Développement d'une application de valorisation de données environnementales ouvertes

Partie R2.01 : Développement orienté objets Semaines 19 à 23

I. Borne - L. Naert

Mai-Juin 2022

Le projet *Développement d'une application de valorisation de données environnementales ouvertes*, proposé par le Parc Naturel Régional (PNR) du Morbihan, rassemble 4 SAÉ sur 6 (S2.01, S2.02, S2.04 et S2.05). L'objectif de ce projet est de fournir le 24 juin au PNR une application permettant la saisie, le traitement et l'affichage de données animalières récoltées par le PNR.

Rôle des différentes ressources de programmation dans la SAÉ

- *Exploitation de base de données* (R2.06) : conception d'une base adaptée aux données existantes fournies par le PNR (période 3), requêtage et définition de vues pertinentes (période 4).
- *Développement orienté objets* (R2.01) : modélisation de la base de données sous forme d'application orientée objet et implantation des traitements mathématiques (statistiques et graphes) en Java (période 4).
- *Développement d'applications avec IHM* (R2.02) : gestion de l'interface graphique pour la saisie de nouvelles données, l'affichage des différents résultats de traitement et le requêtage de la base MySQL depuis Java grâce à l'API JDBC (période 4).
- *Qualité de développement* (R2.03) : travail sur les tests unitaires, l'organisation des sources, la nature du livrable final (période 4).

Ce document détaille le travail à réaliser pour ce projet dans le cadre de la ressource R2.01. Les séances de TD/TP de R2.01 des semaines 19 à 23 seront dédiées au travail de SAÉ décrit dans ce document.

1 Architecture Modèle-Vue-Contrôleur (MVC)

L'application finale suivra une architecture logicielle "Modèle-Vue-Contrôleur" (voir cours R2.02). Cette architecture permet de définir trois modules relativement indépendants :

- Le **modèle** qui décrit le modèle de données choisi ainsi que les traitements (mathématiques par exemple) à effectuer sur les données ;
- La **vue** qui gère les éléments visibles de l'interface graphique (graphes, tableaux, boutons...) et la logique permettant d'afficher ces éléments ;
- Le **contrôleur** qui traite les actions de l'utilisateur en faisant le lien entre modèle et vue (mise à jour d'un affichage de la vue, ajout de données dans le modèle...).

Les modules "vue" et "contrôleur" seront étudiés dans la ressource R2.02 tandis que le module "modèle" sera défini en R2.01.

2 Organisation du code

L'architecture "Modèle-Vue-Contrôleur" devra se retrouver dans l'organisation de vos répertoires pour ce projet. Ainsi, votre dossier `src` doit contenir trois sous-dossiers : `vue`, `modele` et `controleur` correspondant aux trois packages de l'architecture MVC. Dans la suite, nous détaillerons le contenu du package `modele`.

