Agilité : Développement par les tests (2)

TP4 Agilité

Simulateur de forêt : développement par les tests (partie 2)

Jérôme Buisine
jerome.buisine@univ-littoral.fr

5 septembre 2023

Durée : 3 heures

L'objectif de ce TP est la mise en production de la dernière phase (sprint 3) de développement de notre simulateur d'évolution de forêt.

1 Travail

Ce support est le troisième d'une suite de travaux pratiques qui ont pour objectif la mise en place d'un simulateur d'évolution de forêt. Dans ce TP, vous allez continuer à **développer par les tests** (*Test driven development*) tout en améliorant le code produit (à partir de nouvelles métriques de qualité du code). De ce fait, vous allez dans ce TP développer en respectant des consignes spécifiques qui sont les suivantes :

— 1. Créer un milestone sur Gitlab pour ce TP, nommé « seed management ».

— 2. Au sein de ce milestone définir l'ensemble des tâches de développement (important donc de lire l'ensemble du sujet pour comprendre ces différentes tâches).

— 3. À partir d'une branche `feature/Sprint3_TaskX` spécifique, écrire les tests (unitaires et fonctionnels) qui correspondent à la tâche demandée et vérifier qu'ils ne fonctionnent pas.

— 4. Effectuer le développement de la fonctionnalité, puis vérifier que les tests unitaires et fonctionnels sont valides.

— 5. Pour chaque tâche développée, vérifiez la qualité de votre code avec `pylint` (en insistant sur la diminution de sa complexité).

— 6. Si la couverture de code est correcte, c'est-à-dire avec un minimum de pourcentage de couverture $> 80\%$ (à vérifier localement), alors un commit est à réaliser.

— 7. Soumettre en ligne la modification apportée et vérifier que le pipeline d'intégration continue fonctionne correctement. Vérifiez également que la qualité du code est préservée. Si besoin réaliser un commit avec des correctifs.

Pour vous aider : voici un ensemble de ressources qui peuvent vous être utiles pour ce TP :

— 1. Documentation officielle de l'outil `Git`.

— 2. Gitlab : interface web pour la gestion de projets versionnés sous `Git`.

— 3. pyenv : permet la gestion d'environnement Python.

— 4. pylint : permet l'analyse de code relative à des conventions du langage.

— 5. pytest : permet d'écrire facilement des tests simples et peut évoluer pour prendre en charge des tests fonctionnels complexes.

Remarque importante : faites attention au copier/coller à partir du PDF qui peut prendre en compte des caractères inattendus et causer des erreurs.

2 Convention de code du projet

2.1 Les bonnes pratiques en Python

Comme déjà mentionné, l'ensemble du projet est développé en Python. Voici un rappel du lien du document proposant les bases de la programmation orientée objet en Python.

Ce document reprend également les bonnes pratiques de gestion des imports des modules d'une librairie.

2.2 Ajout des mesures de complexité

Nous allons pour ce TP ajouter des mesures de complexité de code au sein de SonarQube. Les rapports proposés seront ainsi plus complets et le code pourra être refactorisé en conséquence.

L'outil `pylint` propose des plugins permettant d'ajouter des mesures spécifiques dans les rapports générés. Pour cela, ajoutez à la commande `pylint` le paramètre suivant au sein de votre pipeline :

```
--load-plugins=pylint.extensions.mccabe
```

2.3 Conventions adoptées par le projet `treevolution`

Pour rappel, le projet est soumis à des consignes de développement précises qui sont à respecter. Voici les différentes conventions demandées :

— Le nom du fichier de classe en minuscule et nom de la classe en « pascal case ».

— Utilisation du « snake case » comme convention de nommage des variables et fonctions/méthodes ;

— Les attributs d'instance possèdent un « _ ». Par exemple : « `self._mon_attribut` ».

— Noms des classes, attributs et fonctions/méthodes en anglais. La documentation peut elle rester en français si l'anglais reste un frein.

— Les variables de classe sont en majuscules (voir des exemples dans le code déjà mis à disposition).

— Chaque classe possédera sa méthode `__str__` pour proposer l'affichage d'une instance.

