tutte e 3 le tipologie di strutture dati implementati, infatti, abbiamo bisogno di una coda con priorità (DHeap<K,T>), un albero, in particolare utilizziamo un TreePtrList<T >, in quanto non abbiamo limite sul numero dei figli di ogni nodo, e risulta più conveniente per via della sua flessibilità in confronto con il vettore posizionale (TreePosVector). Utilizziamo una GraphIncList<T >per modellare la ferrovia immaginaria, e dato che Veroli(il mio paese) è un paese un po' scomodo per spostarsi senza macchina, colleghiamo Veroli a Roma, almeno nelle mie fantasie posso vivere meglio.

La prima cosa da fare è creare una classe che rappresentano le stazioni della città denominata City, che contiene le coordinate della stazione ed il nome della città. Inoltre definiamo gli operatori di comparazioni necessari alla libreria per distinguere i nodi, compresi l'operatore di input ed output per consentire agli archi di poter essere costruiti da File.

Il codice dello svolgimento dell'esercizio si trova nella repository del progetto ai seguenti link:

- Dijkstra.h: Codice che contiene l'algoritmo di Dijkstra.

- City.h: Codice che contiene la classe City.
- Input file: File di input per il riempimento del grafo.

Vengono generati con Graphviz due immagini per aiutare a visualizzare l'input, ovvero il grafo che modella la stazione immaginaria, mostrata in Figura 5, gli archi sono pesati in base alla distanza in Km di linea d'aria tra le città calcolato con un algoritmo che prende in input le due coordinate delle città e restituisce la distanza in Km. L'output invece è mostrato in Figura 6, cioè l'albero dei cammini minimi partendo da Roma.

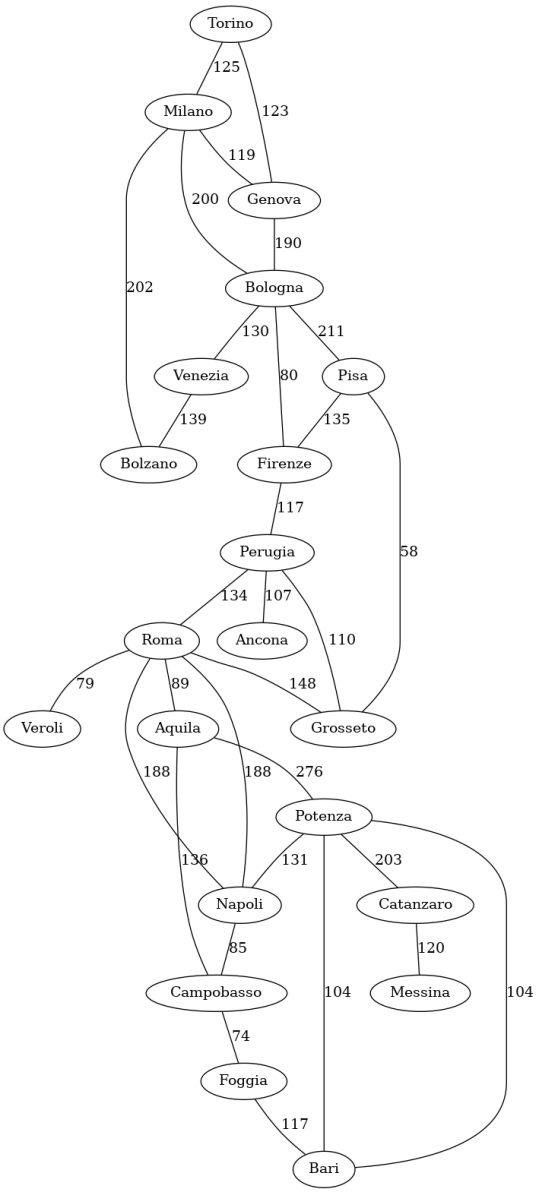


Figura 5: Grafo che modella la stazione immaginaria

7 Documentazione

La documentazione della libreria è stata generata con l'ausilio di Doxygen, e la si trova al seguente link: Documentazione

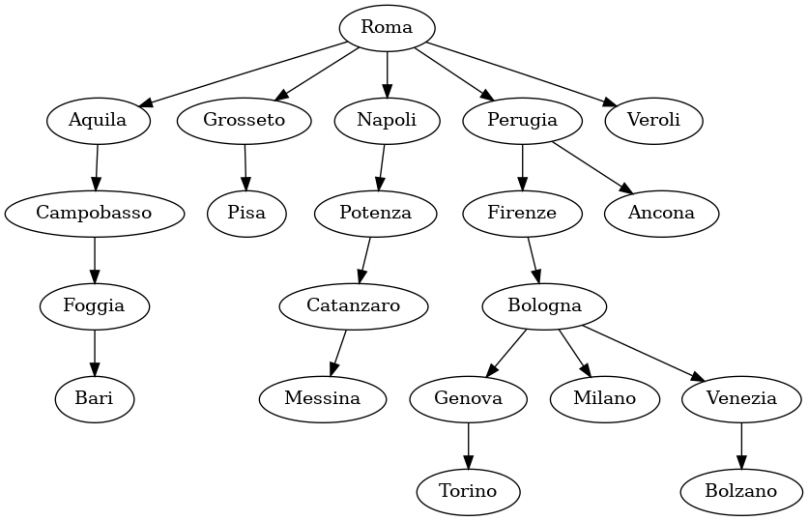


Figura 6: Albero cammini minimi

7.1 Compilazione

Per poter compilare il progetto, è necessario installare CMake, CMake è un'applicazione multi-piattaforma, quindi è disponibile per una varietà di sistemi operativi e consente la compilazione Cross-platform della libreria. E' richiesto lo standard C++ 17.

Per compilare il progetto, è necessaria la versione minima di CMake 3.10, e su Linux una volta installato basta eseguire questi comandi mettendosi dentro la directory del progetto:

Listing 3: Comandi Bash

```
mkdir build
cd build
cmake ..
make
```

7.2 Dipendenze

La libreria non ha dipendenze esterne, dipende solamente dalla libreria standard C++.

7.3 Output grafico

Per poter visualizzare l'output grafico di un albero o un Grafo, si può installare il tool GraphViz, che permette tramite l'output in linguaggio dot, di generare immagini grafiche.