La programmation des agents d'un Système Multi-Agent

Enoncé de la partie du projet à réaliser en binômes

`rev.01 - 14 novembre 2014`

Contenu

Résumé

Ce document décrit la partie du projet que nous vous demandons de réaliser par binômes. Il s'agit de la programmation des agents.

Introduction

Durant les huit premières séances, nous avons progressivement programmé l'environnement du SMA (E5), ainsi que défini la structure de données du système dans son ensemble (E6). Le code Python correspondant à cette première partie du projet est disponible sur l'Université Virtuelle (`http://uv.ulb.ac.be/course/view.php?id=34901`). Il est destiné à vous servir de base pour la programmation de la population d'agents :

1. Nous vous demandons de vous servir des fonctions mises à disposition par le code fourni et de ne pas les réécrire. En effet, la lecture de votre code nous sera largement facilitée par l'utilisation de fonctions "standards".
2. Nous vous suggérons fortement d'adopter la même approche de construction *par couches* successives pour les agents et la population. Il y a, en effet, des similitudes très fortes entre les cellules et les agents, d'une part, ainsi qu'entre l'environnement et la population d'agents, d'autre part.

1 Fonctionnement de notre SMA

Cette section résume brièvement le fonctionnement d'un Système Multi-Agent (SMA), tel qu'il a été présenté lors de la première séance (cf. slides de cette séance, disponibles sur l'Université Virtuelle).

Le principe général d'un SMA est de simuler l'évolution d'une population d'agents placée dans un environnement. La dynamique du système est donnée par deux ensembles de règles s'appliquant respectivement à l'environnement et à la population d'agents.

Les paragraphes ci-dessous résument les caractéristiques les plus importantes des trois éléments constitutifs principaux du SMA que nous considérons dans le cadre de notre projet.

1.1 L'environnement

L'environnement est un quadrillage de cellules. Nous le représentons comme une matrice carrée, dont le nombre de colonnes (ou de lignes) représente la taille. Chaque élément de cette matrice sera appelé une *cellule*.

Une cellule est une surface élémentaire de l'environnement, caractérisée par des propriétés constantes ou variables. Voici les propriétés que nous considérons ici :

— `sugar_capacity` : La capacité d'une cellule représente la quantité maximum que peut contenir une cellule. Le phénomène de régénération de sucre qui a lieu à chaque cycle augmente le niveau de sucre, jusqu'à ce que la capacité de la cellule soit atteinte. La distribution spatiale de la capacité sur l'ensemble des cellules de l'environnement créera des zones plus ou moins riches en sucre, comparables à des zones plus ou moins fertiles. On peut supposer que les agents auront tendance à se concentrer dans les zones plus fertiles (à capacité de sucre plus élevées).

— `sugar_level` : Il s'agit du niveau de sucre d'une cellule à un moment donné. Il varie, dans l'intervalle [0,`sugar_capacity`], principalement sous l'influence des agents qui consomment le sucre et de la règle de régénération qui s'applique à la cellule à chaque cycle.

— `present_agent` : Cette propriété référence l'agent qui serait éventuellement présent sur la cellule. Lorsqu'il n'y en a pas, cette propriété vaut `None`.

1.2 La population d'agents

La population est un ensemble d'agents. Un agent est une entité placée dans l'environnement et qui y évolue. Dans le cadre de ce projet, nous considérons des agents relativement simples qui, à chaque cycle, consomment du sucre et se déplacent.

Chaque agent est caractérisé par des propriétés intrinsèques (son métabolisme, son horizon de vision), qui lui sont aléatoirement attribuées lors de sa création. En plus de ses propriétés constantes, il a également des caractéristiques qui évoluent au fil du temps (exemples : sa position, son niveau de sucre, etc.).

