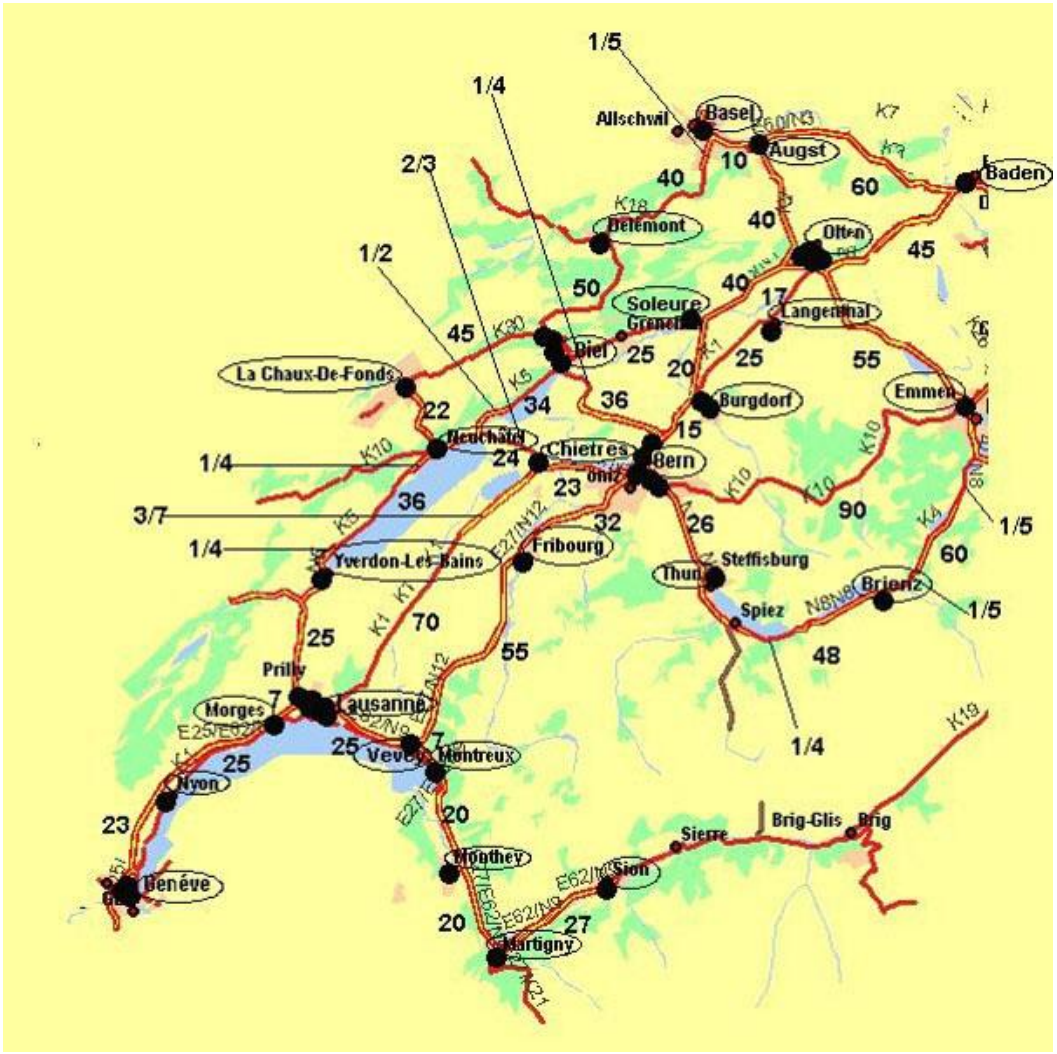
ASD 2: Labo 3

Réseau routier



ASD 2: Labo 3

Réseau routier

- Réseau routier principal de Suisse occidentale
  - Les villes sont reliées par des routes et des autoroutes (ou un mélange des deux: fraction d'autoroute)
- Application d'algorithmes de type:
  - SP Plus court chemin (Dijkstra à implémenter)
  - MST Arbre couvrant de poids minimum
- Pour répondre à des questions du type:
  - Quel est le chemin le plus rapide entre Genève et Emmen en passant par Yverdon ?
  - Minimiser le coût de réfection des routes avec la condition que chaque ville sera accessible par une route rénovée. Prix différent selon le type de route.

