# Dungeon Maker

The goal of this practical is to generate a maze-ish collection of rooms thanks to the *separation steering algorithm*. Roughly speaking, you will generate randomly a collection of rooms, displace then until there are no "collision", and then link them with corridors. For that part, you will choose some main Rooms (the biggest one) and generate a Delaunay triangulation on them. From this triangulation, we will generate a Tree that links every one of these main rooms.

A room will be represented by four numbers : the coordinates of its left-upper corner, its x-dimension and its y-dimension. Notice that these will be the coordinates of the walls. It will be implemented in Python with a simple class (Room) whose constructor takes these four parameters as arguments.

**Do not ask questions before reading the whole subject.**

Here are the different python files in this practical:

- delaunay: contains functions for the Delaunay procedure, do not change this file !
- main: contains the calls to the functions you will define. Uncomment them along your progress.
- mst: contains a function for section 3.2. If you cannot do it, that will allow you to continue
- rng: contains a Pseudo-RNG
- tests: contains some unit tests for your function. Note that tests are not exhaustive
- tp3: contains the functions you have to write !
- utils: contains some more functions used for the display, do not change them !

**Assignment instructions** This work can be done in pairs or alone. A README file MUST be included with the name of students that worked on this assignment, even if you are alone. Not doing this will result in an automatic 0.

You will submit it by april 19, 23h42. A git is available with the following command, replacing $USER by your login.

```
git clone $USER@git.cri.epita.fr:p/2026-s2-nts/tp3-$USER
```

Grading is as follow :

| Part | 1 | 2.1 | 2.2 | 2.3 | 3.1 | 3.2 | 3.3 | 3.4 |
|---|---|---|---|---|---|---|---|---|
| Grade | 0.5 | 1 | 1.5 | 2 | 1 | 3 | 2 | 1 |

# 1 Generating a bunch of rooms

With the help of the Pseudo Random Number Generator you used in earlier works, write a function *generate_rooms* that takes 5 arguments : the number of rooms, the minimal and maximal dimension of a side (note that it should be at least $3 \times 3$ in order not to have an empty room), and the maximal coordinates $x$ and $y$ of the left corner. This function will return a list of Rooms.

# 2 Separate the rooms

## 2.1 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.

Th first point is to be able to decide if two rooms overlap. 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 count as a collision.

## 2.2 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. This will ensure the algorithm terminates. You will choose the displacement that is minimal. For example, in figure 1, the best move is to move the red room down, below the blue one.

The function *best_move* will take as argument two rooms that collide and will move one room following the rules we just enunciated, and returns the integer $0$ if the first room moved, $1$ else.

## 2.3 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”. Implement this in the function *separation_steering_behavior* that takes as argument a list of Rooms. It will use the best_move function.

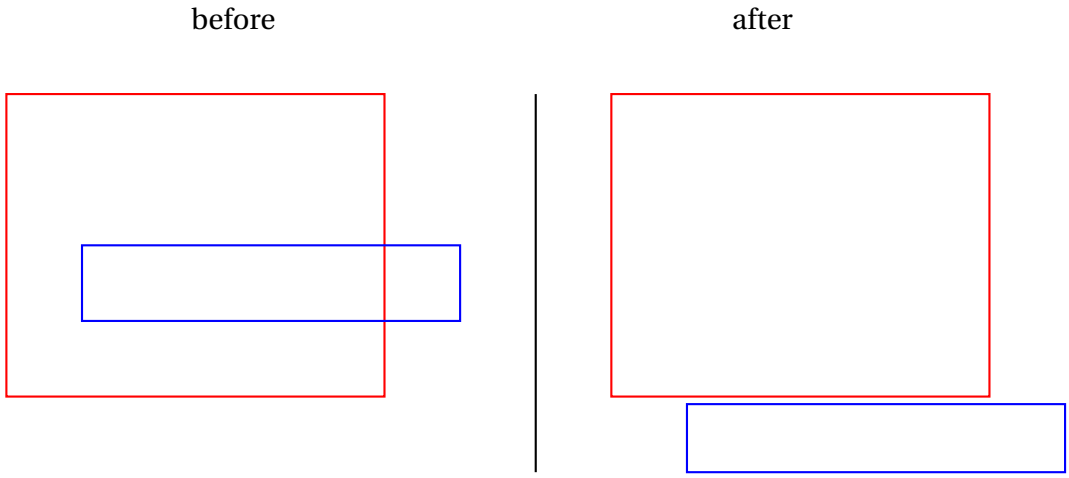


Figure 1: Move one room

# 3 Creating corridors

## 3.1 Choosing main rooms

The next step simply determines which rooms are main hub rooms. This function is quite simple. From a list of Rooms, return a list containing all the rooms of dimensions greater in both dimensions 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 given. It returns a couple (listedges, listcenters) 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 listcenters[2] = (0,0) and listcenters[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. From the result of the Delaunay, you will use one of the two following algorithms: Kruskal algorithm or Prim algorithm (clic on the name to go to Wikipedia pages!).

If you are not able to do it, you can use the function from mst_sol, however you will not gain any point for it. To do so, in main.py, change *mst(...* by *mst_sol.mst(...*

## 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, sizeX = 1 or sizeY = 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 a L-shape : one horizontal and one vertical. The function *make_corridors* takes as arguments the list of center of main rooms and the list of corridors we want to make, and returns a list of 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.