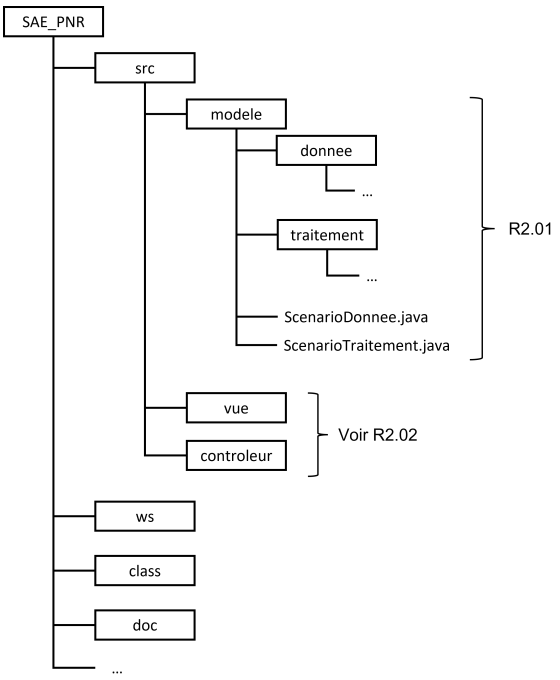

Figure 1: Arborescence de l'application

3 De la base de données MySQL à une architecture orientée objet - Package `modele.donnee`

Pendant les semaines 19 et 20, vous transformerez votre base de données en code Java en utilisant une architecture orientée objets. Il s'agit ici de stocker vos données, non plus dans une base MySQL, mais sous forme d'objets Java de façon à pouvoir exécuter des traitements sur ces derniers. La figure 2 présente le diagramme de classe UML du package `donnee`. Ce diagramme est basé sur une version simplifiée du modèle de données défini en R2.06.

3.1 Énumération

Les champs avec un domaine prédéfini réduits (par exemple le nom de l'espèce pour les chouettes qui peut prendre 3 valeurs : "Effraie", "Chevêche" et "Hulotte") ont été représentés sous forme de types `Enum`. Une énumération en Java permet de lister le nombre fini de valeurs que peut prendre une variable. Une énumération est créée dans un fichier séparé, comme une classe. Par exemple, le fichier `EspeceChouette` qui liste les différentes espèces de chouette contiendra, en plus de l'en-tête approprié, le code :

```
public enum EspeceChouette {
    EFFRAIE,
    CHEVECHE,
    HULOTTE;
}
```

Il est ensuite possible de déclarer et d'initialiser une variable de type `EspeceChouette` de la façon suivante (exemple pour une chouette effraie) :

```
EspeceChouette espece = EspeceChouette.EFFRAIE ;
```

3.2 Description des classes

- `Lieu` : Un lieu est décrit par deux coordonnées (x et y) en Lambert93.
- `Observateur` : un observateur comprend un identifiant unique, un nom et un prénom (l'un ou l'autre peut être une chaine vide).
- `Observation` : `Observation` est une classe abstraite décrite par un id unique, une date, une heure, une liste d'observateurs et un lieu d'observation. Dans la partie R2.01 de ce projet, toutes les dates sont de type `java.sql.Date` et les heures de type `java.sql.Time` même quand cela n'est pas précisé sur le diagramme. La méthode *`especeObs()`* est une méthode abstraite qui renvoie une valeur du type enuméré `EspeceObservee`. Par exemple, `especeObs()` appliqué à un objet de type `ObsBatracien` renverra `EspeceObservee.BATRACIEN`. La méthode `ajouteObservateur (o)` ajoute `o` à l'`ArrayList` si `o` n'est pas déjà dans la liste `lesObservateurs`. De la même façon, la méthode `retireObservateur` retire `o` de la liste si `o` est dans `lesObservateurs`.

