
Migratory Goose

Projet informatique individuel

Tuteur : M. Pierre-Alexandre Favier

Tables des matières

Introduction	3
Procédure d'installation	3
Modélisation	3
Caractéristiques des agents	4
Caractéristiques de l'environnement	4
Interface utilisateur	6
Choix techniques	8
Architecture du programme	8
Classes	9
La classe Bird	9
La classe Map	10
La classe Area	11
La classe Simulation	13
Graphes	15
Interface	16
Tests unitaires	16
Scénarios de tests et observations	17
Les différents scénarios	17
Construction des scénarios de tests	19
Résultats	20
Interprétations et discussion	22
Gestion de projet	23
Bilan et perspectives d'améliorations	25
Bilan sur les objectifs	25
Difficultés rencontrées	26
Pistes d'amélioration et d'évolution du projet	26
Conclusion	27

Introduction

Le projet Migratory Goose a pour but de créer un système multi-agent sur le thème de la migration des oiseaux. Les agents de ce système sont des populations d'oiseaux présentant différents types de comportements migratoires. Elles évoluent sur une carte contenant des éléments influençant leur évolution.

Le choix de ce sujet vient à la fois d'un intérêt pour les systèmes multi-agents et leur fonctionnement et pour le phénomène de migration des oiseaux.

Le but final de cette simulation est de déterminer quelle stratégie migratoire permet à une espèce d'avoir le plus haut taux de croissance de sa population et dans quelles conditions, à travers des scénarios de tests. Par exemple, voir si l'abondance de nourriture permet à la population des sédentaires de croître davantage que les migrateurs, ou au contraire si cela n'a aucun impact.

Ainsi, les objectifs de ce projet sont les suivants :

- Apprentissage d'un nouveau langage (le JavaScript),
- Création d'une map et des agents qui la compose avec la modélisation des comportements de ces agents,
- Réalisation d'une interface de la simulation qui soit simple à utiliser,
- Donner la possibilité à l'utilisateur de modifier les paramètres de la simulation,
- Créer une interface permettant d'avoir la possibilité de tracer des courbes d'évolution du nombre d'individus d'une population en fonction du temps. Cela permettrait par exemple de pouvoir comparer différents types de comportements.

Dans ce rapport, nous allons vous présenter la modélisation du problème réalisée, les choix techniques concernant le code, les scénarios de tests, une partie sur la gestion de projet et enfin un bilan.

Procédure d'installation

La procédure d'installation du programme est également décrite dans le readme.

Le code est disponible à l'adresse suivante : [Github : MigratoryGoose](#). Une fois le code source copié sur sa propre machine, il est nécessaire de lancer la commande `npm install` dans le dossier [MigratoryGoose/src](#) afin de télécharger les librairies dont dépend le projet.

Pour lancer la simulation, il suffit ensuite simplement d'ouvrir le fichier `index.html` contenu dans le dossier `src`. Les instructions d'utilisation de la simulation sont écrites dans la page web.

Modélisation

Migratory Goose est construit comme un système multi-agents. Les agents et l'environnement sont décrits ci-dessous.

Caractéristiques des agents

Les agents de cette simulation sont de deux types : des populations d'oies migratrices et des populations d'oies sédentaires. Les valeurs de durée de vie et de durée de migration choisies pour la modélisation correspondent à celles des oies migratrices, appelées oie cendrées.

Dans la simulation, les agents peuvent effectuer les actions suivantes :

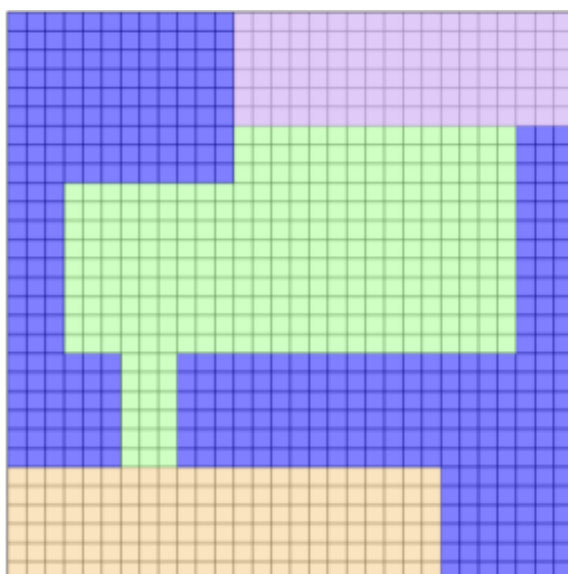
- se déplacer d'une case à une autre
- rester sur la même case
- augmenter l'effectif de la population de 1 ou plusieurs individus
- diminuer l'effectif de la population de 1 ou plusieurs individus
-

L'augmentation ou la diminution du nombre d'individus dans une population est donnée par les paramètres suivants :

- fait augmenter le nombre d'individus dans la population
 - être sur une zone favorable à la reproduction
 - être sur une zone de nourriture
- fait diminuer le nombre d'individus dans la population
 - le temps qui passe représentant le vieillissement de la population
 - être sur une zone de danger

La vitesse de déplacement des agents dépend de la taille de la map. En effet, la taille de la map en hauteur correspond à la durée d'une migration longue, soit 3 mois.

Au niveau de la grille, le nombre de case en hauteur, correspondant à $\frac{\text{hauteur de la map}}{\text{taille d'une case}}$ est égale à 3 mois soit 90 jours.



*Hauteur de la map =
nombre de cases x taille d'une case*

*Soit nbH le nombre de cases,
nbH = Hauteur de la map / taille d'une case*

Pour simplifier, on nomme nbH le nombre de cases en hauteur. On a donc

$$nbH = \frac{\text{hauteur de la map}}{\text{taille d'une case}},$$

et $nbH = 90$ jours.

Cela nous donne *un déplacement d'une case* $= \frac{90}{nbH}$ et *1 mois* $= nbH/3$.

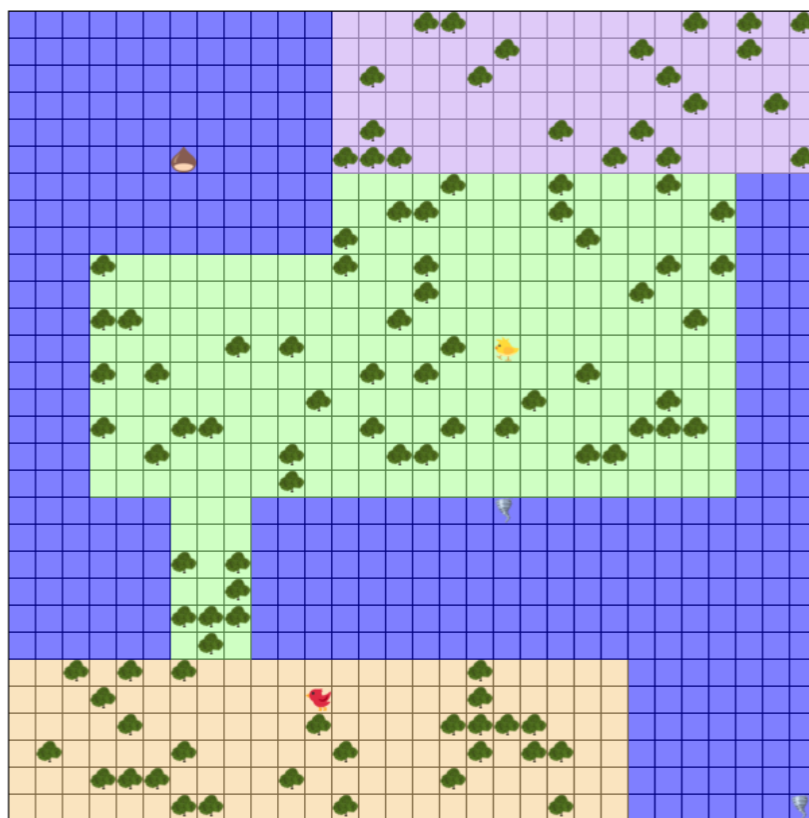
Ces différentes équations nous permettent de calculer les durées de vie et de migration des oiseaux. Par exemple la durée de vie d'une oie migratrice est d'environ 120 mois, ce qui donne

$$120 \times \frac{nbH}{3} \text{ déplacements.}$$

Et un déplacement d'une case équivaut à un tour de simulation. Cela nous permet donc d'avoir le nombre de tours de simulation nécessaires à la traversée d'une population d'oie sur la map.

