**Tableau de matières**

[PARTIE 1 2](#_Toc123601327)

[a. Modélisation informatique : 2](#_Toc123601328)

[b. Affichage de La grille : 2](#_Toc123601329)

[PARTIE 2 6](#_Toc123601330)

[a. Structure de données : 6](#_Toc123601331)

[b. Représentation Informatique : 6](#_Toc123601332)

[c. Choix d’un Algorithme adapté pour le chemin Optimal 7](#_Toc123601333)

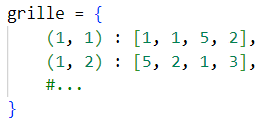
[d. Obtention d’un chemin optimal 10](#_Toc123601334)

# PARTIE 1

## Modélisation informatique :

Nous cherchons une structure de donnée qui nous permet de stocker les informations suivantes :

* Les dimensions du rectangle (issus de la forme rectangulaire de la grille)
* La position de chaque cellule de la grille
* L’épaisseur des 4 murs entourant chaque cellule

La structure de données « dictionnaire » nous permet d’identifier chaque cellule par sa position, cette dernière pourra être modélisée par un couple de la forme : **(ligne, colonne)**. Ainsi, il est possible de modéliser la grille par un dictionnaire dont les mots-clés sont les positions de chaque cellule, et dont les valeurs sont des tableaux de 4 entiers, correspondant aux épaisseurs des murs gauche, haut, droit et bas de cette cellule respectivement. Les dimensions de la grille seront pris en compte par la position du dérnier élement du dictionnaire, qui correspondra à la cellule du bas à droite de la grille. Ci-contre un exemple illustrant cette structure de donné :

## Affichage de La grille :

Pour ce qui est de l’affichage de la grille, nous avons utilisé le module **Turtle** de Python.

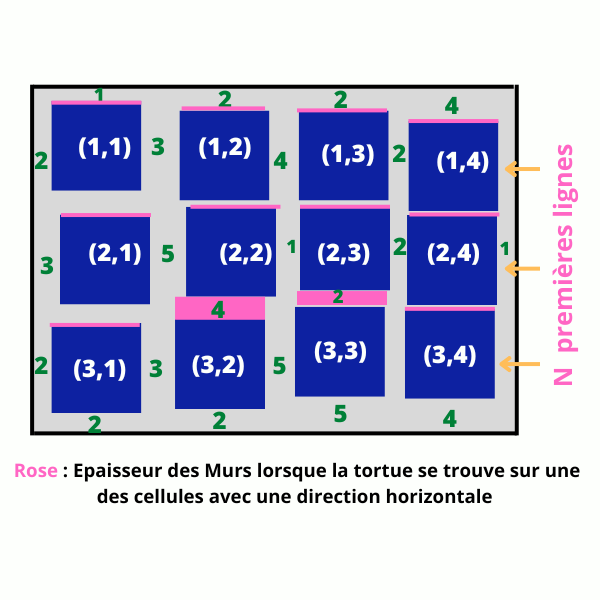
Nous avons utilisé une fonction principale qui sera appelée dans le programme principal « main.py» appellée **dessiner\_grille()** permettant de dessiner une grille, prenant en paramètre une instance de turtle et une grille préalablament créée avec ses dimensions.

Pour donner une explication de cette fonction et comment celle-ci procède afin de dessiner les épaisseurs différentes de chaques murs, il faut savoir qu’elle fait appel à deux fonctions auxilliaires :

1. **Ep\_mur()** : Première fonction prenant en paramètre une grille, la position de la tortue à un instant T (correspondant à une cellule de cette grille) et la direction de la tortue qui peut être **verticale** ou **horizontale**, et qui retournera l’épaisseur du mur à dessiner.

