Agilité : Développement par les tests (1)

TP3 Agilité

Simulateur de forêt : développement par les tests (partie 1)

Jérôme Buisine
jerome.buisine@univ-littoral.fr

22 septembre 2022

Durée : 3 heures

L'objectif de ce TP est la mise en production de la seconde phase (sprint 2) de développement de notre simulateur d'évolution de forêt. Ce sprint concernera l'ajout et l'évolution des branches d'un arbre.

1 Travail

Ce support est le second 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 **développer par les tests** (*Test driven development*). 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 « branch 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/Sprint2_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. 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 ;
— 6. 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 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 : Nous allons tout d'abord définir une représentation générale d'une branche. Pour cela, il nous faut créer la classe `Branch` dans un nouveau module « `models.branch` ». Cette classe abstraite doit posséder :

- Une hauteur (`height`), qui détermine son emplacement sur le tronc de l'arbre ;
- Un angle de position sur le tronc, déterminant son orientation ;
- Une date de naissance (`birth`) ;
- L'instance d'arbre associée ;
- Un état de type `models.state.BranchState` par défaut fixé à `EVOLVE` ;
- Une longueur (`length`), qui définit sa taille (par défaut fixée à 0) ;
- Une longueur maximale (`max_length`) fixée à `None` pour le moment ;
- Une densité de feuillage (`density`) fixée à 0 également ;
- Une méthode `evolve` abstraite qui déterminera par la suite comment évolue la branche. Cette méthode prend en paramètre un contexte tout comme pour l'évolution d'un arbre.

Remarque : les accesseurs et mutateurs sont à développer en conséquence. On note toutefois que seul la hauteur peut-être amenée à être modifiée depuis l'extérieur.

Nous allons, comme pour la classe `Tree` définir une classe spécifiée qui va définir le type de branche liée à notre classe d'arbre `Oak` :

— Créer une classe `OakBranch` qui hérite de la classe abstraite `Branch`. Elle sera définie dans le module `models.branches.oak` ;

— Des variables de classes sont définies pour borner la longueur maximale de la branche et sa densité à son initialisation. Chaque valeur respective est choisie aléatoirement et uniformément dans un intervalle, avec :

```
MIN_LENGTH, MAX_LENGTH = 1, 2.5
MIN_LEAVES_DENSITY, MAX_LEAVES_DENSITY = 0.05, 0.1
```

— La méthode `evolve` permet dans un premier temps de spécifier que la branche grandit de jour en jour. Son gain de longueur quotidien est défini par $0.005 * youth * h_{ratio}$, avec $youth$ le ratio de jeunesse de l'arbre et h_{ratio}, un ratio lié à la hauteur de la branche sur l'arbre. Ce dernier ratio est définit par : $h_{ratio} = 1 - (height/H)$ avec H la hauteur de l'arbre auquel la branche est associée. Ainsi les branches plus proches du sol devraient être plus longues que celles plus hautes.

À tester : `branch_test.py`

→ Créer un arbre de type `Oak`, puis une branche associée ;

→ Vérifier que la méthode `evolve` fonctionne correctement pour cette branche.

Tâche 2 : Prise en compte des branches au sein d'un arbre.

— Prendre en compte qu'un arbre peut posséder des branches ;

— Une méthode `add_branch` permettra l'ajout d'une branche ;

— Un arbre de type `Oak` peut créer une `OakBranch` chaque jour sous certaines conditions :

- L'énergie perçue par le contexte environnant doit être $> 20\%$;
- La santé de l'arbre doit être supérieur à 50 ;
- La probabilité qu'un tel événement arrive est de 2% ;
- L'angle d'orientation de la branche est choisi aléatoirement ;
- Un `OakTree` ne peut pas créer de branches en dessous de 20% de sa hauteur totale. Vous pouvez pour cela définir la variable de classe `PERCENT_BEFORE_BRANCH`.

— Lorsqu'une branche est créée, l'arbre perd en contrepartie 5 de nutrition ;

— Un arbre devra par défaut simuler l'ensemble des branches qu'il possède.

À tester : `branch_test.py`

→ Créer un arbre de type `Oak` et proposer une simulation ;

→ Vérifier que la simulation du monde permet à l'arbre de posséder des branches.

Tâche 3 : Représentation avancée de l'arbre.

— Lorsque l'arbre grandit en hauteur, chaque branche associée à l'arbre voit sa hauteur évoluer également ;

— Définir une méthode `radius`, qui détermine l'envergure de l'arbre actuel. Par simplification, cette méthode définit que le rayon de cet envergure est représenté par la longueur de la plus grande branche de l'arbre ;

— Le contexte fournit à un arbre est à améliorer. Si un voisin d'un arbre est en état de humus, on considère que le contexte calculé pour cet arbre gagne un bonus de humus de 1 ;

— Faire évoluer le calcul de quantité de humus par : $humus = width * height^2 * N_b$, avec N_b le nombre de branches de l'arbre ;

— Un arbre peut avoir une information sur le contexte courant obtenu par le biais d'un attribut `context`. Lorsque l'arbre évolue, il garde une trace du dernier contexte rencontré.