Caractéristiques de l'environnement

L'environnement dans lequel évolue les agents est représenté par la map ci-dessous.



Légende :

Couleur de la zone	Populations d'oiseaux
Orange : zone d'hivernage des migrateurs	Migrateurs
Verte	Sédentaires et migrateurs
Bleue : zone de vol	Migrateurs
Violette : zone de nidification des migrateurs	Migrateurs

Les couleurs ont été choisies de façon à représenter les différents climats : l'orange correspond à une zone au climat chaud, la zone verte à une zone tempérée et la zone violette à un climat froid. La forme globale de la carte a été conçue en référence à une carte d'Europe représentant les différentes migrations des oies cendrées.

Les arbres présents sur la map servent à donner des contraintes supplémentaires au déplacement des oiseaux. En effet, passer sur une case contenant un arbre demande plus d'efforts aux populations migratrices. Cela se modélise par des cases ayant un poids plus important dans l'algorithme A*.

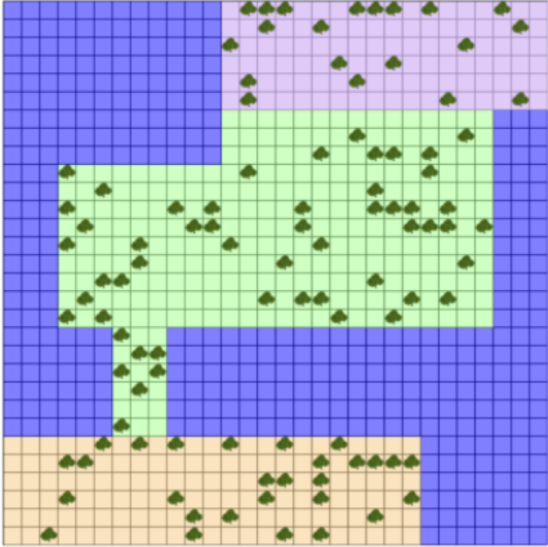


Carte d'Europe des migrations de l'oiseau cendré

Il est à noter cependant qu'il existe un grand nombre de migrations possibles de ces oies : certaines ont des migrations courtes au sein d'un même pays par exemple. Dans le cadre de ce projet, nous nous intéressons uniquement à un seul type de migration pour les comparer aux populations sédentaires. Mais une piste d'évolution intéressante serait de pouvoir rajouter d'autres types de populations migratrices dans la simulation.

Interface utilisateur

L'interface de la simulation a pour but de permettre à l'utilisateur de modifier les différents paramètres comme il le souhaite, afin de pouvoir tester différents types d'environnements. Afin de réaliser cette interface, une maquette a d'abord été créée. Elle a permis de lister les différents éléments nécessaires à cette interface et de trouver un moyen de les intercaler avec la map sans qu'ils ne gênent la lisibilité de celle-ci.



Nombre d'individus dans la population de migrants : 0
Nombre d'individus dans la population de sédentaires : 0

Lancer la simulation Arrêter la simulation

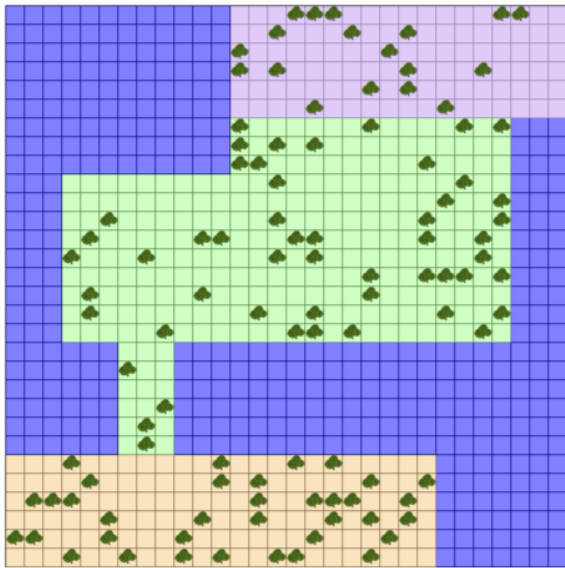
Vitesse de la simulation

	Nombre d'individu initial dans les populations		Nombre d'arbres sur la map	
Migrateurs	10		100	
Sédentaires	10			

	Taux d'apparition des catastrophes naturelles	Taux d'apparition de nourriture	Taux de reproduction
Zone de nidation (violette)	0,3	0,2	0,9
Zone tempérée (verte)	0,3	0,2	0,8
Zone d'hivernage (orange)	0,3	0,2	0,1
Zone d'océan (bleue)	0,6	0	

Réinitialiser
Valider les modifications

Maquette initiale de l'interface



Nombre d'individus dans la population de migrants : 0
Nombre d'individus dans la population de sédentaires : 0

LANCER LA SIMULATION ARRÊTER LA SIMULATION

Paramètres de la simulation

Vitesse de la simulation

Nombre d'individus initial dans la population
10

Nombre d'arbre sur la map
200

	Taux de catastrophes naturelles	Taux de nourriture	Taux de reproduction
Zone Violette	0.01	0.01	0.1
Zone Orange	0.01	0.01	0.1
Zone Verte	0.01	0.01	0.1
Zone Bleue	0.01	0	Fixé à 0

RÉINITIALISER LES PARAMÈTRES
CONFIRMER LES PARAMÈTRES

PARAMÈTRES DU SCÉNARIO 1
PARAMÈTRES DU SCÉNARIO 2
PARAMÈTRES DU SCÉNARIO 3

Interface utilisateur finale

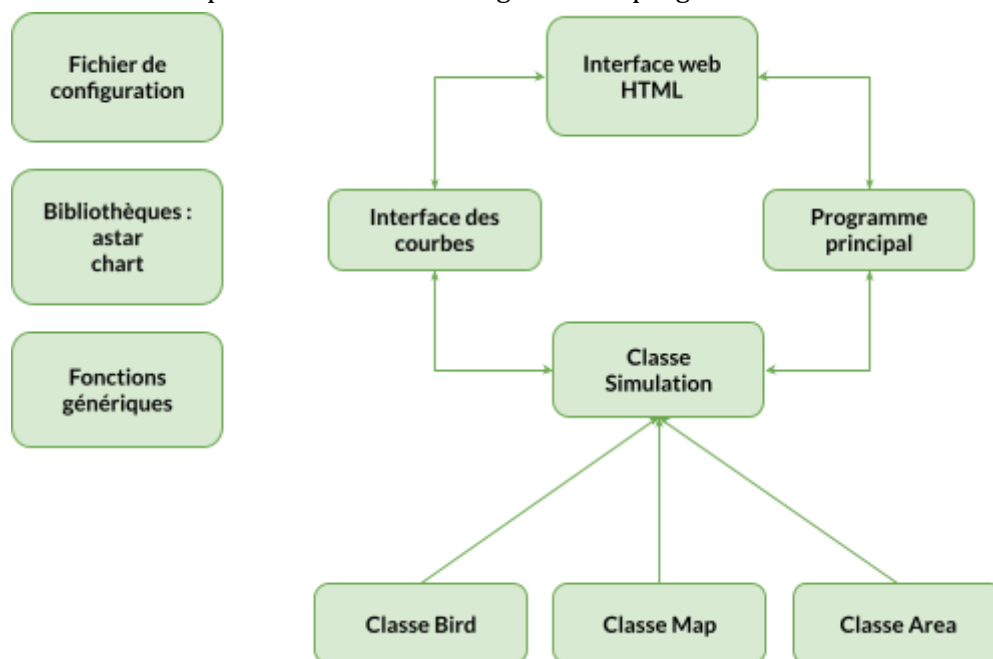
J'ai eu quelques difficultés au niveau des dimensions des éléments CSS qui contiennent l'interface finale. Celle-ci est donc un peu plus grande que la maquette originale. J'ai également ajouté trois boutons permettant de choisir directement les paramètres de test des scénarios de tests, afin de pouvoir les lancer plus rapidement. Cela permet également à un utilisateur qui trouverait fastidieux de rentrer toutes les valeurs à la main de pouvoir tester différents scénarios de simulation.

