Présentation de notre projet de Programmation orientée objet :

1 : Problématique

Le sujet III consiste à créer une simulation de fourmilière. C'est un système complexe qui change de façon saisonnière sur plusieurs années. La fourmilière est habitée par différentes classes de fourmis : les fourmis gardes, les fourmis récolteuses et les fourmis puéricultrices, chacune ayant un rôle spécifique. Les ressources disponibles dans la nature sont affectées de manière différente en fonction de la saison. Les fourmis ont une durée de vie d'un an et doivent faire face à des attaques aléatoires, et gérer leur consommation de ressources. Le but de la simulation est d'expérimenter différents paramètres et comportements pour observer l'évolution de la fourmilière.

2 : Architecture

2.1 : Classes et héritage

Comme nous devons utiliser la notion d'objet dans notre code, les classes sont un aspect important du programme. Ces classes contiennent des attributs, des méthodes, et éventuellement des sous-classes. Dans cette partie, nous allons regarder quels ont été nos choix de classes et d'attributs de classe. Les méthodes seront abordées dans la partie suivante.

Notre première classe est *Fourmi* : C'est une classe évidente pour créer des objets qui prennent la place des fourmis dans le code. Il y a deux attributs pour *Fourmi* :

- *rôle* : Vous verrez par la suite que nous avons également créé des sous-classes pour chaque rôle. Seulement, nous ajoutons l'attribut rôle dans la classe *Fourmi* pour faciliter l'appel des méthodes communes à tous les rôles de fourmi.
- *âge* : Les fourmis ont une durée de vie d'un an (4 saisons/cycles), cet attribut nous servira à connaître quand est-ce qu'il faut les "tuer" dans le programme.

Nous avons aussi trois sous-classes pour la classe *Fourmi*, une pour chaque rôle : *Garde*, *Récolteuse*, *Puéricultrice*. Ces sous-classes reprennent les attributs et méthodes de la classe Fourmi, et permettent de faciliter l'appel de fourmis ayant un rôle spécifique.

Enfin, notre classe la plus importante est la classe *Fourmilière*. En effet, une fourmilière peut être considérée comme un "objet", et c'est ici que nous avons mis en place la majorité des outils pour faire fonctionner notre programme. La classe Fourmilière possède 11 attributs,

dont 9 qui seront saisis en entrée, comme paramètres, déterminés par l'utilisateur :

- Nombre initial de fourmis : le nombre initial de fourmis dans la fourmilière est bien sûr un élément qui impacte beaucoup la survie de la fourmilière.
- Ressources dans la nature : ressources disponibles hors de la fourmilière, que les récolteuses peuvent récupérer. Ce sont ces ressources qui sont impactées par le changement de saison, et doivent d'abord être récoltées et amenées dans le stock avant d'être consommées.
- 3 Proportions de fourmis (pour chaque rôle) : sert à définir le pourcentage de fourmis pour chaque rôle dans une fourmilière (Garde, Récolteuse, Puéricultrice).
- Facteur d'attaques : Permet d'ajuster le nombre d'attaques atteignant la fourmilière chaque saison. Plus celui-ci est élevé, plus il diminue le nombre d'attaques.
- Les "cap", concernant le nombre de fourmis, la quantité de ressources dans la nature, et dans le stock. Ils permettent de limiter ces ordres de grandeur dans la simulation et de limiter le temps d'exécution.

Et 3 autres qui seront initialisés à la création de la fourmilière :

- Une liste de fourmis : Attribut très important, cette liste est composée des fourmis présentent dans la fourmilière, à la création de la fourmilière, chaque fourmi est ajoutée à cette liste avec un rôle dépendant de la probabilité définie en entrée.
- Nouvelles fourmis : Attribut qui servira à créer de nouvelles fourmis en fonction du nombre de puéricultrices. Il est initialisé à zéro. Une méthode sera mise en œuvre à l'aide de cet attribut pour en créer.
- Ressources en stock : Initialise les ressources disponibles à la création de la fourmilière à 0, évoluera en fonction de la consommation et de la récolte.

2.2 : Méthodes

Passons maintenant à la présentation des méthodes.

Fourmi :

Pour la classe *Fourmi*, la seule méthode intéressante est la méthode "*vieillir*", qui, à chaque saison, rajoute une saison supplémentaire à l'âge de la fourmi.

