TDLOG – session 4
Dominoes: exercises

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 October 2020

Upload your work to educnet on 17 October 2020 at the latest

The following exercises must be done in teams of two, and in English (names of the classes, functions, methods, variables, and comments). Please write in the file header both the students' names and the time spent on the exercises. Since these exercises are based on the ones of the previous session, you can either extend your own program, or start from the solution available on educnet.

1 Error handling

The first step is to modify the existing code to properly handle the various possible errors, in particular:

- half-domino with a negative value or a value above 6;
- non-numerical user input, *e. g.* `abc`;
- user input containing an invalid index *e. g.* `0`, `8`, or `4` if the hand has fewer than four dominoes;
- user input whose corresponding dominoes have a total number of pips (or dots) different from 12.

It is mandatory, in each case, to display an explanatory error message and select the appropriate mechanism to handle the error (in particular assertion or exception). Where necessary, dedicated exceptions should be defined (instead of using predefined ones), as it clearly marks which errors are specific to the program.

Note: `int(...)` can be used to convert strings into integer values; it will raise the `ValueError` exception if the conversion fails.

2 Search

The second step is to modify the `Solitaire` class in such a way that it is possible to automatically search for a solution. The modification should lead to the definition of three classes:

- `Solitaire` class, with all the common elements;
- `InteractiveSolitaire` class, inheriting from `Solitaire`, with the elements specific to the interactive game;
- `AutoPlaySolitaire` class, inheriting from `Solitaire`, with the elements specific to the search.

The `Solitaire` and `InteractiveSolitaire` classes hence contain mainly existing code, while the `AutoPlaySolitaire` class contains new code. In particular, it must define a method naively exploring all legal moves from the current state, and a method returning a boolean equal to `True` iff there is a succession of moves such that the player wins.

Note: it is possible, although not required, to *hijack* the exceptions for the search part. Indeed, it is possible to simply code a recursive exploration of all legal moves and raise an exception when a solution is found. When a solution is found, we know that the initial position is winnable and we can stop the exploration. Raising an exception (as long as it is caught appropriately) allows us to stop the exploration. Exceptions are mainly used for error handling, but such uses to handle the *control flow* of the program provide an interesting technique.

3 Optional: Optimization of the search

If you managed to quickly implement the exploration, you can try and implement a classical optimization of this kind of problem: *transposition tables*. A transposition is the fact that a given state of the game can be reached through different combinations of moves. Identifying a transposition is interesting since a position is *winnable* or not independently of how it was reached. It is thus obviously useless to explore the same position again. On the problem at hand, it is more than enough to store non-winnable transpositions. With the optimized search, we can compute a Monte-Carlo estimate of the probably that a game is *winnable*.

A *Python* tip useful to implement transposition tables : in order to be able to use an instance as a set element or a dictionary key, it is necessary for the class to implement the `__hash__` method (and for the instances to be immutable). The `__hash__` method must return the hash value (integer) for the instance, and must be correct with respect to the `__eq__` method:

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