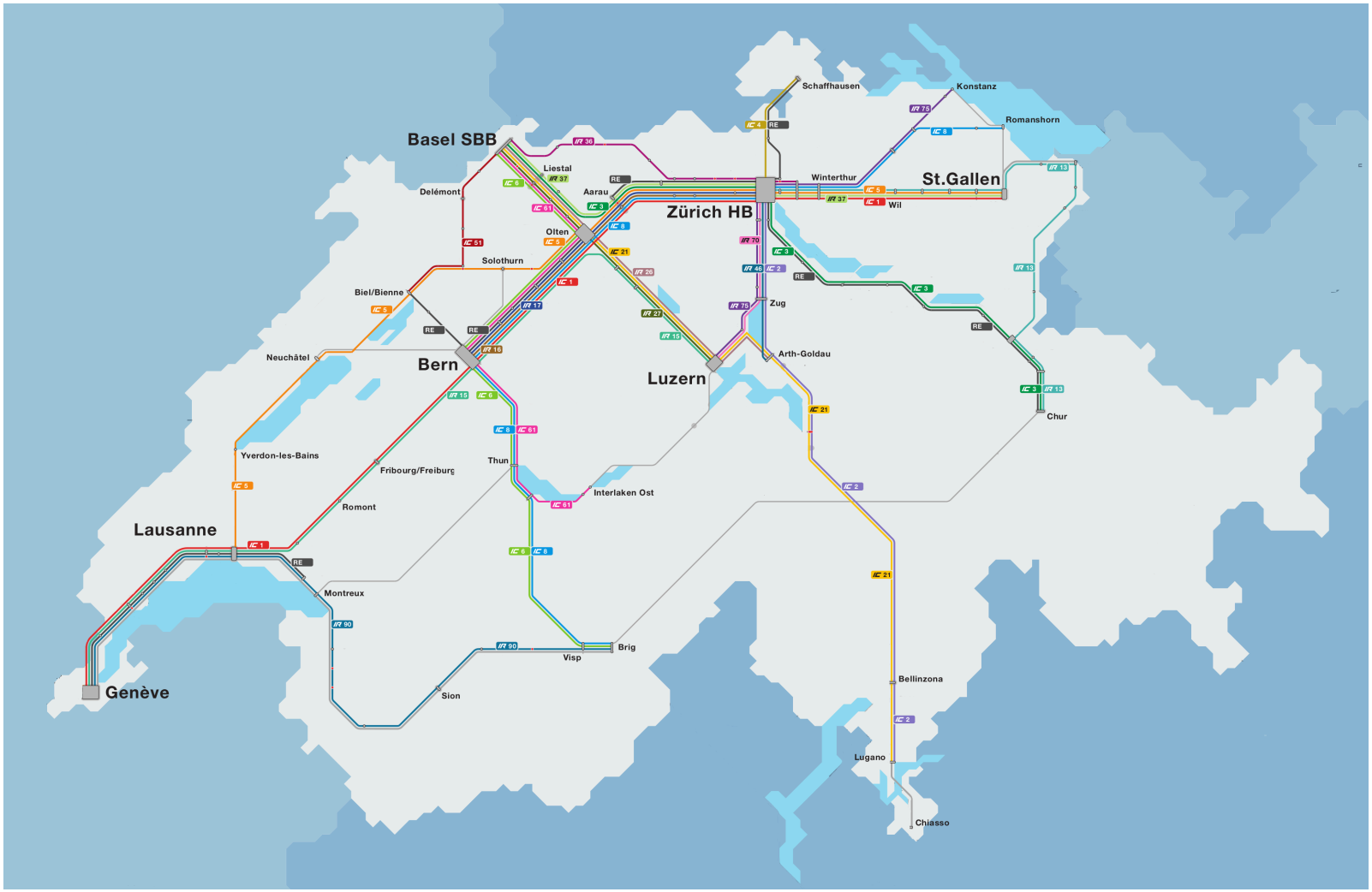
ASD 2: Labo 3

Réseau ferroviaire



ASD 2: Labo 3

Réseau ferroviaire

- Grandes lignes du réseau ferroviaire Suisse
  - Les villes sont reliées par des lignes de voies de chemin de fer
- Application d'algorithmes de type:
  - MST Arbre couvrant de poids minimum
  - SP Plus court chemin (Dijkstra à implémenter)
- Pour répondre à des questions du type:
  - Minimiser le coût de réfection des lignes avec la condition que chaque ville sera accessible par une ligne rénovée. Prix différent selon le nombre de voies sur la ligne.
  - Quel est le chemin le plus rapide entre Genève et Coire en passant par Brigue ?

