Info-F-106 : Projet d'Informatique
Projet 2 – Dungeon Crawler

Anthony Cnudde Gwenaël Joret Abel Laval Tom Lenaerts
Robin Petit Cédric Simar

version du 17 février 2023

Dans ce second projet, il vous est demandé d'implémenter un *dungeon crawler*, *i.e.* un jeu d'exploration de donjon, en terminal.

Ce projet-ci se concentre sur la séparation d'un code en différents fichiers et en différentes classes, et fait appel à des notions algorithmiques vues en INFO-F103 – Algorithmique 1. Les concepts nécessaires seront tout de même rappelés dans l'énoncé.

1 Contexte

Afin de combler votre désir d'aventure, vous décidez de vous préparer à parcourir un donjon sombre et sinueux à la recherche de sa sortie. Pour cela, avant de faire le grand pas, vous choisissez d'écrire un programme d'entrainement. Afin de ne pas vous compliquer arbitrairement la tâche, vous choisissez, raisonnablement, de vous contenter d'un donjon en 2 dimensions, placé sur une grille. Votre objectif est, sur le long terme, de réussir à trouver la sortie depuis n'importe quelle position et peu importe la difficulté du donjon, votre seule consolation étant une torche vous permettant de vous éclairer. Il vous faut cependant faire attention : votre torche fonctionne à l'huile, et cette dernière se consomme à chacun de vos pas. Heureusement pour vous, de petites réserves de carburant à brûler se situent de manière cachée dans les donjons.

Ce programme que vous écrirez risque d'être un peu long à produire donc vous choisissez (et c'est une bonne chose) de vous y prendre par étapes. Tout d'abord vous voulez vous assurer qu'il est possible de visualiser l'entièreté du donjon de manière un minimum esthétique (phase 1). Ensuite, il vous faudra générer aléatoirement un donjon afin de vous préparer à toutes les situations possibles (phase 2). De là, il ne vous restera plus qu'à gérer les déplacements ainsi que la simulation de votre torche facétieuse et de la vision limitée qu'elle vous impose (phase 3). Enfin, si vous en avez la motivation (et la superstition), vous pourrez ajouter des fantômes errants dans les donjons si le cœur vous en dit (phase 4).

2 Phase 1 – représentation et affichage d'une grille

En cette première phase, nous vous demandons d'écrire un code capable d'afficher toute grille possible. Puisque l'objectif est de visualiser plusieurs éléments, il va falloir décider de la représentation de cette grille. Nous considérons ici que la grille est représentée par une matrice de nœuds définissant quelles directions sont valides depuis chaque case. Un nœud doit donc contenir (au moins) 4 booléens (`up`, `lelft`, `down` et `right`) et `grid[i][j].up` doit être évalué à `False` si et seulement s'il y a un mur entre la case `(i, j)` et la case au-dessus.

L'affichage de la grille doit se faire à l'aide de caractères unicodes, comme montré dans la figure 1. Pour vous aider, les caractères à utiliser sont les suivants :

`─`, `┌`, `┐`, `┬`, `└`, `│`, `├`, `┘`, `┴`, `┤`, `┼`

Ils vous sont également donnés dans un fichier `utf8.txt` si vous n'arrivez pas à les copier/-coller depuis ce document.

Dans l'exemple donné en figure 1, le coin supérieur gauche correspond à la coordonnée `(0, 0)` et le coin inférieur droit correspond à la coordonnée `(11, 5)`. Dès lors, si `grid` est le nom de la matrice représentant la grille, alors :

```
ul = grid[0][0]
lr = grid[11][5]
assert ul.up == ul.left == ul.down == lr.right == lr.down == False
assert ul.right == lr.left == lr.up == True
```

