Simulation d'un Ecosystème

-

Rapport Informatique – 1ère Partie

-

Xavier Quélard, Guillaume Thiriet

Groupe 8

17 Mars 2016

Table des matières

Introduction

Le programme présenté tente de simuler le fonctionnement et l'évolution d'un écosystème simple constitué de deux sortes d'animaux : des herbivores se déplaçant en troupeau et des prédateurs solitaires. Les herbivores se nourrissent d'herbe, une ressource omniprésente sur le sol de la carte tandis que les carnivores se nourrissent d'herbivore.
L'objectif de ce programme est de faire évoluer de manière autonome cet écosystème tout en faisant en sorte que celui-ci soit relativement stable (i.e. Faire en sorte que tous les animaux d'une même espèce ne disparaissent pas au bout de quelques tours). Pour ce faire, chaque animal apparait sur la carte avec plusieurs compteurs chiffrés correspondant à l'état de ses besoins (i.e. faim et soif) et à son espérance de vie. En fonction de ces compteurs et de son environnement direct (appelé voisinage), l'animal doit réaliser des actions cohérentes dans le but d'assurer sa propre survie. Ainsi, chaque animal se déplace, se nourrit, se reproduit… Bref, chaque animal vit de manière autonome.

Ce programme est intéressant dans le sens où il réussit à faire cohabiter et évoluer dans un même ensemble des entités ayant des besoins et des caractéristiques différentes et cela de manière cohérente. Notre simulation se veut simple tout en restant relativement réaliste. Pour cela, nous avons limité le nombre de texture à quatre (eau, terre, herbe, pierre) et le nombre d'animaux différents à deux (Des Herbivores et des Prédateurs). Une plus grande diversification de ces variables n'aurait rien apporté d'intéressant en termes de programmation et aurait rendu le problème de l'équilibrage beaucoup plus difficile. Cependant, au-delà de son apparence amusante et ludique, ce programme n'a rien d'anodin. Si, dans le cas présent seul deux sortes d'animaux déambulent dans une carte fictive, ce genre de programme pourrait très bien être appliqué à de véritable réserve naturelle afin de gérer au mieux l'écosystème qui s'y trouve.

1 Description générale du problème :

1.1 Introduction

L'objectif de ce projet était de simuler un écosystème sur une zone limitée qui constitue dorénavant notre *carte*. Ainsi, plusieurs sortes d'animaux devront évoluer et survivre dans cet environnement et pour ce faire, ils devront se nourrir et boire régulièrement afin d'éviter une mort précoce. Chaque animal dispose d'un jeu de caractéristique (vitesse, vision, …) et d'un comportement spécifique variant d'un type a l'autre (Herbivore ≠ Prédateur)

Par ailleurs, le cahier des charges du projet exige que certaines figures informatiques soient présentes dans nos codes. Il s'agit des figures suivantes :

- Découpage du programme en classes
- Utilisation de l'héritage
- Implantation de structure dynamique
- Tests de toutes les classes intermédiaires

1.2 Description des différentes hypothèses réductrices choisies

La simulation d'un écosystème est un projet complexe qui peut vite se transformer en véritable casse-tête si ce dernier est trop ambitieux. Compte tenu de sa difficulté et du temps alloué pour l'écriture des programmes principaux (1 mois et 10 jours), certaines hypothèses réductrices ont dû être réalisée.

1.2.1 Une biodiversité très limitée

Le nombre de texture pour la carte est limité à quatre (eau, terre, herbe, pierre) et le nombre d'animaux différents à deux (Herbivores en troupeau et Prédateurs solitaires). Une diversification de la biodiversité n'apporte en effet rien d'intéressant en termes de programmation.

2 Description de l'application :

2.1 Présentation générale du programme

Notre programme est constitué de 4 classes principales constituant la colonne vertébrale de notre écosystème. Il s'agit des classes *Animal()*, *Etat()*, *Map()* et *AStar()*. Leurs fonctions et mécanismes fondamentaux sont expliqués si après.

Par ailleurs, nous disposons aussi de quatre objets *MAP*, *LIVING, CORPSES* et *Ecosysteme()* dont la fonction est de traiter et stocker les informations caractérisant l'état, la position… de chacun de nos animaux.

Ainsi, *MAP* est un objet issu de la classe *Map()* dont la fonction est de traiter toutes les informations utiles à la constitution de la carte de notre écosystème.
LIVING est une simple liste dont la fonction est de stocker une liste de l'ensemble des êtres vivants présents dans notre écosystème.
CORPSES est une liste chainée dont la fonction est de stocker une liste de l'ensemble des animaux morts durant la simulation.

