Rapport final de projet

SimCitree
Simulateur de forêt

Réalisé par

Mateusz Birembaut

Florian Collé

Walter Deneuville

Matthieu Giaccaglia

Sous la direction de

Marc Joannides

Pour l'obtention du DUT Informatique

Année universitaire 2020-2021

Remerciements

Nous tenons à remercier toutes les personnes qui ont contribué au succès de notre projet tuteuré et qui nous ont aidés lors de la rédaction de ce rapport.

Tout d'abord, nous adressons nos remerciements à notre professeur, M. Marc Joannides qui nous a beaucoup aidés dans l’élaboration de notre projet.

Enfin, nous tenons à remercier toutes les personnes qui nous ont conseillés et relus lors de la rédaction de ce rapport.

Sommaire

Glossaire

In silico: C’est le fait qu’on fasse une recherche ou un essai en utilisant des calculs informatisés ou de modèles informatiques.

Modèle individu-centrés: C’est le fait de modifier les paramètres d’un individu et de voir les effets sur la population entière.

Géré en tore: Géré en tore dans notre cas veut dire que les bords se rejoignent.

Introduction

Dans ce projet, nous cherchons à étudier des écosystèmes à l'aide de simulations numériques, dans notre cas celui d'une forêt. On étudie ses arbres ainsi que leurs interactions en fonction d'un nombre restreint de paramètres, afin d'en observer différents comportements et d'en extraire l'importance que joue chaque paramètre. L'approche de la solution se fera en menant des expériences in silico sur un modèle individu-centré.

En effet, la problématique principale outre la réalisation du projet va-t-être de s'occuper de l'optimisation de l'algorithme. De fait, lorsque la forêt comporte plusieurs milliers d'arbres, le programme risque d'avoir du mal à s'exécuter rapidement si l'algorithme principal n'est pas fait de manière intelligente.

Au final, le projet qui nous a été confié consiste donc à proposer une solution à la réalisation de cette simulation de forêt.

Pour ce faire, nous commencerons par faire une analyse du cahier des charges et donc les besoins fonctions et les spécifications techniques, ensuite nous établirons le rapport technique avec la conception et la réalisation du projet, puis l'étude des résultats. Enfin, nous parlerons de l'organisation du projet avec la démarche personnelle, la planification des tâches ainsi qu'un bilan critique par rapport au cahier des charges.

1. Cahier des charges

Dans cette partie du rapport, nous allons développer le cahier des charges du projet et son analyse, en commençant tout d'abord par une analyse du sujet et de son contexte, ainsi que d'une analyse des besoins fonctionnels, ensuite d'une analyse des besoins non-fonctionnels et pour finir par les spécifications techniques.

1.1 Analyse du sujet et de son contexte

Tout d'abord, nous verrons l'analyse du sujet, pour voir plus en détail en quoi consiste le projet. Nous continuerons avec l'analyse de l'existant, consistant à chercher ce qui a déjà été fait sur le sujet. Enfin, nous finirons avec l'analyse du contexte dans lequel nous établirons comment et pourquoi nous avons choisi les technologies utilisées dans notre projet.

1.1.1 *Analyse du sujet*

Notre sujet traite de la bio-informatique où nous devons simuler le vivant avec des outils informatiques. Cela est au goût du jour du fait que l'écologie est récemment devenue l'une des plus grandes problématiques de l'humanité.

1.1.2 *Analyse de l'existant*

Il existe déjà des projets qui accomplissent l'analyse des interactions de la population d'une forêt à l'aide d'un algorithme. Beaucoup même, vont plus loin que ce qu'il nous est demandé. Mais nous ne nous sommes pas basés sur un projet déjà existant, du fait que c'était une simulation simplifiée et que notre tuteur nous avait déjà montré une version du programme, la sienne, comme exemple qui nous a donc servi de point d'appui sur le résultat qui était attendu de nous.

1.1.3 *Analyse de l'environnement*

L'algorithme et par extension le programme entier est à but scientifique ce qui implique que l'utilisateur ait des connaissances dans le domaine de l'agroforesterie. Cet environnement a influencé nos choix techniques. Par exemple, nous n'avons pas mis de restrictions dans les valeurs pouvant être mises dans les paramètres, car nous estimons que l'utilisateur est conscient des valeurs à définir.

L'application pourra fonctionner sur MacOS, Linux et Windows en ayant installé la dernière version de Java.

1.2 Analyse des besoins fonctionnels

L'utilisateur pourra insérer des données qui permettront de faire fonctionner la simulation en fonction de ses besoins.
Les paramètres à définir sont :

- taux de naissance
- taux de mort
- taux de mort par compétition
- rayon de dispersion
- rayon de compétition
- nombre d'arbre initial

1.3 Analyse des besoins non-fonctionnels

Une des contraintes techniques qui nous étaient imposées était le modèle de l'écosystème, c'est-à-dire que la forêt est assimilée au carré [0;1] x [0;1] géré en tore*.

Nous avions certaines spécifications qui nous étaient imposées, principalement tout ce qui relève de calculs mathématiques (calcul de l'intensité de la compétition, calcul des tirages aléatoires). Ces calculs sont intégrés directement dans l'algorithme, l'utilisateur a juste à rentrer la valeur de la variable et le reste est pris en compte par l'algorithme.

2. Rapport technique

Voici la réalisation du projet en soi, nous allons montrer comment le code à été créé pour le projet. Nous commencerons par les choix pris par l'équipe en amont, puis nous verrons ce qui a été programmé.

2.1 Conception

Lorsqu'il y a une majuscule à arbre, forêt ou voisin, c'est que nous parlons de l'objet

Le projet consiste à créer une forêt sur un environnement limité. Le cahier des charges stipule que les individus à créer et à implémenter ont été limités aux arbres d'un seul type et à la forêt qui représente le terrain.

Le simulateur de forêt nécessite trois classes :

- Forêt
- Arbre
- Voisin

Nous avons dû revoir plusieurs fois la classe Voisin, car elle posait des problèmes. Au départ, elle possédait un objet Arbre (Arbre qui était le voisin) ainsi qu'un objet Double pour la distance, afin d'éviter de la calculer dès que besoin). Ainsi, dès qu'un voisin était trouvé, on ajoutait dans la liste de voisinage des deux arbres respectifs un objet voisin. Le problème était qu'il fallait créer deux objets Voisin et cela posait des problèmes au niveau de la mémoire que la machine alloue pour le programme. Cela montre aussi que Java n'est pas le meilleur langage pour la réalisation du projet si l'on souhaite utiliser le programme sur des simples ordinateurs.

Nous avons donc mis dans la classe Voisin deux objets Arbre qui représente les deux arbres voisins. La distance est toujours présente. La classe Voisin est donc une structure en graphe. Mais à part pour régler le problème de mémoire, cette structure n'a pas été utilisée pour autre chose.

