AeA

TD/TP 2 -- Calcul d'arbres recouvrants de coût minimum

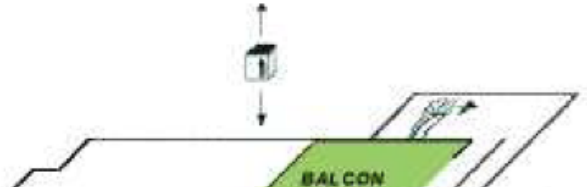
But

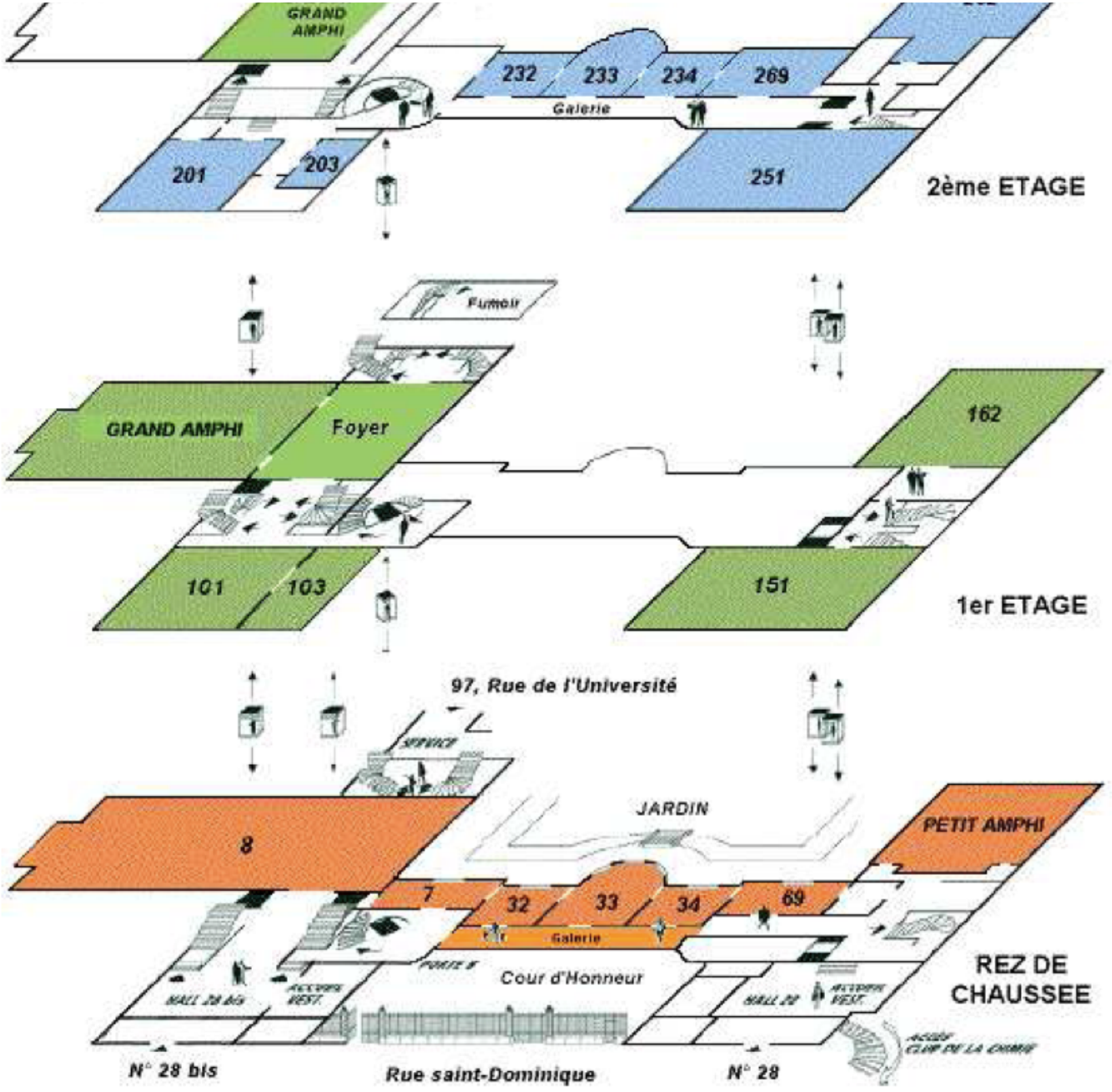
Le but de ce TD/TP est de tester les deux algorithmes classiques de calcul d'arbres recouvrants vus en cours (Prim et Kruskal) et d'en comparer les performances. À la fin du TP une mini-compétition entre vous est proposée.

