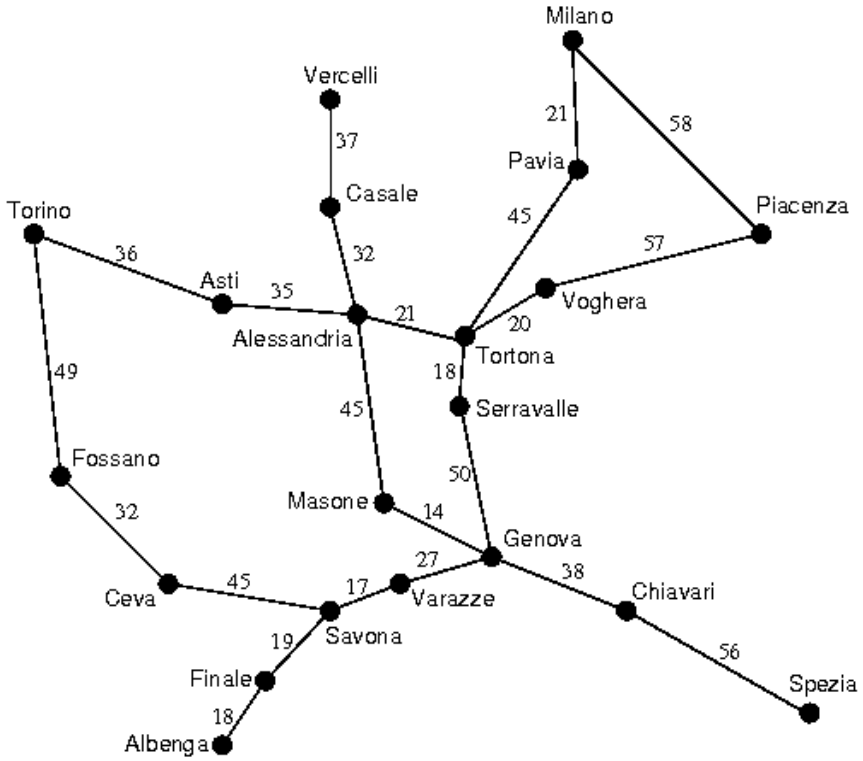
ASD 23/24 - Laboratorio 8

n questo laboratorio, si richiede di implementare il tipo di dato *Grafo non orientato con vertici etichettati e archi pesati*. L'implementazione deve sfruttare l'approccio a **liste di adiacenza**.

1 Motivazione

Si consideri un navigatore satellitare, usato da un commesso viaggiatore per pianificare viaggi tra le città in cui opera i propri commerci. Tra le tante funzioni che il navigatore deve offrire all'utente c'è anche quella di ricercare e suggerire un percorso che, da una qualunque città di partenza, conduca ad un'altra città di arrivo. Normalmente il percorso da cercare sarebbe quello di lunghezza minima, ma per semplicità ci limitiamo ad un percorso qualsiasi purchè sia *aciclico*, ossia tale per cui una stessa località venga visitata al più una volta.
Il navigatore deve poter caricare, da file o da standard input, le mappe delle regioni in cui l'utente si muoverà. A titolo di esempio si consideri la mappa rappresentata sotto:



Tale mappa può essere rappresentata in formato testo come una lista che elenca i segmenti stradali fornendo per ciascuno le due città estreme e la lunghezza in km. Per l'esempio in figura abbiamo:

```
Torino Asti 36
Asti Alessandria 35
Torino Fossano 49
Fossano Ceva 32
Ceva Savona 45
Albenga Finale 18
Savona Finale 19
Varazze Savona 17
Genova Varazze 27
Casale Vercelli 37
Casale Alessandria 32
Alessandria Masone 45
Masone Genova 14
Genova Serravalle 50
Serravalle Tortona 18
Tortona Alessandria 21
Genova Chiavari 38
Chiavari Spezia 56
Tortona Voghera 20
Piacenza Voghera 57
Piacenza Milano 58
```

```
Tortona Pavia 45
Pavia Milano 21
0
```

Si noti lo "0" in fondo, utilizzato per terminare la sequenza di lettura.

Scopo di questo laboratorio è implementare le strutture dati e gli algoritmi necessari al navigatore satellitare per risolvere adeguatamente il problema descritto sopra. L'idea di fondo è che una mappa stradale si può rappresentare come grafo, supponendo che le città siano i vertici e le strade siano gli archi. I vertici risultano etichettati con i nomi delle città. Gli archi, per semplicità, sono non orientati (in pratica si suppone che le strade non abbiano sensi unici). Ciascun arco riceve un peso uguale alla lunghezza in chilometri della relativa tratta stradale.

2 Materiale dato

Nel file `asd-lab8-traccia.zip`, trovate:

- Un file `graph.h` contenente le intestazioni delle funzione da implementare
- Un file `graph.cpp` dove dovete scrivere l'implementazione delle funzioni richieste e anche definire il tipo della struttura
- Un file `graph-main.cpp` contenente un programma principale per testare le funzione via menù
- Un file `graph-test.cpp` contenente un programma principale che avvia una sequenza di test automatici
- Due file `mappa.txt` e `mappetta.txt` con la descrizione di grafi, il primo essendo più piccolo per svolgere test più semplici.
- I file `list-array.h` e `list-array.cpp` contenendo una implementazione delle liste. Questa implementazione potrà essere usata quando si chiedono liste di vertici (per esempio le funzione `adjacentList` e `findPath`), ma non deve essere usata per programmare le liste di adiacenza nella struttura.