De ces trois objets nait le quatrième objet : la classe *Ecosysteme()* servant tout simplement de conteneur aux 3 objets précédents (Cela permet d'alléger l'écriture des classes et méthodes faisant appel aux 3 objets précédents).

2.2 Description des principales Classes et Méthodes :

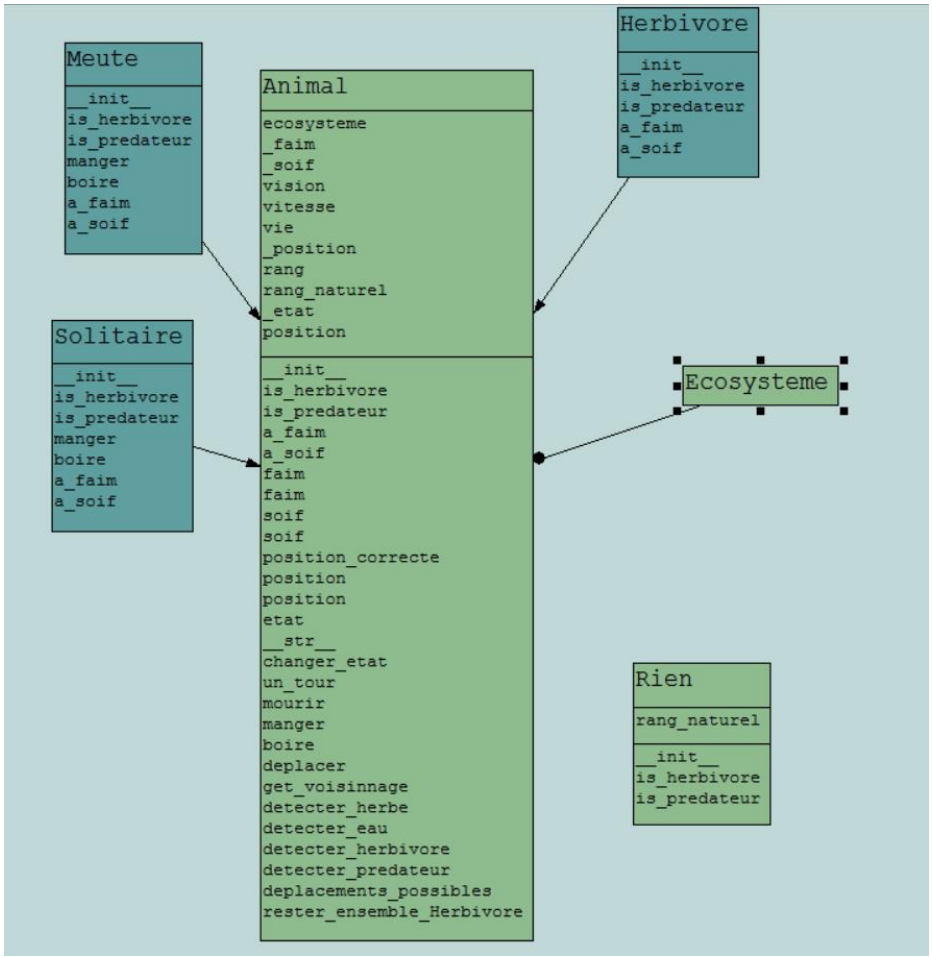
2.2.1 La classe Animal

Ce qu'elle fait : La classe *Animal()* est un pilier de notre écosystème. Son but est de décrire un animal du point de vue de ses caractéristiques intrinsèques (faim, soif, vision, vie…) et des actions basiques que chacun d'eux doit être capable de réaliser (i.e. se nourrir, boire, se déplacer, repérer des Herbivores, des Prédateurs). Attention, cette classe ne définit en aucun cas le comportement d'un animal, cette tâche est laissé à une autre classe : la classe *Etat()*.

Ce qu'elle contient : La classe *Animal()* contient de nombreuse méthodes dont le but est de décrire les caractéristiques intrinsèques de chaque animal. Ainsi, elle possède les méthodes suivantes *faim()*, *soif()* et *position()* dont le but est de définir la fourchette de valeur que peuvent prendre ces variables. Par suite, elle possède une méthode *un_tour()* dont le but est de décrémenter les compteurs de faim, de soif et de vie à chaque tour.

Cette classe contient aussi l'ensemble de méthodes permettant de réaliser les actions de bases nécessaires pour définir plus tard les comportements des dits animaux. Ainsi, sont définis les méthodes suivantes *manger()*, *boire()*, *mourir()*, *deplacer*, *detecter_eau()*, *detecter_herbivore()* et *detecter_predateur()* permettant de réaliser les actions éponymes.