Voici le diagramme de classe qui découle de notre conception :

Diagramme de classes (annexe)

Lors du commencement du projet, nous possédions déjà un algorithme pour faire fonctionner la forêt, que voici :

jsp je crois que c le pseudocode de joannides la
mais on est pas parti d'un algo
on l'a ecrit
nn ?
ah ouais
ouais on est parti en vitesse

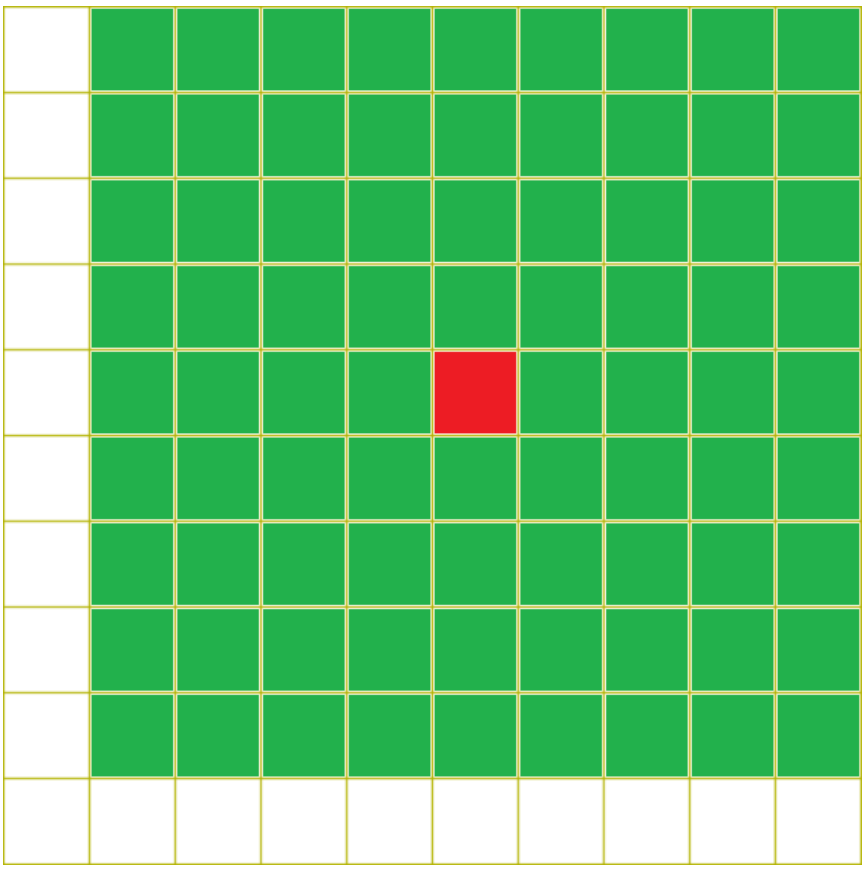
Il ne nous restait donc plus qu'à réfléchir sur comment optimiser le code, surtout lorsqu'un arbre né pour trouver rapidement ses voisins au lieu de regarder pour chaque arbre présent s'il l'était.

Pour cela, nous avons décidé de "couper" le terrain en fonction de la taille du rayon de compétition qu'on notera RC. Ce qui donne :

```
SI (RC > 0 && RC <= 0.4)
     ALORS découpage
```

La valeur 0.4 n'est pas une valeur prise au hasard. Admettons qu'un arbre apparaît en (0.5,0.5) et que le rayon de compétition est de 0.4. L'arbre apparaîtra dans la partie ou il y a tous les arbres qui sont entre 0.5 et < 0.6 en X et Y.

Le schéma ci-dessous nous montre où le programme va chercher les potentiels arbres voisins. Mais si le rayon était supérieur à 0.4, le découpage ne servirait à rien dans notre situation, car il chercherait dans tout le terrain.



Et plus le rayon de compétition est petit, plus le terrain sera découpé.

```
SI ( RC >= 0.1 && RC <= 0.4 )
     Alors découpage en 10*10
SINON SI ( RC >= 0.01 && RC < 0.1 )
     Alors découpage en 100*100
```

etc…

Bien évidemment, nous avons trouvé une meilleure façon de faire que voici :

```
divisionDuTableau <- 1
SI ( RC > 0 && RC <= 0.4 )
ALORS
RCTemp <- RC
TANT QUE ( RC <1 )
     ALORS
     divisionDuTableau <- divisionDuTableau * 10
     RCTemp <- RCTemp * 10
FIN TANT QUE
```

Puis on crée le terrain en fonction de la valeur de divisionDuTableau.

Pour l'interface graphique, nous avons deux scènes. Une pour rentrer les données, l'autre pour voir l'évolution de la simulation.

Scènes Graphique (Annexe)

2.2 Réalisation

Pour la réalisation du simulateur de forêt, nous avons choisi comme langage de programmation Java couplé avec JavaFX pour la partie graphique. Le choix se justifie du fait que nous connaissions mieux Java que Python par exemple, même si ce dernier restait un meilleur choix en termes de calcul scientifique.

2.2.1 Calcul Scientifique

Dans certaines fonctions, nous avons dû arrondir certaines valeurs, car les calculs avec des variables de type double ne renvoient pas la valeur exacte, mais à .0000001 près environ.

Choix de classe pour l'aléatoire :

```
import java.util.Random;
```

Nous avons utilisé la classe random pour simuler l'aléatoire car dans math.random nous pouvons apercevoir un pattern se former, cela fausserait les données de la simulation.

Choix d'un événement :

```
public void applyEvent(){
   double totB = this.tauxNaissance * list.size(); //total Chances Naissance
   double totM = this.tauxMort * list.size(); // total Chances Mort
   double totC = this.tauxIntensiteCTotal;

   double tot = totB+totM+totC;
    double rdm = random.nextDouble()*tot; // entre 0 et 1, il faut alors le rammener sur
le total
```

Le choix de l'événement est fait à partir d'une loi uniforme. On additionne les probabilités de naissance qu'on multiplie par le nombre d'arbres, pareil avec les probabilités de mort ainsi que les probabilités de mort par Compétition. On sépare les deux types de mort (Naturelle et Compétition) car leur tirage n'est pas le même. Les trois sorties possibles sont alors la naissance d'un arbre, la mort d'un arbre de cause naturelle et par compétition. On utilise alors la fonction random qui tire un nombre entre 0 et 1 qu'on ramène sur le total des probabilités. On voit quel événement est tiré.

```
   int indexArbreRandom = random.nextInt(list.size());

   if (rdm <= totB ) //jusqu'à totB,
       addFils(list.get(indexArbreRandom));

   else if(totB <= rdm && rdm <= totB+totM) //de totB au totB+totM
       deleteArbre(list.get(indexArbreRandom), indexArbreRandom);

   else//de totB+totM au total
       deathByCompetition();
}
```

Dans le cas d'une naissance et d'une mort naturelle, un arbre aléatoire est tiré. En effet, pas besoin d'augmenter les chances d'un arbre plus vieux, car dans sa vie, il sera plus testé qu'un arbre né ce qui lui donne plus de chance de donner naissance ou de mourir naturellement. Dans le cas d'une mort par compétition, on appelle une autre fonction qui s'occupe de vérifier les intensités de compétition de chaque arbre et de tirer un arbre en fonction de leur intensité de compétition.