Voici les propriétés que nous considérons ici :

— `metabolism` : Le métabolisme d'un agent représente la quantité de sucre qu'il doit consommer à chaque cycle pour survivre.
Notes :
  — Lorsqu'un agent ne trouve pas de cellule lui fournissant la quantité de sucre nécessaire en un cycle, il peut puiser dans son éventuelle réserve de sucre pour combler le déficit. S'il n'a pas assez de réserves de sucre pour combler ce déficit, il meurt immédiatement.

— `vision` : La vision d'un agent représente la distance (en nombre de cellules) à laquelle il peut scruter son environnement. Concrètement, un agent ayant une vision de 3 pourra "interroger" les cellules qui se trouvent jusqu'à 3 cellules de lui pour connaître leur niveau de sucre.
Notes :
  — La vision d'un agent est fortement directionnelle : il ne peut voir qu'horizontalement ou verticalement (c'est-à-dire, dans une représentation matricielle, selon la ligne ou la colonne de la cellule sur laquelle il se trouve).

— N'oubliez pas que le quadrillage a les propriétés topologiques d'une tore : la vision d'un agent peut "recommencer de l'autre côté" de la matrice lorsque le "bord" de la matrice est atteint.

— `sugar_level` : Le niveau de sucre représente l'énergie (accumulée) de l'agent. Selon la règle de consommation programmée, un agent peut également prendre plus de sucre qu'il ne lui faut pour survivre. Il peut ainsi accumuler du sucre, lui permettant de puiser dans ses réserves lorsqu'il se retrouve sur une cellule qui ne permet pas de subvenir à son besoin de sucre par cycle.

1.3 Les règles

L'ensemble des règles, appliquées respectivement aux agents ou aux cellules, décrit le comportement du SMA.

Les règles des cellules Elles décrivent la manière dont une cellule réagit ou évolue. Le mécanisme principalement à l'oeuvre dans la cellule est la *régénération*. A chaque cycle, le niveau de sucre d'une cellule se régénère automatiquement, jusqu'à ce que ce niveau atteigne la capacité de la cellule (cette capacité une l'une des ses caractéristiques constantes principales).

Les règles des agents Elles décrivent la manière dont un agent se comporte. Les comportements principaux d'un agent sont ses *déplacements* et sa *consommation de sucre*.

2 Enoncé du projet

2.1 Programmation des agents

Le premier objectif de "votre" partie du projet (celle que vous réalisez en binômes) est de compléter le code source qui vous est fourni afin d'*obtenir un SMA* ***fonctionnel***. Concrètement, il s'agira donc de programmer principalement les agents ainsi qu'une population d'agents de manière à pouvoir effectuer des simulations. Les questions ci-dessous sont destinées à vous guider sur ce chemin et ne représentent **pas** des questions auxquelles il suffirait de répondre point par point pour réussir.

L'agent

1. Déterminez une structure de données qui représente un agent (ses propriétés).
2. Programmez les fonctions d'interface ("Getters and setters" dans le code fourni) qui vous permettront d'interagir facilement avec un agent.
3. Programmez les fonctions de base d'un agent : la consommation de sucre et son déplacement. A ce stade, il n'y a pas de stratégie, mais uniquement l'opération qui consiste à déplacer un agent et à le faire consommer le sucre de la cellule sur laquelle il se trouve. Pour rappel, ces fonctions utiliseront les "get" et "set" du point précédent.
4. Programmez des *règles* qui, pour chaque agent, déterminent ses comportements : la consommation/accumulation de sucre et son éventuel déplacement.