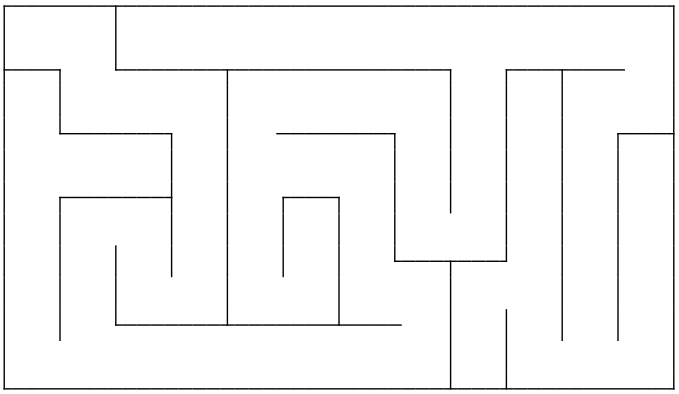

FIGURE 1 – Exemple de donjon (qui s'avère être un labyrinthe) sur une grille de longueur 12 et de hauteur 6.

Dans l'exemple donné en figure 2, nous donnons les valeurs de `up`, `left`, `down` et `right` pour chaque case d'une grille 4×4 et l'affichage attendu.

Code à écrire

Puisque vous allez devoir manipuler beaucoup de coordonnées sur une grille, vous devez écrire une classe `Pos2D` dans un fichier `pos2d.py` représentant un point aux coordonnées entières. Le constructeur `__init__(self, x: int, y: int) -> None` doit construire un point représentant la position `(x, y)`. La classe `Pos2D` doit implémenter la méthode `__eq__(self, other: Pos2D) -> bool` permettant de tester l'égalité entre deux instances via une expression sous la forme `point1 == point2`. Bien que cette fonction vous soit demandée principalement pour les tests automatiques, vous pouvez bien entendu vous en servir dans le reste de votre code.

Afin de pouvoir généraliser la notion de point/position, dans un fichier `box.py`, écrivez une classe `Box` représentant un rectangle défini par deux points.

Les détails d'implémentation de ces deux classes vous sont laissés au choix.

	0	1	2	3
0	up: False left: False down: True right: True	up: False left: True down: False right: True	up: False left: True down: False right: True	up: False left: True down: True right: False
1	up: True left: False down: True right: False	up: False left: False down: True right: True	up: False left: True down: True right: False	up: True left: False down: True right: False
2	up: True left: False down: True right: False	up: True left: False down: False right: True	up: True left: True down: False right: False	up: True left: False down: True right: False
3	up: True left: False down: False right: True	up: False left: True down: False right: True	up: False left: True down: False right: True	up: True left: True down: False right: False

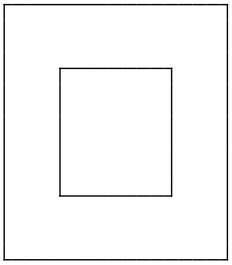

FIGURE 2 – Une boîte dans une boîte.

Vous devez écrire, dans un fichier `grid.py`, une classe `Node` et une classe `Grid` se chargeant, à deux, de la représentation donnée ci-dessus. La classe `Node` doit contenir les 4 booléens demandés, mais le reste vous est laissée totalement au choix ; et la classe `Grid` doit contenir les méthodes suivantes :

— `__init__(self, width: int, height: int) -> None` qui construit une grille par défaut qui est vide, hormis les murs formant le contour ;
— `add_wall(self, pos1: Pos2D, pos2: Pos2D) -> None` qui ajoute un mur entre les positions `pos1` et `pos2` si ces deux cases sont adjacentes ;
— `remove_wall(self, pos1: Pos2D, pos2: Pos2D) -> None` qui retire un tel mur ;
— `isolate_box(self, box: Box) -> None` qui ajoute les murs afin de former le rectangle `box` ;
— `accessible_neighbours(self, pos: Pos2D) -> list[Pos2D]` qui renvoie une liste contenant toutes les cases accessibles depuis `pos`.

Dans un fichier `renderer.py`, Écrivez une classe `GridRenderer` qui se chargera de l'affichage du donjon. Cette classe doit contenir les méthodes suivantes :

— `__init__(self, grid: Grid) -> None` qui construit un *renderer* d'une grille ;
— `show(self) -> None` qui affiche le donjon sur l'écran.