Choix techniques

Architecture du programme

Le programme Migratory Goose est composé d'une interface web HTML associée à des fichiers CSS, de 4 classes JavaScript, et de plusieurs fichiers JavaScript qui contiennent les fonctions annexes, les constantes du programme, la création des courbes permettant le suivi de l'évolution des oiseaux et des libraires.

Le schéma ci-dessous représente l'architecture globale du programme.



Architecture de Migratory Goose

La classe simulation utilise les autres classes pour implémenter une simulation globale. Le fichier program.js permet de faire le lien entre l'interface HTML et les classes en JS.

Dans les paragraphes qui suivent, nous allons voir plus en détails le fonctionnement des principales parties du programme.

Classes

Dans l'architecture du programme, nous avons vu que la classe principale est la classe simulation et qu'elles utilisent trois autres classes : Bird, Map et Area.

La classe Bird

Un objet de type Bird correspond à une population d'oiseaux. Il contient les attributs suivants :

- l'espèce (migrateur ou sédentaire)
- le nombre d'individus dans la population,
- la position de la population sur la map
- l'âge de la population
- l'état de la population (morte ou en vie)
- le taux de reproduction
- le dernier endroit visité (utilisé pour les populations migratrices uniquement)
- l'objectif (c'est-à-dire l'endroit à atteindre, utilisé pour les populations migratrices)
- le temps passé dans une zone
- et enfin l'image de la population affichée sur l'écran

Cette classe contient des fonctions permettant le déplacement des oiseaux en fonction de leur type et des fonctions permettant l'évolution des populations (c'est-à-dire l'augmentation ou la diminution du nombre d'individus).

La fonction *moveBehavior* définit le comportement de la population de l'oiseau en fonction de son type. Dans le cas des oiseaux sédentaires, les déplacements sont aléatoires. Les seules contraintes concernent les zones où peuvent se déplacer les oiseaux. Dans les cas des oiseaux migrateurs, le comportement est un peu plus complexe.

Une population doit rester un certain nombre de tours de simulation sur une même zone, puis migrer sur une autre zone. Pour se déplacer d'une zone à une autre, elle cherche à chaque fois le plus court chemin entre sa case actuelle et sa case objectif.

Pour cela, on exploite une library permettant l'utilisation de l'algorithme A*, par le biais de la fonction *findBestPath*. Cette fonction est appelée pour chaque déplacement de l'oiseau.

La fonction *migratoryMove*, quant à elle, gère le fait qu'une population reste sur une zone un certain nombre de tours avant de changer son objectif. Pour cela, on a plusieurs situations possibles :

```
if (this.spentTime < spentTimeOnWintering) {  
    //L'oiseau reste en zone orange  
    this.oneColorMove(tab, area, 'orange');  
    this.spentTime++;  
}
```

Tant qu'un oiseau n'a pas passé suffisamment de temps dans sa zone, il y reste. Ce temps passé correspond à l'attribut *spentTime* de la population, qui s'incrémente à chaque tour.

```
this.goal = [18, 2];
```

Une fois que l'oiseau a passé suffisamment de temps dans sa zone, il obtient un nouvel objectif, qui est une case dans la zone où il doit migrer.

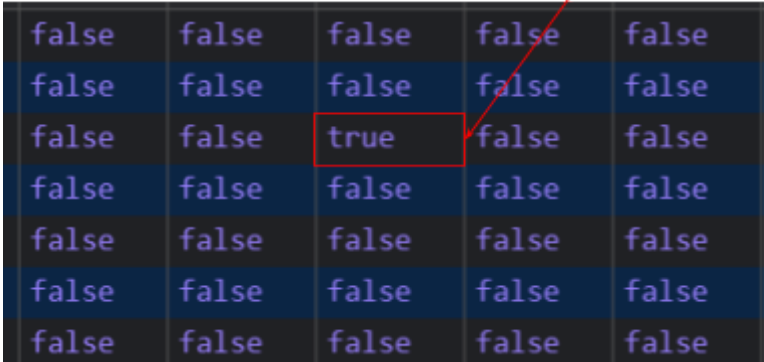
```
if (this.findBestPath(tab, graphTrees) == true) {  
    this.spentTime = 0; //Le temps passé dans cette zone  
    this.lastVisitedColor = 'purple'; //On modifie la
```

Une fois arrivé, l'attribut *spentTime* revient à 0 et sa dernière zone visitée est modifiée, afin

qu'à la migration suivante, il change de nouveau de zone.

La classe Map

Les classes Map est un tableau qui va contenir des booléens. Une case vaudra *true* si elle est occupée par un oiseau et *false* sinon.



Un oiseau qui se promène !

false	false	false	false	false
false	false	false	false	false
false	false	true	false	false
false	false	false	false	false
false	false	false	false	false
false	false	false	false	false
false	false	false	false	false

C'est également cette classe qui va s'occuper de dessiner la grille sur la canvas dans laquelle les oiseaux se déplacent et qui contient les fonctions permettant de rendre une case inoccupée ou occupée.

La classe Area

Un objet de type Area correspond à une case précise de la grille. Chaque case a pour paramètre son type (qui définit son influence sur les oiseaux), sa position, l'élément qu'elle contient (nourriture, arbre, tempête ou vide) et éventuellement l'image de cet élément.

Contrairement à la classe Map, celle-ci ne se préoccupe pas des oiseaux sur la map et de leurs déplacements.

Cette classe possède les fonctions de coloriage des cases de la map en fonction de leur type et d'affichage des différentes images des éléments sur l'écran. Mais c'est également elle qui gère l'apparition aléatoire des éléments (nourriture ou tempête) sur la map en fonction des paramètres entrées par l'utilisateur.

Pour cela on considère 3 événements :

A : "Cette case contient une tempête"

B : "Cette case contient de la nourriture"

C : "Cette case est vide (ou contient un arbre)"

avec $P(X)$ la probabilité de l'événement X et $X \in \{A, B, C\}$.