   ***Note** : Nous vous conseillons de commencer par programmer des règles très simples qui vous permettront de valider le programme dans son ensemble. Vous aurez ensuite tout le loisir de concevoir des mécanismes et des stratégies plus évoluées (cf. ci-dessous).*

La population d'agents

1. Déterminez une structure de données qui représente une population d'agents (ses propriétés).
2. Programmez les fonctions d'interface ("Getters and setters" dans le code fourni) qui vous permettront d'interagir facilement avec la population dans son ensemble.
3. Programmez une fonction d'application de règles qui applique l'ensemble des règles d'agents à chaque agent. Notez que l'ordre des agents dans la population n'est pas neutre si vous utilisez une liste d'agents pour représenter la population (ce que nous vous conseillons) et que vous appliquez les règles selon l'ordre induit par cette liste. Pour diminuer ce biais, nous vous demandons d'appliquer les règles aux agents dans un ordre aléatoire : l'ordre des agents sera généré à chaque cycle par une permutation aléatoire des indices.

2.2 Programmation de règles applicables aux agents

Avertissement : Arrivés à ce stade, vous devriez avoir un SMA opérationnel. Nous vous conseillons de ne pas aborder cette partie du projet si votre système ne fonctionne pas encore.

Nous vous demandons à présent de programmer des règles plus spécifiques liées aux agents. Notez que les stratégies décrites ci-dessous demandent de programmer des règles de consommation de sucre *et/ou* de déplacement.

Commençons par un agent "égoïste" (ou "anxieux de la vie", si vous préférez), soucieux uniquement de son bien être individuel. Cet agent consommera à chaque cycle le maximum de sucre auquel il aura accès et essayera de se constituer une réserve de sucre la plus importante possible. Nous considérerons deux cas de figures :

RA1 À chaque cycle, un agent peut se déplacer aussi loin que porte sa vision. La stratégie la plus efficace (à titre individuel) sera par conséquent de choisir, pour son déplacement, la cellule dans son champ de vision qui offre le plus de sucre et de consommer tout le sucre qui s'y trouve.

RA2 À chaque cycle, un agent ne peut se déplacer que d'*une seule* cellule, quand bien même sa vision lui permet d'explorer un horizon plus éloigné (par exemple, à trois cellules, si sa propriété "vision" vaut trois). Apparaît ici la nécessité d'une stratégie un peu plus évoluée, qui doit faire le compromis entre la promesse d'un gain futur (une cellule à niveau de sucre élevé à plus d'une cellule de distance), la nécessité de "traverser" une zone plus pauvre en sucre et, enfin, le risque de voir le sucre "convoité" être consommé par un autre agent qui atteindrait la cellule plus rapidement.

Nous compliquons encore la donne. Jusqu'ici, seul l'intérêt propre d'un agent entrait en ligne de compte pour sa stratégie de déplacement. Nous considérons à présent un agent soucieux d'une répartition équilibrée des resources avec les autres agents qui l'entourent. Si sa survie reste sa première priorité, il adaptera néanmoins son comportement pour que les autres agents de la population puissent également survivre.

RA3 Un agent peut à nouveau se déplacer aussi loin que porte sa vision et il choisira la cellule qui lui permet de subvenir le plus ses besoin, c'est-a-dire qui offre un niveau de sucre le plus proche possible, mais au minimum équivalent à son métabolisme.

RA4 Nous considérons à présent qu'un agent peut interroger une cellule qu'il envisage pour un déplacement pour connaître le niveau de sucre moyen des agents "autour d'elle". Ceci équivaudrait à une information représentant le "niveau de vie moyen" de la population dans cette zone.[1] Nous envisagerons le cas d'agents altruistes qui auront tendance à renoncer à se déplacer vers une cellule, pour laquelle cet indicateur serait inférieur à son propre niveau de sucre. Bien sûr, ce geste désintéressé suppose qu'il existe une autre cellule dans son champ de vision qui permette à l'agent de trouver le sucre dont il a besoin pour survivre.