Dans un premier temps, elle vérifie si la direction entrée en paramètre est horizontale ou verticale, nous prendrons uniquement l’exemple d’horizontale car lorsque la direction est verticale, le fonctionnement est identique sauf que l’on récupère les épaisseurs des murs verticaux. Si la tortue se trouve sur une cellule avec une direction horizontale, dans ce cas, **deux possibilités** s’offrent à nous :

* Soit la tortue se trouve sur une cellule des **n premières lignes** de notre grille, dans ce cas, il suffit de consulter le dictionnaire représentant notre grille à la position de la cellule ou se trouve la tortue actuellement et récupérer l’épaisseur du mur haut de cette cellule.

**Exemple** **:**

* **Une image contenant texte

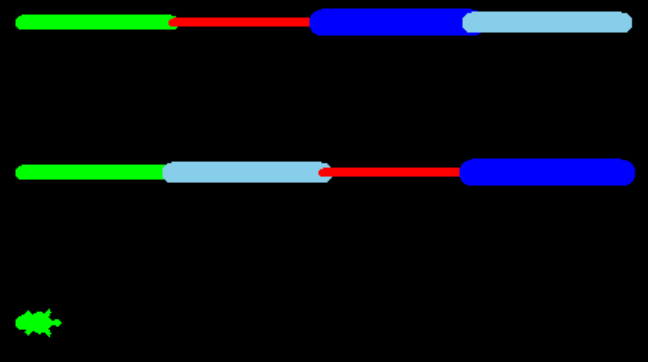
  Description générée automatiquement**Soit la tortue ne se trouve pas sur une des n premières lignes exemple juste aux dessus (ligne 4) et auquels cas il faut tout de même récuperer les épaisseurs des murs du dessous de la dernière ligne, pour ce faire on fait une vérification dans notre code en vérifiant si la direction est bien horizontale et que si la position entré en paramètres est égale à la n-ième ligne de notre grille +1 (dans notre cas 4), c’est-à-dire quelle est bien dans le cas ou il faut l’épaisseur du mur de dessous. Nous retournons cette fois-ci dans notre grille représentée par un dictionnaire l’épaisseur du mur en dessous de celui-considéré et non celui du dessus.

1. **Colorer\_mur()** : Fonction qui prend en paramètre une instance de turtle et une épaisseur entre 1 et 5 puis en fonction de l’épaisseur dessine avec une couleur spéficifique à l’épaisseur ( ex : Jaune pour une épaisseur de 5 et rouge pour une épaisseur de 1).

Une fois ces deux fonctions auxilliaires implémentées, nous faisons appel à la fonction **dessiner\_grille()** qui, dans un premier lieu, va modifier certains paramètres de notre instance de turtle, mettant ainsi le background de la fenetre en noir, positionnant la tortue en haut à gauche de la fenetre et réglant la rapidité par défaut du traçage de la tortue à 5.

**Puis**, enfin, à l’aide de deux boucles for, une première pour les lignes de notre grille, et une deuxième pour les colonnes, la tortue va parcourir chaque cellule de la grille afin de récupérer l’épaisseur du mur à dessiner horizontalement. On appliquera la fonction **colorer\_mur()**,vue juste au dessus, qui va régler la couleur du traçage, en fonction de l’épaisseur récupérée. Par la suite, on avancera la tortue d’un « pas » afin de tracer le mur correspondant. Cela ainsi de suite jusqu’à ce que la boucle for interne se termine, c’est-à-dire que notre tortue a dessiné la totalité des murs du haut de la première ligne.

Ensuite, on répète cela pour les autres lignes sauf que la tortue doit se repositionner à la bonne place afin de dessiner les murs de la deuxième ligne, ect…



Pour ce faire lorsque la tortue aura fini de dessiner les épaisseurs des murs de la première ligne c’est-à-dire quand elle sera positionné sur la dernière colonne on va la faire reculer du pas qu’on l’a fait avancer fois le nombre de colonnes afin qu’elle puisse revenenir à sa position puis une fois cela fait il va falloir l’emmener sur la deuxième ligne, pour cela on va la faire tourner à droite d’un angle de 90 degré et la faire avancer du pas en question puis elle sera prete a dessiner les deuxièmes épaisseurs des murs de la deuxième ligne.