Par ailleurs, la classe Animal contient aussi deux sous-classes héritant de la classe Animal et caractérisant de manière unique chacune des sortes d'animaux constituant notre écosystème. Ce sont les sous-classes *Herbivore(Animal)* et *Solitaire(Animal)*. Ces méthodes définissent notamment les seuils de faim et de soif des deux espèces.



2.2.2 La classe Map

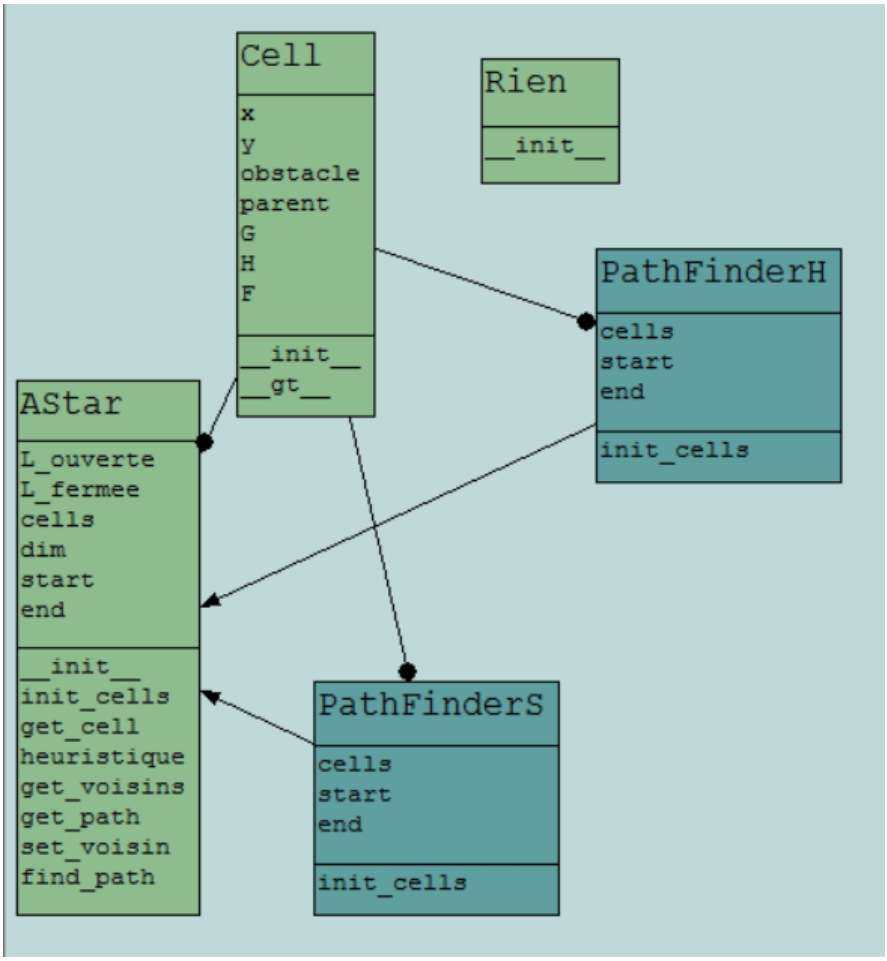
Ce qu'elle fait : La classe *Map()* est une classe permettant de générer une Carte dans laquelle vont pouvoir circuler nos animaux. Cette carte prend la forme d'une Matrice de liste de 2 éléments. Le premier élément de cette liste correspond à un indice codant la terrain (0 = sol, 1 = herbe, 2 = eau). Le second élément correspond à une instance de la classe Animal (Une sous-classe de la classe Animal) spécifiant la présence ou non d'un animal sur la case correspondante. Si un Herbivore ou un Prédateur est présent sur la case, les classes utilisées sont les sous-classes *Herbivore(Animal)* ou Solitaire*(Animal).* Dans le cas contraire (i.e. aucun animal n'est présent sur la case), une classe *Rien()* est utilisée.

Ce qu'elle contient : La classe *Map()* contient l'ensemble des méthodes permettant aux animaux définies dans la classe *Animal()* d'interagir avec la carte Map. Ainsi, la méthode *voisinage(position,vision)* renvoi le voisinage d'un animal situé sur une case repéré par position. Les méthodes *deplacer(position1, position2)* et *suppression(position)* permette quant à elle de déplacer un animal sur la carte/supprimer un animal de la carte.