2.3 L'ordre de mouvement des agents

Dans un premier temps, nous vous avons demandé de programmer l'ordre "d'activation" des agents à chaque cycle en utilisant tout simplement leur indice au sein de la liste de la population à laquelle ils appartiennent. Cette approche est toutefois excessivement rudimentaire et bénéficie systématiquement aux agents qui se trouvent en début de liste. Ce biais peut être annuler ou utilisé pour modéliser un autre modèle de vie en société. Nous vous demandons d'implémenter les deux modèles suivants :

OA1 L'ordre "d'activation" des agents est choisi aléatoirement à chaque cycle. Ceci permet de ne privilégier aucun agent en particulier et de donner, en moyenne, la même chance à chacun d'eux.

OA2 Une autre approche consisterait à simuler une régulation de l'ordre d'activation en fonction du niveau de sucre. En particulier, à chaque cycle, les agents seraient successivement activés dans l'ordre croissant de leur niveau de sucre courant. Ceci créerait un biais tendant à favoriser les "plus faibles".

3 Délivrables attendus

3.1 Le code source

Votre code source sera complet et opérationnel, écrit en Python 3.4 (ou ultérieur). Pour ce faire, vous supposerez que les fichiers (modules) que nous vous fournissons se trouvent dans le même répertoire (dossier) que vos fichiers.

Votre code source sera également lisible et clairement structuré. Nous vous demandons donc d'utiliser les "docstrings" pour commenter votre code. Lorsque vous jugez que le code pourrait ne pas être assez clair par lui-même, nous vous suggérons d'intégrer un commentaire (pas dans le docstring !) pour donner une brève explication. Vous pouvez vous inspirer de la manière dont les commentaires du code source fourni (environnement et cellules) ont été rédigés.

Pour nous faciliter la lecture de votre projet, nous vous demandons de référencer *explicitement* les règles ci-dessus (exemple : **RA1**) dans les commentaires du code source que vous nous fournirez. Notez que nous vous demandons de ne faire référence à une règle que lorsque la fonction est spécifique à cette règle. Il est inutile d'ajouter la référence d'une règle dans toutes les fonctions qui sont appelées par les fonctions "principales". Par exemple, si vous avez une

1. La détermination de ce niveau moyen de sucre des agents se trouvant dans le voisinage d'une cellule représente la plus grande difficulté pour cette question. Nous vous suggérons de programmer cette fonction dans le module `mas_environnement.py`. La détermination du voisinage d'une cellule ainsi que le choix de la taille de ce voisinage sont laissés à votre choix.

"fonction-règle" liée à la règle **RA1**, et que celle-ci utilise la distance euclidienne, ne référencez pas la règle dans le code lié à cette fonction de calcul de distance.

Si l'une des fonctions fournies avec l'énoncé doit être modifiée ou si une fonction devait être ajoutée, nous vous demandons de créer un module spécifique séparé, utilisant le suffixe `_mod`, tant pour le nom de fichier, que pour l'alias utilisé dans le code source. Voici un exemple pour préciser ce que nous attendons :

Admettons que vous ayez choisi d'utiliser, dans vos calculs, la distance de Manhattan [2] au lieu de la distance Euclidienne. Vous devriez, en toute logique, compléter le module `mas_utils.py`, qui contient toutes les fonctions d'aide (et, notamment, la fonction de distance Euclidienne `eucl_dist`). Dans ce cas, nous vous demandons, plutôt que de modifier ce fichier, d'en créer un nouveau, nommé `mas_utils_mod.py` ("`mod`" pour modification), et d'y intégrer la définition de votre fonction. En supposant que vous ayez nommée cette nouvelle fonction `manhattan_dist`, le module dans lequel cette fonction serait appelée aurait, dans son en-tête et parmi d'autres, l'instruction d'import suivante :

```
import mas_utils_mod as u_mod
```

Dans le corps du code, l'appel à votre fonction de distance de Manhattan se ferait alors, par exemple, par

```
...
dist = u_mod.manhattan_dist(cell_ref1, cell_ref2)
...
```

3.2 Le rapport

Nous vous demandons de nous fournir, outre votre code source, un rapport faisant **entre deux et quatre pages**. Etant donné la taille limitée du rapport, vous ne pourrez être complets. Nous vous demandons dès lors d'être *synthétiques*. En outre, nous sommes particulièrement sensibles aux aspects suivants :

