Projet de qualité logicielle

1 Description du projet

Le but de ce projet, à réaliser en binôme, est d'appliquer les bonnes pratiques de qualité vues en cours. Il est pensé pour être réalisé pendant les créneaux de cours mais des heures supplémentaires peuvent être utiles selon votre avancement.

Le projet consiste à développer un jeu en respectant la méthodologie et l'ordre de développement proposé. L'idée est de mettre en place rapidement les outils pour pouvoir observer et mesurer la qualité du logiciel (tests automatisés, rapports de couverture des tests et rapport de qualité par analyse statique du code), et de montrer comment conserver un bon niveau de qualité en intégrant des modifications le long du développement du logiciel. Le logiciel est un jeu de go à faire dans le langage de programmation Python (version 3.7 ou plus récente).

En plus de l'avancement de votre projet, vous serez principalement évalués sur la démarche de qualité adoptée tout au long du développement. Voici une liste de libertés/contraintes pour le développement de ce projet.

- Vous ne devez **pas** écrire tout votre code d'un coup puis le tester, même si la quantité de code nécessaire pour développer ce projet est petite. L'idée est de maintenir une qualité tout en faisant évoluer le code. C'est toute la difficulté d'un développement réel, où la phase initiale de développement ne représente souvent qu'une partie faible du travail.
- Votre projet doit contenir au moins la documentation minimale attendue dans tout projet. C'est-à-dire comment l'installer, comment s'en servir et un historique des changements.
- Vous devez faire en sorte que vos bugs et corrections de bugs soient traçables. Le plus simple pour ça est est de vous servir des issues GitLab.
- Ce projet peut être la bonne occasion pour essayer différentes méthodologies de travail comme le Test-Driven Development ou le Pair programming.
- Vous pouvez vous servir des outils de GitLab pour suivre l'avancement de votre projet, notamment des milestones et du kanban.

2 Utilisation demandée de Git

Votre projet doit être développé dans un dépôt Git. Vous devez utiliser un workflow Git qui aide à maintenir une certaine qualité tout le long du développement. Ce workflow doit comprendre des branches de fonctionnalités dont l'intégration doit rester visible dans l'historique Git du projet. Il doit aussi contenir une branche qui permet d'avoir la dernière version stable de votre code.

Vous êtes libres d'utiliser n'importe quel workflow qui respecte ces contraintes (Gitflow par exemple), mais je vous conseille fortement le workflow suivant.

- Maintenir une branche principale de développement (`main`). Passé un certain stade d'avancement du projet (ici, ce stade sera atteint à la fin de la section Mise en place de tests et de suivi de qualité) cette branche doit toujours être dans un bon état de qualité. Tous les tests du projet doivent passer sans erreur et les différentes métriques de qualité de votre code doivent être *bonnes* : par exemple pas d'erreurs/warnings en analyse statique, couverture de 95 % du code par les tests...
- La quasi-totalité des développements doivent être faits dans une branche qui leur est dédiée, que ce soit pour ajouter une nouvelle fonctionnalité (*feature branch*) ou pour corriger un bug. Idéalement, la branche n'est intégrée à `main` qu'une fois qu'on est sûr qu'elle *marche* : le nouveau code de la branche doit être couvert par des tests, sans casser les autres tests.
- Les autres changements (documentation, procédure de *release*...) peuvent également être faits dans une branche qui leur est dédiée, ou directement dans la branche `main` si les changements sont légers et peu susceptibles de *casser* votre code. Si vous faites de gros changements (sur le code ou non), dédier une branche à ces changements est à préférer, pour une meilleure traçabilité mais aussi pour permettre une *review* des changements par d'autres développeurs avant de les intégrer.
- Maintenir une branche stable de votre projet (`releases`). Cette branche devrait toujours contenir la dernière *release* stable du projet. Pour garder cette branche simple et lisible, vous pouvez faire en sorte qu'elle ne contienne que le commit initial du projet et un *merge commit* par *release*. La commande `git log --oneline --first-parent releases` permet de vérifier que c'est bien le cas.