Pour s'echauffer, on commence simplement avec l'exercice suivant.

Exercice 1 : Modélisation d'un exemple simple

On se propose d'aider à la réalisation du plan du réseau informatique visant à cabler l'ensemble de l'immeuble dessiné ci-dessous.





Le problème

Le coût de connexion entre deux pièces est de :

- 1 si les pièces sont au même étage et ont une cloison commune.
- 2 s'il faut traverser une autre pièce.
- 4 s'il faut traverser un hall ou un couloir.
- 5 s'il faut changer d'un niveau.
- 10 s'il faut changer de deux niveaux.

Que cherche-t-on à faire pour réaliser un cablage avec un coût minimal ?

Résolution

- Écrire une classe java qui représente un graphe valué. Vous êtes libre d'adopter la représentation qui vous semble la plus adéquate en fonction de l'algorithme que vous voulez implémenter. Votre classe pourra par exemple s'inspirer de l'interface suivante :
  - n.b.: Nous allons manipuler des graphes valués par la suite !
  - n.b.: Implémenter seulement les méthodes qui vous semblent utiles. Ne pas hésiter à revoir votre structure si besoin est !

```
public interface Graph {
     public void addVertex();
     public void addVertexNumber(int i)
                    throws
VertexAlreadyExistException;
     public void addEdge(Vertex v1,
Vertex v2);
                    throws
VertexNotFoundException;
```

```
    public void addEdge(int i, int j)
                    throws
VertexNotFoundException;
    ...
    public Vertex getVertex(int i);
    ...
    public Iterator<Edge>
getSortedEdgeIterator();
    ...
}
```

- Donner le réseau de coût minimal en choisissant l'algorithme que vous préférez.
- Comparez votre solution avec celle du voisin.

Exercice 2 : Graphes aléatoires

Notre but maintenant est d'implémenter et de comparer les deux algorithmes (Prim et Kruskal) vus en cours. Pour cela on va créer un jeu de test en utilisant des graphes aléatoires.

Graphe d'Erdos-Renyi (1959)

De façon générale, il est important d'être capable de générer des entrées aléatoires pour tester nos algorithmes et pouvoir prouver de façon expérimentale leurs propriétés. Dans le cadre des algorithmes sur les graphes, nous voulons générer des graphes aléatoires de façon simple. Il existe différentes façons de générer des graphes aléatoires avec des propriétés bien définies. Ici, on s'interesse à un modèle simple et

bien connu sous le nom du modèle d'Erdös-Renyi (du nom de deux mathématiciens célèbres Paul Erdös et Alfred Renyi).

Plus précisement, soit n un entier positif et p un nombre réel compris entre 0 et 1. On procède alors comme suit :

1. On génère n sommets numérotés de 1 à n.
2. Pour chaque paire de sommet i et j on connecte les deux sommets avec une probabilité p.

- Écrire un classe java ayant une méthode qui permet de générer un graphe aléatoire selon le modèle d'Erdos-Renyi.

```
public interface RandomGraphGenerator {
     Graph generateErdosRenyiGraph(int n,
float p)
                          throws
IllegalArgumentException;
}
```

Pour ajouter des pondérations sur les graphes ainsi générés, nous nous proposons de générer pour chaque arête un poids correspondant à un entier choisit de façon aléatoire et uniforme entre 1 et N, avec N un entier assez grand, e.g. N=n^4.