3 Funzioni da implementare

I prototipi delle funzioni da implementare sono forniti nel file `graph.h` come descritto qui sotto e dovrete realizzare l'implementazione nel file `graph.cpp`. Notate che la struttura `struct vertexNode` non è data e deve essere scritta da voi sempre in questo ultimo file. **Importante:** È utile mettere un campo `bool isVisited` che potrà essere usato per la ricerca di cammino.

```
namespace graph {
  typedef string Label; //etichetta dei vertici
  typedef unsigned int Weight; //peso degli archi

  struct vertexNode; // da definire nel file graph.cpp

  typedef vertexNode* Graph; // un grafo e' identificato dal
  //puntatore al primo vertice inserito

  const Graph emptyGraph = nullptr;

  /*********************************************************************/
  /*               Funzione da implementare                            */
  /*********************************************************************/
  // Restituisce il grafo vuoto
  Graph createEmptyGraph();

  // Aggiunge nuovo vertice con etichetta la stringa.
  // Se non e gia presente, ritorna true, altrimenti fallisce e ritorna false
  bool addVertex(Label, Graph&);

  // Aggiunge nuovo arco tra i due vertici con etichette le due stringe e peso
  // l'intero. Fallisce se non sono presenti tutti e due i nodi o se l'arco
  // tra i due e' gia' presente. Se fallisce ritorna false,
  // altrimenti ritorna true
  bool addEdge(Label, Label, Weight, Graph&);

  // Restituisce true se il grafo e' vuoto, false altrimenti
  bool isEmpty(const Graph&);

  // Ritorna il numero di vertici del grafo
```

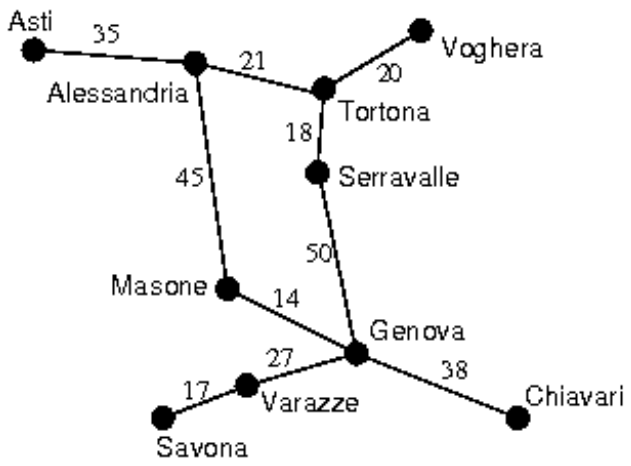
```
  unsigned int numVertices(const Graph&);

  // Ritorna il numero di archi del grafo
  unsigned int numEdges(const Graph&);

  // Calcola e ritorna (nel secondo parametro) il grado del vertice.
  // Restituisce un valore falso se il vertice non esiste,
  // altrimenti ritorna true
  bool nodeDegree(Label, unsigned int&, const Graph&);

  // Verifica se due vertici sono adiacenti (ovvero se esiste un arco)
  bool areAdjacent(Label, Label, const Graph&);

  // Ritorna la lista di adiacenza di un vertice
  // corrispondente alla lista dei label dei vertici adiacenti
  list::List adjacentList(Label, const Graph&);

  // Calcola, se esiste, un cammino tra due vertici
  // Il primo argomento e il vertice di partenza
  // Il secondo argomento e il vertice di arrivo
  // Il terzo argomento sara la lista delle etichette degli
  // vertici visitati sul cammino (senza il vertice di partenza,
  // ma con il vertice di arrivo)
  // Si assume che il chiamante fornisca inizialmente una lista vuota.
  // Il quarto argomento e il peso del cammino
  // La funziona ritorna false se non c'e un cammino tra i due vertici
  // Se il vertice di partenza e uguale al vertice di arrivo, la funzione
  // ritorna true, e il peso e 0 e la lista e' vuota
  bool findPath(Label, Label, list::List&, Weight&, const Graph& g);

  // Svuota un grafo
  void clear(Graph&);

  // Stampa il grafo
  // Per ogni vertice stampa su una riga l'etichetta del vertice seguito di ':'
  // poi le etichette dei vertici adiacenti con fra parentesi il peso associato,
  // separate da virgole
  void printGraph(const Graph&);
}
```

4 Tests manuali

Il file `graph-main.cpp` contiene il `main` di un programma per aiutarvi a svolgere dei tests.
Per potere usare questo programma con la vostra nuova implementazione, potete compilarlo così:
`g++ -std=c++11 -Wall list-array.cpp graph.cpp graph-main.cpp -o graph-main`
e poi eseguirlo con `./graph-main`.

Per fare le verifiche iniziali, è fornito un esempio di mappa più piccola nel file `mappetta.txt`.

```
Asti Alessandria 35
Varazze Savona 17
Genova Varazze 27
Alessandria Masone 45
Masone Genova 14
Genova Serravalle 50
Serravalle Tortona 18
Tortona Alessandria 21
Genova Chiavari 38
Tortona Voghera 20
0
```



5 Tests automatici

Nel file `graph-test.cpp`, abbiamo programmato una sequenza di tests che si eseguono automaticamente grazie i quali verifichiamo che le funzioni implementate si comportino bene. Per usare questo programma invece, potete compilare così:
`g++ -std=c++11 -Wall list-array.cpp graph.cpp graph-test.cpp -o graph-test`
e poi eseguirlo con `./graph-test`.

6 Consegna

Per la consegna, creare uno `zip` con tutti i file forniti.