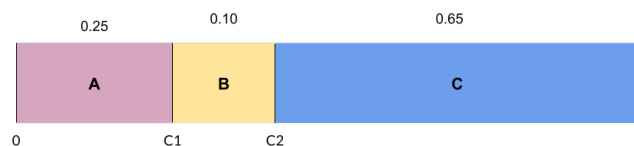
On considère dans le schéma ci-dessous que le rectangle coloré correspond à l'espace des états possibles. Chaque couleur est proportionnelle à la probabilité d'être dans cet état.

$$P(A) + P(B) + P(C) = 1$$

$$P(A) = 0.25$$

$$P(B) = 0.1$$

$$P(C) = 0.65$$



Ce qui nous intéresse, ce sont les frontières C1 et C2. De ce schéma, il vient que :

$$C1 = P(A)$$

$$C2 = P(A) + P(B)$$

Au niveau du code cela correspond à deux fonctions. *checkSumProbabilities* fait la somme des deux éléments et vérifie que cette somme est inférieure à 1. Cela correspond à trouver C2 avec $P(A) + P(B)$. Si la somme est bien inférieure à 1, les deux éléments sont stockés, sinon ils sont remplacés par des valeurs par défaut.

```
checkSumProbabilities(elt1, elt2, correctElt1, correctElt2) {  
  let sum = elt1 + elt2;  
  if (sum > 1) {  
    elt1 = correctElt1;  
    elt2 = correctElt2;  
    showErrorMsg();  
  }  
  return [elt1, elt2];  
}
```

Fonction *checkSumProbabilities*

Ensuite, la fonction *createElement* va récupérer les valeurs entrées par l'utilisateur et le type de la case actuelle.

```
let rdn = Math.random();
let hurricaneValue = 0;
let foodValue = 0;

switch (this.areaType) {
  case 'blue':
    hurricaneValue = getParameter("disasterRateBlue");
    foodValue = getParameter("foodRateBlue");
    break;
```

Une fois les valeurs *hurricaneValue* et *foodValue* récupérées (qui correspondent à $P(A)$ et $P(B)$ dans le schéma) on utilise *checkSumProbabilities* pour avoir la valeur qui correspond à notre C2 dans le schéma.

```
hurricaneValue = this.checkSumProbabilities(hurricaneValue, foodValue, 0.01, 0.01)[0];
foodValue = this.checkSumProbabilities(hurricaneValue, foodValue, 0.01, 0.01)[1];
```

On peut ensuite placer l'élément en fonction du nombre aléatoire *rdn* défini au début de la fonction.

```
if (this.hasElement == 'no') //La case est vide
{
  if (rdn < hurricaneValue) {
    this.hasElement = 'hurricane';
  } else if (rdn < foodValue + hurricaneValue) {
    this.hasElement = 'food';
  }
}
```

La classe Simulation

Un seul objet Simulation est créé dans le programme. Les attributs d'une simulation sont différentes populations d'oiseaux, une Map, un tableau d'objets Area, un graph qui se calque sur la Map pour permettre de faire des recherches du plus court chemin et des nombres aléatoires. Les deux fonctions principales de cette classe sont *initialisation* et *move*, qui vont, respectivement, mettre tous les éléments de la simulation en place et modéliser un pas de temps dans la simulation.

Nous allons observer un peu plus en détail quelques actions importantes de la fonction *move*.

Dans un premier temps, on vérifie l'état de la population. Si celle-ci n'a plus d'individus, elle est supprimée de la simulation.

```
//Déplacement des oiseaux et interactions avec l'environnement
for (let bird of this.birds) {
  //On supprime de la carte les populations qui n'ont plus d'oiseaux
  if (bird.state == 'death') {
    this.birds.splice(this.birds[bird], 1);
  }
}
```

Ensuite on rend la case où se trouve la population d'oiseaux occupée, et on libère la case précédente. L'ordre des actions est le suivant :

- la case sur laquelle je suis avant mon déplacement devient inoccupée
- je me déplace immédiatement
- la nouvelle case sur laquelle je suis devient occupée

```
//Déplacement des oiseaux et interactions avec l'environnement
this.map.turnUnoccupied(bird.positionX, bird.positionY);
bird.moveBehavior(this.map.matrice, this.areas, this.graph);
this.map.turnOccupied(bird.positionX, bird.positionY);
```

Les éléments présents sur la map restent plusieurs tours. Ce nombre de tours est donné par la variable *elementTime*. Cette variable est incrémentée à chaque tour, et revient à 0 lors du sixième tour. Ainsi, les images des éléments et leur valeur sur les cases sont conservées pendant 5 tours. Au bout du 5ème tour, on réinitialise les cases avant de replacer les éléments, sauf celles qui contiennent un arbre, étant donné que les arbres ne bougent pas. Les nouveaux éléments sont positionnés de façon aléatoire sur la map.

```
if (this.elementTime >= 5) {
  for (let i = 0; i < heightMap / cellSize; i++) {
    for (let j = 0; j < widthMap / cellSize; j++) {
      area = this.areas[i][j];
      if (area.hasElement != 'tree') {
        area.hasElement = 'no';
      }
    }
  }
}
```

Un des derniers éléments importants de cette fonction est le lien qu'elle fait avec l'affichage des courbes. À chaque tour, on récupère le nombre d'oiseaux dans chaque population et on l'affiche sur l'écran en dessous de la simulation. On le récupère également dans les variables *nbSedentaryData* et *nbMigratoryData* qui seront ensuite utilisées comme données pour l'affichage des graphiques.

```
if (bird.species == 'migratory') {  
  nbMigratory.innerHTML = bird.nbIndividuals;  
  nbMigratoryData = bird.nbIndividuals;  
} else if (bird.species == 'sedentary') {  
  nbSedentary.innerHTML = bird.nbIndividuals;  
  nbSedentaryData = bird.nbIndividuals;  
}
```

Graphes

La création des courbes permettant de voir l'évolution de la population des oiseaux s'appuie sur la bibliothèque chart.js. Cette bibliothèque permet de créer un graphe simplement de la même façon qu'un objet, à partir d'un canvas préexistant :

```
let birdsChart = new Chart(canvasGraphContext, config);
```

Dans la partie *config* on retrouve les éléments graphiques des courbes (couleurs, type du graph etc.) et les données.

C'est ici la partie donnée qui nous intéresse. En effet pour chaque courbe, les données en ordonnées correspondent aux variables globales du programme contenant le nombre d'oiseaux dans chaque population :

```
label: "Population d'oiseaux sédentaires",  
fill: false,  
backgroundColor: "#ffff80",  
borderColor: "#ffff80",  
data: sedentaryBirdsData,  
fill: false,
```

Les labels qui correspondent aux valeurs en abscisses sont quant à elles créées au fur et à mesure de la simulation.

```
const addData = (chart, label) => {  
  chart.data.labels.push(iteration); //Abcisse du graph  
  //Valeurs en ordonnée des deux courbes  
  chart.data.datasets[0].data.push(nbMigratoryData);  
  chart.data.datasets[1].data.push(nbSedentaryData);  
  chart.update();  
  iteration++;  
}
```

La fonction ci-dessus *addData* est appelée à chaque itération du programme pour ajouter un nouveau point sur les deux courbes grâce aux fonctions *push* qui font partie de la bibliothèque chart.js.

Interface

L'interface est principalement composée de deux canvas HTML, l'un pour la map ou se déplacent les populations d'oiseaux et l'un qui contient les courbes.

```
<canvas id="map" width="800" height="800"></canvas>
```

```
<canvas id="graph" width="300" height="300"></canvas>
```

Ces canvas sont ensuite modifiés grâce aux différents fichiers JS.

Le dernier élément important est une div qui contient les paramètres modifiables par l'utilisateur.

```
<div id="userInterface">
```

Une partie de ces paramètres est contenue dans un tableau, ce qui permettait un agencement plus propre sur l'interface.

```
<table>
  <tr>
    <td>&nbsp;</td>
    <td>Taux de catastrophes naturelles</td>
    <td>Taux de nourriture</td>
    <td>Taux de reproduction</td>
  </tr>
```

Les fichiers CSS et l'agencement global de la page web proviennent d'un template du site templated.co. J'ai fait le choix d'utiliser un template déjà réalisé pour ne pas avoir à passer trop de temps sur le CSS de la page, parce que ce n'était pas le cœur du sujet. Et cela permettait d'avoir une page "jolie" rapidement.