- `ObsBatracien` : Une observation de batracien est un sous-type d'`Observation` avec, en plus, un attribut énuméré précisant l'espèce précise du batracien et quatre entiers indiquant le résultat de l'observation. Le constructeur de `ObsBatracien` prend en paramètre un tableau d'entiers `resObs` de 4 cases contenant le nombre d'adultes (`resObs[0]`), d'amplexus (`resObs[1]`), de têtards (`resObs[2]`) et de pontes (`resObs[3]`).
- `ObsHippocampe` : Une observation d'Hippocampe est également un sous-type d'`Observation`. Dans le constructeur, l'attribut `estGestant` est mis par défaut à `false`. Attention, le setter `setEstGestant ()` est un peu particulier : chez les hippocampes, seuls les mâles peuvent être gestants. Si le sexe de l'hippocampe est `MALE`, le setter se comporte donc normalement. Par contre, un message d'erreur doit être envoyé si le paramètre de `setEstGestant` est `true` alors que le sexe de l'hippocampe est `FEMELLE`. Dans ce cas, l'état de `estGestant` ne doit pas être modifié. Si le sexe est `INCONNU` et que le paramètre est `true` : l'attribut `estGestant` doit être mis à `true` et le sexe doit être modifié pour `MALE`.
- `ObsGCI` : une observation de gravelot consiste en un contenu observé (œuf, poussin...) dans une certaine quantité (par exemple : observation de 2 œufs).
- `NidGCI` : Un nid de gravelot est décrit par un identifiant unique, un nombre d'envol(s) (égal à 0 à la création du nid et mis à jour grâce au setters), le nom de la plage sur laquelle se trouve le nid et la liste des observations concernant le nid (`ArrayList<ObsGCI>` vide à la création du nid). `dateDebutObs()` (resp. `dateFinObs()`) renvoie la date de la première (resp. dernière) observation du nid. `NidGCI` implémente l'interface `IObs<ObsGCI>` qui décrit des méthodes permettent d'ajouter une ou plusieurs observations à la liste, de retirer l'ensemble ou l'une des observations. `nbObs()` donne le nombre d'observations du nid. `retireObs()` renvoie `false` si l'observation n'était pas dans la liste et n'a donc pas pu être retirée.
- `ObsLoutre` : les observations pour les loutres consistent en la visite de lieux précis pour y détecter des épreintes. Si le lieu n'est pas visité, l'indice sera à `NON_PROSPECTION`. Sinon, il sera `POSITIF` ou `NEGATIF` en fonction de la présence ou de l'absence de l'épreinte.
- `ObsChouette` : l'observation d'une chouette peut-être réalisée soit visuellement, soit par la détection de sons caractéristiques, soit par les deux.
- `Chouette` : Une chouette est décrite par son identifiant (une chaîne de caractère), son sexe, son espèce et les observations de cette chouette (`ArrayList<ObsChouette>` vide à la création de l'objet). Chouette implémente l'interface `IObs<ObsChouette>` qui décrit des méthodes permettent d'ajouter une ou plusieurs observations à la liste, de retirer l'ensemble ou l'une des observations. `nbObs()` donne le nombre d'observations de la chouette.

3.3 Setter/Getter

Un setter et un getter doivent être définis pour chaque attribut de chaque classe. Pour des raisons de lisibilités, ces méthodes ne sont pas présentes dans le diagramme de la figure 2. Par

défaut, les getters et setters n'implémenteront pas de copie défensive à moins que vous n'en ressentiez fortement le besoin. Les setters prennent en paramètre un argument du même type que l'attribut à modifier. Ne modifiez la valeur de l'attribut avec le setter que si le paramètre est valide.

3.4 Travail demandé (semaine 19 et 20)

Pendant les semaines 19 à 20, vous implémenterez les classes et les énumérations du diagramme de la figure 2. Les paramètres des méthodes doivent être testés et un message d'erreur envoyé en cas de paramètre non valide. N'hésitez pas à lancer un `IllegalArgumentException` si le paramètre d'un constructeur est invalide et que cela empêche l'initialisation d'un objet.

Vous testerez votre code **progressivement**[1] dans un fichier `ScenarioDonnee.java` (dans le dossier `modele` mais hors du dossier `donnee`) dans lequel vous créerez, manipulerez et afficherez au moins une instance de chacune des classes.

Ce premier travail (sources et javadoc générée) est à rendre sur Moodle pour le **dimanche 22 mai avant 23h59**.

Conseil : N'hésitez pas à avancer cette première partie rapidement en utilisant les créneaux de SAE dédiés.

[1] Bonne pratique : dès qu'une classe a été écrite, elle doit être testée !

