TP1: Le Village de Thierceville

Romain CAILLIERE & Cédric BURON

11 janvier 2022

Introduction

C'est une triste histoire bien connue : tous les cent ans, le village de Thierceville est envahi de lycanthropes se transformant en bêtes féroces. Toute personne mordue par une telle créature les rejoint aussitôt, rejoignant les rangs des créatures des ténèbres.

Mais cette fois-ci, les villageois ont décidé d'anticiper, et ont invité des apothicaires à s'installer dans le village. La réputation du village a aussi attiré un certain nombre de chasseurs de monstres qui comptent bien débarrasser les lieux de tout danger.

Dans ce TP, nous vous proposons de concevoir une simulation du village de Thierceville comprenant les villageois, les loups-garous, les chasseurs et les apothicaires. La simulation du village sera très simplifiée : l'environnement ne sera pas représenté. L'intéraction entre agents se limitera à des actions extrêmement simples et les agents ne chercheront pas à se coordonner. Ces différents aspects des systèmes multi-agents (SMA) feront l'objet du second TP.

Afin de modéliser le village de Thierceville, nous utiliserons la plate-forme `mesa`. `mesa` est une plate-forme multi-agents utilisée pour modéliser notamment des agents situés. Elle intègre entre autre un environnement graphique facile d'utilisation, ainsi que la possibilité de tracer des courbes permettant de visualiser l'évolution des métriques d'une simulation. `mesa` est écrite en python et inclut un client léger pour voir les graphiques et la simulation.

Mise en place

Pour ce TP et les suivants, vous aurez besoin d'une version récente de python 3, plus spécifiquement une version de python supérieure ou égale à la version 3.7. Si vous n'avez pas encore de version de python installée, je vous recommande Anaconda [1]. Vous pouvez également installer un IDE si vous le souhaitez, ou développer directement depuis un éditeur de texte en lançant les scripts depuis la ligne de commande.

Une fois python installé, vous devrez installer le package `mesa`. Pour ce faire, ouvrez un terminal et tapez :

```
pip install -r requirements.txt
```

En cas de besoin ou de doute, vous trouverez la documentation de `mesa` ici : `https://mesa.readthedocs.io/en/master/`.

Une fois `mesa` lancé, téléchargez importez le TP1 situé sur le git data-ensta :

```
git clone git@gitlab.com:romcai/thierceville-mesa.git
```

Décompressez-le si besoin et ouvrez le dossier dans votre explorateur/IDE. Vous verrez que deux fichiers sont inclus : l'un correspond au modèle python, et l'autre est le code javascript correspondant aux éléments graphiques les contenant.

Une fois le projet importé, lancez-le avec la commande :

1. `https://www.anaconda.com/`

```
python village.py
```

Cela lancera d'un côté le serveur python qui fera tourner le code du modèle et des agents, et de l'autre le code du client html/JavaScript. Cela devrait aussi lancer votre navigateur sur la page du simulateur. Si la page ne s"ouvre pas et que le serveur python s'est lancé, ouvrez votre navigateur et accédez à l'URL `http://127.0.0.1:8521/`.

L'interface graphique de mesa se présente comme la fig. 1. L'entête de la page continent le nom du modèle (ici Village). Le bouton `About` permet d'avoir accès à la description du modèle. Cela peut en particulier permettre d'expliquer à un utilisateur tiers ce qui est vu à l'écran et l'analyser. Sur la partie droite de l'entête, les trois boutons permettent de contrôler le déroulement de la simulation : un bouton permettant de lancer ou mettre en pause la simulation, un autre permettant de faire un unique tour et un troisième permettant de la remettre à zéro.