2.2.3 La classe AStar

Ce qu'elle fait : La classe *AStar()* contient le Pathfinder. Ce Pathfinder a pour objectif de gérer les déplacements de chaque animal lorsque la position d'arrivée est connue (Lors de la traque d'une proie par exemple, la position de la proie étant alors la position d'arrivée) Ce Pathfinder trouve son application dans la classe *Etat()* lors de la recherche d'eau ou de nourriture notamment. Attention, la fonction de cette classe est différente de la méthode *deplacer()* définie dans la classe *Map()*. Dans notre cas, l'animal ne se « téléporte » pas, il se déplace case par case en contournant les obstacles.

Ce qu'elle contient : La classe *AStar()* contient un ensemble de méthode ne servant qu'à faire fonctionner le Pathfinder AStar. Elles ne seront plus jamais utilisées par la suite contrairement à de nombreuses méthodes définit dans la classe *Map()* et dans la classe *Animal()* servant pour le codage des comportements des différentes sortes d'animaux dans la classe *Etat()* (Cf. Paragraphe suivant).
Ce programme a été inspiré du Pathfinder AStar trouvé sur le site Web suivant http://www.laurentluce.com/posts/solving-mazes-using-python-simple-recursivity-and-a-search/
A la suite de la classe *AStar()*, 2 sous-classes filles *PathFinderH(AStar)* et *PathFinderS(AStar)* définissent un Pathfinder particulier pour chacune de nos 2 espèces. Ainsi, le Pathfinder appliqué aux Herbivores n'est pas le même que le Pathfinder appliqué aux Prédateurs Solitaires. Ces Pathfinders seront utilisés par la suite dans les différentes sous-classes d'*Etat()* définissant les divers comportements de nos animaux.



2.2.4 La classe Etat

Ce qu'elle fait : La classe *Etat()* est la classe décrivant le comportement de nos 2 sortes d'animaux (Herbivore et Prédateur Solitaire) en fonction de leur niveau de soif, leur niveau de fin et de leur voisinage. Le comportement de chaque animal est mis à jour à chaque tour et définit l'ensemble des actions à exécuter pendant ce tour. Cette classe ainsi que les sous-classes qui lui sont associé utilisent massivement les méthodes crées dans la classe *Map()*, *Animal()* et *AStar()*, c'est pourquoi elles ont été codées en dernier.

Ce qu'elle contient : La classe *Etat()* contient de nombreuses sous-classes héritant chacune de la classe *Etat()*. Chacune de ces sous-classes correspond à un comportement particulier. Ainsi, les sous-classes *Herbivore_normal(Etat)*, *Herbivore_faim(Etat)*, *Herbivore_soif(Etat)* décrivent le comportement des herbivores lorsque ces derniers sont repus, affamés ou assoiffés. Ils existent des sous-classes analogues nommées *Solitaire_normal(Etat)*, *Solitaire_faim(Etat)*, *Solitaire_soif(Etat)* pour coder les comportements des prédateurs solitaires.

3 Conclusions :

3.1 Tests unitaire

La plupart des méthodes et sous-classes utilisées dans les classes principales *Map()* et *AStar()* ont été testé une par une. L'objectif était d'identifier puis de résoudre les exceptions ainsi que de réparer tout programme défaillant. Pour cela, l'utilisation *d'assertion* (effectué par l'instruction « *assert* » en python) a été fortement privilégié. En effet, ces dernières ont l'avantage d'être activable/désactivable à volonté une fois les tests terminés. Ces assertions ont été pensé de la manière suivante. Une méthode est testée, si le résultat renvoyé est « *False* » : une exception est levée et le testeur en est informé par le biais de l'instruction « *assert* ». Dans le cas où la méthode testée fonctionne, l'instruction « *assert* » n'est pas utilisé et l'utilisateur est averti du bon fonctionnement de la méthode par le biais d'un petit texte renvoyé par l'instruction « *print* » suivant l'instruction « *assert* ».

Concernant la classe *Etat()*, les tests ont été effectués différemment. Un mini-écosystème muni d'un animal de chaque espèce a été créé. Toutes les sous-classe héritant d'*Etat()* ont ensuite été testée une par une à l'intérieur de ce petit écosystème en faisant varier les caractéristiques des animaux concernées.

3.2 Ce qui doit encore être fait

Certaines méthodes/sous-classes de notre programme n'étant pas encore suffisamment au point, nous avons préféré les laisser à l'état de commentaire pour l'instant. Ces méthodes/sous-classes seront traitées dans les jours à venir mais ne sont pas fonctionnelles actuellement. Les herbivores ne voyagent pas encore en troupeau come cela étaient prévu initialement.

3.2.1 Méthode rester_ensemble_Herbivore()

La méthode *rester_ensemble_Herbivore()* est écrite mais doit encore être déboguer et insérer dans les comportements décrits dans la classe *Etat()*.