- Adoptez une **structure claire et synthétique**. Pour ce rapport, nous vous suggérons la structure suivante :
  1. *Introduction* (Mise en contexte en quelques phrases)
  2. *Eléments clefs de votre implémentation* (Comment avez-vous répondu au cahier des charges) : fonctionnalités de l'agent, moyens techniques d'implémentation (méthodes).
  3. *Synthèse de votre/vos apport(s) personnel(s)*
  4. *Conclusion* (Qu'avez-vous retiré de ce projet.)
- **Justifiez vos choix** par un raisonnement scientifique. Quand plusieurs approches ou solutions (techniques) sont possibles, nous vous demandons de les énoncer et de motiver ce qui vous a amené à en choisir une.
- **Soignez la forme**. La formulation, l'orthographe et la mise en page contribuent à la clarté de votre rapport. Négliger ces aspects peut, au contraire, irriter le lecteur et compliquer sa compréhension.

2. `http://fr.wikipedia.org/wiki/Distance_de_Manhattan`

Notes

— Ne nous transmettez pas une version annotée du code source en annexe du rapport. Tous les commentaires liés au code source sont à y intégrer par les commentaires prévus à cet effet.

3.3 Date de remise

La date ultime de remise est le **vendredi 12 décembre 2014**, la date de réception du mail faisant foi (cf. modalités ci-dessous).

3.4 Format de soumission du projet

— Nous vous demandons de soumettre votre projet par **e-mail** à l'adresse suivante :

`infoh100.1415.projet@gmail.com`

Tout e-mail envoyé à une autre adresse (d'un assistant ou de Thierry Massart, par exemple) **sera ignoré**.

— Le sujet de l'e-mail sera : "INFO-H-100 - Remise du projet"

— L'e-mail contiendra, dans son corps, le nom complet et le numéro de matricule des deux membres de l'équipe.

— Un seul et unique fichier *archive* (compressé au format .zip ou .tar.gz) sera annexé à l'e-mail. Il contiendra les fichiers suivants :

1. `mas_sim.py` - Le fichier contenant le code Python qui permet de lancer toutes les simulations. Ce fichier contiendra au moins une simulation complète et devra être executable
2. `mas_population.py` - Le fichier contenant le code Python du module relatif à la population d'agents.
3. `mas_agent.py` - Le fichier contenant le code Python du module relatif à un agent.
4. `rapport.pdf` - Votre rapport au format PDF.
5. `readme.txt` - Un fichier texte qui liste l'ensemble des fichiers du répertoire et les décrit brièvement.

Si nécessaire, votre fichier archive pourra également contenir *quelques* autres fichiers, mais ils devront être décrit avec plus de détail dans le fichier `readme.txt`.

Avertissement : Pour nous éviter des recherches laborieuses et une perte de temps inutile, nous attendons de votre part le *strict respect des consignes* ci-dessus. Leur non respect entraînera une réduction d'un point (sur vingt) de la note finale !

4 Evaluation

Les aspects pris en compte pour l'évaluation sont les suivants :

— Respect du cahier des charges

— Programmation

    - Bon fonctionnement du programme
    - Structure et clarté du code fourni
    - Respect des règles de bonne pratique
- Le rapport
    - Structure du rapport
    - Qualité de la synthèse
    - Mise en page
- Respect des consignes
    - Délais de remise (également du mail "d'annonce" des binômes)
    - Format des fichiers remis (contenu du fichier archive)
- Apport personnel
    - Intérêt et difficulté de l'apport
    - Réflexion menée (expliquée dans le rapport)
    - Qualité de la programmation

Votre éventuel bonus lié à la présentation faite durant les séances d'exercices viendra s'ajouter à la note finale du projet et n'est pas comptabilisée dans cette grille.