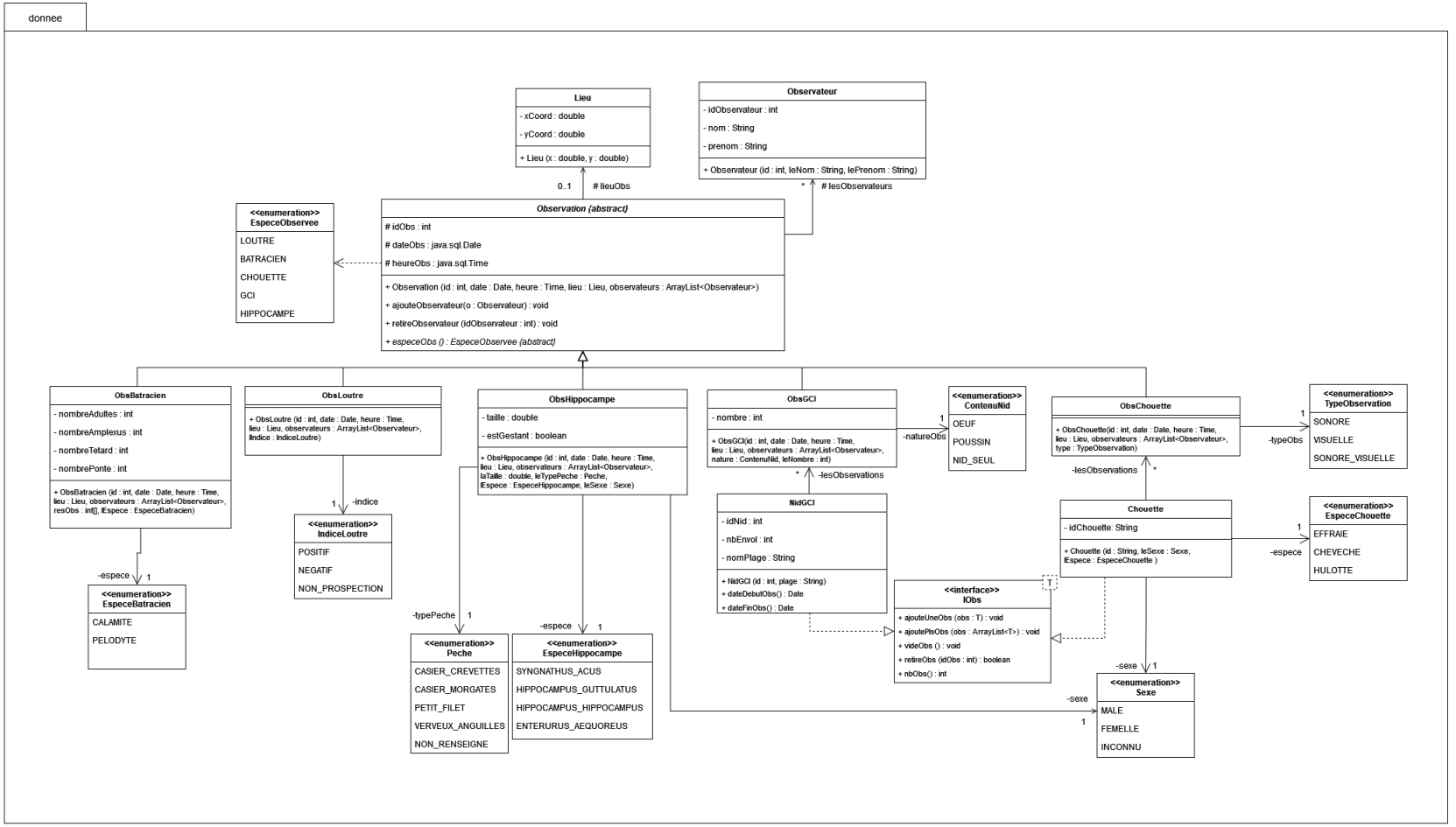

Figure 2: Diagramme de classe du package `donnee`

4 Ajout de traitements mathématiques - Package `modele.traitement`

L'accumulation de données n'a aucun intérêt si les données ne sont pas étudiées. Pour ajouter du sens aux données fournies par le PNR, il peut être intéressant d'appliquer les connaissances et savoirs-faires acquis dans les ressources *Graphes* (R2.07) et *Outils numériques pour les statistiques descriptives* (R2.08) à nos données de façon à extraire des indicateurs statistiques (moyennes, médiane, écart-type...) et des représentations graphiques (dessins de graphes, diagrammes, histogrammes...) pertinents. Des métriques statistiques, si choisies avec soin, pourraient par exemple permettre au PNR de valoriser un programme de sauvegarde des gravelots ou un protocole d'acquisition de données sonores de crapaud calamites.