Indications :

- On considère qu'un arbre est le voisin d'un autre, si le rayon (envergure) de ce potentiel voisin couvre la position (coordonnée) d'un arbre ;
- Utilisez la méthode `is_inside_circle` de la classe `Point` qui permet de vérifier si une instance de `Point` intersecte un cercle.

À tester : `tree_test.py`

→ Créer un contexte de simulation adapté ;

→ Vérifier que la simulation permet à un arbre de gagner en envergure ;

→ Vérifiez qu'un arbre peut bénéficier de humus (contexte courant de l'arbre) lorsque l'un de ses voisins est en état humus. Faire attention à l'ordre de simulation des arbres.

Tâche 4 : Modélisation des risques de cassures de branches. La météo fournit la vitesse de vent ainsi que sa direction. Il nous faut prendre en compte dans notre modèle de risque de cassure de branche ces deux paramètres. Une branche est composée d'un risque plus fort de cassure quand son angle et la direction du vent sont orthogonaux entre eux.

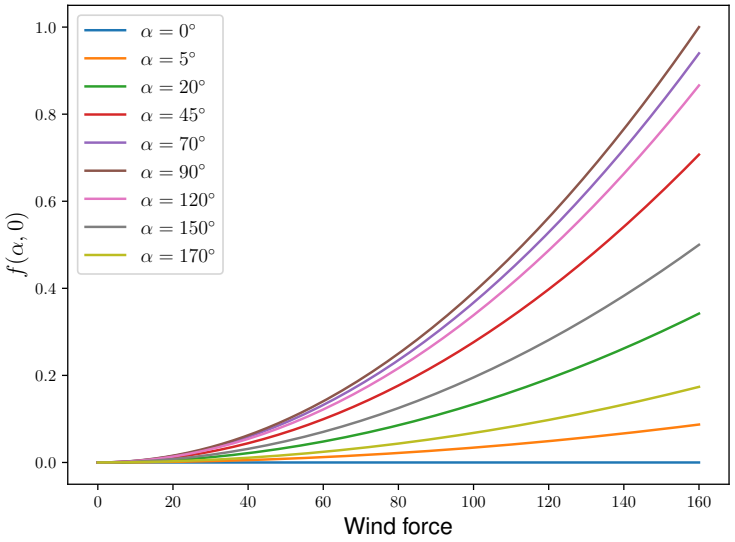
Définissons le modèle d'évaluation du risque de cassure qu'une branche peut subir face au vent comme suit :

$$f(\alpha, \beta) = \sin\left(|\alpha - \beta| \bmod 180\right) * \left(\frac{w}{W}\right)^2$$

avec :

- α l'angle du vent ;
- β l'angle de la branche sur l'arbre ;
- w la vitesse de vent du jour actuel ;
- W la vitesse maximale de vent au sein du simulateur.

Voici un exemple de représentation de l'évaluation de ce risque avec $\beta = 0$ et $W = 160$ pour plusieurs directions et vitesses de vent :



À partir de cette proposition de modèle :

— Écrire une méthode dans la classe `Weather` qui prend en paramètre l'angle de la branche et retourne le niveau de danger ;

— La cassure d'une branche de type `OakBranch`, qui sera définie par un état `BROKEN`, intervient dans un certain cas :

- Si la santé de l'arbre est inférieure à 35 ;
- Le niveau de danger lié au vent évalué est supérieur à 0.95 ;
- La probabilité que l'événement survienne est de 5%.

— Seules les branches non cassées peuvent encore évoluer ;

— La santé de l'arbre de type `Oak` est modifiée comme étant le produit de la nutrition et le ratio de branches non cassées ;

— L'arbre de type `Oak` peut également chuter si sa santé est fortement dégradée (< 10) et si le vent dépasse 140 km/h. Cet événement à 20% de chance de survenir.

Indications :

◆ La méthode `sin` de la librairie `math` de Python attend une valeur en **radian**. Il est possible de convertir un angle en radian par : $rad = \alpha * (\pi/180)$.

À tester : `branch_test.py`

→ Proposer un cas de simulation adapté pour évaluer l'évolution de branches ;

→ Vérifier qu'une branche peut être cassée et que par conséquent elle n'évolue plus ;

→ Vérifier que la santé de l'arbre est calculée comme attendue.

Tâche 5 : Prise en compte du contexte avoisinant. L'intensité lumineuse perçue pour chaque arbre est pour le moment directe. En réalité ce n'est pas le cas, un arbre peut en cacher un autre et atténuer à cause de son feuillage l'intensité lumineuse perçue. Pour cela, nous définissons l'ensemble B, comprenant l'ensemble des branches (non cassées) des arbres voisins qui sont à une hauteur plus importante que la hauteur d'un arbre étudié. L'énergie lumineuse perçue E par un arbre est définit par :

$$E = \max\left(I - \sum_{i=1}^{N} d_i * \left(\frac{l_i}{r_i}\right), 0\right)$$

avec :

- N le nombre d'éléments dans B, soit $||B||$ (son cardinal) ;
- I l'intensité lumineuse émise par le soleil ;
- d_i la densité de feuillage de la branche $b_i \in B$;
- l_i la longueur de la branche $b_i \in B$;
- r_i le rayon de l'arbre auquel la branche b_i est associée.