Faites en sorte que votre historique Git soit lisible. En particulier, vos messages de commits doivent être clairs et concis. Privilégiez l'intégration de branches *fast forward* tout en conservant un commit de fusion, afin de maximiser la lisibilité de votre historique et la traçabilité de vos intégrations.

3 Étapes du projet

3.1 Mise en place de tests et de suivi de qualité

Cette phase a pour but de mettre en place de bonnes bases pour la suite du développement. Voici des instructions précises pour cette étape du projet.

1. Sur l'instance GitLab de l'UGA, faites un *fork* du code de base fourni. Ce *fork* sera le dépôt Git dans lequel vous développerez votre projet. Placez ce *fork* dans le *namespace* personnel d'un des développeurs du projet et choisissez une visibilité publique ou interne à l'instance GitLab. Faites en sorte que tous les membres du projet en soient membres en tant que développeurs.
2. Choisissez une licence pour votre projet et documentez votre projet en conséquence — choosealicence peut faciliter votre prise de décision si vous souhaitez faire du libre.
3. Familiarisez-vous avec le code de base fourni. Ce code propose deux modules Python : un pour la gestion basique du plateau de jeu (sans gérer aucune règle du jeu), et un autre chargé de la gestion du jeu. Ces modules proposent quelques fonctions dont vous aurez très probablement besoin pour développer le jeu.
4. Localement (sur votre machine), lancez sur le code source du projet l'analyseur statique de votre choix (pylint, flake8…). Ces outils devraient vous afficher un rapport d'analyse sur votre code source (qui devraient au moins se plaindre de la longueur des lignes dans le code initial fourni). Ces outils comportent de nombreuses options pour choisir quels conseils il est important ou non de suivre pour le projet (lancez-les avec l'option `--help` pour en savoir plus).
5. Localement, lancez les tests du projet (je vous ai écrit un premier scénario de test dans `tests/test_board.py`). Ces tests devraient lancer un seul test qui échoue puisque la classe `Board` n'est pas implémentée. Si les tests échouent pour une autre raison, ce n'est pas ce qui est attendu. En particulier, si le test n'arrive pas à importer le *package* `gomi3`, faire en sorte que votre environnement d'exécution ait une variable d'environnement `PYTHONPATH` qui contienne la racine de votre projet peut vous aider. La documentation de pytest vous sera utile ici, en particulier getting started et usage.
6. Localement, faites en sorte d'obtenir rapport de couverture de vos tests grâce à pytest-cov. Vous devriez obtenir une couverture de 0 % sur les deux fichiers fournis.
7. Automatisez le lancement de votre analyse statique, le lancement de vos tests et la génération des rapports d'analyse et de couverture de vos tests grâce à GitLab CI. La documentation de GitLab CI vous sera très utile ici, en particulier son getting started et la référence syntaxique du fichier yaml.
8. Faites en sorte que votre script Gitlab CI échoue si les tests du projet ne passent pas ou si votre analyseur statique détecte des erreurs ou des *warnings* sur votre code.

Après avoir réalisé toutes ces opérations et après m'avoir appelé pour que je valide l'état de votre projet, commencez à préparer une *release* de votre projet.

- Modifiez le contenu de `README.rst` pour qu'il réflète mieux l'état actuel du projet.
- Votre projet doit respecter Semantic Versioning 2.0.0. Lisez la documentation de Semantic Versioning pour déterminer le nom de la version initiale de votre projet.
- Lisez keep a changelog et créez un changelog initial pour votre projet.

Faites ensuite la *release* de votre projet grâce à un *tag annoté* git. Selon le workflow Git utilisé, vous devrez également faire différentes opérations Git ici, comme mettre à jour la branche qui contient la dernière release du projet.

Appelez-moi pour vérifier la manière dont vous avez *releasé* votre projet. Vous devrez répéter ces opérations à chaque étape du projet et puisque je me placerai dans chacune de vos *releases* pour évaluer votre travail, il vaut mieux que vous ayez un retour rapide là-dessus.