La figure 3 décrit le contenu du package `traitement` permettant de faire des calculs statistiques et des graphes à partir du modèle de données développé dans la partie précédente.

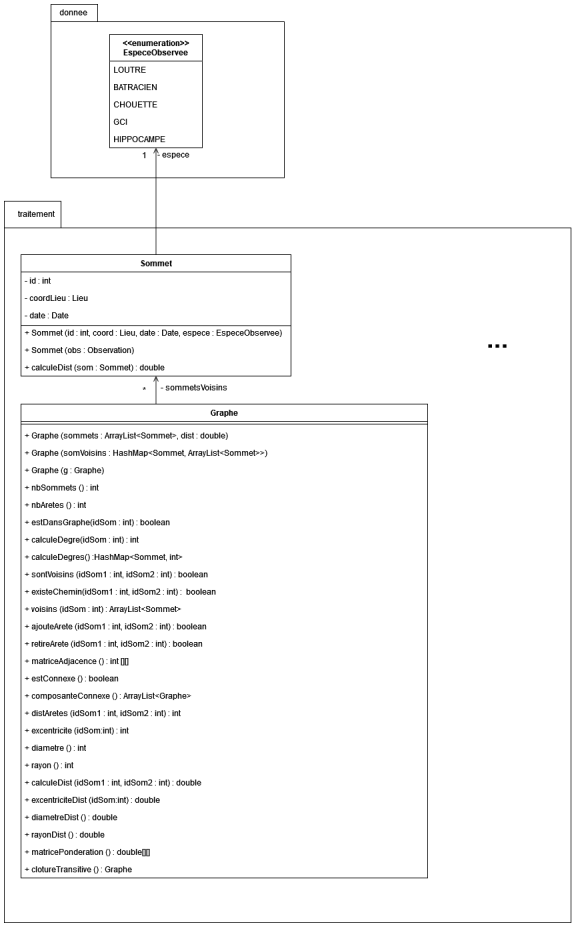

Figure 3: Diagramme de classe du package `traitement`.

4.1 Graphe (non orienté)

Un graphe est décrit par un unique attribut représentant le réseau composé des sommets et de leurs voisins directs[2]. Cet attribut est un dictionnaire (`HashMap`) dont les clefs sont chacun des sommets du graphe et les valeurs la liste des sommets reliés à chaque sommet (i.e. liste des voisins). Ainsi, le type en java sera : `HashMap<Sommet, ArrayList<Sommet>>`.
Prenons le graphe de la figure 4. En considérant que les sommets sont désignés exclusivement par un numéro, voici la structure de la `HashMap` correspondante : `[(1,(3)), (2, (3)), (3,(1,2)),(4,())]`.

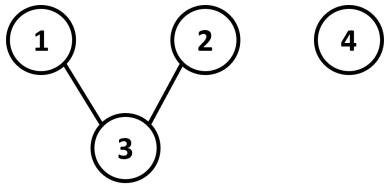

Figure 4: Exemple de graphe

4.1.1 Sommet

Un sommet désigne ici une `Observation` très simplifiée. Il est décrit par un identifiant unique (celui de l'observation), le lieu et la date de l'observation ainsi que l'animal observé (du type énuméré `EspeceObservee` définie dans le package `donnees`). Un sommet peut être créé en donnant des valeurs à chacun des attributs avec les paramètres correspondants (premier constructeur du diagramme) ou directement en extrayant les informations pertinentes d'une `Observation` (deuxième constructeur). Vous pouvez définir les setters et getters de `Sommet` en fonction de vos besoins.

La méthode `calculeDist (som : Sommet)` renvoie la distance (`double`) entre le sommet courant et le paramètre `som`. Pour calculer cette distance, on utilise la localisation (attribut `coordLieu`) des deux sommets considérés.