— Exploiter au maximum la metadata « `property` » pour les getteurs/setters d'attributs d'une classe (notamment si l'attribut est abstrait et spécialisé).

— Ajouter la docstring pour chaque : module/classe/fonction/méthode....

— Plus généralement, respecter les conventions de bonne conduites de codage proposées par la *PEP* (Python Enhancement Proposals) et qui seront à vérifier par l'outil pylint.

3 Développements

Tâche 1 : Représentation d'une graine d'une branche. Pour cela, il nous faut créer la classe `Seed` dans un nouveau module « `models.seed` ». Cette classe abstraite doit posséder :

— Une date de naissance (`birth`).

— L'instance d'arbre associée.

— L'instance de branche associée.

— Un état de type `models.state.SeedState` par défaut fixé à `ON_BRANCH`.

— Un nombre de jour après sa naissance avant d'arriver à maturité. Ce nombre de jour est pour le moment inconnu car à spécifier.

— Un nombre de jour maximal qu'une graine peut passer sur le sol après une chute. Ce nombre de jour est pour le moment inconnu car à spécifier.

— Un état `fallen` qui déterminera si la graine est tombée ou non.

— Une date qui conserve le jour de sa chute.

— Une méthode `evolve` abstraite qui déterminera par la suite comment évolue la graine. Cette méthode prend en paramètre une météo (et non un contexte). Par défaut cette méthode incorpore :

- la vérification de transition qu'une graine passe de l'état `ON_BRANCH` à `WAITING`. Ce qui signifie qu'elle attend sur le sol.
- la vérification qu'elle soit en état `DEAD` après avoir atteint le nombre de jour maximal qu'elle peut passer sur le sol.

Remarque : les accesseurs et mutateurs sont à développer en conséquence.

Nous allons, comme pour les classes `Tree` et `Branch` définir une classe spécifiée qui va définir le type de graine liée à notre classe de branche `OakBranch` :

— Créer une classe `OakNut` qui hérite de la classe abstraite `Seed`. Elle sera définie dans le module `models.seeds.oak`.

— Des variables de classes sont définies pour borner les jours d'attente pour atteindre la maturité et ceux d'attente sur le sol. Chaque valeur respective est choisie aléatoirement et uniformément dans un intervalle, avec :

```
MIN_MATURITY, MAX_MATURITY = 30, 40
MIN_SURVIVE, MAX_SURVIVE = 20, 25
```

— La graine de ce type d'arbre évolue spécifiquement de la manière suivante quand elle tombe sur le sol :

- Elle a 80% de chance d'être dévorée, c'est-à-dire de passer dans l'état `EATEN`.
- Elle a également 2% de chance de se transformer en arbre si elle n'est pas dévorée ce jour-là.

À tester : `seed_test.py`

→ Créer un arbre de type `Oak`, une branche de type `OakBranch` associée ainsi qu'une graine de type `OakNut`.

→ Vérifier que la méthode `evolve` fonctionne correctement pour cette graine : que les états transites correctement.

Tâche 2 : Gestion des graines au sein d'une branche.

— Prendre en compte qu'une branche peut contenir une liste de graines.

— Pour notre branche de type `OakBranch`, on considère qu'elle peut générer une graine sous les conditions suivantes :

- La santé de l'arbre doit être supérieure à 30.

- La probabilité qu'une graine germe sur la branche est de 1%.
- Générer une graine fait perdre 1 de nutrition à l'arbre.

— Une branche se doit maintenant de simuler l'évolution de ses graines.

— Si une branche est cassée, elle voit ses graines tomber.

À tester : `world_test.py`

→ Créer un contexte de simulation adapté.

→ Vérifier que la branche est composée de graines lors de la simulation.

Tâche 3 : Intégration des graines pour fleuraison potentielle au sein du monde. Pour le moment l'état de transition d'une graine n'est pas gérer.

— Faire en sorte qu'à la chute d'une graine, celle-ci tombe (pour le moment) aléatoirement dans un carré autour de l'arbre de côté 10.