Cette méthode est liée à deux autres méthodes présentes dans la classe *Fourmilière* :

envieillir : cette méthode utilise "*vieillir*" de la classe Fourmi pour ajouter une saison de plus aux fourmis présentes dans la fourmilière concernée a chaque saison simulée.

tuer_anciens : Quand l'âge d'une fourmi atteint quatre saisons, cette méthode permet de la "tuer" en supprimant la fourmi vieille de la liste des fourmis.

Garde, *Récolteuse*, *Puéricultrice* :

Les trois sous-classes de *Fourmi* héritent des ces méthodes, et n'ont pas de méthode spécifique. Ce sont donc les mêmes que pour *Fourmi*.

Fourmilière :

La classe Fourmilière, elle, a beaucoup de méthodes importantes :

- Des méthodes qui renvoie la valeur d'un attribut :
    - *get_ressources_stock* : Pour avoir la quantité de ressources en stock dans la fourmilière.
    - *get_ressources_nature* : Pour obtenir la quantité de ressources disponible dans la nature.
    - *get_nombre_nouvelle_fourmis* : Pour connaître le nombre de fourmis que possèdent la fourmilière à un moment donné.

- Des méthodes permettant d'avoir la répartition des fourmi entres les rôles en en faisant le compte :
    - *get_nombre_gardes*
    - *get_nombre_recolteuses*
    - *get_nombre_puericultrices*
- Deux méthodes permettant d'obtenir la taille de la fourmilière (avec la longueur de la liste *fourmis*), et l'autre le nombre de naissances pour la saison en court dans la fourmilière (en récupérant la valeur de l'attribut *nouvelles_fourmis*)
    - *get_nombre_fourmis*
    - *get_nombre_nouvelle_fourmis*

Gestion de la consommation :

Tout d'abord, voici quelques méthodes qui seront utilisées ensuite dans les calculs de consommation, et qui permettent de fixer certains paramètres impactant la consommation.

- *facteur_recolte* : En fonction des proportions choisies pour les rôles, et de leur consommation respective, cette méthode calcule la quantité que chaque fourmi récolteuse doit récupérer pour nourrir la fourmilière.
- *sum_conso* : Calcule la consommation totale de la fourmilière, en utilisant la quantité de fourmis de chaque rôle.
- *cap_res* : Limite la quantité de ressources, en stock et dans la nature, en fonction du paramètre choisi, permet de limiter l'ordre de grandeur de ces valeurs, et ainsi de diminuer le temps d'exécution du code.

Mises à jour des quantité de ressources :

maj_ressources : C'est la méthode qui s'occupe de la gestion des ressources. Voici les différentes étapes de cette gestion :

Mettre en place la quantité de nourriture à manger pour les différents types de fourmis : La nourriture consommée que nous administrons aux fourmis au moment où nous exécutons cette méthode est leur nourriture pour toute la saison. La consigne exige que les gardes mangent plus que les récolteuses, qui elles-même mangent plus que les puéricultrices. Nous avons donc choisi de nourrir les gardes avec trois unités de ressources, les récolteuses deux, et les puéricultrices une. Ce à quoi s'ajoutent les fourmis juvéniles : Nous avons décidé de les nourrir avec une demi-unité de ressources. Puisque chaque fourmi puéricultrice élève une fourmi juvénile, il y a autant de puéricultrices que de juvéniles : c'est pourquoi, dans le programme, on "nourrit" les puéricultrices avec une unité ressource et demi.

Récolter les ressources : Le faire de manière intelligente, ce n'est pas si facile. Notamment, il ne faut pas que les fourmis s'emparent de toutes les ressources présentes dans la nature, car sinon il n'y en aura plus la saison d'après étant donné que le stock d'une saison se base sur celui de la saison d'avant. Ainsi, nous avons fait en sorte que quoi qu'il arrive, il reste toujours au moins cinquante unités de ressources disponibles à la fin d'une saison. Nous avons également décidé qu'une fourmi récolteuse peut récolter un nombre d'unités de ressource par saison défini dans *facteur_recolte*. Il y a donc deux cas de figure :

Si la quantité de ressources dans la nature est supérieure au nombre de récolteuses multiplié par ce facteur, plus cinquante : toutes les récolteuses peuvent recueillir les unités de ressources (c'est à dire le nombre de fourmis récolteuse fois le facteur récolte), et laisser le reste dans la nature.

