Practical 2: Separation steering and dungeons

1 Introduction and motivation

There are several ways to create a maze-ish dungeon. In this practical, you are going to implement one of them. The principle is to generate several random rooms, that may overlap, and separate them with an algorithm called *separation steering behaviour*. Then, we create corridors to link these rooms. In order to make these corridors plausible, we choose them according to a Delaunay triangulation on the rooms.
This way to create dungeons is used for example in some indie games, such as *Tiny Keep*.
The files where you will have to write functions are :

— `p1_room_creation`

— `p2_corridors`

The `main.py` file contains the sequence of instructions for the display.
The symbol ♣ means that there are unit tests for the associated function. To use the unit tests, run the command `pytest tests/namefile/namefunction.py` from the tp2 directory. For example, if I want to test the function `collision_detection` which is in the file `p1_room_creation.py`, I launch `pytest tests/p1_room_creation/collision_detection.py`.

Important : Imports must not be changed !

2 Rooms, generation and collision

2.1 Generating a bunch of rooms

A room will have a rectangular shape, and integer dimensions. Walls will have a thickness of 1 square, so rooms' dimensions will be at least 3×3 in order to avoid empty rooms. To model those, a `Room` class is provided. In this one, a room is modeled by its size and the position of one of its corner (the upper left). As we will use a matrix to represent the structure, the x coordinate is vertical (with first line being line 0) and the y coordinate is horizontal.
Figure 1 illustrates this with two rooms that overlap. The blue room has for parameters :

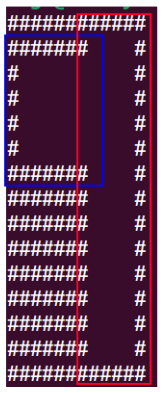
— `size_x = 6`

— `size_y = 8`

— `corner_x = 1`

— `corner_y = 0`

W1 ♣ With the help of the Pseudo Random Number Generator given in `rng.py`, write a function `generate_rooms` that takes 5 arguments : the number of rooms, the minimal and maximal dimension of a side, and the maximal coordinates x and y of any corner. This function will return a list of Rooms.



```
############
#######    #
#          #
#          #
#          #
#          #
#######    #
#######    #
#######    #
#######    #
#######    #
#######    #
#######    #
#######    #
############
```

FIGURE 1 – Two rooms generated

2.2 Collision detection

Now we can move on to the separation part. There are many rooms mashed together and they should not be overlapping at all. The first point is to be able to decide if two rooms overlap.

W2 Implement the function `collision_detection` that takes as argument two Rooms and returns `True` if their intersection is non-empty. To make things simpler, if only the walls intersect, then it will still be detected as a collision.

```
collision_detection(Room(11, 3, 1, 4), Room(4, 8, 7, 1))
>>> True

collision_detection(Room(4, 2, 2, 1), Room(11, 3, 1, 4))
>>> False
```

2.3 Coordinates changes for a room

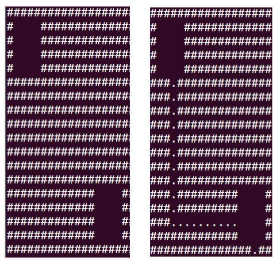
When two rooms collide, one will have to be moved. For that, you will always move rooms right and down. (Reminder : as we use a matrix to represent the structure, for coordinates (x,y), right is y'>y and down is x'>x). This will ensure the algorithm terminates. You will choose the displacement that is minimal between the four following possibilities : moving first room right or down, and moving second room right or down. For example, in Fig. 1, the best move is to move the red room by two units to the right (that is, add 2 to its `corner_y` component).

W3 ♣ Write the function `best_move` will take as argument two rooms that are in collision and will move one room following the rules we just enunciated (by changing the content of the corresponding object), and will return the integer 0 if the first room is moved, 1 else.

```
best_move(red_room, blue_room)
>>> 0
```

2.4 Separation steering algorithm

Now that we have our tools, we can implement the algorithm, which can roughly be summarized by “while there are two rooms colliding, move one”.



```
###################
#    ##############
#    ##############
#    ##############
#    ##############
###################
###################
###################
###################
###################
###################
###################
###################
##############    #
##############    #
##############    #
##############    #
###################
```

```
###################
#    ##############
#    ##############
#    ##############
#    ##############
###.###############
###.###############
###.###############
###.###############
###.###############
###.###############
###.###############
###.###############
###.##########    #
###.##########    #
###...........    #
##############    #
################.##
```

FIGURE 2 – Two rooms before and after adding the corridors

W4 Implement this in the function `separation_steering_behavior` that takes as argument a list of Rooms. It will use the `best_move` function.

3 Corridors

3.1 Choosing main rooms

The next step simply determines which rooms are main hub rooms.

W5♣ Write the function `choose_main_rooms` that takes as argument a list of Rooms, return a list containing all the rooms of dimensions greater than the parameters.

3.2 Building a Minimal Spanning Tree

Now we take all the centers of the selected rooms and feed that into the Delaunay procedure (already done in main). Lucky you, the Delaunay procedure is already provided. It returns a couple (`list_edges`, `list_centers`). Edges are couples of integers i, j where i and j are indexes for the list of centers. For example, if there is a couple $(2, 5)$ in the list of edges, and `list_centers[2]` $= (0, 0)$ and `list_centers[5]` $= (3, 7)$, it means that there is an edge from $(0, 0)$ to $(3, 7)$. The minimum spanning tree will ensure that all main rooms are connected, but also that the path is unique.
The weight of an edge is given by is euclidean norm.

W6♣ Write the function `minimal_spanning_tree`. It returns a list of edges in the tree. This list needs no particular order.

```
minimal_spanning_tree([(0,1), (2,2), (1,10)], [(0,1), (0,2), (1,2)])
>>> [(0,1), (1,2)]
```

3.3 Corridors

For the final part, we want to add corridors. A corridor will be a Room with one dimension equal to 1 (that is, `size_x` $= 1$ or `size_y` $= 1$). (Note that contrary to previous Rooms, we will not add 2 to each dimension to take into account the walls). For each edge of the minimal spanning tree, we will create two corridors, in an L-shape : one horizontal and one vertical. This is illustrated in Fig. 2.

W7 The function `make_corridors` that takes as arguments the list of center of main rooms and the list of corridors we want to make, and returns a list of Rooms (which are corridors).

3.4 Filter the rooms

Finally, filter the rooms : keep only the ones that intersect with at least one corridor. Thus, it will add some of the small rooms we discarded when we chose the main rooms.

W8 ♣ Write this function.