Mort par compétition :

```
private void deathByCompetition() {
   System.out.println("Death By Competition");
   double tot = 0;int i = 0;
   double rdm = Math.random()* tauxIntensiteCTotal; // entre 0 et 1, il faut
alors le rammener sur le total
```

```
    for(Arbre a: list){
        tot += a.getIntensiteCompetition();
        if ( rdm < tot) { //jusqu'à totB,
            deleteArbre(list.get(i), i);
            return;
        }
        i++;
    }
}
```

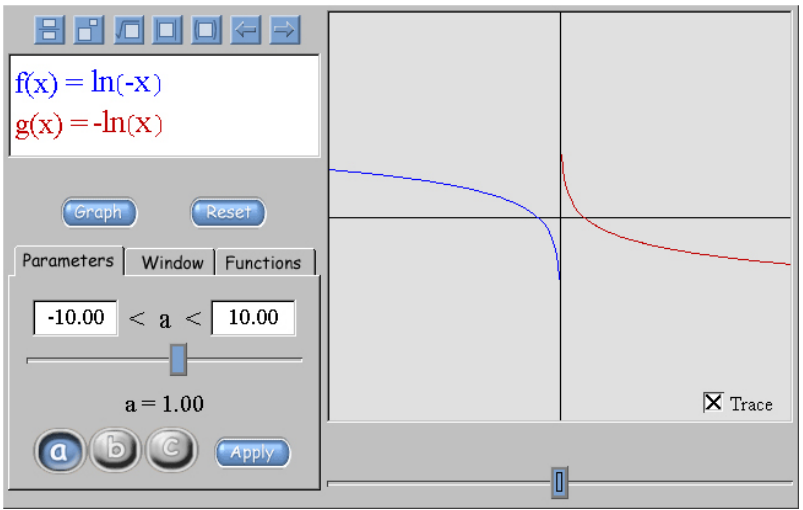
tot représente la valeur des intensités de compétition totale, au lieu de parcourir tous les arbres pour la créer en un temps et l'utiliser dans un autre, pour optimiser le code nous faisons les **deux à la fois**. **i** représente l'index d'un arbre. On réutilise la fonction Math.random() qu'on ramène sur le taux d'intensité total, cette valeur est stockée dans rdm.

Ensuite, nous parcourons la liste d'arbres, on rajoute à tot l'intensité de Compétition de l'arbre, on la compare à la valeur de random. On la compare jusqu'à ce qu'on trouve quel arbre est mort. On prend alors l'index de l'arbre courant stocké dans i et on supprime son arbre de la liste, sinon on incrémente i.

Durée du prochain événement :

```
public double getDureeNextEvent(){
    double event = -Math.log(random.nextFloat())
            / getTauxGlobal();
    return  event;
}
private double getTauxGlobal() {
    return ( (this.tauxNaissance+this.tauxMort) * list.size()) +
tauxIntensiteCTotal;
}
```

Le choix de durée du prochain événement se fait grâce à la fonction getDureeNextEvent() selon la formule mathématique $-\ln(x)/\sum(\lambda_n+\lambda_m+\lambda_c)$. L'évolution est exponentielle. Les lambdas représentent les taux de naissance, mort naturelle et d'intensité. La fonction getTauxGlobal() s'occupe de calculer tout cela.



On utilise que la partie de la courbe rouge en positif. Cela signifie que dès qu'un évènement arrive, il arrivera dans tous les cas à cause de la limite. Plus il y a

d'arbres, plus grand sera le dénominateur et donc la durée sera plus courte.

2.2.2 Optimisation

Nous avons expliqué comment le programme a été optimisé lors de la partie de conception. Donc après avoir décidé en combien de parties sera découper le terrain, il faut savoir comme il est simulé. Pour cela, nous avons créé une matrice de deux dimensions. Et dans chaque case, il y a une liste d'arbres qui sont dans les coordonnées de l'index. Par exemple dans List[5][5] il y a les arbres qui sont entre 0.5 et < 0.6 comme dit précédemment dans la conception.

Mais afin de manipuler facilement toutes ces listes, nous avons utilisé des ArrayList. Ce qui donne :

```
private final ArrayList<ArrayList<ArrayList<Arbre>>>
tableauDivision = new ArrayList<>();
private int division = 1;
```

Ainsi, on instancie le terrain avec ce morceau de code :

```
if (rayonCompetition > 0 && rayonCompetition <= 0.4) {
   double rayonCompetitionTemp = this.rayonCompetition;
   while (rayonCompetitionTemp < 1) {
       this.division *= 10;
       rayonCompetitionTemp *= 10;
```

```
    }
}

for (int i = 0; i<division;i++) {
    tableauDivision.add(new ArrayList<>());
    for (int j = 0; j < division; j++)
        tableauDivision.get(i).add(new ArrayList<>());
}
```

Puis lorsqu'un arbre est né, il faut faire cette vérification afin d'éviter de problème. Si division != 1 signifie qu'il y a eu un découpage du terrain :

```
list.add(arbreAdd);
if (division != 1) {
    checkVoisinFast(arbreAdd);
     tableauDivision.get((int) (coordonneeX * division)).get((int)
(coordonneeY * division)).add(arbreAdd);
}
```

Mettre (int) devant un double ou un calcul qui retourne un double permet de récupérer d'arrondir la valeur à l'unité inférieure. Cela permet de prévenir dans quelle case du tableau il faut ajouter l'arbre.

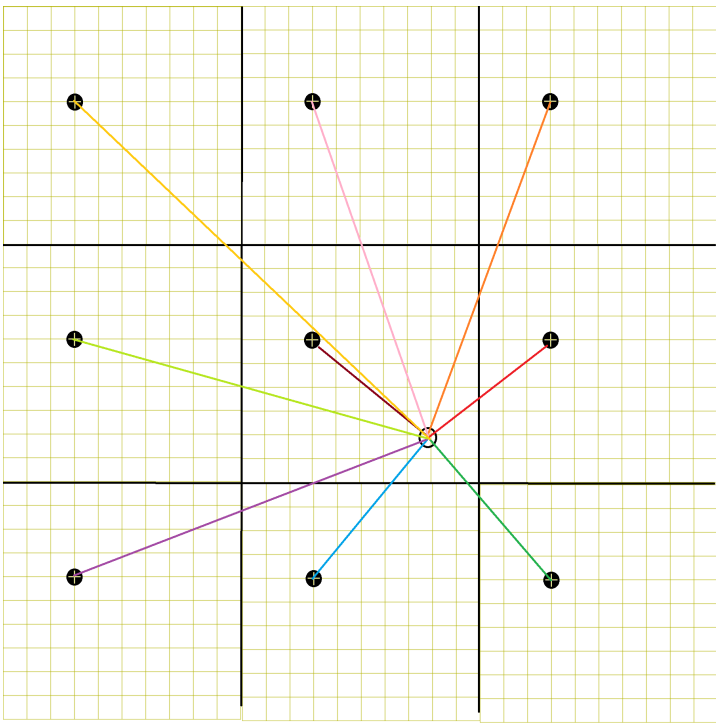
Et on fait de même pour la mort d'un arbre :

```
list.remove(arbreSupp);
if (division != 1)
    tableauDivision.get((int) (arbreSupp.getX()
*division)).get((int) (arbreSupp.getY()
*division)).remove(arbreSupp);
```

Dû à ce choix pour l'optimisation, nous avons dû créer deux fonctions pour chercher les arbres voisins à un arbre né.

- checkVoisinsFast() qui est optimisé pour un terrain découpé.
- checkVoisinsSlow() qui regarde bêtement si chaque arbre est voisin à l'arbre né.

De plus, du fait que le terrain soit géré en tore, il faut vérifier toutes les distances possibles qu'il y a entre deux arbres.



Comme on peut le voir dans cette représentation grossière, il y a 9 distances possibles au total, il peut y avoir plus mais nous sommes partis du principe que le scientifique ne mettra pas un rayon de compétition de plus de 1. Cependant, nous ne pouvons pas nous permettre de calculer ces 9 distances à chaque fois.