Ainsi, en considérant cette approche, il est possible d'ajouter les développements suivants :

— Bien considérer l'ensemble B des branches voisines non cassées et plus hautes que l'arbre courant ;

— Prendre en compte l'énergie E filtré à la conception d'un contexte environnant pour faire évoluer un arbre ;

— L'énergie lumineuse perçue par un arbre ne peut pas être négative.

Indications :

- On considère qu'un arbre est le voisin d'un autre, si le rayon (envergure) de ce potentiel voisin couvre la position (coordonnée) d'un arbre ;
- Utilisez la méthode `is_inside_circle` de la classe `Point` qui permet de vérifier si une instance de `Point` intersecte un cercle.

À tester : `world_test.py`

→ Proposer un cas de simulation adapté pour évaluer une baisse d'énergie émise ;

→ Vérifier que l'énergie proposée en contexte pour un arbre est diminuée lorsque qu'un voisin de hauteur plus conséquente est présent.

En l'état le simulateur devrait proposer une seconde version stable. Le prochain TP, permettra d'ajouter de nouvelles fonctionnalités telles la gestion de graines d'arbres et la propagation d'une espèce.

À ce stade, si vous simulez une forêt de 5 arbres avec le contexte suivant :

— La graine aléatoire fixée à 42 ;

— Une date de départ fixée à « 2022-09-10 » ;

— Un monde de taille 200×200 ;

— Les arbres placés aléatoirement ;

— Une simulation de 3000 jours.

Avec un affichage des arbres tous les 1000 jours simulés, vous devriez obtenir (ou être proche de) :

```
At 2025-06-06:
-- (name: Oak, height: 4.52, width: 0.36, coordinate: (x: 147.29, y: 135.34),
        health: 47.94, nutrient: 47.94, age: 2, max_age: 9.46,
        radius: 1.02, branches: 14, state: TreeState.TREE, humus_days: None)
-- (name: Oak, height: 4.26, width: 0.34, coordinate: (x: 84.38, y: 5.96),
        health: 66.83, nutrient: 66.83, age: 2, max_age: 6.09,
        radius: 0.92, branches: 8, state: TreeState.TREE, humus_days: None)
-- (name: Oak, height: 4.45, width: 0.36, coordinate: (x: 5.31, y: 39.77),
        health: 42.70, nutrient: 42.70, age: 2, max_age: 8.25,
        radius: 1.12, branches: 19, state: TreeState.TREE, humus_days: None)
-- (name: Oak, height: 4.50, width: 0.36, coordinate: (x: 44.09, y: 117.85),
        health: 47.94, nutrient: 47.94, age: 2, max_age: 9.05,
        radius: 0.99, branches: 14, state: TreeState.TREE, humus_days: None)
-- (name: Oak, height: 4.32, width: 0.35, coordinate: (x: 161.16, y: 139.63),
        health: 67.94, nutrient: 67.94, age: 2, max_age: 6.70,
        radius: 1.24, branches: 13, state: TreeState.TREE, humus_days: None)
At 2028-03-02:
-- (name: Oak, height: 5.18, width: 0.41, coordinate: (x: 147.29, y: 135.34),
        health: 65.59, nutrient: 65.59, age: 5, max_age: 9.46,
        radius: 1.75, branches: 17, state: TreeState.TREE, humus_days: None)
-- (name: Oak, height: 6.01, width: 0.48, coordinate: (x: 84.38, y: 5.96),
        health: 49.48, nutrient: 49.48, age: 5, max_age: 6.09,
        radius: 1.64, branches: 18, state: TreeState.TREE, humus_days: None)
-- (name: Oak, height: 6.09, width: 0.49, coordinate: (x: 5.31, y: 39.77),
        health: 19.83, nutrient: 75.35, age: 5, max_age: 8.25,
        radius: 1.27, branches: 19, state: TreeState.TREE, humus_days: None)
```

```
-- (name: Oak, height: 5.02, width: 0.40, coordinate: (x: 44.09, y: 117.85),
        health: 20.48, nutrient: 55.59, age: 5, max_age: 9.05,
        radius: 1.80, branches: 19, state: TreeState.TREE, humus_days: None)
At 2030-11-27:
-- (name: Oak, height: 5.18, width: 0.41, coordinate: (x: 147.29, y: 135.34),
        health: 40.41, nutrient: 40.41, age: 8, max_age: 9.46,
        radius: 2.35, branches: 25, state: TreeState.TREE, humus_days: None)
```

---

4 Remise des travaux

N'oubliez pas de réaliser un dernier commit de vos travaux. Taguez ensuite votre projet avec le tag « tp3 » et soumettez-le sur le serveur Gitlab. Vérifiez que le pipeline reste fonctionnel. Cette version de projet fera office de rendu.