5 Modalités diverses

5.1 Problème de binôme

Nous sommes conscients que la réalisation d'un projet en groupe génère parfois des difficultés. Si l'un des objectifs du projet est de vous confronter à ces problèmes et de vous apprendre à les gérer (car c'est une situation que vous rencontrerez tout au long de votre parcours universitaire et professionnel), il est des situations que nous ne voulons pas vous laisser "porter" seuls. Parmi les situations qui sortent du cadre de l'*apprentissage de travail en groupe*, il y a, par exemple :

- Votre binôme a "disparu de la circulation". Particulièrement en BA1, certains étudiants abandonnent malheureusement l'année avant la session de janvier, vous laissant ainsi seul face à une charge de travail prévue pour deux étudiants. Cet abandon se fait souvent progressivement et est donc parfois difficile à identifier : il vous est souvent difficile de trouver le bon moment pour intervenir auprès d'une "instance extérieure".
- Vous avez un conflit (ouvert ou non) avec votre binôme, ce conflit menaçant la bonne réalisation du projet. Ici encore, il peut être difficile de décider du moment où il est nécessaire de demander de l'aide.

Afin que vous ne vous sentiez pas isolés face à ce type de situations et pour vous éviter de devoir "assumer", par solidarité (honorable, mais parfois déplacée) avec votre binôme, les conséquences d'un dysfonctionnement sur lequel vous ne vous sentez pas avoir de prise, nous vous proposons de prendre contact avec la Coach Polytech, Aline De Greef (`Aline.De.Greef@ulb.ac.be`) du Bureau d'appui pédagogique (BAPP).

Elle vous recevra pour discuter de votre problème, pour prendre du recul par rapport à la situation et pour envisager avec vous une solution. Les solutions les plus classiques étant :

— *Une médiation entre vous et votre binôme*. L'idée étant qu'un tiers (le BAPP, en l'occurence) permet, parfois, de résoudre un contentieux.

— *Vous serrez les dents*. Il n'est pas exclu que votre analyse subjective de la situation puisse être "recadrée" et que ceci vous permette de jeter un autre regard sur votre situation, qui soit plus orienté vers la solution que vers le problème.

— *La dissolution du groupe*. Dans ce cas, nous en serions avertis par le BAPP et nous trouverions une solution pratique pour en tenir compte dans l'évaluation, sans que cela ne porte préjudice à vous ni à votre binôme.

Nous avons choisi de faire intervenir le BAPP, plutôt que de vous proposer de prendre contact avec nous, les assistants, pour ne pas vous mettre dans la situation d'avoir l'impression de dénoncer un autre étudiant. En effet, notre rôle d'évaluateur du projet pourrait vous faire craindre (à tort ou à raison), que nous ne soyons pas/plus impartiaux dans notre jugement.

Notre objectif étant que vous soyez dans les meilleures conditions possibles pour réaliser votre projet, nous vous conseillons de ne pas hésitez à contacter Aline De Greef en cas de problème (sérieux).

5.2 Retard

Tout projet remis en retard sera sanctionné par une réduction de 25% des points : la note du projet serait multipliée par un facteur de pénalité de 0,75. Si le retard dépasse 24 heures, la réduction sera de 50%. Plus aucun projet ne sera accepté au-delà de 48 heures de retard après l'échéance initiale.

Un conseil : Prévoyez de remettre votre projet bien à temps (de préférence pas un quart d'heure avant l'échéance), en anticipant d'éventuelles difficultés techniques (indisponibilité ou saturation des serveurs, etc.). La gestion de celles-ci relève de votre responsabilité.

En cas de motif légitime de retard, merci de nous prévenir. Une pièce justificative vous sera demandée.

5.3 Plagiat

Les étudiants qui se rendraient coupables de plagiat verront les points de leur projet annulés. En outre, excepté le cas où il serait possible de prouver qu'un groupe se serait fait abuser par un ou plusieurs autres, nous considérerons toutes les parties comme également impliquées. La sanction s'appliquera par conséquent à tous de la même manière.