Dans le cas où le rayon de compétition est inférieur ou égal à 0.4, cela ne nous pose pas de problème car on arrive à savoir quelle est la distance choisie vu qu'il y en a qu'une seule de possible.

Pour cela il faut récupérer les valeurs de X (abscisse) minimum et maximum en fonction du X de l'arbre et du rayon de compétition, de même pour Y (ordonnée). Puis on instancie deux variable qui permettent de savoir s'il y a un débordement en X et en Y:

- - 1 S'il y a débordement en X ou Y négativement.
- 0 S'il n'y a pas de débordement.
- + 1 S'il y a débordement en X ou Y positivement.

Maintenant qu'on a la plage d'ordonnées ou il est possible d'avoir un voisin pour le nouvel arbre. Il faut donc boucler sur les coordonnées minimales jusqu'au coordonnées maximums. Et on en profite pour vérifier si les coordonnées sont inférieures à 0 ou supérieures à 1 afin de déterminer s'il y a un débordement. Puis on envoie cette valeur à une fonction qui calcule si la distance entre deux arbres permet de dire s'ils sont voisins.

Voici le code qui fait tout cela :

```
private void checkVoisinFast(Arbre arbreNouveau){
   double X = arbreNouveau.getX();
   double Y = arbreNouveau.getY();

   int xmin = (int) ((X - rayonCompetition) * division);
   int xmax = (int) ((X + rayonCompetition) * division);
   int ymin = (int) ((Y - rayonCompetition) * division);
   int ymax = (int) ((Y + rayonCompetition) * division);

   int debordX;
   int debordY;

   for (int i = xmin; i <= xmax; i ++) {

       debordX = 0;
       int indexX = i;
       if (indexX < 0) {
           indexX += division;
           debordX = 1;
       } else if (indexX >= division) {
           indexX -= division;
           debordX = -1;
       }

       for (int j = ymin; j < ymax; j ++) {

           debordY = 0;
```

```
            int indexY = j;

            if (indexY < 0) {
                indexY += division;
                debordY = 1;
            } else if (indexY >= division) {
                indexY -= division;
                debordY = -1;
            }

            for (Arbre arbreCourant : tableauDivision.get(indexX).get(indexY))
                checkInsideRayon(arbreNouveau, arbreCourant, debordX, debordY);
        }
    }
}
```

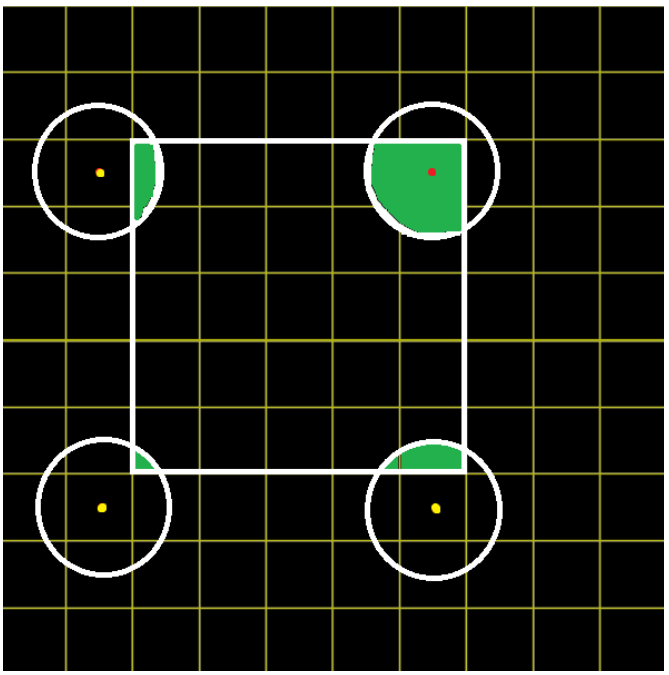
```
private void checkInsideRayon (Arbre arbre , Arbre arbreCourant , int
debordementX, int debordementY) {

    double distance = Math.hypot(( (arbre.getX() + debordementX) -
arbreCourant.getX()), ( (arbre.getY()+debordementY) - arbreCourant.getY() ));
    if (distance <= rayonCompetition)
        addEachOther(arbre,arbreCourant,distance);
}

private void addEachOther(Arbre arbre, Arbre arbreCourant, double distance) {
    double tauxCompetition = Math.round((((1-distance) / rayonCompetition) *
tauxIntensiteC) * 1000000000) / 1000000000d;
    Voisin voisin = new Voisin(arbre, arbreCourant, tauxCompetition);
    augmenterTauxIntensiteCTotal(tauxCompetition * 2);
}
```

Mais dans le cas inverse, il y a plusieurs distances qui sont possibles, nous avons donc dû chercher un moyen pour éviter de faire 9 calculs à chaque fois qu'on regarde si un arbre est voisin.

Nous vérifions si les coordonnées + le rayon de compétition dépasse du repère. Prenons ce cas par exemple : où il y a un débordement en x et y.



En rouge, l'arbre où l'on souhaite ajouter ses voisins, et autour, en blanc, son "cercle" de compétition. Le terrain est géré en tore, ces bords se rejoignent. Nous vérifions ensuite si un arbre se trouve dans l'un de ces cercles, si c'est le cas, on ajoute cet arbre en tant que voisins en prenant la distance la plus courte car il peut exister plusieurs distances entre deux arbres qui restent inférieurs au rayon de compétition. Puis on ajoute les deux arbres en tant que voisins.

```
private void checkVoisinsSlow(Arbre arbre) {
   double rayon = rayonCompetition;

   for ( Arbre arbreCourant : list) {
       if (arbreCourant != arbre) {

           if (arbre.getX() + rayonCompetition > 1 && arbre.getY() + rayonCompetition >
1)
               checkInsideRayonQuatre(arbre, arbreCourant, -1, -1);
```

```
private void checkInsideRayonQuatre(Arbre arbre , Arbre arbreCourant , int debordementX,
int debordementY) {

   double distance0 = Math.hypot(( arbre.getX()- arbreCourant.getX()), ((arbre.getY()
) - arbreCourant.getY() ));

   double distance1 = Math.hypot(( (arbre.getX() + debordementX) - arbreCourant.getX()),
((arbre.getY() + debordementY) - arbreCourant.getY() ));

   double distance2 = Math.hypot(( (arbre.getX() + debordementX) - arbreCourant.getX()),
(arbre.getY()- arbreCourant.getY() ));
```

```
   double distance3 = Math.hypot(( arbre.getX()- arbreCourant.getX()), ((arbre.getY() +
debordementY) - arbreCourant.getY() ));


   double distancePlusPetite = distance0;

   if (distance1 <= distancePlusPetite)
       distancePlusPetite = distance1;

   if (distance2 <= distancePlusPetite)
       distancePlusPetite = distance2;

   if (distance3 <= distancePlusPetite)
       distancePlusPetite = distance3;

   if(distancePlusPetite < rayonCompetition)
       addEachOther(arbre,arbreCourant,distancePlusPetite);
}
```

2.2.3 Interface Graphique

Afin de réaliser l'interface graphique, nous avons donc utilisé JavaFX. Nous possédons deux scènes, donc deux fichier FXML avec leur controller respectif. L'un récupérait les données saisies par l'utilisateur et l'autre permettant d'afficher l'évolution en temps réel de la forêt. Cette dernière utilise un diagramme en nuage de points couplé avec la bibliographie jfxutils de Gillius. Elle permet de zoomer sur le diagramme, ce qui est pratique pour voir l'évolution dans une zone en particulier. Nous avons juste modifié un morceau de code pour bloquer le dézoom à 1 en X et Y car au-delà il n'y a rien qui s'affiche sur le diagramme. L'interface utilise aussi la bibliothèque Chrono créé par JeffProd. Il nous permettra d'avoir une unité de référence pour l'avancement du simulateur.