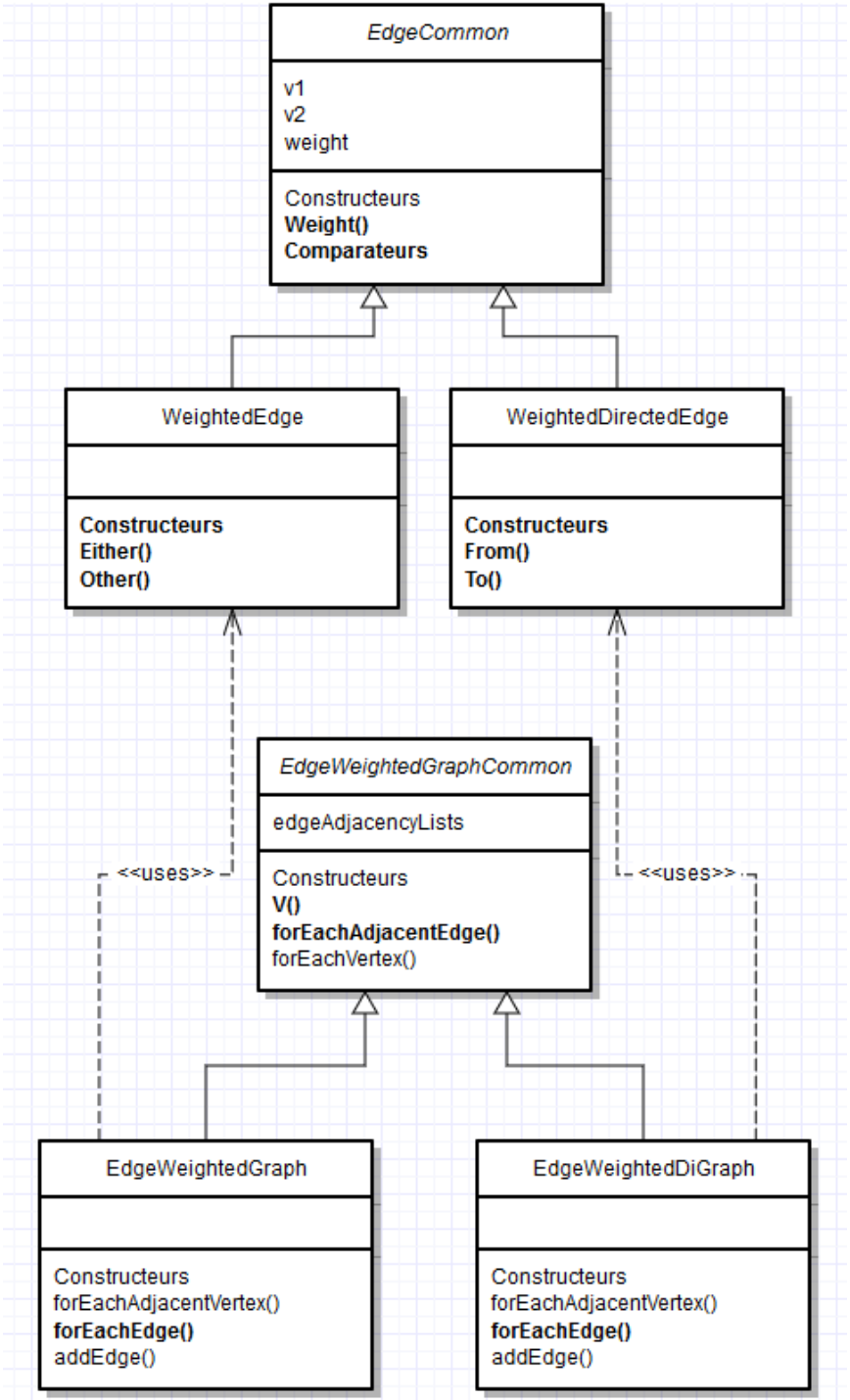
ASD 2: Labo 3
Code fourni

- Nous vous fournissons les implémentations de graphes pondérés (orienté et non-orienté)
- Dans le fichier *EdgeWeightedGraphCommon.h* vous trouverez la classe abstraite *EdgeWeightedGraphCommon* qui contient le code en commun
  - Dans *EdgeWeightedGraph.h* vous avez l'implémentation d'un graphe pondéré **non-orienté**: classe *EdgeWeightedGraph*
  - Dans *EdgeWeightedDigraph.h* vous avez l'implémentation d'un graphe pondéré **orienté**: classe *EdgeWeightedDiGraph*

ASD 2: Labo 3
Code fourni

- De manière similaire, ces 3 fichiers fournissent les structures représentant des *Edges:*
  - *EdgeCommon (EdgeWeightedGraphCommon.h)*
    Représente le code en commun entre une arrête et un arc pondéré, on utilisera uniquement ses «sous-classes»:
    - *WeightedEdge (EdgeWeightedGraph.h)* hérite de *EdgeCommon*
      Edge non-orienté → **arête pondérée**
    - *WeightedDirectedEdge (EdgeWeightedDigraph.h)* hérite de *EdgeCommon*
      Edge orienté → **arc pondéré**

Edges: arrêtes ou arcs pondérés



Graphes pondérés orientés ou non

ASD 2: Labo 3
Code fourni

- Dans la première partie du laboratoire, on souhaite appliquer des algorithmes d'arbres couvrants de poids minimum à un réseau ferroviaire.
- La classe *TrainNetwork* fournie, permet de lire le fichier *txt* de définition du réseau ferroviaire. Elle ne possède en revanche pas les méthodes nécessaires permettant d'appliquer les algorithmes directement dessus.