3.2 Développement des bases du jeu (pas du go)

À partir de maintenant, la plupart des changements doivent être réalisés dans des branches dédiées. Ces changements doivent être intégrés les uns après les autres dans `main`, uniquement après avoir vérifié que le code respecte le niveau de qualité souhaité (ce qui se fait en configurant votre script de CI pour qu'il échoue si le niveau n'est pas atteint). Voici les branches successives qui me semblent pertinentes pour cette étape du projet.

1. Implémentez le minimum de code dans `board.py` pour que le test existant passe. Cela devrait correspondre aux méthodes `__init__`, `load` et `color_at` de la classe `Board`.

2. Implémentez les autres méthodes simples de `board.py` (pas `stone_group_at`) et testez-les. Pensez en particulier à tester les cas limites de vos fonctions, comme vérifier que votre code a le comportement attendu au cœur du plateau, sur un bord ou dans un coin. Pensez également à tester des scénarios d'erreur de vos fonctions. Par exemple, vouloir lire ou modifier une intersection aux coordonnées en dehors du plateau doit être une erreur — ce qui se fait via des Exceptions en Python. Veillez à utiliser le type d'exception adéquats plutôt qu'une exception générique lorsqu'il existe dans la bibliothèque standard python (liste des Built-in Exceptions).

3. Implémentez la méthode `stone_group_at` de `board.py` grâce à ces règles du go :

   - *Deux intersections sont dites voisines quand elles sont sur la même ligne et sans autre intersection entre elles.*
   - *Deux pierres sont voisines si elles occupent des intersections voisines.*
   - *Une chaîne [ou groupe] est un ensemble de une ou plusieurs pierres de même couleur voisines de proche en proche.*

   En des termes plus formels, cela veut dire qu'un plateau de go est un graphe non orienté où chaque intersection du plateau est un nœud, et où deux nœuds sont connectés s'ils sont voisins sur le plateau (un nœud au cœur du plateau a 4 voisins, un nœud sur un bord en a 3, et un dans un coin n'en a que 2). Chaque nœud contient une information de couleur (intersection vide, pierre noire sur l'intersection ou pierre blanche sur l'intersection). Cette modélisation en tant que graphe permet d'utiliser directement des algorithmes de graphe, comme l'algorithme de parcours en largeur qui peut fortement vous aider à calculer quelles cases font parties d'un groupe, si vous arrêtez le parcours dès que la couleur de l'intersection actuelle change.

   Testez fortement cette méthode puisqu'elle est une des plus délicates à réaliser dans ce projet. N'hésitez pas à créer d'autres fonctions pour vous aider à rédiger ces tests, par exemple pour vérifier que deux ensembles d'intersections sont les mêmes.

4. Faites une implémentation de base de `game.py` de telle sorte qu'on puisse commencer à faire une suite d'actions pour jouer au jeu. **N'implémentez pas toutes les règles du go à cette étape.** Voici précisément ce qui est demandé :

   - Un coup valide est de placer une pierre sur une intersection vide (ou de *passer*).
   - Le jeu doit s'arrêter lorsque deux coups d'affilé consistent à *passer* son tour.
   - Quand un joueur place une nouvelle pierre, si un ou des groupes adverses adjacents à cette pierre se retrouvent sans liberté, ils doivent être capturés et retirés du plateau.

   Créer une fonction qui permet de calculer les libertés d'un groupe, et bien tester cette fonction, est fortement conseillé ici. Ne testez que **légèrement** la méthode `play_move` de `Game` à cette étape puisque les règles du jeu sont incomplètes pour l'instant.

5. Développez un programme avec une interface utilisateur (en ligne de commande) qui permette de jouer au jeu, de telle sorte que l'interface soit appelable de manière interactive ou non. Par exemple si votre code lit les entrées utilisateur ligne par ligne il sera très facile de jouer des scénarios de jeu en donnant un fichier d'entrée à votre programme.