Pour mettre à jour le diagramme, nous avons créé deux variables static, une de type Forêt dans la classe Main afin d'y avoir accès dans les Controller des FXML, ainsi qu'une de type XYChart.Series<Number, Number>, qui permet de mettre les points dans le diagramme, pour que la classe Forêt la mettent à jour dès qu'un arbre meurt ou né.

Pour avoir accès à ces variables, il suffisait de faire :

Main.nomDeLaVariable;

Lorsqu'on utilise JavaFX, si on a besoin d'attendre un certain temps avant de faire des calculs, on doit utiliser AnimationTimer. Car les autres classes permettant de marquer une pause dans le programme ne permettent pas la mise à jour de l'interface graphique. Cette dernière se fera uniquement à la toute fin du programme, donc lorsque tous les arbres sont morts.

Voici la construction du code de l'AnimationTimer :

On instancie deux variable permettant de savoir s'il y a un évènement :

```
animationTimer = new AnimationTimer() {
   private long lastTimeEvenement = 0;
   private double nextTimeEvent = Main.foret.getDureeNextEvent();
};
```

Puis on réécrit la fonction Handle d'AnimationTimer qui permet de faire certaine action en fonction du temps :

```
animationTimer = new AnimationTimer() {
   private long lastTimeEvenement = 0;
   private double nextTimeEvent = Main.foret.getDureeNextEvent();

   @Override
   public void handle(long now) {
   }

};
```

Ensuite il faut regarder s'il y a un évènement et si c'est le cas, on fait les actions nécessaire :
A savoir que la variable now renvoie le temps en nanoseconde, c'est pour ça qu'on la divise par 1 milliard, c'est pour la récupérer en seconde :

```
animationTimer = new AnimationTimer() {
   private long lastTimeEvenement = 0;
   private double nextTimeEvent = Main.foret.getDureeNextEvent();

   @Override
   public void handle(long now) {
```

```
        if ((now - lastTimeEvenement)/ 1_000_000_000.0 >= nextTimeEvent &&
Main.foret.getList().size() != 0){

            nbEvent++;

            Main.foret.applyEvent();
            labelNbEvent.setText(String.valueOf(nbEvent));

labelNbArbres.setText(String.valueOf(Main.foret.getList().size()));
            ecrireFichier();

            lastTimeEvenement = now;
            nextTimeEvent = Main.foret.getDureeNextEvent();
        }
    }
};
```

Bien évidemment, il faut arrêter le timer s'il n'y a plus d'arbre dans la forêt :

```
animationTimer = new AnimationTimer() {
    private long lastTimeEvenement = 0;
    private double nextTimeEvent = Main.foret.getDureeNextEvent();

    @Override
    public void handle(long now) {

        if ((now - lastTimeEvenement)/ 1_000_000_000.0 >= nextEvent &&
Main.foret.getList().size() != 0){

            nbEvent++;

            Main.foret.applyEvent();
            labelNbEvent.setText(String.valueOf(nbEvent));

labelNbArbres.setText(String.valueOf(Main.foret.getList().size()));
            ecrireFichier();

            lastTimeEvenement = now;
            nextTimeEvent = Main.foret.getDureeNextEvent();
        }
        if (Main.foret.getList().size() == 0) {
            stop();
```

```
        }
    }
};
```

3. Résultats

Cette section du rapport montre comment le logiciel a été validé et testé dans son environnement de fonctionnement.

3.1 Installation

Le projet étant programmé sous le langage Java, il faudra installer la dernière version de Java sur la machine devant lancer le programme.

Lien officiel pour installer Java : https://www.java.com/fr/.

Le projet peut être soit lancé depuis l'IDE avec IntelliJ IDEA, mais il est aussi possible de créer un exécutable du projet en .jar.

3.2 Test/Validation

Le simulateur se reposant principalement sur de l'aléatoire pour ses calculs, il est assez difficile de réaliser des tests sur ce dernier. Difficile ne voulant pas dire impossible, nous avons pu quand même tester certains points du simulateur :

- Vérifier si un arbre est bien né entre les coordonnées 0 et 1.
- Vérifier si un arbre possède vraiment tous ses voisins.
- L'affichage correspond avec la partie back-end.
- Comparer notre simulateur à une version créée par notre tuteur.
- Vérifier si le fichier texte généré correspond avec la partie back-end

4. Gestion d'un projet

Enfin, dans cette dernière partie, nous présenterons notre méthodologie. En commençant par la Démarche personnelle, suivie par la Planification des tâches pour finir par un Bilan.

4.1 Démarche personnelles

L'organisation du projet s'est faite autour de différentes méthodes et de logiciels. Nous nous sommes réparti le travail grâce à GitLab et selon la méthode agile SCRUM. Du fait de notre taille, le Project Owner était aussi le ScrumMaster.

Une des problématiques lors de la réalisation du projet était la division des tâches, où nous avons eu du mal à définir les tâches nécessaires à la réalisation du projet. Malgré tout, nous avons divisé le projet en sous parties, plus simples à réaliser. Ces sous-parties sont appelées issues.

Les membres avaient cependant un grand degré d'autonomie, ils pouvaient choisir les issues qu'ils souhaitaient réaliser ou aider à réaliser. Le projet se

Nous étions sous la tutelle de M.Joannides, suivant la méthode SCRUM nécessitant un feed-back, nous le contactions toutes les deux semaines pour faire le

point, en faisant une réunion, sur notre avancement et corriger les problèmes. Il arrivait que nous posions nos questions par mail afin de gagner du temps. Cependant, les réunions se sont faites moindres en approchant la fin du projet où nous consacrons plus de temps à la rédaction du rapport et nos questions n'étaient pas suffisantes pour justifier la mise en place d'une réunion. Ces réunions étaient plus simples à organiser grâce au COVID-19 qui, au global, a rendu la réalisation du projet plus simple.

4.2 Planification des tâches

Nous avons planifié les tâches selon la méthode SCRUM, le problème de création de sous-parties bien définies a fait que souvent l'implémentation de code se faisait ensemble. Cela représentait une perte de temps, car il était plus difficile de travailler individuellement.

Le projet s'est déroulé durant le Semestre 3 qui s'avère être très chargé, nous avons dû faire de nombreuses modifications de plannings pour rendre ce projet dans les temps ainsi que nos autres travaux.

4.3 Bilan critique par rapport au cahier des charges

Sur les fonctionnalités du projet, tout ce qui nous était demandé pour le projet a été implémenté et fonctionne sans problème. On a mis une plus grande importance sur le côté de l'interface graphique que ce qu'il nous été demandé, c'était principalement pour nous aider à avoir une vision sur ce qu'on fait et en avoir une représentation claire.

D'un point de vue organisation dans un premier temps, le projet s'est déroulé comme prévu, à un rythme constant.