3 Phase 2 – génération d'un donjon et arbres couvrants

3.1 Graphes et arbres

Notons que ce que nous appelons une *grille* est un cas particulier de *graphe* : toute grille est un graphe, mais attention, la majorité des graphes ne sont pas des grilles. En effet dans un graphe quelconque, il n'y a pas de restriction sur les paires de *sommets* (ici des cases donc) qui peuvent être jointes par une arête, mais dans une grille, seules des cases *qui se touchent* peuvent être adjacentes. Pour cette raison, nous ne vous demandons pas d'écrire une classe `Graph` pouvant

représenter n'importe quel type de graphe, mais bien d'utiliser les conditions supplémentaires d'une grille pour une représentation et une utilisation plus efficace.

Cependant, puisqu'une grille est un graphe, nous pouvons la traiter comme tel et utiliser les concepts de graphes qui nous intéressent. En particulier, nous allons ici parler d'*arbres*, et plus précisément d'arbres *couvrants* ainsi que de *parcours* de graphes.

Un *arbre* est un graphe qui est à la fois *connexe* et *acyclique*. La connexité veut dire qu'il existe un chemin joignant chaque paire de sommet. Dans le cas d'une grille, cela veut dire qu'il est possible d'aller vers n'importe quelle case (k, ℓ) en partant de n'importe quelle case (i, j). Acyclique veut dire qu'il n'y a pas de cycle, c'est-à-dire qu'en partant d'un sommet, il est impossible de retomber sur ce même sommet en ne prenant chaque arête qu'au plus une fois. Dans le cas d'une grille, cela veut dire qu'à moins de revenir sur ses pas, il est impossible de retomber sur la case de départ.

Ces deux conditions amènent à une propriété fondamentale des arbres : si v et w sont deux sommets d'un arbre, alors il existe *un unique* chemin permettant d'aller de v à w.

Si nous partons d'un graphe G, un *arbre couvrant* de G est un arbre dont les arêtes sont des arêtes de G et qui contient *tous* les sommets de G. Dans le cas d'une grille, un arbre couvrant est donc un *labyrinthe* de la grille.

Attention : au cours d'algorithmique 1 (INFO-F103), les arbres considérés sont *enracinés*, *i.e.* nous avons toujours un sommet particulier, appelé *racine*, duquel découle le reste de la construction. Ce n'est pas le cas ici : un arbre couvrant d'une grille ne contient aucun sommet spécial. Notons également que les grilles sont des graphes *non-dirigés* et que les arbres sous-jacents le sont également.

Pour un graphe G, un *parcours* de G est un ordre de traitement des sommets qui doit satisfaire certaines propriétés en fonction du type de parcours. Nous nous intéresserons ici aux parcours en profondeur (également appelé DFS pour *Depth First Search* en anglais). Un tel parcours se fait comme suit, pour `v` un sommet quelconque :

```
TRAITEMENT(v)
on traite v
POUR CHAQUE voisin w de v
    SI w n'a pas encore été traité
        on appelle récursivement le traitement de w
```

Un tel parcours porte son nom du fait qu'en partant d'un sommet initial v_0, on va aller le plus *profondément* possible dans le graphe (jusqu'à ne plus avoir de voisin qui n'a pas encore été traité) avant de passer au voisin suivant. Notons bien qu'en général, l'ordre de parcours n'est pas unique ! En effet, en fonction de l'ordre dans lequel les voisins sont traités, il est possible de faire plusieurs (en réalité *beaucoup* !) de parcours en profondeur différents. La figure 3 donne trois exemples de parcours en profondeur partant du coin supérieur gauche sur une grille 3 sur 3 (les chiffres sur la grille servent juste à expliciter le nom des sommets).