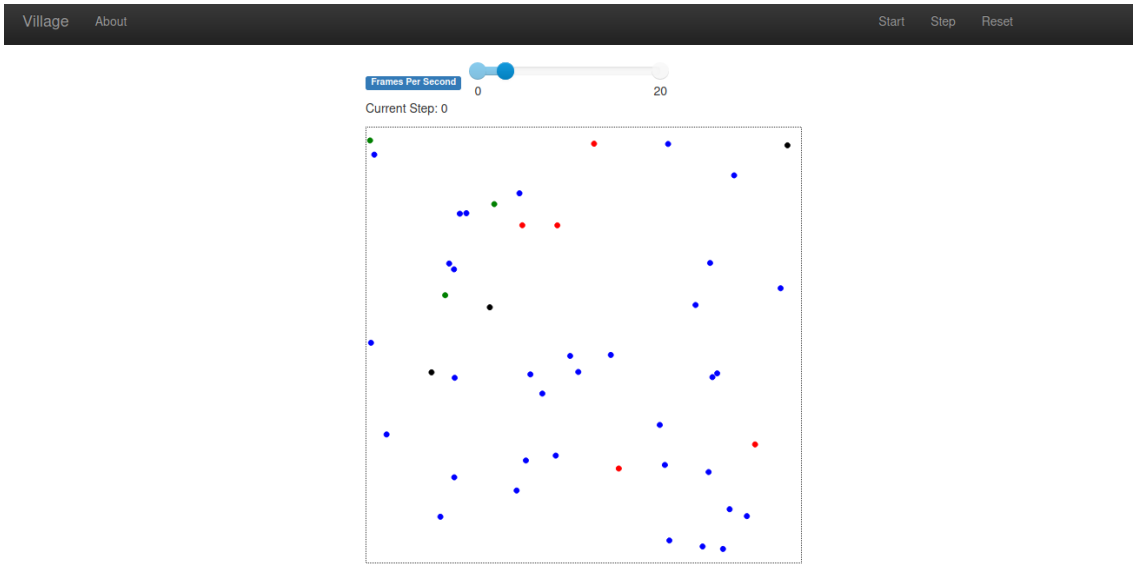

FIGURE 1 – Interface graphique de Mesa

Sous l'entête se trouve une barre permettant de régler le nombre de tours de simulation par seconde. Plus ce nombre sera important, plus la simulation ira vite. Notez cependant qu'un nombre de tours par seconde réglé à 0 permettra de faire tourner la simulation à sa vitesse maximale. Le nombre de tours écoulés se trouve sous la barre. Au-dessous se trouve la fenêtre de simulation, où l'on peut voir les agents. Après avoir observé le déroulement d'une simulation, revenez au code python. Notez que le fait de changer le modèle relancera automatiquement la simulation. Si vous voulez éviter que la simulation ne se relance de manière impromptue, je vous conseille d'arrêter python lorsque vous revenez à l'édition.

Mesa contient la description de plusieurs éléments :

Le modèle représente les éléments communs à la simulation, qu'il s'agisse des paramètres en termes de nombres d'agents ou des éléments qui en sont indépendants. Le modèle est aussi doté d'un *scheduler* qui est chargé d'activer les agents. Dans le cadre de ce cours, nous utiliserons systématiquement le `RandomActivation` scheduler qui active les agents dans un ordre aléatoire.

Les agents sont des classes qui héritent de la classe `mesa.Agent`. Les autres éléments incluent surtout des outils de visualisation : dans ce modèle, comme dans les suivants, vous verrez le canvas que nous utiliserons, c'est-à-dire l'espace dans lequel se déplacent les agents. Cet élément correspond à un élément présent dans le code JavaScript, de la même manière que les éléments comme les cercles et les rectangles utilisés pour représenter les agents.

Le bloc `main` sert quant à lui de définir la manière dont les blocs graphiques s'agencent. Pour le moment, seul un unique bloc graphique est présent (celui du simulateur). Nous en ajouterons d'autres dans la suite de ce TP. Le serveur sert également à régler les valeurs des options du modèle, ici le nombre de villageois. Nous verrons dans la suite du TP la manière de régler ces éléments directement dans l'interface graphique du client.

Vous verrez plusieurs éléments pour chaque agent :

- la méthode `step` indique ce que l'agent fait à son tour
- la méthode `portrayal method` indique la manière dont l'agent doit être représenté sur le simulateur.

Implémentation de la simulation

La simulation donnée est très simple, et donne seulement une représentation des agents `Villagers`. La première partie de ce TP consiste à la compléter.

Les lycanthropes

La première tâche consistera à implémenter le fait qu'une personne puisse être un lycanthrope. Pour cela, nous allons enrichir l'espèce personne. Ajoutez à la classe `Villager` un booléen indiquant si la personne est un loup-garou ou non. Faites en sorte que les lycanthropes soient affichés en rouge. Il doit y avoir 5 lycanthropes au départ de la simulation. Dans le modèle, il vous est possible d'ajouter un paramètre de la même manière que `n_villagers`. Ce paramètre doit figurer dans le constructeur du modèle `Village.__init__`, mais aussi dans la création de ce modèle, sous la forme d'une chaîne de caractère dans la liste constituant le dernier paramètre du constructeur de `ModularServer`.

```
class Village(mesa.Model):
    def __init__(self, n_villagers):
        mesa.Model.__init__(self)
        #...
        for _ in range(n_villagers):
            self.schedule.add(Villager(random.random() * 600,
                random.random() * 600, 10,
                random.randint(1, 600), self))
#...
if __name__ == "__main__":
        server = ModularServer(Village,
                               [ContinuousCanvas],
                               "Village",
                               {"n_villagers": 20})
        #...
```

Les lycanthropes peuvent se trouver dans deux états : transformés ou non. Afin de les différentier, changez la taille des lycanthropes transformés et faites la passer à 6. Initialement, les lycanthropes ne sont pas transformés. Ils se transforment de manière aléatoire, avec une probabilité de 10%. Pour ajouter un nouveau comportement au lycanthrope, il vous faudra modifier la méthode `step`.