Dans un second temps, il y a eu quelques retours en arrière dû à la complexité du thème abordé, nous avons cependant pu corriger ces problèmes sans perdre trop de temps grâce aux réunions qui étaient hebdomadaires.

Enfin, la grosse problématique du projet n'est pas d'écrire énormément de lignes de code et les faire marcher, mais elle se trouve plus au niveau de la réflexion, où les choix pris et leurs justifications vaudraient plus que le résultat en lui-même. Cela a pu rendre le projet plus compliqué à diviser en sous-tâches, nous avons remédié à ce problème en travaillant plus souvent en groupe.

Grâce à quoi, nous avons réussi à rendre un projet dans les temps et fidèle au cahier des charges.

Au final, si nous étions amenés à rencontrer un projet similaire, nous prendrions beaucoup plus de temps au départ pour bien s'approprier le sujet, ce qui assurerait une bonne division des tâches et ainsi un gain de temps.

Conclusion

En conclusion, lors de ce projet d'équipe, nous avons appris à répondre aux attentes d'un cahier des charges avec pour objectif principal de créer un simulateur de forêt avec le langage de programmation de notre choix. Pour ce faire, nous avons choisi la méthode de travail agile SCRUM.

Durant la réalisation du projet, nous avons appris à prendre des choix techniques, à faire preuve d'autonomie, le travail d'équipe ainsi que de travailler avec un "client" ayant des attentes spécifiques, joué ici par notre tuteur. Tout d'abord, on a pris en main le sujet assez compliqué. Ensuite, on a fait des choix : Javafx au lieu de Python, l'optimisation de notre code, la méthode de travail. Le code en soit n'est pas si volumineux que ça, mais l'enjeu principal du projet est l'optimisation du code. Enfin, on a divisé la tâche principal en plein de sous-tâches bien moins conséquentes et on s'est réparti le travail. L'organisation consistait en majorité à faire des réunions régulières entre nous et hebdomadaires avec le prof. Tout cela a permis à ce qu'on rende le projet dans les temps.

Finalement, on a appris à bien mieux utiliser Java FX et StarUML à travers l'utilisation de différents graphiques pour la conception du code. Ainsi qu'à optimiser un code et l'importance de la division de tâches qui avec l'utilisation de méthodes agiles telles que SCRUM permettent de gagner en efficacité et d'éviter de grosses pertes de temps dues à des mauvaises compréhensions qu'elles soient minimales ou non. Un des problèmes rencontrés a été la difficulté du sujet qui a rendu plus

difficile la division des tâches, nous avons dû organiser plusieurs réunions avec notre tuteur pour s'assurer une bonne compréhension des attentes, mais il nous ait quand même arrivé de faire fausse route au départ, ce qui représentait au maximum une semaine de retard, il y a eu une perte de temps là-dedans. Cependant, une fois réalisée, cette division des tâches et l'organisation choisie ont prouvé être très utiles lors du confinement qui a bouleversé notre planning.

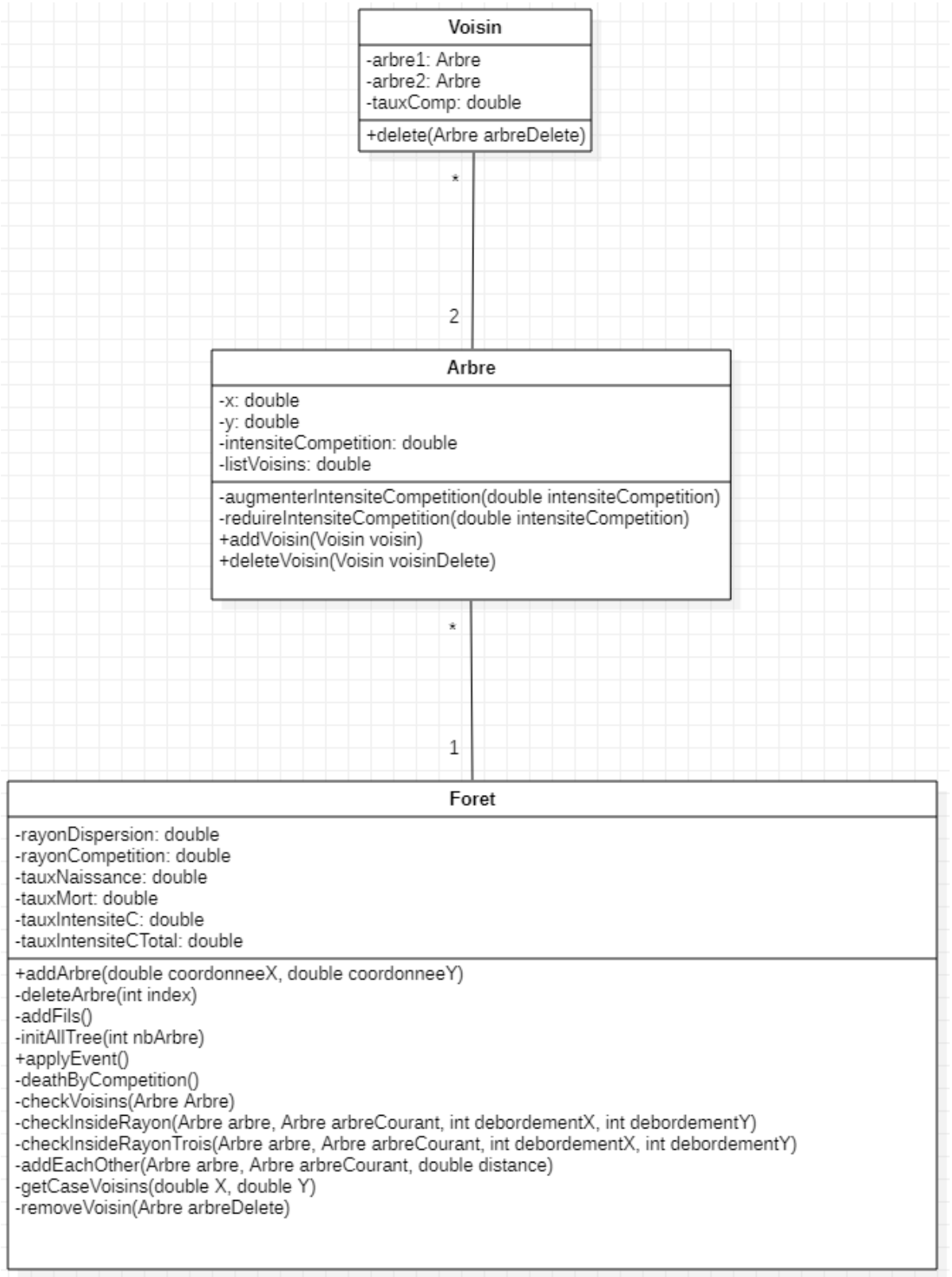
Pour finir, des évolutions possibles du projet pourraient être de changer de langage de programmation, passer au python qui est plus tourné vers les calculs scientifiques ce qui nous permettrait d'encore plus optimiser le code. Au niveau du projet en lui-même, on pourrait s'occuper de l'âge de l'arbre pour qu'ils ne naissent pas adultes ou rajouter des climats, biomes et différents types d'arbres.