0	1	2
3	4	5
6	7	8

Exemples de parcours en profondeur :
- `0, 1, 2, 5, 4, 3, 6, 7, 8;`
- `0, 1, 4, 3, 6, 7, 8, 5, 2;`
- `0, 3, 6, 7, 8, 5, 2, 1, 4.`

FIGURE 3 – Exemples de parcours en profondeur (DFS) sur une grille 3 sur 3.

3.2 Labyrinthes, donjons et pièces

Afin de générer un donjon depuis une grille, nous allons tout d'abord en faire un labyrinthe. Nous définissons ici un *labyrinthe* comme étant une sous-grille de la grille qui passe par toutes les cases mais telle que pour chaque paire de positions (i, j), (i', j'), il existe un *unique* chemin entre (i, j) et (i', j'). En reprenant ce qui a été dit ci-dessus, un labyrinthe est donc un *arbre couvrant* d'une grille.

Afin d'extraire un arbre couvrant d'une grille, il suffit de faire un parcours en profondeur de cette grille et de ne considérer que les arêtes qui sont prises lors du parcours. Notons que nous voulons que le donjon soit généré aléatoirement et change à chaque exécution du programme. Il vous faudra donc parcourir les voisins d'un sommet dans un ordre aléatoire. Vous pouvez pour cela, par exemple, utiliser la fonction `random.shuffle(x: list) -> None` qui prend une liste `x` et paramètre et la *mélange* en place (donc sans la renvoyer).

L'union de deux arbres couvrants T_1 et T_2 va donner un sous-graphe H du graphe initial G (où H ne sera plus un arbre, sauf si les deux arbres couvrants T_1 et T_2 sont égaux).

La figure 4 donne un exemple pour une grille de taille 5 sur 5 en montrant deux labyrinthes générés depuis des arbres couvrants différents et le donjon (sans pièce) défini comme l'union de ces deux labyrinthes.

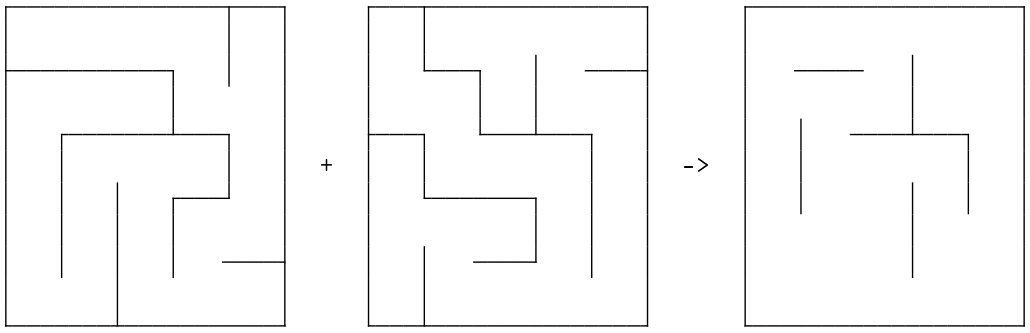

FIGURE 4 – Exemple d'union d'arbres couvrants pour former un donjon sans pièce.

Nous appelons un *donjon* tout sous-graphe connexe (dans lequel toute position est accessible depuis toute autre position) d'une grille (donc un labyrinthe est un cas particulier de donjon). Nous vous demandons d'écrire un code qui permet de générer un donjon étant défini par deux arbres couvrants ainsi qu'un certain nombre de *pièces* défini par passage de paramètre au programme. Une pièce est ici simplement un rectangle avec un nombre prédéfini d'entrées/sorties. Attention, nous imposons également que deux pièces générées ne peuvent pas se toucher et doivent en particulier avoir au moins 2 cases entre elles. Cela implique également que deux pièces ne peuvent pas avoir d'intersection (*i.e.* de case en commun). De plus les pièces ne peuvent pas toucher les bords extérieurs du donjon. La figure 5 montre deux situations impossible pour la génération des pièces.