- Modifier vos programmes pour générer des graphes aléatoires pondérés. Comme précédemment, vous êtes libre de choisir la structure de données que vous souhaitez.

Test de performances

- Écrire une classe java permettant de calculer pour un graphe donnée un arbre recouvrant en utilisant respectivement l'algorithme de Prim et de Kruskal.

```
public interface MSTTools {
    public ... runPrim(Graph g);
    public ... runKruskal(Graph g);
}
```

- En jouant sur la taille n du graphe généré et pour différentes valeurs du paramêtre p (le paramêtre p détermine le nombre d'arêtes du graphe), observez comment vos algorithmes se comporte en terme de temps de calcul et de mémoire utilisée :
    - En utilisant l'outil gnuplot par exemple, vous pourrez dessiner des courbes qui rendent compte de vos expérimentations puis les interpréter plus facilement !

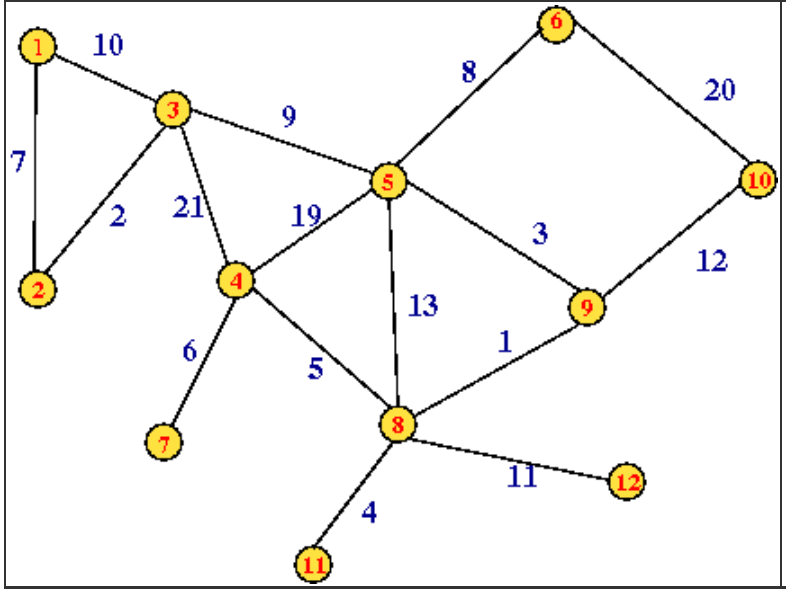
Un peu de compétition

- Écrire une classe java permettant de transformer votre représentation de graphe sous forme texte et vice versa. Plus précisément, nous adoptons le format texte décrit dans ce qui suit. Pour éviter les duplicats, une arête est représentée une seule fois pour une seule extrémité):

```
public abstract class GraphTools {
```

```
    public static void toTextFormat(Graph
g, String file)
                    throws Exception;
    public static Graph
loadGraphFromFile(String file)
                    throws Exception;
}
```

Format texte à utiliser
... sommet_i : voisin_1 poids_1 voisins_2 poids_2 ... ...

Exemple de graphe	Format texte correspondant
	1 2 7 3 10 2 3 2 3 4 21 5 9 4 5 19 7 6 8 5 5 6 8 9 3 8 13 6 10 20 8 9 1 11 4 12 11 9 10 12

- Le but maintenant est de calculer un MST le plus efficacement possible et de mettre vos algorithmes en compétitions. Échangez vos graphes avec les voisins et comparez les performances de vos deux algorithmes: Prim et Kruskal précédemment implémentés.
    - Vous pouvez vous amuser à échanger vos graphes au format texte --ou vos générateurs java-- et à faire des moyennes sur les temps de calcul et les tailles de vos graphes.

- Pour gagner vous avez le droit maintenant d'utiliser n'importe quelle technique même non abordée en cours (pour l'algo de MST en lui même ou pour la représentation du graphe) ! La seule règle à respecter est de ne pas bidouiller les graphes générés et de respecter le modèle Erdos-Renyi : le but est de ganger en moyenne et non sur des instances particulières !