ASD 2: Labo 3
Wrappers

- Pour appliquer ces algorithmes sur le réseau ferroviaire, il y a 2 manières de faire:
  - Utiliser l'instance du *TrainNetwork* pour remplir une instance de, par exemple si on à besoin d'un graphe orienté, *EdgeWeightedDiGraph*
    Cette solution nécessite de dupliquer intégralement le graphe en mémoire, ce n'est pas souhaité
  - La seconde façon de faire, celle qui est demandée dans ce labo, est d'encapsuler (de wrapper) le *TrainNetwork* dans une structure qui traduira l'API du *TrainNetwork* dans une API utilisable par les algorithmes

ASD 2: Labo 3
Wrappers

- L'API nécessaire pour être utilisable par les algorithmes SP ou MST est la suivante:
  - `int V() const`
  - `void forEachEdge(Func f) const`
  - `void forEachAdjacentEdge(int v, Func f) const`

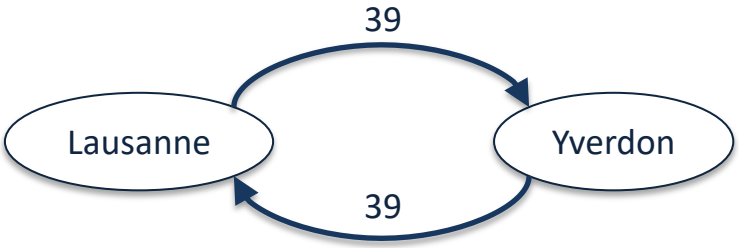
  `f` étant une fonction qui sera appelée sur les Edges

- C'est donc ces méthodes que vos wrappers devront implémenter

ASD 2: Labo 3
Code fourni

- Les algorithmes d'arbre couvrant de poids minimum utilisent des arêtes et les algorithmes de plus courts chemins des arcs.
- Il y aura donc 2 types de wrappers selon que l'algorithme utilise des arêtes ou des arcs.
- Pour les algorithmes utilisant des arcs, il faudra bien faire attention à transformer les lignes du réseau ferroviaire en 2 *WeightedDirectedEdge* opposés, chacun composé d'un sommet de départ, d'un sommet d'arrivée et d'un poids. C'est à ce moment que la fonction de coût intervient, elle doit permettre de calculer un poids pour une ligne.

Ligne ferroviaire du fichier txt
Wrappers pour les MST (arête)



Wrappers pour les SP (arcs)

ASD 2: Labo 3
Code fourni

Variables importantes de la classe *TrainNetwork :*

- *cities* : tableau des villes du réseau (les nœuds*)*
- *lines* : tableau des lignes du réseau

  Chaque ligne possède les variables (poids) :
  - *cities* : les deux villes que la ligne relie
  - *length* : la longueur de la ligne (en kilomètre)
  - *duration* : la durée en minute qui sépare les deux villes de la ligne
  - *nbTracks* : le nombre de voies de la ligne
- *cityIdx* : map permettant de trouver l'indice d'une ville à partir de son nom

ASD 2: Labo 3
Wrapper

Utilisation de *Wrappers*:

```
TrainNetwork tn("reseau.txt");

TrainGraphWrapper tgw(tn);
auto mst = MinimumSpanningTree<TrainGraphWrapper>::Kruskal(tgw);


TrainDiGraphWrapper tdgw(tn);
DijkstraSP<TrainDiGraphWrapper> sp(tdgw, v);
```

- Ne stocker que des références !
- Vous pouvez définir autant de *Wrappers* que nécessaires, soit avec une fonction de coût définie explicitement dans la classe, soit avec un paramètre permettant de passer une fonction de coût au constructeur (plus élégant)

ASD 2: Labo 3
Algorithme de Dijsktra

- Dans une deuxième partie, vous devrez implémenter l'algorithme de *Dijsktra*:
  - Nous vous fournissons la méthode *testShortestPath()* qui compare vos résultats avec ceux de notre implémentation de *BellmanFord.* Utilisation des fichiers de différentes tailles fournis
- Les temps d'exécution de *Dijsktra* (algorithme uniquement) doit rester très court
  - (l'implémentation de *BellmanFord* fournie peut prendre 1-2 minutes sur le graphe à 10'000 sommets, *Dijsktra* sera plus rapide)