4.1.2 Méthodes de la classe Graphe

Remarques :

1. Dans certaines méthodes où un sommet est désigné par un identifiant, il peut être nécessaire de tester la présence du sommet dans le graphe et de renvoyer un message d'erreur dans le cas contraire.

2. La partie suivante donne un exemple détaillé avec les résultats attendus pour la plupart des méthodes décrites dans cette partie.

[2]On appelle voisins deux sommets reliés par une arête.

3. Il peut être intéressant d'implémenter une méthode `toString()` dans `Sommet` (affiche les attributs d'un sommet). De la même façon, un `toString()` dans `Graphe` renvoyant l'état de la `HashMap` du graphe courant peut être utile pour tester les méthodes de `Graphe`.

Méthodes à implémenter obligatoirement :

- `Graphe (sommets : ArrayList<Sommet>, dist : double)` : Constructeur. Prend en paramètre la liste des sommets appartenant aux graphes. Deux sommets sont reliés par une arête (i.e. sont "voisins") si la distance entre eux est inférieure ou égale à `dist`.
- `Graphe (somVoisins : HashMap<Sommet, ArrayList<Sommet>>)` : constructeur.
- `Graphe (g : Graphe)` : Constructeur par copie.
- `nbSommets ()` : nombre de sommets du graphe.
- `nbAretes ()` : nombre d'arêtes du graphe (attention à ne pas les compter plusieurs fois !).
- `estDansGraphe(idSom : int)` : `true` si le sommet désigné par `idSom` est un des sommets du graphe, `false` sinon.
- `calculeDegre(idSom: int)` : calcule le degré (nombre de voisins) du sommet désigné par `idSom`.
- `calculeDegres()` : retourne une `HashMap<Sommet, Integer>` dont les clefs sont les sommets et les valeurs le degré de chaque sommet.
- `sontVoisins (idSom1 : int, idSom2 : int)` : `true` si les sommets sont voisins, `false` sinon.
- `existeChemin (idSom1 : int, idSom2 : int)` : `true` si on peut aller du sommet d'identifiant `idSom1` au sommet d'identifiant `idSom2` en passant par des arêtes successives, `false` sinon.
- `voisins (idSom: int)` : retourne la liste des voisins du sommet désigné par `idSom`.
- `ajouteArete (idSom1 : int, idSom2 : int)` : si `idSom1` et `idSom2` représentent des sommets du graphe, rajoute une arête entre les deux et renvoie `true`, `false` sinon.
- `retireArete (idSom1 : int, idSom2 : int)` : si `idSom1` et `idSom2` représentent des sommets du graphe et sont reliés par une arête, retire l'arête entre les deux et renvoie `true`, `false` sinon.
- `matriceAdjacence ()` : retourne une matrice (`int[][]`) `res` de dimensions (`nbSommet`) et (`nbSommets + 1`) où la première ligne (`res[i][0]` avec `i` allant de `0` à `nbSommets -1`) reprend les identifiants des sommets dans l'ordre croissant. Les autres lignes composent la matrice d'adjacence elle-même : un `1` signifie que deux sommets sont voisins directs, un `0` qu'ils ne le sont pas.

- `estConnexe ()` : `true` si le graphe est connexe (i.e. tout sommet peut être relié à tout autre sommet par une arête ou une suite d'arêtes ("graphe en 1 seul morceau")), `false` sinon.
- `composanteConnexe ()` : renvoie la liste des graphes connexes composant le graphe courant.
- `distAretes (idSom1 : int, idSom2 : int)` : s'il existe un chemin entre `idSom1` et `idSom2`, renvoie le nombre d'arêtes entre eux (minimal si plusieurs chemins possibles), `0` s'il n'existe pas de chemin mais que les deux sommets sont dans le graphe, `-1` sinon.
- `excentricite (idSom: int)` : retourne le nombre maximal d'arêtes du chemin entre le sommet en paramètre et les autres sommets du graphe. Si l'excentricité est calculée sur un graphe non connexe, retourner -1.
- `diametre ()` : retourne le maximum des excentricités. Si le diamètre est calculé sur un graphe non connexe, retourner -1.
- `rayon ()` : retourne le minimum des excentricités. Si le rayon est calculé sur un graphe non connexe, retourner -1.