6. Packagez votre code pour que votre programme soit installable par pip — il n'est pas demandé ici de rendre votre code disponible dans un ensemble de paquets Python comme

PyPI, l'installation depuis des sources locales suffit. Il existe beaucoup de manières de packager du Python. Il est ici demandé d'utiliser un `pyproject.toml` (cf. PEP 517 et PEP 518). Vous êtes libres d'utiliser le *backend* de votre choix tant qu'il est raisonnablement connu et bien packagé.

Une fois toutes ces opérations faites, mettez à jour toute votre documentation et sortez une nouvelle *release*. En particulier n'oubliez pas de mettre à jour comment installer votre projet, comment appeler votre programme et de documenter les changements que vous avez apportés. **Note** : même si l'usage de votre programme est écrit dans votre documentation, on s'attend à ce que le programme lui-même puisse nous dire comment s'en servir (typiquement via `--help`).

3.3 Développement d'un jeu (presque du go)

Modifiez votre code précédent pour qu'il respecte les règles essentielles du go.

- Le *suicide* est un coup illégal au go. Placer une pierre qui a pour effet de former un groupe qui ne contient pas de liberté à partir de la nouvelle pierre est interdit. Ce test doit être réalisé après avoir retiré d'éventuels groupes *morts* adverses adjacents à la nouvelle pierre, afin de permettre la capture de groupes adverses. Détails sur le sucide en go.
- La règle de ko permet d'éviter les parties de longueur infinies. Il vous est demandé ici d'implémenter cette règle de manière très simple de telle sorte que les cycles courts de captures successives d'une pierre soient interdits (cf. exemple de situation simple de ko). **Il ne vous est pas demandé ici d'implémenter la règle complète de ko** (aussi dite de superko). La plupart des programmes de go affichent un symbole (un carré) quand il est interdit de jouer sur une intersection à cause de la règle du ko, vous pouvez faire la même chose dans votre interface utilisateur.

À partir d'ici, votre projet devrait avoir le minimum qui permette de jouer au go ! Mettez à jour toute documentation nécessaire, assurez-vous d'être satisfait de l'interface d'appel de votre programme, de son interface interactive et sortez une nouvelle *release*.

3.4 Développement de fonctionnalités *avancées*

La suite du développement du projet est **optionnelle**. Les développements additionnels seront récompensés dans l'évaluation s'ils sont réalisés en suivant la même démarche de qualité que celle proposée dans les étapes précédentes. Notez cependant que la quantité de points bonus qu'ils apportent est faible par rapport au temps nécessaire pour réaliser ces fonctionnalités.

Vous êtes libres d'implémenter les fonctionnalités optionnelles qui vous intéressent dans cette liste (ou d'autres si vous le souhaitez) et pouvez faire de nouvelles *releases* de votre projet les contenant.

- Comptage des points pour déterminer qui gagne la partie. Il existe différentes manières de compter les points en go (cf. fin de partie). Beaucoup de logiciels ne le font pas directement mais demandent aux utilisateurs de définir quels groupes ils considèrent comme morts afin de calculer les territoires et le nombre de pierres capturées. Définir automatiquement quels groupes sont morts ou vivants est difficile à cause de situations complexes comme le seki.
- Règle complète du ko. Il est interdit de jouer une pierre si l'état résultant sur le plateau a déjà été observé lors de la partie. Conserver tous les états précédents de la partie rend cela facile — des fonctions de hachage permettront d'être efficace ET léger en mémoire.
- Jeu contre une IA. Intégrer une IA existante comme KataGo ou GNU Go serait intéressant pour votre projet, et faisable de manière assez transparente en ajoutant du support pour le protocole GTP dans votre projet. Les ~~fous~~ téméraires pourront aussi tenter de coder la leur (bon courage !).
- Interface graphique.
- Jeu en réseau — le protocole GTP peut encore vous intéresser ici.
- Possibilité d'annuler son coup précédent.
- Enregistrement ou chargement de parties (par exemple dans le format SGF).