TDLOG – séance n° 4
Dominos : exercices

Xavier Clerc – xavier.clerc@enpc.fr
Mathieu Bernard - mathieu.a.bernard@inria.fr
Clémentine Fourrier - clementine.fourrier@inria.fr
Basile Starynkevitch - basile-freelance@starynkevitch.net

12 octobre 2020

À rendre au plus tard le 17 octobre 2020 sur educnet

Les exercices suivants sont à faire en binôme, et en anglais (noms des classes, fonctions, méthodes, variables et commentaires). Merci de placer en en-tête de votre fichier les noms des deux étudiants ainsi que le temps consacré aux exercices. S'agissant de la suite des exercices de la séance précédente, vous pouvez au choix repartir de votre propre programme ou du corrigé disponible sur educnet.

1 Gestion des erreurs

La première étape consiste à modifier le code existant pour traiter les différentes erreurs possibles, en particulier :

- demi-domino de valeur négative ou supérieure à 6 ;
- entrée de l'utilisateur non numérique, *p. ex.* `abc` ;
- entrée de l'utilisateur contenant un indice invalide, *p. ex.* `0`, `8`, ou `4` si la main contient moins de quatre dominos ;
- entrée de l'utilisateur désignant des dominos dont la somme des points n'est pas égale à 12.

Il faut, dans chaque cas, afficher un message d'erreur explicatif et choisir le bon mécanisme de gestion de l'erreur (en particulier assertion ou exception). Il est demandé, là où nécessaire, de définir de nouvelles classes d'exceptions plutôt que de réutiliser des classes prédéfinies par *Python*. Cela permet de séparer plus nettement les erreurs spécifiquement liées au programme.

Note : `int(...)` permet de convertir une chaîne de caractères en entier et lève l'exception `ValueError` si la conversion échoue.

2 Recherche de solution

La seconde étape consiste à modifier la classe `Solitaire` pour permettre la recherche automatique de solution. Cette modification doit aboutir à la définition de trois classes :

- classe `Solitaire`, qui contient les éléments communs ;
- classe `InteractiveSolitaire`, qui hérite de `Solitaire` et y ajoute les éléments nécessaires au jeu interactif ;
- classe `AutoPlaySolitaire`, qui hérite de `Solitaire` et y ajoute les éléments nécessaires à la recherche de solution.

Les classes `Solitaire` et `InteractiveSolitaire` contiennent donc essentiellement un réarrangement du code existant, tandis que la classe `AutoPlaySolitaire` ajoute du code. En particulier, elle doit définir une méthode permettant l'exploration de tous les coups possibles depuis un l'état courant (on se contente d'une exploration naïve) et une méthode qui renvoie un booléen valant `True` ssi il existe une suite de coups permettant de gagner.

Note : il est possible, mais non requis, de *détourner* les exceptions pour résoudre le problème de la recherche de solution. On peut, en effet, simplement coder une exploration récursive de tous les coups légaux et lever une exception lorsque l'on trouve une solution. De fait, lorsque l'on a trouvé une solution, on sait que la position initiale est gagnable et on peut stopper l'exploration. Lever une exception (pour peu qu'elle soit rattrapée de manière adéquate) permet de stopper l'exploration. Si les exceptions sont avant tout utilisées pour la gestion d'erreurs, ce type d'utilisation pour la gestion du *flux de contrôle* du programme fournit une technique intéressante.

3 Optionnel : optimisation de la recherche de solution

Si vous avez réussi à implémenter rapidement l'exploration, vous pouvez vous intéresser à une optimisation classique de ce type de problème par le biais de *tables de transposition*. Une transposition est le fait qu'un même état du jeu soit atteignable par différentes suites de coups. Identifier une transposition est intéressant puisqu'à l'évidence le caractère *gagnable* d'une position ne dépend pas de l'enchaînement de coups qui a conduit à la position. Il est donc inutile d'explorer à nouveau la position. Dans le cas qui nous intéresse, identifier les transpositions perdantes suffit à améliorer nettement les performances. Avec la version optimisée, on peut alors faire une estimation par Monte-Carlo de la probabilité qu'une partie soit *gagnable*.

Une précision *Python* utile pour la mise en place de tables de transpositions : si vous souhaitez utiliser une instance comme clef d'un dictionnaire ou élément d'un ensemble, il est nécessaire que la classe de cette instance implémente la méthode `__hash__` (et que les instances soient non mutables). La méthode `__hash__` doit renvoyer la valeur de hachage (entier) de l'instance et le contrat qui la lie à la méthode `__eq__` est :

$$\forall x \forall y \ x.\texttt{\_\_eq\_\_}(y) == \texttt{True} \Rightarrow x.\texttt{\_\_hash\_\_}() == y.\texttt{\_\_hash\_\_}()$$