Notez également que l'intérieur des pièces doit être vide et ne peut pas contenir de murs. La figure 6 donne un exemple de donjon à deux pièces, chacune ayant une unique ouverture et généré depuis deux arbres couvrants, le tout sur une grille 12 sur 6.

Attention à l'ordre dans lequel vous procédez : si vous générez les pièces sur un arbre couvrant (ou sur une union d'arbres couvrants), rien ne vous garantit que le résultat est bien connexe (*i.e.* qu'il n'y a pas des régions inaccessibles dans le donjon). Nous vous recommandons donc de

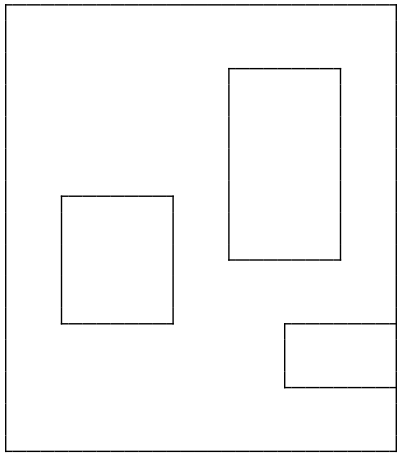

FIGURE 5 – Donjon non-admissible à 3 pièces : l'une touche le bord droit et la pièce en haut à droite n'est à une distance que de 1 des deux autres pièces.

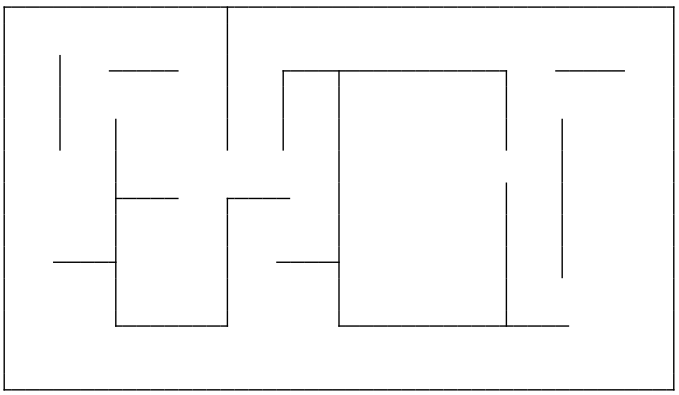

FIGURE 6

générer les pièces en premier, et de générer un arbre couvrant sur base de cela. En effet, en procédant de la sorte (et en sachant que l'intérieur des pièces doit être vide), vous pouvez vous assurer de la connexité de votre donjon.

Code à écrire

Votre programme va devoir prendre en paramètre la paramétrisation du donjon à générer. Pour cela, vous devez *obligatoirement* utiliser le module standard `argparse` (si ce n'est pas le cas, votre projet ne sera pas corrigé). Les paramètres à gérer sont les suivants :

— `width` et `height` sont des paramètres obligatoires correspondant respectivement à la longueur et à la hauteur de la grille sur base de laquelle le donjon va être généré ;

— `--rooms <N>` permet de spécifier le nombre (possiblement nul) de pièces à introduire dans

le donjon (prend 5 comme valeur par défaut) ;
- `--seed <SEED>` permet de spécifier la *seed* (graine) utilisée pour la génération du donjon, ce qui vous sera utile pour le debuggage de votre code (`None` par défaut pour ne pas seeder explicitement la génération du donjon) ;
- `--minwidth <w>`, `--maxwidth <W>`, `--minheight <h>` et `--maxheight <H>` permettent de spécifier les dimensions minimales et maximales que peuvent prendre chacune des pièces qui vont être ajoutées au donjon (les valeurs minimales prennent 4 et les valeurs maximales prennent 8 comme valeur par défaut) ;
- `--openings <N>` permet de spécifier le nombre d'ouvertures (entrées/sorties) qui sont ajoutées à chaque pièce (2 par défaut) ;
- `--hard` permet de spécifier que le donjon à générer doit être basé sur un labyrinthe et pas sur une union d'arbres couvrants.