ASD 2: Labo 3

Réseau routier

- Vous devrez implémenter *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)

ASD 2: Labo 3
Wrapper

- Un *wrapper* est un objet encapsulant un autre objet dans le but de fournir une API particulière ou d'ajouter de l'abstraction (l'utilisateur passe par le *wrapper* pour accéder à l'objet «wrappé»).
- Dans ce laboratoire, il faut définir différents *wrappers* sur *RoadNetwork* (représentant le réseau routier) fournissant l'API nécessaire aux algorithmes de plus court chemin et arbre recouvrant minimum

ASD 2: Labo 3
Réseau routier

Utilisation de *Wrappers*:

```
RoadNetwork rn("reseau.txt");


RoadGraphWrapper rgw(rn);
auto mst = ASD2::MinimumSpanningTree<RoadGraphWrapper>::Kruskal(rgw);


RoadDiGraphWrapper rdgw(rn);
ASD2::DijkstraSP<RoadDiGraphWrapper> sp(rdgw, v);
```

- Ne stocker que des références !
- La fonction de coût peut être «hard-codée» mais une solution plus élégante est la bienvenue

ASD 2: Labo 3
Réseau routier

Les Wrappers doivent définir un type Edge (orienté ou non) et implémenter les méthodes suivantes:

Algorithme	V()	forEachEdge(*)	forEachAdjacentEdge(*)
Kruskal	✓	✓	
LazyPrim	✓		✓
EagerPrim	✓		✓
BellmanFordSP	✓	✓	
BellmanFordQueueSP	✓		✓
DijkstraSP	?	?	?