Tests unitaires

Des tests unitaires ont été faits sur plusieurs parties du programme. Ils sont trouvables dans le fichier unitTests.js

Ils se lancent automatiquement lors de l'ouverture de la page HTML et leurs résultats sont visibles dans la console de la page web.

```
Lancement des tests unitaires :
Tests effectués : 8
Nombre de tests échoués : 0
Nombre de tests réussis : 8
Fin des tests unitaires.
```

Un exemple d'exécution des tests unitaires

Ces tests ne testent pas toutes les fonctions présentes dans le programme, car certaines fonctions, comme l'affichage des oiseaux sur l'écran par exemple, ont un résultat facilement visible "à l'œil nu". Ils ne recouvrent pas les fonctions qualitatives.

Un exemple de test unitaire est présenté ci-dessous :

```
const testResetParameters = () => {  
  nbTests++;  
  const expectParam = 200;  
  resetParameters();  
  if (getParameter("nbTrees") !== expectParam) {  
    failedTests++;  
    console.error('Function resetParameters failed');  
  }  
}  
testResetParameters();
```

Test unitaire de la fonction resetParameters

Cette fonction test la fonction *resetParameters* qui s'active lorsque l'utilisateur clique sur un bouton et qui permet de remettre à leurs valeurs de défaut tous les paramètres de la simulation. *expectParam* correspond à chaque fois au résultat attendu. Si la fonction testée ne renvoie pas le bon résultat, un message d'erreur s'affiche dans la console. On incrémente également le nombre de tests qui ont échoué et le nombre de tests au total pour pouvoir l'afficher dans la console et vérifier l'exécution de l'ensemble des tests.

Nous aurions pu également utiliser une bibliothèque de test JS pour la construction des tests unitaires. Mais cela nécessitait un temps d'apprentissage qui n'avait pas été prévu dans le planning et qui aurait donc retardé le projet. La construction des tests "à la main" a également permis de retravailler des acquis du précédent semestre, où le travail sur les tests avait été fait en C#.

Scénarios de tests et observations

Afin d'observer quels sont les paramètres d'influences sur les populations d'oiseaux, des petits scénarios de test ont été réalisés. Il est à noter que nous ne faisons pas d'hypothèse sur les résultats que nous allons obtenir. L'objectif est simplement de voir quelles vont être les variations observées, en terme de nombre d'individus et d'essayer d'en tirer des conclusions ou des interprétations intéressantes.

Les différents scénarios

Nous allons comparer trois simulations avec des jeux de données différents. Le scénario 1 correspond à la situation "témoin".

Le scénario 2 correspond à une saison de sécheresse où le taux d'apparition de nourriture dans chaque zone est très faible.

Le scénario 3 correspond au contraire à une saison orageuse et pluvieuse, où les tempêtes prolifèrent.

Le récapitulatif des différents jeux de données utilisés pour les trois simulations est donné dans le tableau ci-dessous (les paramètres sur fond jaune sont ceux qui ont été modifiés par rapport au premier scénario).

		Scénario de test 1	Scénario de test 2	Scénario de test 3
Nombre d'itérations		300		
Nombre initial d'individus dans chaque population		20		
Nombre d'arbres		200		
Jeux de données	Taux d'apparition de nourriture dans toutes les zones sauf bleue	0.05	0.01	0.05
	Taux d'apparition de nourriture en zone bleue	0	0	0
	Taux d'apparition d'un danger en zone orange	0.02	0.02	0.08
	Taux d'apparition d'un danger en zone bleue	0.08	0.08	0.16
	Taux d'apparition d'un danger en zones verte et violette	0.05	0.05	0.1

Les taux de reproduction sont les mêmes dans les trois scénarios :

Taux de reproduction en zone orange	0.01
Taux de reproduction en zone bleue	0
Taux de reproduction en zone verte	0.2
Taux de reproduction en zone violette	0.6

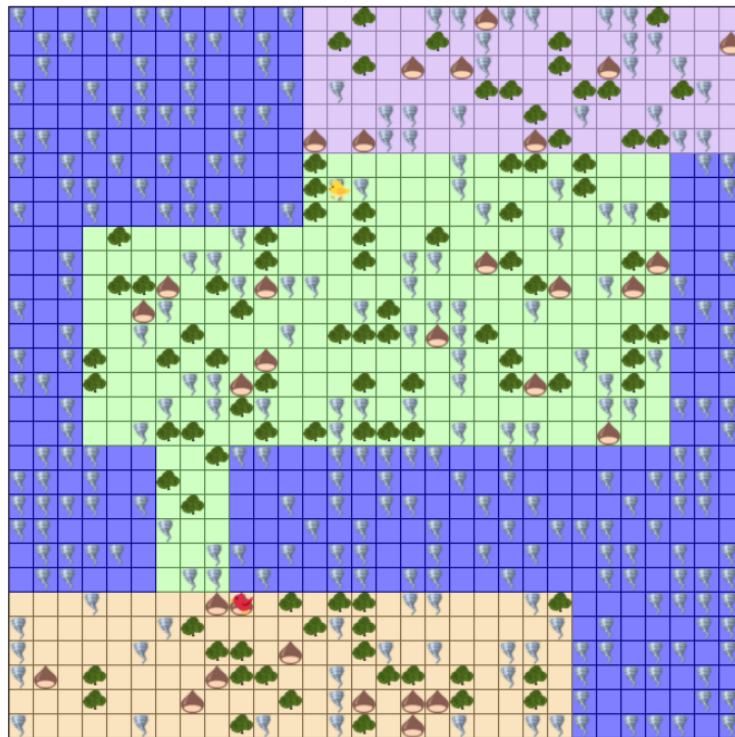
Construction des scénarios de tests

Dans cette partie, nous allons décrire la façon dont les paramètres (taux de nourriture, taux de tempête..) des tests ont été choisis.

Le scénario 1 doit représenter une situation témoin. Il était donc important que l'évolution moyenne des oiseaux dans cette situation ait une croissance ou une décroissance faible, voire nulle, pour les deux populations. Les paramètres ont donc été choisis, après plusieurs tests, de façon à avoir des évolutions faibles. Il n'a pas été simple de trouver des situations où les deux populations d'oiseaux sont stables. De ce fait, en cas de variations importantes, la croissance des populations a été favorisée par rapport à leur décroissance.

Pour les scénarios 2 et 3, on conserve un certain nombre de paramètres à l'identique du scénario témoin, afin d'isoler l'effet d'un paramètre à la fois sur la population. Respectivement le taux de nourriture et le taux de tempête.

De plus, il a fallu choisir des valeurs qui permettaient de ne pas avoir une surcharge à l'écran. Si des taux sont supérieurs à 20% dans la simulation, cela donne l'impression que l'élément correspondant est présent partout sur la carte (*cf image ci-dessous*).