Dans un fichier `generation.py`, écrivez une classe `DungeonGenerator` ayant les méthodes suivantes :
- `__init__(self, params: argparse.Namespace) -> None`, le constructeur ;
- `generate(self) -> dict` qui génère le donjon (avec le bon nombre de pièces, d'ouvertures par pièce, en concordance avec le mode difficile, etc.) et qui renvoie un dictionnaire contenant un unique élément (vous compléterez cette méthode par la suite) : à la clef `'grid'`, associez l'instance de `Grid` représentant le donjon.

Ajoutez une méthode `spanning_tree(self) -> Grid` à la classe `Grid` permettant d'extraire un arbre couvrant *aléatoire* de la grille (attention : `self` ne doit pas être modifié par cette méthode).

4 Phase 3 – Personnage, visibilité et déplacements

Puisque l'objectif est de se déplacer dans le donjon afin d'en trouver la sortie, il va falloir gérer les inputs pour le déplacement. De plus, comme mentionné plus haut, votre vision au sein du donjon se fait uniquement via une torche dont la luminosité diminue à intervalle régulier (c'est-à-dire après un nombre prédéfini de mouvements dans le donjon). Afin de rendre le tout plus accessible, nous voulons également avoir la possibilité de rajouter du fuel à la torche. Pour cela, le donjon va contenir un nombre prédéfini de bonus de visibilité.

En terme d'affichage, le personnage est représenté par un `X` sur le donjon, la sortie du donjon est représentée par le caractère `#` et les bonus de visibilité sont représentés par un `@`. Un tel bonus est utilisé dès qu'il est ramassé, et doit donc disparaitre du donjon. La restriction de visibilité se traduit en un cercle autour du personnage contenant les informations du donjon et au delà duquel les caractères seront invisibles. Plus précisément, seules les cases à distance (euclidienne) du personnage inférieures ou égales au rayon de visibilité seront visibles et les autres cases afficheront uniquement des espaces pour maintenir l'alignement.

La figure 8 montre la vision attendue de la grille de la figure 7 avec différents rayons de visibilité. Par exemple sur la grille suivante :

La simulation de la torche qui s'éteint petit à petit est faite en décrémentant le rayon de visibilité du personnage à intervalle régulier. Les déplacements se font via les touches `zqsd` (haut, gauche, bas, droit). Attention à ne considérer que les mouvements valides (vous ne pouvez bien entendu pas passer à travers les murs...) tant pour les déplacements que pour les décréments du rayon de la torche.

Si le rayon de vision de la torche arrive à 0, alors la partie est perdue. Si par contre vous arrivez à la sortie avant que votre torche ne rende l'âme, la partie est gagnée.

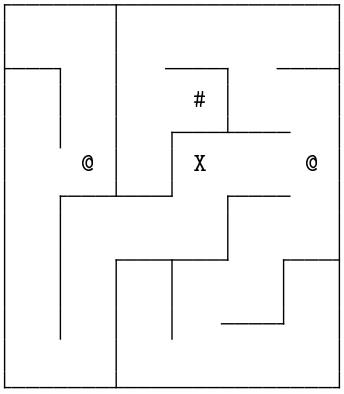

FIGURE 7 – Exemple de donjon sans restriction de visibilité.

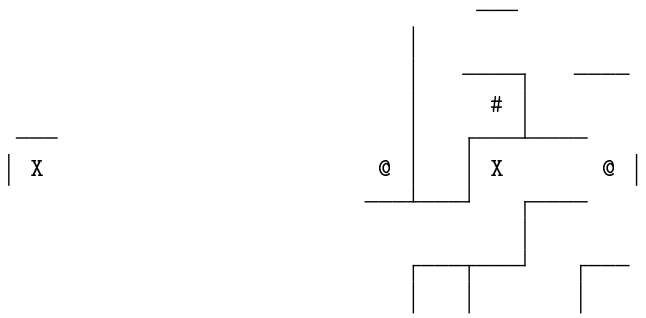

FIGURE 8 – Rayon de visibilité de 1 à gauche et de 5 à droite.

