Goal

Your goal in the Machine Learning assignment is to develop an AI for the cutting edge new match- 3 game using Reinforcement Learning techniques.

You are free to pick whatever exact method you consider most appropriate. Please note that it is sufficient to return a proof-of-concept solution which behaves somewhat smartly, and there is no need to push your agent's performance to the absolute max possible.

Material to return

- Code for the AI, including everything necessary to run the AI (e.g. trained weights etc.)
- Code used for training (if separate from the AI code)
- Short instructions how to run your solution
- *Short* summary of which approach you chose, and why

The Game

Basic rules

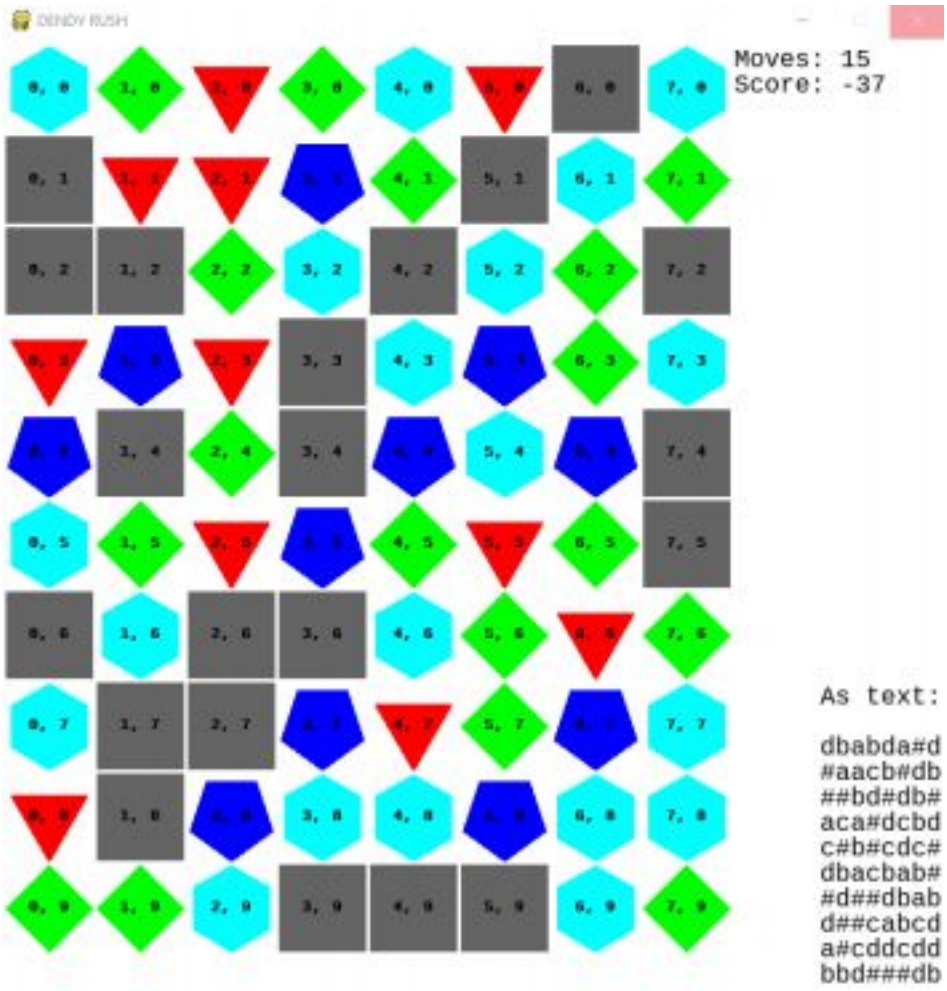
The game is played on an 8x10 grid containing gems (colored polygons) and obstacles (big gray boxes). See **Error! Reference source not found.** for a screenshot of the game. Note that colors and shapes are mapped one-to-one, and the main purpose of having the shapes in addition to colors is to aid players who have difficulties perceiving colors. Each gem is labeled with its position for convenience.

A game consists of 25 moves. In each move, the player swaps two adjacent tokens. After this, three things happen until the playing field stabilizes:

- **Matching**: Any horizontal or vertical sequence of at least three neighboring gems of the same color matches. Matching gems disappear, and the player obtains points for the match. *Only gems can match, obstacles cannot match!*
- **Clearing obstacles:** Any obstacle that is adjacent to a matching gem also disappears. Clearing obstacles does not give points.
- **Falling:** Tokens fall if there are empty cells below in the same column, and columns are filled with random tokens from the top until the entire field is filled again. Note that falling gems can create new matches, which will give the player additional points and may lead to a longer chain reaction.