Lorsqu'un lycanthrope est transformé, il peut s'attaquer à une autre personne si celle-ci est à une portée de 40. Modifiez à nouveau la méthode `step` de manière à permettre à un lycanthrope de s'attaquer à une personne. Pour implémenter ce nouveau comportement, je vous recommande vivement de vous appuyer sur des *list comprehensions*. Si vous ne connaissez pas les list comprehensions, *merci de l'indiquer dans le fichier* `responses.md`. L'autre difficulté de ce réflexe est qu'il agit sur l'autre personne. Lorsqu'un lycanthrope attaque une personne, il la transforme en lycanthrope.

Question 1– Comment avez-vous fait pour que l'autre agent soit modifié ? Cela vous paraît-il compatible avec la définition d'agent que vous avez vue en cours ? Argumentez votre réponse.

Les apothicaires

Créez une classe `Cleric`. Ces agents ont les mêmes capacités de déplacement que les villageois. Donnez-lui pour aspect un cercle de rayon 3 et de couleur verte. Les apothicaires ont un comportement de soin, leur permettant de rechanger un lycanthrope en humain à condition que ce dernier soit à une distance de 30 et qu'elle ne soit pas transformée. La simulation prend en compte un unique `Cleric`.

Les chasseurs

Créez une classe `Hunter`. Ces agents sont capables de se déplacer de la même manière que les villageois. Dans la simulation, les chasseurs seront représentés par un cercle noir de rayon 3. Implémentez le comportement de chasse des chasseurs : ils sont capables de tuer un lycanthrope si celui-ci est transformé et s'il se trouve à une distance de 40. Tuer un agent se fait en le supprimant du `Model.schedule`. Il y a 2 chasseurs dans notre simulation.

La simulation est maintenant complète ! Vous pouvez la lancer et en observer le résultat.

Question 2– Commentez le résultat de la simulation : Vers quoi le système converge-t-il ? En combien de cycles ? À votre avis, quel est l'impact de la présence de l'apothicaire ? Celui de la quantité d'agents de chaque espèce ? Justifiez votre réponse.

Expérimentations

Maintenant que vous avez exprimé des conjectures, il va falloir les tester. Pour cela, il va falloir améliorer les simulations.

Graphiques

Nous allons commencer par créer des indicateurs permettant de mesurer plus finement la manière dont le système évolue. Mesa intègre un environnement d'expérimentation que nous allons exploiter.

Nous allons laisser l'affichage de la simulation et afficher au-dessous les graphiques. Un graphique est un `VizualizationElement`, et plus particulièrement un `ChartModule`. En parallèle, il va falloir rassembler des informations que nous souhaitons afficher. Pour ce faire, nous allons utiliser un `DataCollector`. Le DataCollector est un nouveau champ du modèle. Attention, gardez le nom du champ que vous allez utiliser, il vous sera nécessaire par la suite.

L'utilisation du DataCollector se fait en l'initialisant dans le modèle. Le constructeur de DataCollector prend en entrée un dictionnaire de `model_reporters`, ayant d'un côté un nom (qu'il vous faudra aussi garder en tête) et une fonction associant au modèle une valeur. Notez que dans ce cadre, l'utilisation des `lambda` vous sera très utile. Si vous ne connaissez pas les lambda expressions, *merci de l'indiquer dans le fichier* `responses.md`.

Implémentez un graphique affichant le nombre de personnes (non lycanthropes) en fonction du temps. Lancez à nouveau la simulation. Vous voyez maintenant deux parties, incluant la courbe que vous avez créée. Lorsque vous lancez la simulation, vous voyez la valeur de votre métrique évoluer en direct, comme le montre la fig. 2.

Retournez à la fenêtre d'édition. Nous allons ajouter d'autres métriques à notre page, afin de créer un tableau de bord qui permette de se faire une meilleure idée de l'évolution du système. Tentez d'ajouter d'autres graphes à l'onglet. Ajoutez trois graphes au tableau de bord représentant le nombre de lycanthropes, le nombre de lycanthropes transformés et le nombre total d'agents.

Question 3– Enregistrez les courbes en cliquant avec le bouton droit sur la courbe, puis "Enregistrez l'image sous…" et comparez ces résultats à vos conjectures. Qu'en concluez-vous ?