Code à écrire

Le rayon initial de visibilité doit être donné au programme via l'argument `--view-radius <R>` et vaut 6 par défaut. Le nombre de bonus de visibilité à générer est spécifié par l'argument `--bonuses <N>` et vaut 2 par défaut. La valeur avec laquelle le rayon de visibilité est augmenté par bonus est spécifiée par l'argument `--bonus-radius <R>` et vaut 3 par défaut. Le nombre de mouvements correspondant à un décrément du rayon de visibilité de la torche est spécifié par le paramètre `--torch-delay <N>` et vaut 7 par défaut.

Dans un fichier `player.py`, écrivez une classe `Player` représentant la personne tentant de sortir du donjon. Les détails d'implémentation de cette classe vous sont laissés au choix.

Adaptez la méthode `DungeonGenerator.generate` afin que le dictionnaire renvoyé contienne également :

— une clef `'bonuses'` associée à une liste contenant la position de tous les bonus de visibilité (`list[Pos2D]`) ;
— une clef `'start_position'` associée à la position de départ du personnage (`Pos2D`) ;
— une clef `'exit_position'` associée à la position de la sortie du donjon (`Pos2D`).

Dans le fichier `renderer.py`, ajoutez une classe `Renderer` qui se chargera d'afficher le donjon avec tous ses éléments et les restrictions de visibilité. Son implémentation vous est laissée libre, mais attention à la duplication de code.

5 Phase 4 – bonus

De manière facultative, nous vous proposons également d'ajouter des fantômes hantant votre donjon. Les fantômes errent sans but de manière aléatoire dans le donjon, mais attention : en toucher un amènera immédiatement à la perte de la partie. Les fantômes doivent être affichés avec le caractère `G` sur le donjon.

Le comportement des fantômes peut être modifié par les trois paramètres suivants :

— `--ghosts <N>` donne le nombre de fantômes à générer sur le donjon (0 par défaut) ;
— `--ghosts-delay <N>` donne le délai entre deux mouvements des fantômes (2 par défaut) ;
— `--ghosts-walls` précise que les fantômes ne peuvent pas traverser les murs.

Adaptez la méthode `DungeonGenerator.generate` de sorte à ce que le dictionnaire de retour contienne une clef `'ghosts'` associée à la représentation que vous choisissez pour vos fantômes.

L'implémentation de ces fonctionnalités permettent d'obtenir jusqu'à 4 points bonus sur la note finale.

6 Vision d'ensemble du code et des fichiers demandés

Voici un récapitulatif (par ordre alphabétique) des fichiers source attendus et des classes qu'ils doivent contenir lors de la remise de votre projet :

— `box.py` contient la classe `Box` ;
— `generation.py` contient la classe `DungeonGenerator` ;
— `grid.py` contient les classes `Node` et `Grid` ;
— `main.py` est le point d'entrée de votre programme ;
— `player.py` contient la classe `Player` ;
— `pos2d.py` contient la classe `Pos2D` ;
— `renderer.py` contient les classes `GridRenderer` et `Renderer`.

Ces précisions désignent uniquement ce qui vous est imposé, mais nous vous invitons bien entendu à ajouter autant de fichiers/fonctions/classes/méthodes que vous jugez pertinent afin que votre code soit bien découpé et soit propre.