If you have played so-called match three games such as Candy Crush or Bejeweled, you are probably already familiar with similar mechanics. If not, just watch the game for a bit to get a hang of it.

Figure 1: The game of



Scoring

The player receives points for each match, depending on how many gems are involved in the match:

Gems matched	Points
3	8
4	15
5 or more	24

If multiple matches are created in a single turn, the points for all matches are added.

Furthermore, if the player makes a move during which no match is generated, they receive a penalty of minus 5 points.

Game end and goal

The game ends after 25 moves. The player's goal is to reach the maximum cumulative score at the end

of the game

Implementation

Files

With the programming assignment, several files are provided:

`game.py, graphical.py`	These files contain the code for the game
`LiberationMono-Regular.ttf`	This font file is needed for rendering the text in the game
`random_ai.py`	This file contains a very simple AI playing entirely randomly as an example. This can serve as a base for your own code. This file can also be used to validate the game can run correctly on your machine

Getting started

requires Python 3 with the `pygame` package installed. You can install the package globally using

```
pip install pygame
```

or just for the current user using

```
pip install -U pygame –user
```

Once you installed `pygame`, try running `random_ai.py` to verify everything works correctly. You should see the window with random moves being played.

Inserting your own code

You can launch from your code in two steps:

1. Create a new `graphical.Game`, providing a few callbacks as arguments
2. Call `run()` on the `graphical.Game`

Your code will interact with through three callbacks provided to the `graphical.Game` constructor: `ai_callback`, `transition_callback` and `end_of_game_callback`.

- `ai_callback` is called when it is the player's turn to decide on the next move.
  - `ai_callback` receives three arguments:
    - the current grid state
    - the current score
    - the number of moves left
  - `ai_callback` is expected to return a single move, which will then be executed by the game (see below for the exact data format expected)
- `transition_callback` is called after a move has been made. It can be used to observe the outcome of a move.
  - `transition_callback` receives five arguments:
    - the grid state before the move

        - the move made
        - the number of points received for the move
        - the grid state after the move
        - the number of moves left after the move
    - `transition_callback` does not need to return anything
- `end_of_game_callback` is called when a game finishes. It can be used to observe an entire game
    - `end_of_game_callback` receives four arguments:
        - a list of all grid states during the match
        - a list of all scores after each turn (same length as grid state list)
        - a list of all moves made (exactly one less than grid states)
        - the final score of the game
    - `end_of_game_callback` is expected to return a Boolean value, `True` meaning that another game should be played, and `False` causing the application to exit after the current game's score has been displayed to the user

Additionally the `graphical.Game` constructor can receive two optional arguments:

- `speed`: this argument can be used to speed up or slow down the game. E.g. and argument of 10 will result in 10x speedup. Note that there is a maximum speed at which the game can be run, determined by your screen's refresh rate
- `seed`: this argument is used to seed the game's random number generator. This can be used for debugging purposes.

Finally, feel free to take a look at `game.py`, which contains all of the game logic without any of the graphics. Feel free to use the `GameLogic` class in there to simulate games faster than possible with the graphical interface.

Data format

While most data is provided in the obvious format (e.g. points are integers, etc.), two formats need a bit of explanation:

- **Grid states**: Grid states are provided to your code as strings, with the rows of the grid being separated by \n characters. In the string, a letter indicates a gem ('a'=red triangle, 'b'=green square, 'c'=blue pentagon, 'd'=cyan hexagon), while the # symbol indicates and obstacle. For convenience, the window shows the textual representation of the current (or most recent stable) grid state in the bottom right. Refer to Figure 1 for an example of a grid state and its encoding.
- **Moves**: Moves are provided as three-tuples `(x, y, direction)`.
    - The `x` and `y` components are integers indicating the position of one of the tokens that will be swapped. The coordinate system is zero based and starts in the top left, the `x` axis is horizontal and they `y` axis points down. So e.g. the top left corner is `(0, 0)`, the bottom left corner is `(0, 9)` and the bottom right corner is `(7, 9)`. For convenience, each cell is labelled with its position in thewindow.
    - The `direction` value is a Boolean value. A value of `True` indicates a vertical swap,

while a value of `False` indicates a horizontal swap. The coordinate the token at `(x, y)` is swapped with is obtained by adding one to either `x` or `y`, depending on `direction`. Two examples:

- `(3, 4, True)` swaps the tokens at `(3, 4)` and `(3, 5)`
- `(2, 2, False)` swaps the tokens at `(2, 2)` and `(3, 2)`

- Returning an invalid move (coordinates out of range) will result in no change in the grid. The player still loses a move and receives a score penalty for not matching anything