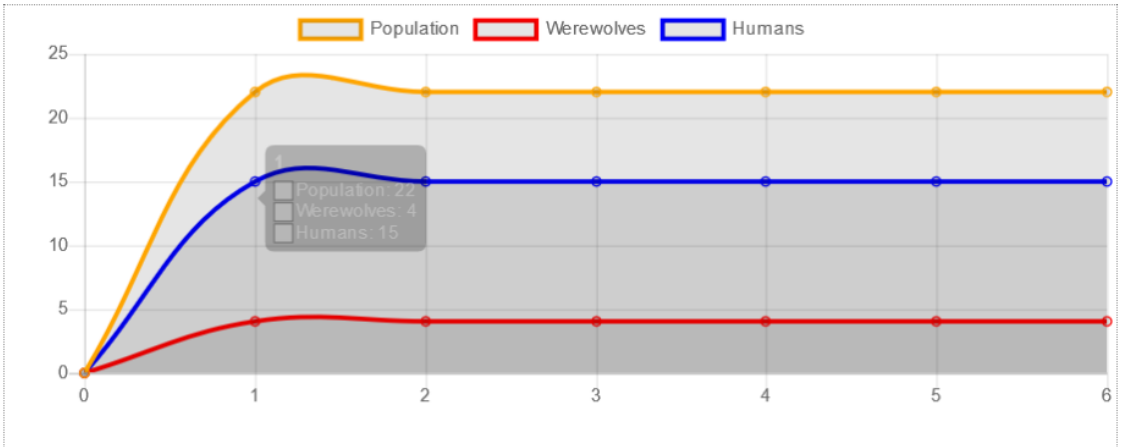

FIGURE 2 – Exemple de courbe

Variations de paramètres

Voyons maintenant comment on peut faire varier les paramètres pour observer leur influence sur l'expérimentation. Un moyen de faire est de changer manuellement les valeurs entrées dans le code et relancer les simulations les unes après les autres. Mais cela n'est pas pratique, et demande de revenir sans arrêt au code. Il est donc possible d'ajouter des paramètres à l'expérimentation afin de les faire varier directement dans la perspective correspondante. Pour cela, il faut les déclarer directement dans le lancement du serveur, dans les paramètres permettant de lancer le serveur et changer le type des paramètres par un `ModularVisualization.UserSettableParameter`. Ici, nous utiliserons des `slider`, des sélecteurs utilisés pour les plages de valeur :

```
mesa.visualization.ModularVisualization.UserSettableParameter('slider',
        "name of slider", default_value, minimal_value, maximal_value, interval)
```

Question 4– Ajoutez aux paramètres le nombre de villageois sains, le nombre de lycanthropes, le nombre de chasseurs et le nombre d'apothicaires. Enregistrez les courbes qui vous paraissent pertinentes et commentez-les. Cela correspond-il à vos hypothèses ? Qu'en concluez-vous ?
Question 5– Sans faire les expériences associées, quels sont, selon vous, les paramètres qui auraient une influence sur le résultat de la simulation ? Argumentez ces hypothèses.

Plan d'expérience

Même en ajoutant les paramètres à la fenêtre de simulation, il peut s'avérer long et fastidieux de tester une batterie de valeurs pour un même paramètre. mesa intègre le moyen de créer plusieurs simulations en faisant varier (ou non) ces paramètres. Pour cela, nous allons utiliser les `batch`.

Dans cette partie, nous cherchons à évaluer l'impact du nombre d'apothicaires dans un village avec 1 chasseur.

Question 6– Formulez une hypothèse argumentée sur le résultat de cette expérience.

Pour cela, créez une nouvelle fonction `run_batch()` après la fonction `run_single_server()`. Cette fonction créera d'abord un dictionnaire qui donnera pour chaque paramètre de la fonction `__init__()` du modèle `Village` une plage de valeur. Les valeurs de `n_villagers`, `n_werewolves`, `n_hunters`, respectivement à 50, 5 et 1. Le paramètre `n_clerics` variera lui dans `range(0, 6, 1)`. Une fois ce dictionnaire créé, instanciez un `Batchrunner`. Voici la signature de son constructeur : `BatchRunner(model, params_dict, model_reporter)`. Vous utiliserez le même `model_reporter` que pour le modèle individuel. Lancez le batchrun grâce à la méthode `BatchRunner.run_all()`

Ce `model_reporter` vous permettra de récupérer une `pandas.DataFrame` en utilisant la méthode `BatchRunner.get_model_vars_dataframe()`.

Question 7– Comment interprétez-vous les résultats de cette expérience ? Qu'en concluez-vous ?

Bonus

Supposons que l'on souhaite faire varier toutes les variables, sur toutes les valeurs permises par les `slider` ; quel problème voyez-vous ? Comment peut-on le résoudre ? Cherchez dans la documentation de mesa et implémentez votre solution.

Si certains éléments du framework mesa vous ont posé problème, merci de l'indiquer à la fin du fichier `reponses.md`