6 Idées de "contributions personnelles"

6.1 Extension du comportement des agents

Note : Si vous ajoutez de nouvelles propriétés aux agents, nous vous prions d'en fournir une brève description (y compris vos motivations et l'utilité de la modélisation de ce comportement pour des systèmes "réels"), mais également des résultats de ce comportement.

Mécanisme reproduction des agents. Dans sa version de base, les agents d'une population ne peuvent que mourir ou survivre. La dynamique du système peut être enrichie en introduisant, par exemple, l'un des mécanismes de multiplication ou de reproduction suivants :

— "La *multiplication asexuée* est un mode de reproduction, qui correspond à la capacité des organismes vivants de se multiplier seuls, sans partenaire, sans faire intervenir la fusion de deux gamètes de sexes opposés. On observe la multiplication asexuée chez les pluricellulaires (animaux et végétaux) et chez les organismes unicellulaires. En botanique, le terme souvent employé pour la multiplication asexuée des végétaux est multiplication végétative." [3]

— La *reproduction sexuée* consiste à faire se rencontrer deux agents (que nous supposerons être "hermaphrodites", c'est-à-dire sans sexe déterminé) et de faire naître un troisième agents de leur "union". Ce nouvel agent aura généralement une combinaison des traits (propriétés intrinsèques) de ses parents.

La mort par vieillesse Le mécanisme de reproduction peut être complété par celui de *durée de vie*. Dans la version de base, un agent ne meurt que s'il ne trouve pas suffisamment de sucre pour répondre aux besoins de son métabolisme. S'il trouve de quoi se nourrir, un agent peut vivre indéfiniment. Notez qu'il ne sera probablement pas très intéressant de programmer ce mécanisme sans la reproduction (quelle qu'en soit la forme), car sinon, la population disparaîtra après un nombre probablement assez réduit de cycles.

Mise en présence de deux populations. L'implémentation de plusieurs populations d'agents étendra considérablement la richesse des phénomènes que vous pourriez observer. Vous pourriez, entre autres, tester plusieurs scénarios :

— Combat entre deux agents de groupes différents quand ils sont sur des cellules adjacentes

— Partage de sucre entre agents d'un même groupe

Pour ce cas, posez-vous également la question de la représentation des agents (comment les distinguer ?) et du type d'information qu'il pourrait être utile de sauver dans le fichier résultats.

Introduction d'une monnaie d'échange. Le livre d'Epstein et Axtell propose d'utiliser l'*épice* comme monnaie d'échange. Vous pouvez ainsi introduire des échanges de sucre et d'épices dans le système et construire un véritable modèle économique.

6.2 Extension du système et de ses possibilités

Amélioration du fichier de configuration. Comme discuté en séance d'exercices, l'utilité d'un fichier de configuration est d'éviter la modification du code source. Un nombre de possibilités, pourtant très utiles en pratique, ne sont pourtant pas encore incluses dans la gestion actuelle des fichiers de configuration. Voici quelques suggestions concrètes qui améliorerait ce système :

3. Source : `http://fr.wikipedia.org/wiki/Multiplication_asexuee`.

— Permettre de préciser les *règles* qui s'appliquerons aux cellules et aux agents. Il pourrait, par exemple, s'agir des noms de fonctions existantes que vous avez programmées.
***Note** : Pratiquement, vous devriez prévoir la possibilité de déclarer plusieurs fonctions. Le plus simple (pour l'utilisateur) serait d'utiliser le même nom de propriété (par exemple "`CELL_RULE = ...`") pour chaque fonction. L'ordre dans le fichier déterminerait celui de l'exécution.*

— Permettre de donner le nom d'un fichier Python qui contient le code pour quelques règles particulières.