Egalement, les poids des arcs/arrêtes sont générés par les *Wrappers*

ASD 2: Labo 3
Code fourni

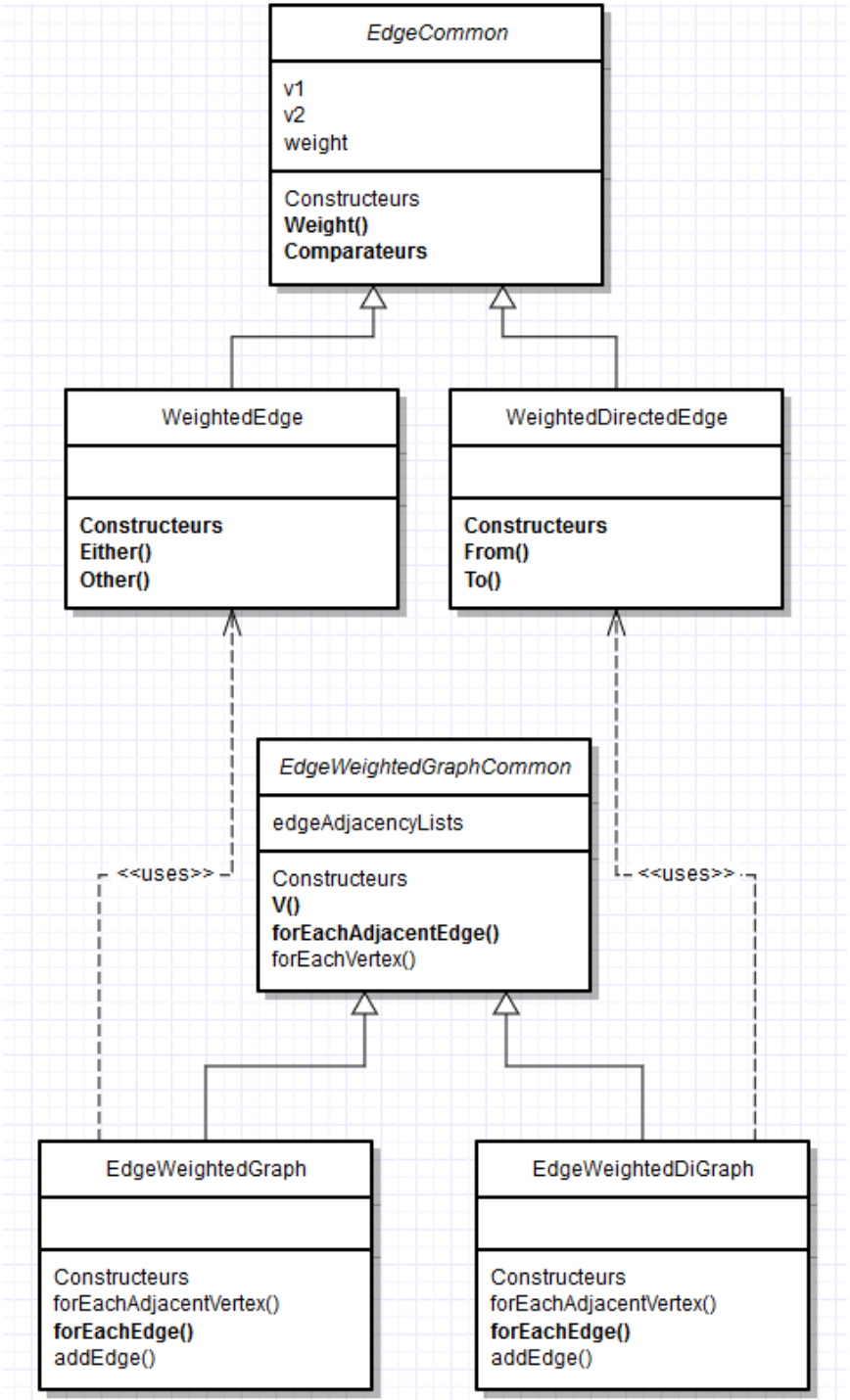
- Nous vous fournissons les implémentations de graphes pondérés (orienté et non-orienté)
- Dans le fichier *EdgeWeightedGraphCommon.h* vous trouverez la classe abstraite *EdgeWeightedGraphCommon* qui contient le code en commun
  - Dans *EdgeWeightedGraph.h* vous avez l'implémentation d'un graphe pondéré **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 → **arrête 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 (implémentation de Dijkstra), la méthode fournie *testShortestPath()*, appelée dans le main, va charger depuis plusieurs fichiers *txt* des graphes. Elle utilise le constructeur de *EdgeWeightedDiGraph<double>(filename)*, on aura donc une instance d'un graphe orienté pondéré, sur laquelle on va appliquer les algorithmes de *BellmanFordSP* (fourni) et de *DijkstraSP* (implémenté par vous). Elle vérifiera ensuite que les sorties des 2 algorithmes de plus courts chemins sont identiques → cela vérifie votre implémentation de *Dijkstra*

ASD 2: Labo 3
Code fourni

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

ASD 2: Labo 3
Code fourni

- Pour appliquer ces algorithmes sur le réseau routier, il y a 2 manières de faire:
  - Utiliser l'instance du *RoadNetwork* 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 *RoadNetwork* dans une structure qui traduira l'API du *RoadNetwork* dans une API utilisable par les algorithmes

ASD 2: Labo 3
Code fourni

- 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
- En prenant l'exemple de *BellmanFordSP* (fourni)
  - Il utilise la méthode V() pour initialiser ses structures
  - Il utilise la méthode forEachEdge(**`Func f`**), l'implémentation de **f** s'attend à recevoir des Edges orientés (*WeightedDirectedEdge* ou objet avec une API équivalente)
  - La méthode `forEachAdjacentEdge()` n'est pas utilisée, elle n'est donc pas obligatoire dans ce Wrapper

ASD 2: Labo 3
Code fourni

- Si on souhaite faire un *Wrapper* du réseau routier compatible avec *BellmanFordSP*, il faudra implémenter:
  - La méthode `V()` qui retourne le nombre de sommets = le nombre de villes
  - La méthode `forEachEdge(Func F)` qui appliquera la méthode `f` reçue en paramètre à tous les arcs = à toutes les routes
    Il faudra dans ce cas transformer chaque route en 1 (ou plusieurs) *WeightedDirectedEdge*, 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 route