Bibliographie

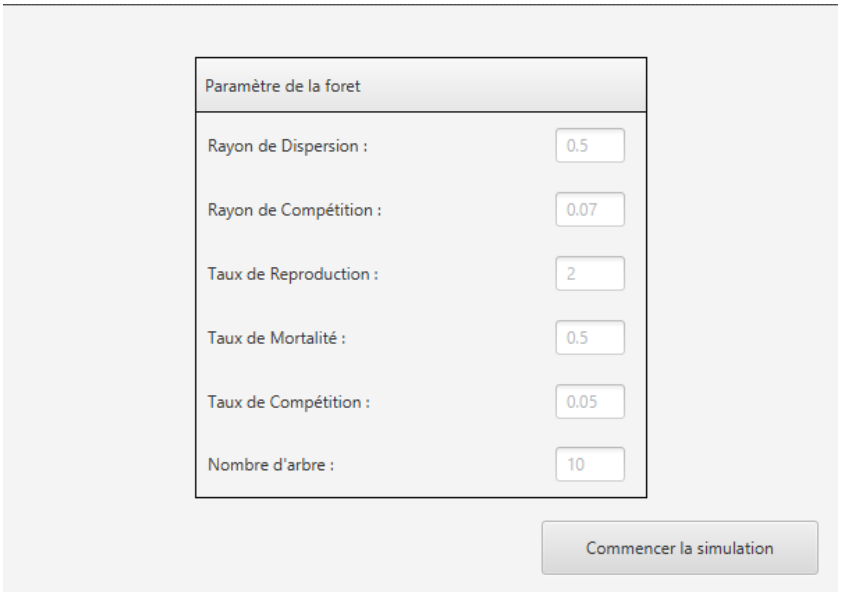
[1]
J. Winnebeck, *gillius/jfxutils*. 2020.
https://github.com/gillius/jfxutils
[2]
JeffProd, « Un chronomètre en Java ».
https://fr.jeffprod.com/blog/2015/un-chronometre-en-java/

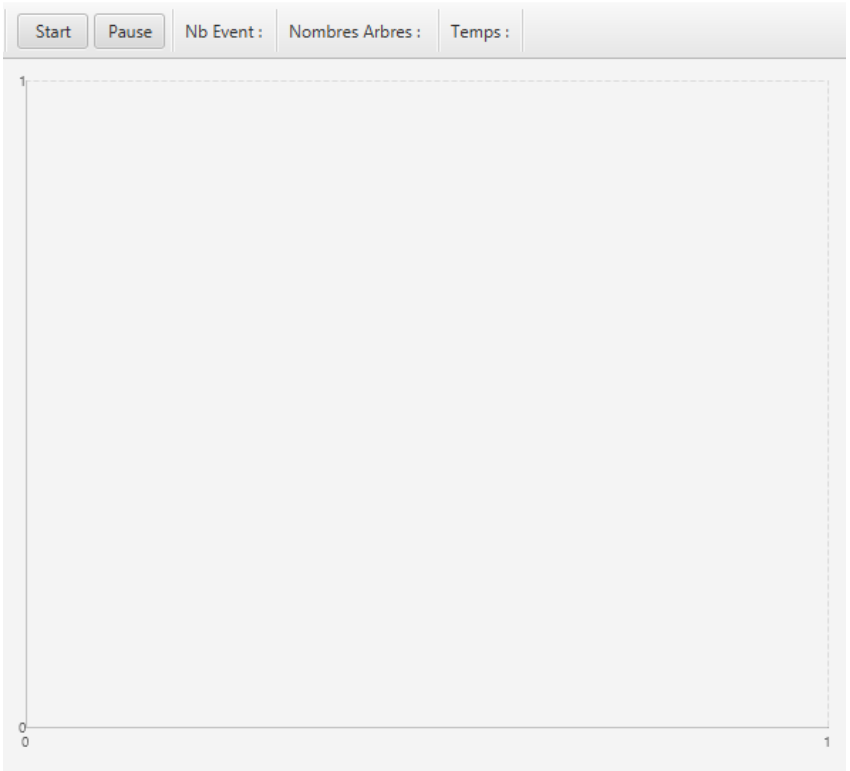
Annexes technique

Diagrammes de classes :



Scènes Graphique :





Comptes rendues de réunion :

Introduction:

Principe

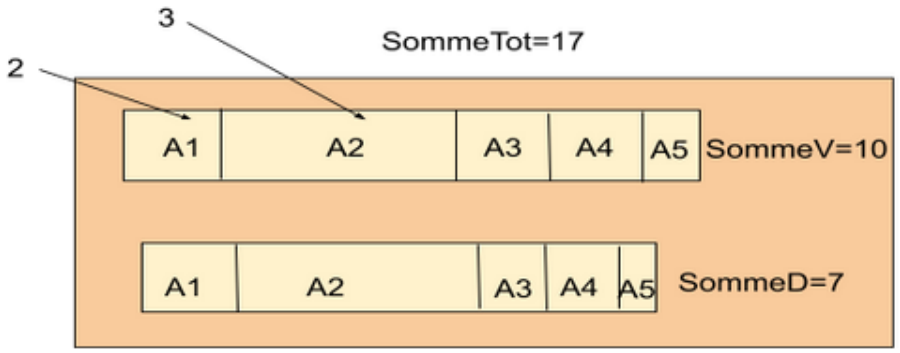
On suit le développement d'une population d'arbre sur un terrain fixe. On doit créer des paramètres qui changent l'évolution et qui sont modifiables sur l'interface. La nature étant **aléatoire** on utilise une loi pour simuler l'apparition d'arbres dans le temps *in visso*.

Def:
Fréquence λ -> Proba d'un événement
Somme des fréquences -> Proba des événements
On utilise la somme des λ: pour simplifier le code

Règles

-Arbres ne peuvent pas grandir sur la même case
-Arbre peut souffrir de compétition(-Espérance)
-Arbre regagne son espérance si la compétition n'est plus
-Terrain en **carré** mais **infini**
-Les bords du carré se rejoignent.
-Le code et l'optimisation sont les plus importants (#interface)
-Loi qui donne la proba de naissance
-Espérance de vie est fixe
-Enfants d'un arbre entrent en compétition avec le père
-Lebensraum des arbres, le +proche le moins bon

Notes



Ici on utilise que la **fréquence** de la **somme totale**.
Dès que le test est un succès, on utilise les probas des **sous groupes** pour trouver quel évènement est à faire.(Entre vie et mort, puis entre les arbres)

-Choix du langage: java

Date: 14/10/2020

Réunion 1:

Notes

Pas beaucoup de diagrammes à faire

LambdaGeneral = N*lambdab+N*lambdad+Somme(Lambdac)

Choix d'un évènement: -ln(rand())/lambda general

On a bien avancé

Corrections :

- Espérance de vie = 1/lambda
- Fréquence de Mortalité = lambda
- tauxNaissance, rayon de dispersion, rayon de compétition, Espérance et Fréquence de mortalité sont constants mais pas l'intensité de compétition
- Changer l'ArrayList

Enjeux :

Observation d'une **Limite d'arbres**

Observation d'une **croissance exponentielle**

Améliorations

Compteur d'arbres

Mise en place d'une fréquence de Mortalité

Mise en place du choix d'évènement(lambda global)

Effectuer les corrections

04/11/20

Notes rendez notes

taux naissance et mort sont pareil pour tous les arbres
lambda c (competition) individuel
rayonComp, esperanceVie, rayonDisperion, chanceRepro sont les mêmes pour tous les arbres
donc on peut mettre ces parametres dnas la forêt
un arbre c'est juste intensité (lambda c) et position
à modéliser en pseudo code pas en complexité structure
---
lambda et taux :