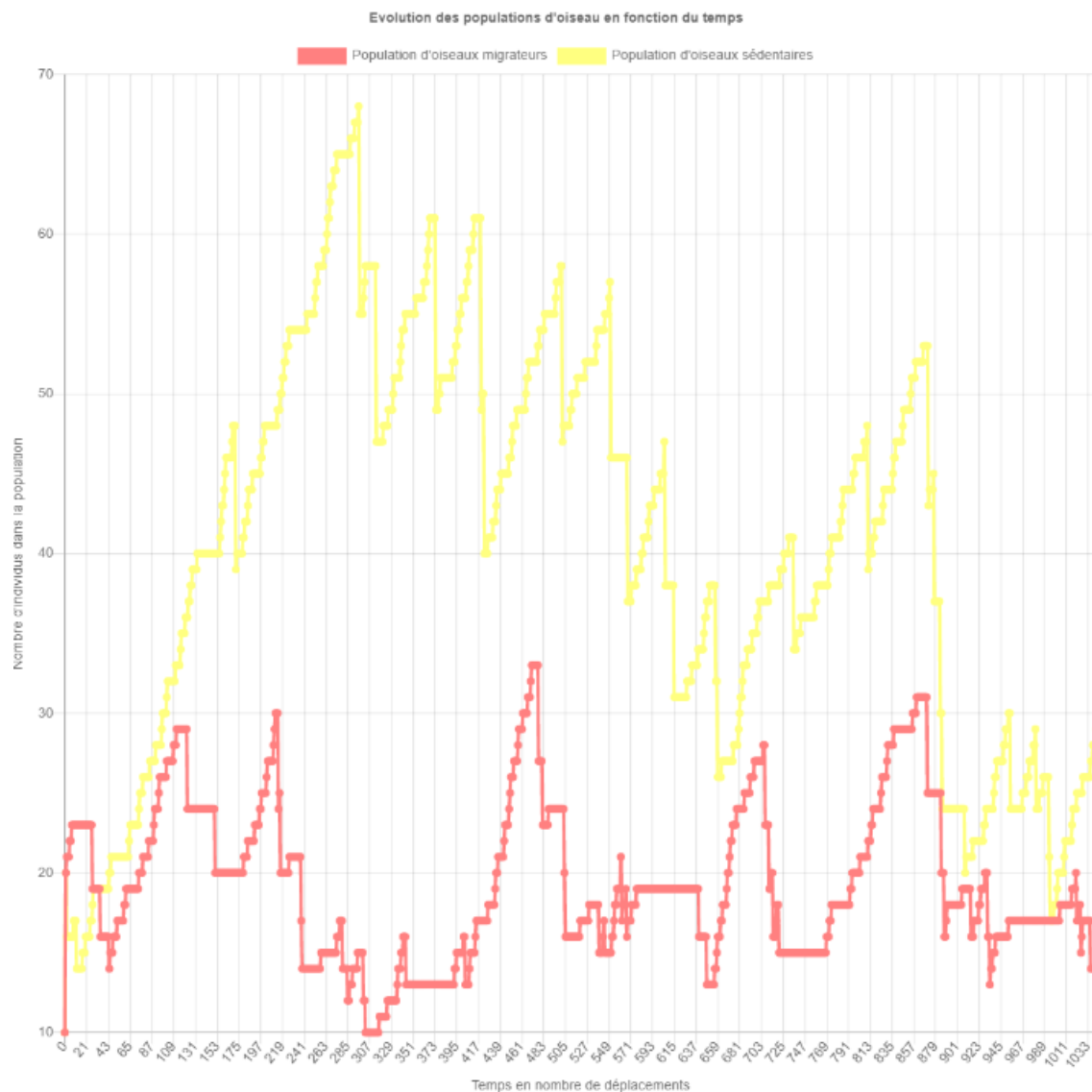
C'est l'apocalypse !

De la même façon, le nombre d'arbres a été choisi de sorte à ce qu'il y en ait suffisamment pour modifier le chemin des oiseaux migrateurs à chaque migration, sans que cela ne surcharge l'écran. Il faut également laisser des passages libres aux déplacements des oiseaux pour que l'algorithme A* puisse fonctionner. Si toutes les cases ont la même valeur, cela revient à n'avoir aucun arbre sur la simulation.

Après quelques tests, le nombre de 200 arbres a semblé être un bon compromis.

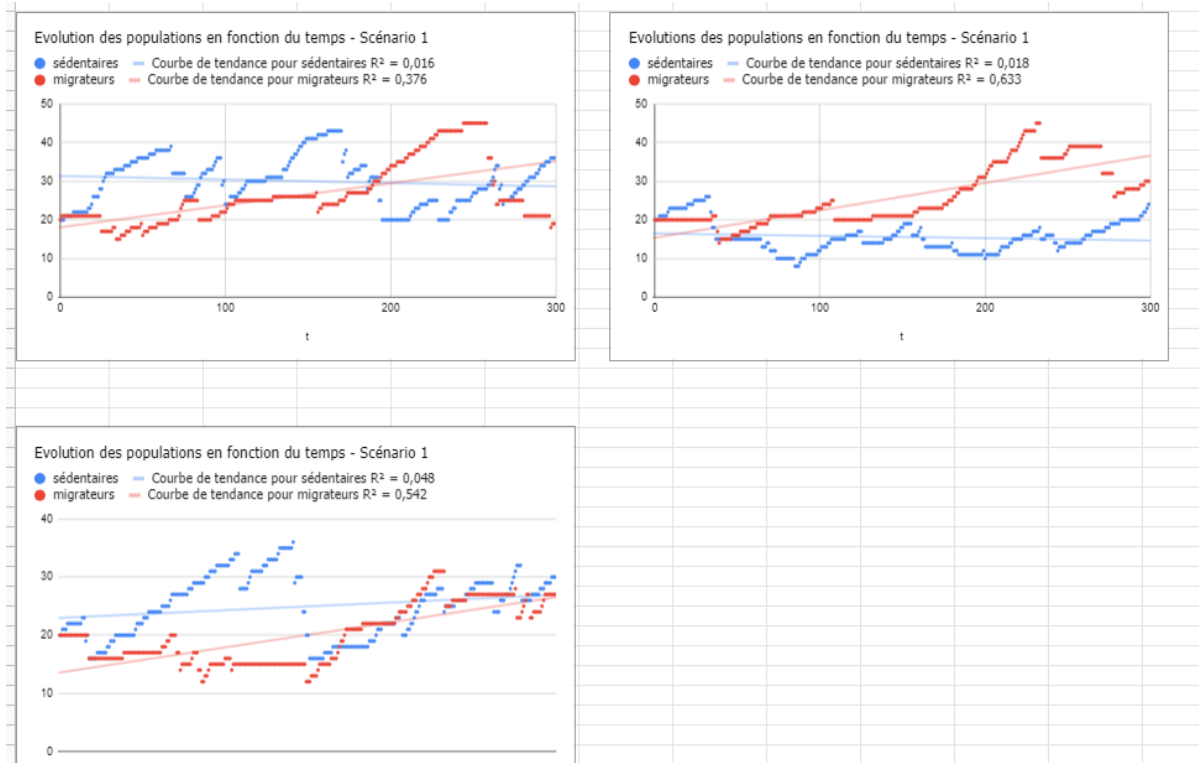
Le nombre de 300 itérations pour chaque test est un compromis entre le temps de la simulation et l'obtention de courbes avec des variations intéressantes. Pour rappel, une itération correspondant au déplacement d'une case d'un oiseau. On va ensuite observer, dans la partie résultat, qu'on obtient notamment des "cycles" qui peuvent être intéressants à analyser avec ce nombre d'itérations.

Sur l'image ci-dessous, la simulation a tourné pendant plus de 1000 itérations. On constate qu'on retrouve le même phénomène de cycle, à plus grande échelle.



Résultats

Lorsqu'on fait tourner la simulation plusieurs fois, sur un même jeu de paramètre, on constate que les situations peuvent grandement différer d'une fois sur l'autre (*cf image ci-dessous*). Cela est probablement dû aux caractères aléatoires des déplacements des oiseaux et des éléments. Afin d'avoir des résultats un peu plus fiables, j'ai donc choisi de faire tourner la simulation plusieurs fois pour chaque jeu de données et de faire des moyennes des résultats obtenus.