Mais si la quantité de ressources ne respecte pas cette première condition, (mais doit quand même être supérieure à la limite), dans ce cas les récolteuses ne peuvent pas prendre autant de ressources que défini par le facteur. Elles prennent donc toutes les ressources de la nature, mais en laissant la limite calculée, (qui est dynamique pour suivre l'évolution de la fourmilière).

Les ressources récoltées vont ensuite dans le stock de la fourmilière. Et les quantités, en stock et dans la nature, sont ensuite mise à jour.

Nourrir les fourmis : Les ressources que consomment les fourmis viennent du stock de la fourmilière, désormais à jour. Il y a là aussi deux cas de figure :

S'il y a plus de ressources dans le stock que la quantité de nourriture pour nourrir toutes les fourmis, on enlève cette quantité au stock, elle est consommée. Toutes les fourmis ont été convenablement nourries, il n'y a pas de mort due au manque de ressources.

Mais si il n'y a pas assez de ressources dans le stock pour nourrir toutes les fourmis, alors certaines vont être tuées. L'idée est qu'on parcourt la liste des

fourmis, et que pour chacun des rôles, on va tuer suffisamment de fourmis pour combler le manque en répartissant la même quantité manquante pour chaque rôle.

Par exemple :

S' il manque 18 ressources dans le stock. On considère qu'il manque 6 ressources pour chaque rôle. Donc 2 gardes vont être tués (car il consomme 3 et 1/9 de 18 = 2), 3 récolteuses (conso = 2 et 1/6 de 18 = 3) et 6 puéricultrices (conso = 1 et 1/3 de 18 = 6)

Finalement, la méthode retourne des informations qui pourront être visualisées plus tard (le total de la consommation et le nombre de fourmis de chaque rôle ayant été tuées (0 si il n'y a pas eu de famine).

Gestion des attaques :

Maintenant, regardons les méthodes qui gèrent le déroulement des attaques extérieures sur la fourmilière :

get_nb_attaques : Sert à déterminer le nombre d'attaques sur la fourmilière lors d'une saison. C'est une valeur aléatoire, que nous avons choisi de calculer comme ceci : le programme choisit un nombre aléatoire entre 0 et le nombre de fourmis dans la fourmilière divisé par un facteur, saisi en paramètre, pour diminuer, ou augmenter l'impact de celles-ci.

gerer_attaques : Permet de simuler les attaques. La variable infos_attaques est une liste contenant deux éléments. Le premier élément est le nombre d'attaques sur la fourmilière, et le second élément est le nombre de morts dues aux attaques. Comment calcule-t-on les morts dues aux attaques ? Puisqu'un garde est capable de contrer une seule attaque à lui seul, il y a deux cas de figure :

Si il y a plus de gardes que d'attaques, dans ce cas aucune fourmi ne meurt. Le second élément de *infos_attaques* est égal à zéro.

Si il y a moins de gardes que d'attaques, alors toutes les attaques qui n'ont pas été contrées par un garde tuent une fourmi. Ainsi, on supprime autant de fourmis de la liste qu'il y a eu d'attaques non-contrées, et ce de manière totalement aléatoire.

Gestion des naissances :

nouvelle_fourmis : Ici, le but est de faire apparaître dans la liste les fourmis élevées par une puéricultrice.

Notre programme utilise la fonction "choice" de la bibliothèque random pour répartir les rôles aux nouvelles fourmis, en respectant les probabilités de rôles de la fourmilière.

Également, une liste "new" avec trois éléments nous permet de savoir combien de fourmis de chaque type a été créée. Il y a autant de nouvelles fourmis que de puéricultrices présentes dans la fourmilière. Quand les rôles sont tous répartis, on

ajoute le nombre de nouvelles fourmis à l'attribut "nouvelles_fourmis", tout en les comptant, afin de pouvoir afficher ces informations ensuite.

Vieillissement :

Les méthodes *vieillir* et *tuer_anciens* permettent de faire vieillir toutes les fourmis, et d'enlever les fourmis trop âgées de la liste des fourmis appartenant à la fourmilière, de les “tuer”.

Saisons :

Quatres méthodes sont créées pour simuler l’impact de la saison sur la quantité de ressources dans la nature. Ainsi une méthode par saison sera utilisée et appelée pour mettre à jour les ressources dans la nature.

Celles-ci sont appelées dans la méthode *saison*. qui prend en argument la saison concernée permet de gérer les ressources quelque soit la saison, et est utilisé lors de la simulation.