— La graine est ajoutée au monde et n'appartient plus à l'arbre. Une graine qui tombe en dehors du monde est simplement supprimée.

— Le monde simule chaque jour les graines qui lui ont été affectées et retourne également l'ensemble des graines dans sa représentation du monde (en plus de la date, de la météo et des arbres simulés).

— Une graine qui atteint un état arbre lors de sont évolution engendre la création d'un arbre à sa coordonnée et est supprimée de la liste des graines du monde.

— Au contraire, une graine mangée ou morte est supprimée du monde.

— Une complexité linéaire est présente dû au fait de l'ajout constant d'arbre. Limiter le monde afin qu'il ne simule au final que 10 arbres aléatoirement chaque jour.

À tester : `world_test.py`

→ Créer un contexte de simulation adapté pour avoir au moins un cas de création d'arbre.

→ Vérifier que le monde peut se voir affecter des graines.

→ Vérifier que la naissance d'un arbre à partir d'une graine est possible.

Tâche 4 : Simulation de la chute d'une graine en fonction du vent. Actuellement la chute de la graine est réalisée aléatoirement. Nous allons adapter la chute pour la rendre plus physiquement réaliste et adaptée à la météo. Pour calculer cette information, nous allons nous baser sur l'équation d'un corps en chute libre définie par :

$$y(t) = v_0 t + y_0 - \frac{1}{2} g t^2$$

avec :

— v_0 la vélocité initiale (m/s).

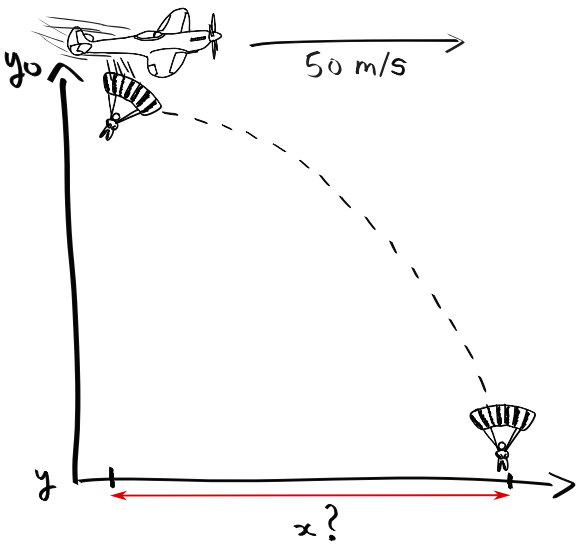
— y_0 l'altitude initiale (m).

— $y(t)$ l'altitude en fonction du temps (m).

— t le temps écoulé (s).

— g est l'accélération due à la gravité (9.81 m/s près de la surface de la terre).

Cette équation permet de calculer l'évolution de la chute du corps verticalement. Dans notre cas, le problème pour objectif de savoir le distance parcourue d'un élément en chute et en mouvement :



Le schéma ci-dessus illustre un exemple, où une personne saute en parachute depuis un avion (on ne considère pas ici les frottements). Cela revient à un parcours partant d'un point vertical initial y_0 jusqu'à un point final vertical y (le sol). Imaginons que y_0 soit égal à 2000 (m) d'altitude. Il faut savoir que v_{0y}, la vélocité verticale, vaut en réalité au départ de la chute 0. On peut de ce fait simplifier l'équation par :

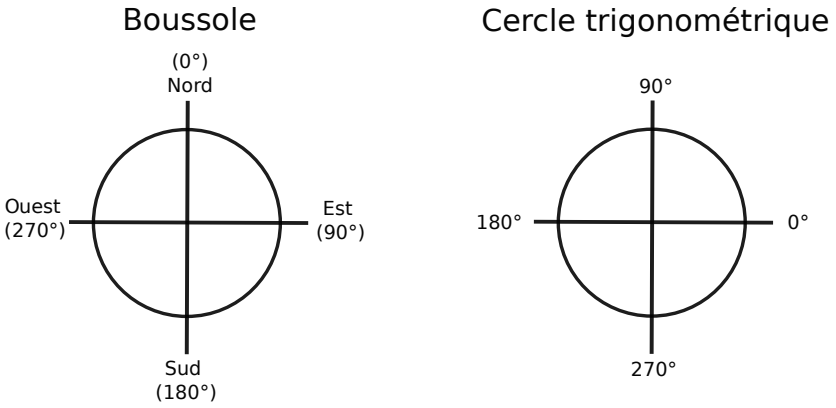
$$\begin{aligned} y &= y_0 - \frac{1}{2}gt^2 \\ \Leftrightarrow 0 &= 2000 - 4.9t^2 \\ \Leftrightarrow t &= \sqrt{\frac{2000}{4.9}} \approx 20.20 \end{aligned}$$