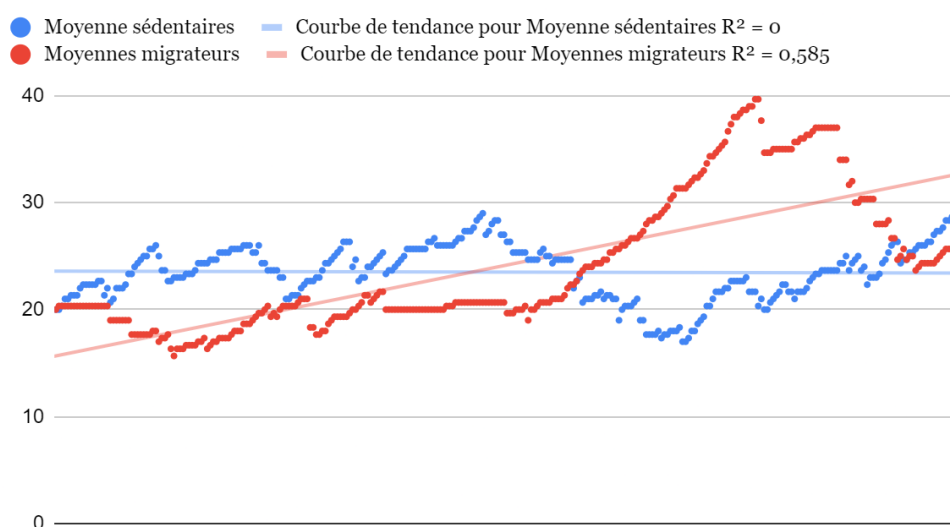


Sur cette image, on voit trois graphes qui correspondent à un même jeu de paramètres et qui sont pourtant assez différents.

Sur les courbes qui vont être présentées ci-dessous, j'ai affiché à chaque fois la droite de régression linéaire et le R^2 . Pour certaines courbes, le R^2 est assez mauvais, mais il est tout de même intéressant de les conserver pour les comparer, car on observe une différence nette entre les types de population d'oiseaux.

Pour le premier scénario, on obtient les courbes moyennées suivantes :

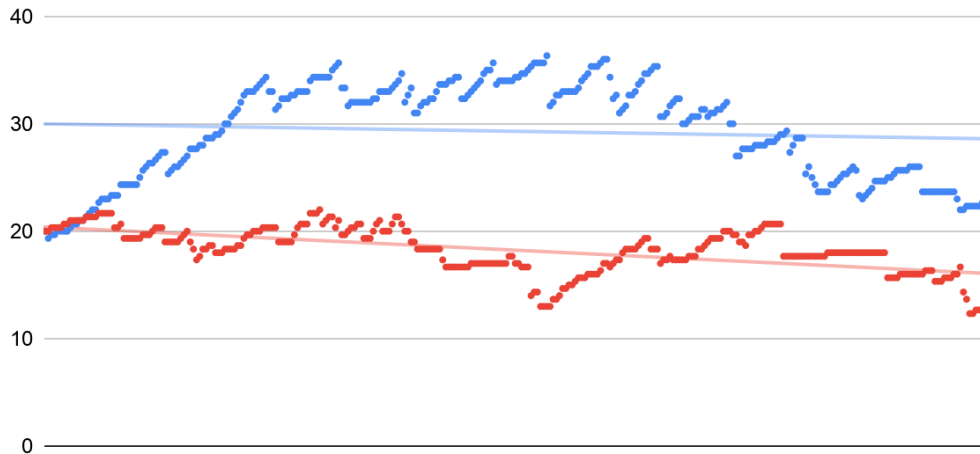
Evolution moyenne de l'évolution de la population en fonction du temps - Scénario 1



Pour le scénario 2, on obtient les courbes moyennées suivantes :

Evolution moyenne de l'évolution de la population en fonction du temps - Scénario 2

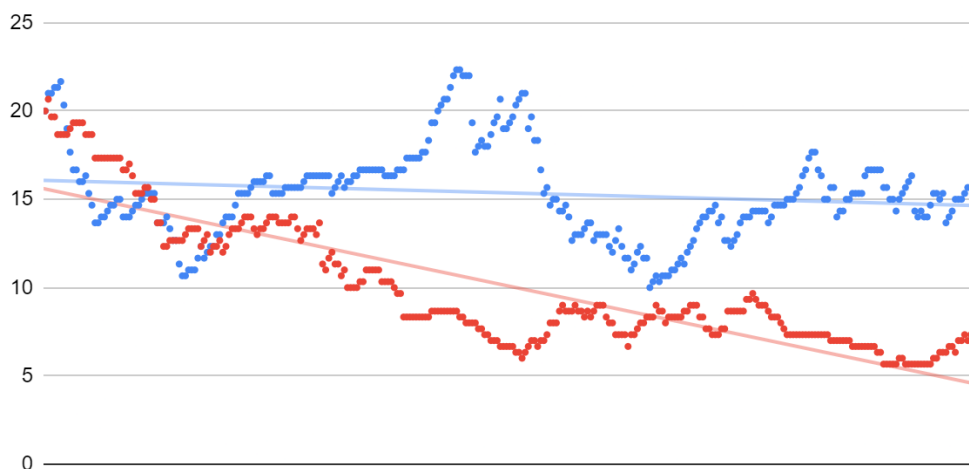
- Moyenne sédentaires — Courbe de tendance pour Moyenne sédentaires $R^2 = 0,008$
- Moyennes migrants — Courbe de tendance pour Moyennes migrants $R^2 = 0,354$



Pour le scénario 3, on obtient les courbes moyennées suivantes :

Evolution moyenne de l'évolution de la population en fonction du temps - Scénario 3

- Moyenne sédentaires — Courbe de tendance pour Moyenne sédentaires $R^2 = 0,026$
- Moyennes migrants — Courbe de tendance pour Moyennes migrants $R^2 = 0,758$



L'interprétation de ces courbes est faite dans la partie suivante.

Interprétations et discussion

Au vu des résultats obtenus, plusieurs éléments peuvent être intéressants à notifier.

Sur les trois scénarios de test, les R^2 sont très proches de 0 pour les populations sédentaires. Mais les droites de régression linéaires obtenue sont également proches d'une équation de la forme $y = \text{constante}$. En observant la forme des courbes, on constate des évolutions qui présentent des formes de sinusoides, plutôt irrégulières. De ces trois éléments, on peut émettre l'hypothèse que l'évolution des populations sédentaires suit une forme de cycle, qui rappelle les cycles obtenus en biologie des populations sur des modèles proies-prédateurs.

Ces cycles sont aussi présents chez les populations migratrices, mais avec des amplitudes moins grandes. Si on regarde les échelles de temps, on se rend compte qu'elles collent avec les déplacements migratoires des oiseaux. Il n'est donc pas très étonnant de retrouver ces cycles. Le nombre d'individus augmente lorsque la population est dans une zone favorable à la reproduction, pour redescendre ensuite. Néanmoins, la zone de reproduction (zone violette) étant plus restreinte que les zones d'hivernage et de vol (zones verte, bleue et orange), on a globalement une décroissance du nombre d'individus.

Les courbes obtenues sur les scénarios 2 et 3 sont décroissantes, contrairement à celles du scénario témoin. Cela semble logique puisque dans l'un des cas, la nourriture se fait plus rare et dans les autres les tempêtes se font plus présentes. Néanmoins, il semble que ces contraintes ont un effet beaucoup plus important sur les populations d'oiseaux migrateurs que celles des sédentaires. Cela peut s'expliquer par leur type de comportement. Les oiseaux migrateurs ne se reproduisent que sur une certaine zone : en dehors de cette zone ils sont donc plus sensibles aux aléas climatiques. Les oiseaux sédentaires vivant toujours dans une même zone, avec un taux de reproduction plus constant, ont une évolution plus stable.

Gestion de projet

Le tableau ci-dessous présente les différentes fonctionnalités du projet sous forme d'un planning.