— Permettre de préciser le format du ou des fichiers de résultats qui seront générés.

Intégration d'un système de *campagnes de simulations*. Le principe d'un système multi-agent repose sur l'exécution d'une simulation. Celle-ci est basée sur une initialisation partiellement aléatoire (position initiale et propriétés intrinsèques – vision et métabolisme – des agents). Pour diminuer le biais introduit par l'aspect stochastique de l'approche et ainsi permettre d'induire des comportements généraux à partir d'expériences particulières, on procède généralement à la répétition de celles-ci : les paramètres sont maintenus constants et on répète l'expérience un nombre suffisant de fois pour permettre un traitement statistique des résultats.

Vous pourriez envisager d'intégrer un tel mécanisme dans le programme. Notez que ceci vous demandera notamment de réfléchir à la sélection des informations qu'il faudra "enregistrer" pour chaque expérience, tout en restant le plus général possible dans votre approche. Les informations qui devront être sauvées devront également être sélectionnées.

6.3 Améliorations graphiques

Génération de graphiques de résultats. Il pourrait être utile de pouvoir générer et sauver des graphiques immédiatement à partir de votre programme, plutôt que de devoir importer un fichier résultat dans une autre application. Pour ce cas, vous pouvez également étendre le "vocabulaire" du fichier de configuration pour permettre d'y ajouter le type, le format, le nom de fichier, etc. du/des graphique(s) que vous voudriez voir généré(s) par le programme.

Enrichissement des informations affichées. Il est possible d'améliorer l'affichage du système pour qu'il donne plus d'information. Une extension assez immédiate serait, par exemple, de faire varier la couleur de chaque agent pour indiquer son niveau de sucre actuel, celui-ci pouvant être interprété comme une mesure de sa santé. On pourrait ainsi, éventuellement, mettre en évidence des zones où se concentrent des agents ayant un niveau de sucre proche.

Note : Vous pouvez également vous inspirer des idées reprises dans le livre de référence d'Epstein et Axtell.

7 La base de code fourni

L'Annexe A donne une représentation schématisée de l'organisation du code source, ainsi que des différentes fonctions qui permettent d'interagir avec chaque entité. Pour ne pas trop l'alourdir, les méthodes d'interface de base ("getters and setters") n'ont pas été représentées dans ce graphique.

S'il utilise très largement les fonctions développées au fil des séances d'exercices, il y a toutefois quelques remarques particulières à faire au sujet du code qui vous est fourni comme base

à vos développements. Ces points n'ont pas, pour une bonne partie, été abordées durant les séances.

— Une cellule est intégrée dans un environnement, qui lui-même fait partie d'un SMA. La structure de données développée rend compte de cet emboîtement (Figure **??**). Nous avons en outre programmé des fonctions qui permettent de manipuler chacun de ses éléments individuellement (ex. `cell_set_sugar_level`, `env_get_size`, etc.). Chaque entité de niveau supérieur "connaît" les éléments qu'ils contient : à partir de la variable `mas` (représentant le SMA dans son ensemble), on peut accéder à l'environnement ; à partir de celui-ci, on peut accéder à toutes les cellules. Il en va de même pour la population et les agents.
Le "chemin inverse" n'est, par contre, pas aussi naturel. A priori, une cellule ne sait pas dans quel environnement elle est contenue et un environnement ne sait pas à quel SMA il appartient. Ceci peut poser des difficultés pratiques, surtout pour la programmation des *règles* des agents. Aussi, pour le code fourni, nous avons introduit une propriété supplémentaire à chaque entité, qui référence l'entité de niveau supérieur à laquelle elle appartient : L'environnement, par exemple, reçoit, à l'initialisation, la référence du système auquel il appartient. La fonction `env_get_mas(env)` permet ensuite, de "remonter" au système. Ce mécanisme permet de "naviguer" d'une entité à l'autre de manière cohérente et simple.

A Diagramme général du code du SMA