Au printemps les ressources dans la nature se voient être doublées. En été les ressources augmentent de 50%, en hiver elles baissent de 25%, et en automne elles n'évoluent pas.

Simulation :

simuler_saison est une méthode appelée dans une boucle, elle prend en entrée la saison actuelle de la simulation, et effectue la séquence suivante :

1 - Application des saisons sur les ressources dans la nature
2 - Vieillissement des fourmis
3 - Mort des fourmis trop âgées
4 - Création de nouvelles fourmis
5 - Attaques sur la fourmilière
6 - Mises à jours des ressources, Récoltes et Consommation

La méthode finit par retourner des informations concernant les attaques, et les naissances pour l’affichage qui suit.

Simulation :

Ensuite, nous avons créé une fonction pour la simulation, facilement exécutable, et dont l’affichage saisonnier complet est optionnel. De plus, nous calculons le temps d'exécution, et sauvegardons les informations de la simulation dans un dataframe, pouvant ainsi être analysé ultérieurement, et affiché. Quatres graphiques sont créés dans cette fonction :
Un montrant l'évolution du nombre de fourmis, générale et par rôle. Le second montre le nombre d’attaques, et de mort dues à celles-ci. le troisième affiche les quantités de ressources, et le dernier le nombre de naissances pour chaque type de fourmis.

3 : Personnalisations

Comme vous avez pu le voir précédemment, nous avons ajouté de nombreuses personnalisations, le sujet laissant une grande place à la créativité. Ainsi nous avons ajouté :

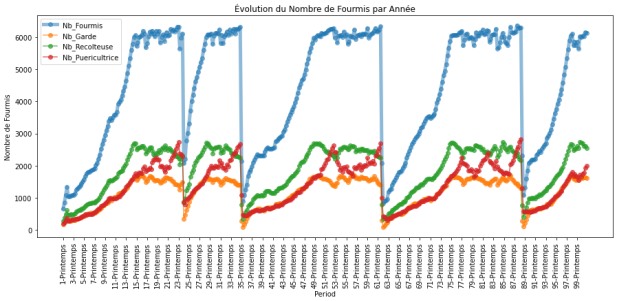
- Le réglage d'un facteur attaques influant sur leurs impacts
- Un facteur récolte adaptant la charge imposée à chaque récolteuse
- Une limite dans la récolte des fourmis qui est dynamique
- Une situation de famine en cas de manque de ressources
- Une part d'aléatoire dans la distribution des rôles
- Un cap concernant 3 valeurs : le nombre de fourmis, les ressources en stock, et dans la nature
- La récupération de nombreuses informations concernant la fourmilière
- La visualisation de ces mêmes informations

4 : Résultats

Quand nous exécutons notre code, nous obtenons plusieurs sortes de résultats. Le graphique ci-dessous est le type de résultat que nous obtenons le plus souvent. Il représente l'évolution du nombre de fourmis sur une période de dix ans. On constate que l'évolution du nombre de fourmis forme un cycle qui se répète plusieurs fois. Au début, le nombre de fourmis augmente drastiquement, sur les 5 à 10 premières années. Puis, il fait une chute libre et redescend presque autant. Cette boucle se produit une nouvelle fois : le nombre de fourmis augmente, dépasse même celui du printemps de la cinquième année, mais finit par rechuter lourdement entre l'hiver de la sixième année et le printemps de la septième.
Les causes de ces chutes soudaines et vertigineuses peuvent être variées. L'augmentation du nombre d'attaques, le manque de ressources etc… Les causes changent d'une simulation à l'autre. Nous vous invitons à tester le programme en modifiant les paramètres, et observer les changements.

Voici par exemple le graphique,
sur 100 ans cette fois, avec les paramètres suivant :

- 500 fourmis initiales
- 10 000 unités de ressources dans la nature
- 25% de gardes, 40% de récolteuses, 25% de puéricultrices
- Facteur attaque de 3
- Limites de fourmis à 6 000, de ressources dans le stock à 100 000, dans la nature à 60 000.



5 : Conclusion (Proposition d'extensions)

Nous aurions pu encore étendre la simulation, en ajoutant d'autres fonctionnalités :
Par exemple la création d'une 2ème fourmilière lorsque la 1ère vie longtemps. L'introduction de probabilités de victoires pour les gardes dans les attaques, augmentant ainsi leur impact. L'introduction de nouveaux évènements aléatoires impactant la fourmilière. Une adaptation de la récolte à la saison, supposant que les fourmis perçoivent cette information.