Echéance	Description des tâches	État d'avancement
16/02	Rédaction de la modélisation du projet et réflexions sur les risques	100%
23/02	<p>Avoir terminé les fonctionnalités concernant la simulation suivantes :</p> <ul style="list-style-type: none"> - La carte est répartie en différentes zones selon le code couleur indiquée. - 2 populations d'oiseaux (une de chaque type) se déplacent de façon aléatoire sur la carte. - Des cases "bonus" et "malus" apparaissent de façon aléatoire en fonction des taux d'apparition de chaque zone. - Le nombre d'individus de chaque population apparaît à côté de la map. - Le nombre d'individus de chaque population augmente et diminue en fonction des éléments rencontrés par les agents. - Tous les $(nbCaseLongeur/3)*120$ déplacements (correspond à la durée de vie de l'oiseau), la population décroît de 10% des individus. - Les oiseaux sédentaires restent sur leur zone. 	100%

	<i>(nbCaseLongeur/3) correspond aux nombre de déplacement d'une population sur la map pour effectuer un mois de temps.</i>	
02/03	<p>Avoir terminé les fonctionnalités suivantes :</p> <ul style="list-style-type: none"> - Les oiseaux migrateurs présentent le comportement suivant : <ul style="list-style-type: none"> - Je suis en zone d'hivernage : j'y reste 4 mois puis mon objectif est de rejoindre la zone de reproduction la plus proche. - Je suis en zone de reproduction : j'y reste 2 mois puis mon objectif est de rejoindre la zone d'hivernage la plus proche. - Je suis dans un autre type de zone : si la dernière zone visitée est l'hivernage alors je rejoins la zone de reproduction la plus proche. Si la dernière zone visitée est celle de reproduction alors je rejoins la zone d'hivernage la plus proche. 	100%
17/03	<p>Avoir terminées les fonctionnalités concernant l'affichage graphique suivantes :</p> <ul style="list-style-type: none"> - L'utilisateur peut démarrer et stopper la simulation à tout moment grâce à deux boutons. - L'utilisateur peut modifier la vitesse de la simulation à l'aide d'un slider : de 250 ms à 2 secondes. - L'utilisateur peut choisir le nombre d'oiseaux de départ dans chaque population. 	100%
18/03	Rendu de l'état d'avancement du projet	
22/03	<p>Avoir terminées les fonctionnalités concernant l'affichage graphique suivantes :</p> <ul style="list-style-type: none"> - Lorsque la simulation est en cours, les nombres d'oiseaux, de populations ainsi que les paramètres ne peuvent plus être modifiés. - L'utilisateur peut modifier le taux d'apparition d'une case bonus selon les différentes zones - L'utilisateur peut modifier le taux d'apparition d'une case malus selon les différentes zones 	100%
30/03	<p>Avoir terminé les fonctionnalités concernant l'affichage des courbes suivantes :</p> <ul style="list-style-type: none"> - Un graphe "nombre d'individus en fonction du temps" est créé. - Pour chaque population d'oiseaux, une courbe est créée sur le graphe. 	100%
05/04	Avoir terminé les fonctionnalités concernant l'affichage des courbes suivantes :	100%

	<ul style="list-style-type: none"> - Un point d'une courbe se crée à chaque déplacement. - Les courbes s'affichent en temps réel au cours de la simulation. - Un bouton permet de réinitialiser l'affichage des courbes. - Un bouton permet d'enregistrer les courbes afin de pouvoir les conserver et les comparer à d'autres. 	
13/04	Avoir terminé les tests unitaires du projet	80%
20/04	Préparation de la vidéo de présentation	100%
27/04	Rendu final des livrables	

Le respect des deadlines s'est globalement bien déroulé. La principale difficulté concernant le planning a été la vidéo de présentation. Elle n'avait pas été anticipée au début du projet et réaliser une vidéo prend plus de temps que de préparer une soutenance.

En ce qui concerne les tests unitaires, nous aurions pu tester davantage de fonctions afin d'être plus exhaustif. Néanmoins ce n'était pas l'objectif principal de ce projet, c'est pourquoi ils ne sont pas tout à fait complets.

Bilan et perspectives d'améliorations

Bilan sur les objectifs

Objectif	Validé : oui ou non Si non, pourquoi ?
Apprentissage du JS	Oui
Création d'une map et des agents qui la compose avec la modélisation des comportements de ces agents	Oui
Réalisation d'une interface de la simulation qui soit simple à utiliser	Je ne suis pas convaincue par l'ergonomie de l'interface utilisateur. Avec plus de temps j'aurais aimé mettre en place des tests utilisateurs de cette interface. Mais ce n'était pas le cœur du projet.
Donner la possibilité à l'utilisateur de modifier les paramètres de la simulation	Oui
Créer une interface permettant d'avoir la possibilité de tracer des courbes d'évolution	Oui

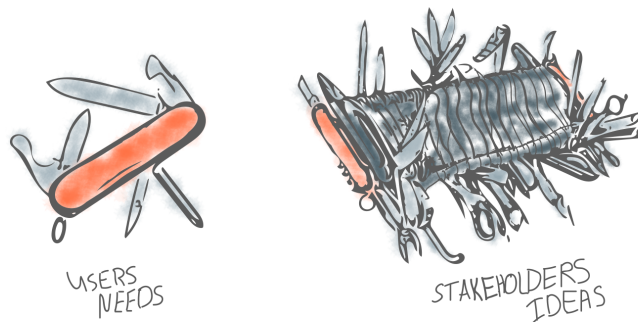
du nombre d'individus d'une population en fonction du temps	
Pouvoir mettre différents types de populations migratrices sur la simulation	Il est possible d'ajouter des populations dans le code, puisque que ce sont des objets de la simulation. Néanmoins il n'est pas possible de leur donner un autre type que "migrateur" ou "sédentaire" et les graphes sont conçus pour afficher seulement deux courbes.

Difficultés rencontrées

Le fait de devoir cadrer seule son projet, c'est-à-dire de définir quelles sont les limites pertinentes ou au contraire quels sont les ajouts à faire qui sont importants n'a pas été évident. Un des problèmes qui en découle est celui du features creep, c'est-à-dire l'envie de toujours ajouter plus de nouvelles fonctionnalités au fur et à mesure, ce qui donne l'impression que le projet est infini.

Cela m'a fait prendre un peu de retard dans la rédaction du rapport, car j'avais l'impression qu'en termes de fonctionnalité de la simulation et de code, ça n'était jamais suffisant.

Une des solutions pour contrer ce problème a été d'essayer de se fixer l'objectif d'aller jusqu'au bout de la problématique de départ, à savoir la comparaison entre les différentes populations à l'aide de courbes, avant de se concentrer sur de nouvelles possibilités d'enrichissement. C'est ainsi que je n'ai pas eu le temps de faire évoluer plus de deux populations différentes sur la carte. Mais il me semblait plus intéressant de s'en tenir à deux et de passer du temps sur l'analyse des résultats obtenus.



Une petite illustration du features creep

Pistes d'amélioration et d'évolution du projet

Parmi les améliorations possibles du projet, on pourrait avoir la rectification du CSS et l'amélioration de l'ergonomie de l'interface utilisateur.

Afin d'aller plus loin dans le projet, il serait intéressant d'avoir l'évolution de plusieurs populations présentant d'autres variétés de comportement. On pourrait également enrichir la simulation avec l'ajout de davantage de paramètres : des maladies qui se propagent lorsque

deux populations se rencontrent, ou des effets de l'intervention humaine par la modification de l'environnement par exemple.

Il aurait été intéressant d'avoir des fonctions permettant de faire directement les moyennes de plusieurs simulations, pour ne pas à avoir à le faire de façon manuelle. On aurait pu également avoir des bibliothèques supplémentaires pour traiter de façon automatique les données obtenues à chaque simulation.

Conclusion

Malgré les difficultés rencontrées, j'ai beaucoup apprécié travailler sur ce projet. Il m'a donné l'occasion de réfléchir sur des aspects de modélisation globale et de trouver des façons d'interpréter des résultats qui ne sont pas très parlants au premier abord. J'ai pu également progresser en JavaScript, qui est un langage versatile et souvent demandé comme compétence sur les annonces d'embauche ou de stage.