Ainsi, à partir de t, il est possible de calculer la distance parcourue par :

$$\begin{aligned} d &= v_{0x} * t \\ &= 50 * t \approx 1010.15 \end{aligned}$$

À noter que v_{0x} correspond à la vélocité initiale horizontale, soit dans notre exemple la vitesse de l'avion qui vaut $50\,\mathrm{m/s}$. Notre parachutiste a parcouru environ 1010 mètres.

Dans le cadre du projet, nous allons utiliser la distance parcourue pour déterminer la nouvelle coordonnée de la graine à partir du cercle trigonométrique. Pour cela, il faut tout d'abord prendre en considération que l'angle de direction du vent liée à une boussole (représentation de notre monde) n'a pas la même représentation que celui-ci du cercle trigonométrique, utile au calcul des nouvelles coordonnées :



Finalement, à partir d'une distance d et d'un angle de direction du vent α (système de représentation par boussole), il est possible au sein du projet de calculer la nouvelle coordonnée par :

$$x' = x + \cos(-\alpha + 90) * d$$
$$y' = y + \sin(-\alpha + 90) * d$$

À partir de ce modèle de simulation de chute simplifié, calculer correctement le point de chute de chaque graine :

- Externaliser ce calcul dans une fonction externe dans un nouveau module.
- Faire attention aux unités attendues et les conversions. Le vent de la météo est actuellement représenté en km/h.
- Faire également attention aux fonctions sinus et cosinus Python qui attendent un angle en radian.

À tester : `seed_test.py`

→ Vérifier sur quelques exemples que les coordonnées calculées sont bien représentatives de l'attendu (angle du vent et vitesse).

→ Vérifier qu'il n'y ait aucune régression (si besoin adaptez) et que le simulateur fonctionne correctement.

Tâche 5 : Application de simulation avec interface graphique.

- Exploiter le programme `src/main_visu.py` en l'adaptant si besoin à votre code.
- Visualiser une simulation complète de 10000 jours.

Pour une seule espèce d'arbre simulée, celle `Oak`, vous devriez avoir comme fin de sortie de l'interface graphique pour 10000 jours simulés :

```
2050-01-20: (trees: 68, seeds: 0)
2050-01-21: (trees: 68, seeds: 1)
2050-01-22: (trees: 68, seeds: 1)
2050-01-23: (trees: 68, seeds: 3)
2050-01-24: (trees: 68, seeds: 3)
2050-01-25: (trees: 69, seeds: 2)
2050-01-26: (trees: 69, seeds: 3)
2050-01-27: (trees: 69, seeds: 2)
```

```
2050-01-28: (trees: 69, seeds: 1)
2050-01-29: (trees: 69, seeds: 1)
2050-01-30: (trees: 69, seeds: 2)
2050-01-31: (trees: 69, seeds: 2)
2050-02-01: (trees: 69, seeds: 2)
2050-02-02: (trees: 68, seeds: 3)
```

Tâche 6 : Ajouter une nouvelle espèce d'arbre dérivée de la classe `Oak` avec quelques comportements spécifiques de votre choix. Vérifiez qu'il est possible de simuler avec deux espèces d'arbres différentes.

4 Remise des travaux

N'oubliez pas de réaliser un dernier commit de vos travaux. Taguez ensuite votre projet avec le tag « `tp4` » et soumettez-le sur le serveur Gitlab. Vérifiez que le pipeline reste fonctionnel. Cette version de projet fera office de rendu.