3.2.2 Méthode fuite()

De même pour le comportement de fuite de ces derniers. La méthode décrivant le comportement de fuite est écrite mais doit encore être déboguer et appliquer à une nouvelle sous-classe *Herbivore_fuire(Etat)*.

3.3 Perspectives et améliorations

3.3.1 Un Ecosystème plus varié

La création de deux autres sortes de prédateur a été envisagées mais ces derniers sont restés à l'état de pensées et de schémas par manque de temps. Il s'agissait d'un prédateur *charognard* se nourrissant uniquement d'animal mort et d'un prédateur chassant en *meute*. Leur création aurait été intéressante puisque leur manière de vivre différait radicalement de celle du prédateur solitaire et de celle de l'herbivore en troupeau crées dans notre projet. Le code décrivant le comportement du prédateur chassant en meute a été initié mais nous n'avons cependant pas eu le temps de trouver une méthode algorithmique efficace pour que ces derniers se regroupe, chasse de manière réaliste en encerclant leur proie et se nourrisse, tout cela en « petite » meute de 7 à 10 membres maximum.

3.3.2 Une frontière pour notre réserve naturelle

Constituer la frontière de notre écosystème fut un problème difficile à résoudre puisque cette dernière entrait en conflit avec toutes les méthodes utilisant la caractéristique « vision » de l'animal. Deux solutions se sont offertes à nous.
La première consistait à faire en sorte que les animaux ne puissent pas se rapprocher de la frontière de manière à ce que toutes les cases présentes autour d'eux soient accessible par « champ de vision » (Ainsi, un animal ayant une vision de 2 ne pourra s'approcher à moins de 2 cases de la frontières).
La seconde solution dîtes solution de « la chaine de montagne » consistait à entourer notre carte d'une chaine de montagne intraversable (i.e. des cases de type « Pierre ») suffisamment large pour que toutes les cases présentes autour des animaux côtoyant ces montagnes soient accessible par « champ de vision ».

Nous avons finalement opté pour la seconde solution plus rapide à coder et permettant en plus la gestion du problème des déplacements en bordure de la carte. Un animal ne pourra en effet jamais sortir de la carte puisque cette dernière est cernée par une chaine de montagne infranchissable. En revanche, bien que plus efficace et plus facile à coder, cette méthode n'est pas des plus réalistes et aurait dû laisser sa place à la méthode numéro une si le temps l'avait permis.

3.3.3 De l'amour dans l'air ??

L'accent a été porté sur la cohabitation et l'évolution sur une carte fictive de nos 2 espèces d'animaux. Mais rien n'a été pensé pour renouveler les générations et par conséquent notre carte finit inévitablement par se transformer en un grand cimetière sans trace de vie. Dans l'optique de créer un véritable écosystème viable sur la durée, il serait intéressant de concevoir un système de naissance. Il serait en effet amusant de voir si une espèce prend le dessus d'une autre ou si tout finit (comme nous le souhaitons) par s'équilibrer naturellement.
Nous songions, pour cela à ajouter un paramètre « reproduction » établit à 1 ou 2 dans les caractéristiques de chaque espèce (Classe *Animal()*) représentant le nombre d'enfant potentiellement créable par un animal au cours de sa vie. La naissance d'un nouvel être pourrait alors potentiellement avoir lieu à chaque tour et passerait par le biais d'une instruction du type *random.randint()*. Evidemment, cela n'est pas très réaliste puisqu'il n'y a pas de Mâle ni de Femelle et que toutes naissance à lieu de manière aléatoire mais il s'agit d'une méthode simple et efficace pour palier au problème de la naissance.

Ouverture :

Pour conclure, nous allons exprimer très simplement nos objectifs concernant la 2ème partie de ce Projet Informatique.

- La création d'une carte dynamique 2D de notre écosystème dans laquelle nous pourrons voir nos animaux naître, vivre et mourir. Cette interface graphique sera créée en utilisant une de bibliothèques suivantes (PyQT, wxPython ou Tkinter…). Le choix n'est pas encore arrêté.
- Instaurer une gestion des naissances par la création d'une sous-classe *Reproduction()* dans la classe principal *Etat()* s'occupant des comportements de nos divers animaux
- Créer une nouvelle espèce d'animaux Charognards comme l'avait suggéré le cahier des charges du Projet Ecosystème.
- Eventuellement, si le temps nous le permet, améliorer le système de frontière en remplaçant l'astuce de la « *chaine de montagne* » par un système de « *frontière répulsive* »
  (Cf. Perspectives et améliorations)