Méthodes facultatives :

- `calculeDist (idSom1 : int, idSom2 : int)` : s'il existe un chemin entre `idSom1` et `idSom2`, calcule la somme (minimales) des distances entre eux, `0` s'il n'existe pas de chemin mais que les deux sommets sont dans le graphe, `-1` sinon.
- `excentriciteDist (idSom: int)` : retourne la distance maximale du chemin entre le sommet en paramètre et les autres sommets du graphe. Si l'excentricité est calculée sur un graphe non connexe, retourner -1.
- `diametreDist ()` : retourne le maximum des excentricités de distance. Si le diamètre est calculé sur un graphe non connexe, retourner -1.
- `rayonDist ()` : retourne le minimum des excentricités. Si le rayon est calculé sur un graphe non connexe, retourner -1.
- `matricePonderation ()` : retourne une matrice (`int[][]`) `res` de dimensions (`nbSommet`) et (`nbSommets + 1`) où la première ligne (`res[i][0]` avec `i` allant de `0` à `nbSommets -1`) reprend les identifiants des sommets dans l'ordre croissant. Les autres lignes composent la matrice de pondération elle-même : les `1` de la matrice **d'adjacence** sont remplacés par les distances entre les sommets voisins, un `0` désignent deux sommets non voisins.
- `clotureTransitive ()` : renvoie un nouveau graphe qui est la clôture transitive du graphe courant (cf. cours Graphe). Dès qu'un chemin existe entre deux sommets non voisins, ces sommets deviennent voisins (ajout d'une arête entre les deux sommets).

4.1.3 Exemple détaillé

Prenons l'exemple des sommets de la figure 5 (gauche). Ces sommets sont espacés d'une certaine distance, calculable grâce à leur coordonnées Lambert. Le schéma au centre de la figure 5 montre le graphe construit à partir de ces sommets et en considérant une distance maximale `dist` de 4 entre les sommets. Les segments grisés en pointillées correspondent à des distances supérieures à `dist`. Les segments noirs représentent les distances inférieures ou égales à 4 qui donneront les arêtes du graphe. Enfin, le schéma de droite donne le graphe final (avec la valeur des distances entre les sommets).

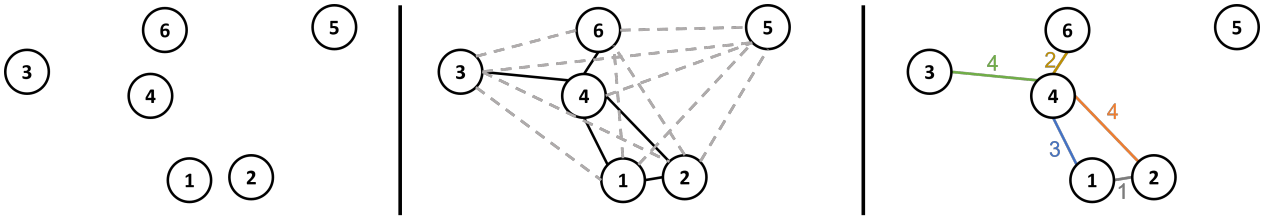

Figure 5: Gauche : Ensemble de sommets localisés. Centre : distance supérieures (en pointillés gris) et inférieures ou égales à `dist` (trait continu noir). Droite : Dessin du graphe résultant en précisant les distances.

En considérant que les sommets sont désignés exclusivement par un numéro, voici la structure de la `HashMap` correspondante :
`[(1,(2,4)), (2, (1,4)), (3,(4)),(4,(1,2,3,6)),(5,()),(6,(4))]`
La variable pointant vers ce graphe est nommée `grapheEx` dans cet exemple.