N * taux naiss indivi
tous les arbres ont lambda b de naissance et lambda d de mort
lambda c1 + lambda c2 + ... + lambda cn pour compétition

Tirage arbres:

tirage selon loi discrète seul dif si mort naturel alors c la même case sinon pas la même taille de case par mort de compétition
---
lorsque le taux grandit alors le tirage va être petit alors -ln(rand())
plus le taux est elevée plus le temps qui s'écoule va être faible
---
Vaudrait mieux partir d'une config aléatoire uniforme
on décrit pas population mais relation des individus

on peut soit faire quelques arbres et voir soit commencez avec beaucoup d'arbres et voir
on espère une convergeance vers un état d'équilibre
choix des paramètres qui va influer sur la
position doit être random
doit faire les paramètres de vie d'abord, puis mortalité seul pour essayer

truc intéressant : temps courrant, taille population
en l'absence de mortalité normalement croissance exponnentielle

en l'absence de naissance normalement décroissance exponnentielle
avec compétition (de)croissance vers un état d'équilibre.
un fois que structure de voisinage, arraylist compliqué (tableau de tableau avec les voisins dedans)
si on garde arraylist : avec comp on doit parcourir toute la forêt pour les voisins
complexcité n^2
quand naissance d'un arbre ajout dans liste des voisins et morts enlever liste des voisins.

Réunion 2:

Notes

Taux de 0.5 = 0.5 fois par seconde
TauxReprod 2 fois par an
TauxMort 0.5 il faut attendre 2ans

Corrections :

- Espérance de vie = 1/lambda
- Fréquence de Mortalité = lambda
- tauxNaissance, rayon de dispersion, rayon de compétition, Espérance et Fréquence de mortalité sont constants mais pas l'intensité de compétition
- Changer l'ArrayList

Enjeux :

Observation d'une **Limite d'arbres**

Observation d'une **croissance exponentielle**

Améliorations

Compteur d'arbres

Mise en place d'une fréquence de Mortalité

Mise en place du choix d'évènement(lambda global)

Effectuer les corrections

09/11/20

Notes rendez-vous 2 9/11

définir taux = c'est l'utilisateur qui s'en occupe.
//calcul temps d'attente : taux 0.5 temps attends 2 secondes

tirage aléatoire du temps :

return -ln(Math.rand()) / lambda (Math.rand() pas ouf pour calcul scientifique)
nextFloat() du coup(?) (entre 0 et 1)
classe random aussi
ptet class qui renvoie direct une loi exp quand on donne lambda(pas ouf)
risque de pattern avec Math.rand()
nb tjrs positif, pas besoin de faire 1/lambda par le chiffre.

ex:
tps courant 10,3827 ans
-ln(rand()) / lambda = 0.4
pop 253
le temps d'attente avant le prochaine événement est de 0.4
=>t= t + 0.4
=>nv evenement -> naissance / mort

birth rate 2 = graine tous les 6mois 1/2 donc 6mois.
dure 4 ans car death rate 0.5, ça arrive tous les 1/0.5ans donc deux ans.
competition integrity si 0.01 il faut 50 arbres pour diviser l'esp de vie / 2
rayon de disp pas forcement égale à rayon de comp.
prendre en compte la distance avec l'itensité de compétion du paramètre donnée (ici 0.01)
n * (lambdaB + lambdaD) = lambdaGlobal

_____ _____
lambab + lambdad
on mets tous les lamba (b et d) dans une grande liste
pour saovir si c mort ou vie
tire random 0 et 1
si c rouge(vie) on tire uniformement l'arbre qui va se reproduire
loi uniforme
si ça tombe avant lambda1 / lamdaBarre
alors c'est l'evenement 1
après on ajoute (lambda1 + lambda2 + ... + lambdan) / lambdabarre
alors c'est l'evenement n

Réunion 3:

notes comptes rendu numéro 3 24/11

choix de structures cruciales
doit être claires de pourquoi (arguments de pourquoi on a fait comme ça)
choix conception fait pas performance et pertinance
creuser + pourquoi on décide de le faire dans cette forme

avant de tuer arbre 100 on regarde dans liste de ses voisins si y a 10 on enlève a liste de 10 arbre 100 et par extension on enleve arbre 100 à tous ses voisins
arbre 1000 qui nait a comme voisins a 100 (pas réussi à suivre son exemple)
pas vraiment des voisins mais plutot des noeuds / arrêtes
on fait structure de graphe
si jamais on en avait parler avant on aurait pu faire ça en structure de graphe car ya des librairies javais qui aide pour les graphes
Interessant d'un point de vue utilisation du composant, ce qu'on fait plus bas niveau
Mais utilisation de librairies aide de faire des trucs plus poussées mais on comprends pas forcement ce que les autres ont fait
FAIRE POINT CENTRAL DU RAPPORT SUR LE CHOIX D'AVOIR FAIRE CA EN MODE DE GRAPHE

2struct de graphges qui coexistent
struc de graphes localisée (voisinage)
struct de cadrillage qui est posé sur le carée 0,1
on devrait pouvoir relier les cases à un arbre
dans un case pour connaitre les arbres de cette case (on parcours tout les arbres dans cette case en maintenant une liste d'arbre)
Il faut justifier les choix à expliquer (encore une fois c important)
un truc en rapport avec les cases à schematiser sur le diagrammes
nos choix sont globalements pertinants, la structure tient le coup
choix d'une double structure qui parrait une redondance d'informations mais qui aboutit à un benef de perf
pas l'ajout le prob de perf mais plus la recherche avec la solution de base de parcourir tout
---
tirage pour abre aléatoire pour mort compétition:
horloge globale, d'abord type d'evenement (naissance,mortNat, lambdatauxcompet)
prob : choisir l'arbre
tirer nombre uniforme entre 0 et somme lambdaBarre puis on regarde
si ce nombre U < lambda1 alors lambda1
si lambda1 < u < lambda2 alors lambda2 ...
Si lambda1+lambda2 < u < lambda1+2+3 alors lambda3
competition est le plus couteux, ça marche c'est sur mais moyen de l'optimiser
ptet moyen de l'optimiser avec la structure en graphe qu'on utilise dans le voisinage
peut faire ça dichtomie log(n) (?) moyen que ça se fasse avec complexcité
à chaque evenement rajouter une ligne dans un ficher avec date et la taille de population.

[QUATRIÈME DE COUVERTURE]

Résumé :

Le projet du simulateur de forêt est un logiciel de simulation dont le but est de permettre à des scientifiques de faire des tests sur des populations d'arbres *in silico*. Il est censé simuler l'évolution d'une population d'arbres dans une forêt avec un nombre de paramètres limités. Le but étant d'observer les interactions entre les individus en fonction de ces paramètres.

Le logiciel a été développé dans le langage de programmation Java et avec le Framework JavaFX.

Mots-clés : Java, JavaFX, Simulateur de forêt, Bioinformatique, Écologie, In silico.

Summary :

The Forest Simulator Project is a simulation software designed to allow scientists to test populations of trees in silico. It is supposed to simulate the evolution of a tree population in a forest with a limited number of parameters. The goal is to observe the interactions between individuals according to these parameters.

The software was developed in the Java programming language and with the JavaFX Framework.

Keywords : Java, JavaFX, Forest Simulator, Bioinformatic, Ecology, in silico.