7 Récapitulatif des paramètres à donner au programme

```
$ python3 main.py -h
usage: main.py [-h] [--rooms ROOMS] [--bonuses BONUSES] [--seed SEED]
               [--view-radius VIEW_RADIUS] [--torch-delay TORCH_DELAY]
               [--bonus-radius BONUS_RADIUS] [--minwidth MINWIDTH]
               [--maxwidth MAXWIDTH] [--minheight MINHEIGHT]
               [--maxheight MAXHEIGHT] [--openings OPENINGS] [--hard]
               [--ghosts GHOSTS] [--ghosts-delay GHOSTS_DELAY] [--ghosts-walls]
               width height

INFO-F106 - Dungeon Crawler

positional arguments:
  width                 Width of the dungeon
  height                Height of the dungeon

optional arguments:
  -h, --help            show this help message and exit
  --rooms ROOMS         Number of rooms to generate (default: 5)
  --bonuses BONUSES     Number of bonuses to generate (default: 2)
  --seed SEED           Seed for the RNG (default: None)
  --view-radius VIEW_RADIUS
                        Rendering distance around the player (default: 6)
  --torch-delay TORCH_DELAY
                        Number of moves between 2 torch decays (default: 7)
  --bonus-radius BONUS_RADIUS
                        Visibility radius augmentation by bonus (default: 3)
  --minwidth MINWIDTH   Min width of a room (default: 4)
  --maxwidth MAXWIDTH   Max width of a room (default: 8)
  --minheight MINHEIGHT
                        Min height of a room (default: 4)
  --maxheight MAXHEIGHT
                        Max height of a room (default: 8)
  --openings OPENINGS   Number of openings per room (default: 2)
  --hard                Turn the dungeon into a maze (default: False)
  --ghosts GHOSTS       Number of ghosts to spawn (default: 0)
  --ghosts-delay GHOSTS_DELAY
                        Ghosts will wonder around once every (default: 2)
  --ghosts-walls        Ghosts cannot go through walls (default: False)
```

Si vous ajoutez des paramètres à votre programme (par exemple si vous préférez utiliser `wasd` sur votre clavier qwerty), assurez-vous que leur comportement soit bien documenté et qu'ils aient une valeur par défaut (et que cette dernière ne modifie pas le comportement demandé du programme).

8 Consignes de remise

Afin de créer votre repository pour votre projet 2 sur GitHub Classroom, vous devrez utiliser le lien suivant :

`https://classroom.github.com/a/nvpEuYr6`

Attention, c'est un nouveau repository, ne réutilisez pas celui de la partie 1 !

Rappelons que, tout comme pour la première partie de ce cours, les consignes de remise sont à prendre au sens le plus strict et tout manquement à ces consignes résultera en une note nulle non négociable.

Votre programme doit se lancer par le fichier `main.py`, et tous vos fichiers doivent se trouver *à la racine* de votre repository (*i.e.* pas dans un sous-dossier).

Tous vos fichiers doivent commencer par un docstring sous la forme :

```
"""
Nom : X
Prénom : X
Matricule : X
"""
```

Si vous ajoutez du contenu non demandé dans cet énoncé, précisez-en la nature, l'objectif et le comportement dans un fichier `README` au format soit `.txt` soit `.md`.

Faites bien attention à *toutes* les communications faites via l'UV car *toutes* les précisions ou modifications données de cette manière font partie *intégrante* de l'énoncé.

Un fichier de tests unitaires vous est donné sur l'UV. Vous pouvez le lancer via la commande `pytest --capture=tee-sys -v test.py` en le mettant dans le même dossier que le reste de votre code. Ces tests sont non seulement là pour vous aider à développer votre projet de votre côté et pour vous aider à trouver de potentielles erreurs, mais vont également servir d'évaluation intermédiaire **obligatoire**. En effet, le 19 mars à 22h00, le code présent sur votre repository sera automatiquement téléchargé et ces mêmes tests seront lancés sur votre code. Il faut **impérativement** que l'entièreté de ces 18 tests (qui correspondent à la phase 1 et une partie de la phase 2) passe afin que la remise intermédiaire soit validée et donc que votre remise finale soit corrigée. Sans cela, votre projet sera considéré comme non-remis et une note nulle vous sera assignée. Assurez-vous donc bien que pour cette remise, votre repository soit bien à jour (*i.e.* veillez à ne pas oublier de `git push`) !

La remise finale du projet est fixée le dimanche 2 avril à 22h00. À nouveau, assurez-vous que votre repository soit bien à jour pour la remise car aucun retard ne sera toléré.