`grapheEx.nbSommets()` vaut 6
`grapheEx.nbAretes()` vaut 5
`grapheEx.estDansGraphe(7)` vaut `False`
`grapheEx.estDansGraphe(5)` vaut `True`
`grapheEx.calculeDegre(3)` vaut 1
`grapheEx.calculeDegre(4)` vaut 4
`grapheEx.calculeDegre(5)` vaut 0
`grapheEx.calculeDegres()` vaut `[(1,2), (2, 2), (3,1),(4,4),(5,0),(6,1)]`
`grapheEx.sontVoisins(3,4)` vaut `true`
`grapheEx.sontVoisins(6,2)` vaut `false`
`grapheEx.existeChemin(5,6)` vaut `false`
`grapheEx.existeChemin(6,2)` vaut `true`
`grapheEx.matriceAdjacence()` vaut

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

`grapheEx.estConnexe()` vaut `false` (à cause du sommet 5 isolé)
`grapheEx.composanteConnexe()` retournera un ArrayList de deux graphes. Le premier constitué des sommets (1,2,3,4,6) et le deuxième constitué du sommet (5).
`grapheEx.distAretes(6,2)` vaut 2
`grapheEx.excentricite(6)` vaut -1 (graphe non connexe)
`grapheEx.matricePonderation()` vaut

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 1 & 0 & 3 & 0 & 0 \\ 1 & 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 \\ 3 & 4 & 4 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 \end{pmatrix}$$

`grapheEx.clotureTransitive()` retournera le graphe de la figure 6.

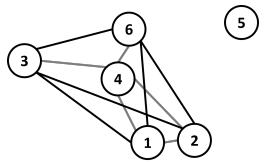

Figure 6: Clôture transitive de `grapheEx` (arêtes existantes en gris, nouvelles arêtes en noir).

On ne considère maintenant que le graphe `grapheEx2` constitué par les sommets (1,2,3,4,6).
`grapheEx2.estConnexe()` vaut `true`
`grapheEx2.excentricite(6)` vaut 2
`grapheEx2.excentricite(4)` vaut 1
`grapheEx2.diametre()` vaut 2
`grapheEx2.rayon()` vaut 1
`grapheEx2.calculeDist(6,2)` vaut 6
`grapheEx2.calculeDist(3,2)` vaut 8
`grapheEx2.excentriciteDist(6)` vaut 6
`grapheEx2.diametreDist()` vaut 8
`grapheEx2.rayonDist()` vaut 4

4.2 Statistiques descriptives

Pour ce qui est des indicateurs et des représentations statistiques, vous avez carte blanche ! Utilisez les connaissances acquises en R2.08 et les exemples de compte-rendus rédigés par le PNR pour valoriser les données par des métriques statistiques pertinentes. N'hésitez pas à créer les classes et les outils dont vous avez besoin.

4.3 Travail demandé (semaine 21 et 23)

Pendant les semaines 21 à 23 vous implémenterez les classes de la figure 3 et quelques indicateurs statistiques.

Vous testerez votre code dans un fichier `ScenarioTraitement.java` dans lequel vous montrerez le bon fonctionnement des méthodes développées.

Ce second travail (sources et javadoc générée) est à rendre sur Moodle pour le **dimanche 12 juin avant 23h59**.

5 Pour aller plus loin

Des classes et attributs ont été retirés par rapport au diagramme de classe de BDD, si vous estimez que certaines classes ou certains attributs sont nécessaires pour vos traitements, n'hésitez pas à les ajouter. Vous avez ensuite jusqu'au 24 juin pour enrichir votre application avec de nouvelles fonctionnalités. A vous de jouer !

Mais attention, avant d'aller plus loin, vérifiez que les fonctionnalités principales de votre application sont bien en place (saisie de nouvelles données dans la base et affichage de vues simples).