Une image contenant texte, intérieur, très coloré

Description générée automatiquementUne fois tout les murs horizontaux dessinés, nous n’avons plus qu’a réitérer le processus pour les murs verticaux en repositionnant la tortue à sa position initiale, puisque par défaut, elle est à la dérnière ligne actuellement ayant dessiné tout les murs horizontaux.

Ci-contre vous trouverez un exemple lorsque la tortue a dessiné les murs horizontaux et qu’elle remonte à sa position initiale afin de dessiner les murs verticaux de la même manière qu’elle a dessiné les murs horizontaux.

Une image contenant texte, horloge, intérieur, rayon

Description générée automatiquementUne fois que la construction des murs horizontaux et verticaux réalisée, on fait appel a une fonction nommée **ajoute\_coor()** qui ajoutera les coordonées à chaque cellule, pour ce faire, nous appliquons quelques méthodes sur Turtle, telles que : **write()**, **forwrard()**, ect … ( cf. **DessinerGrille.py** ligne 72-85).

# PARTIE 2

## Structure de données :

A picture containing shape

Description automatically generatedA picture containing text, clock

Description automatically generatedLa structure de données permettant de tenir compte des différents chemins de parcours de la grille et de l’épaisseur de chaque mur est un graph non orienté pondéré. Les sommets de ce graph seront les cellules de la grille, ils seront identifiés par leurs positions. Les arêtes du graph désigneront un mur entre les deux cellules adjacentes, ces dernières sont représentées par les deux sommets adjacents à l’arête. Le poids de chaque arête désigne l’épaisseur du mur correspondant. Les murs se trouvant à l’extérieur de la grille seront représentés par des arêtes sortantes et rentrantes dans la cellule adjacente au mur en question.

Voici un exemple d’un tel graphe, et la grille correspondante :

## Représentation Informatique :

Nous avons choisi de représenter ce graph par sa liste d’adjacence. Pour ce faire, la liste d’adjacence sera un dictionnaire, les mots-clés de celui-ci sont les positions des cellules de la grille, tant dis que les valeurs sont eux-mêmes des dictionnaires, les mots-clés de ceux-ci sont dans l’ordre : ‘g’, ‘h’, ‘d’ et ‘b’ (initiales pour gauche, haut, droit et bas resp.), leurs valeurs sont des tableaux de la forme : **[ position, épaisseur ]**, correspondant à la position de la cellule adjacente au mur en question, ainsi que l’épaisseur de ce mur.

Une image contenant table

Description générée automatiquementVoici un exemple illustrant cette structure :

Choix implémentations de La création de la grille :

* Pour les choix d’implémentation, nous avons pensé à créer une méthode dans la Classe **Grille.py** qui prends en paramètre le nombre de lignes et colonnes, et qui crée une grille avec des épaisseurs aléatoires entre chaque cellule (exemple pour lignes=3 et colonne =4) la méthode nous retourne la grille du dessus.

Pour ce faire notre méthode crée à chaque tour une cellule **en vérifiant quelques spécifications** c’est-à-dire **si c’est** la cellule de la première ligne et première colonne alors elle crée des épaisseur aléatoires pour ces 4 murs adjacents tandis que pour les cellules qui sont déjà bordées par une cellule qui à déjà été créer tel que la cellule en position (2,2) elle va récupérer l’épaisseur du mur bas de la cellule (1,2) qui aura déjà été créer et l’affecter à la valeur ‘h’ de la cellule (2,2) car l’épaisseur du mur est commun aux deux cellules. Idem pour la valeur ‘g’ qui sera récupérer via sa cellule gauche voisine c’est-à-dire la cellule (2,1). Puis les épaisseurs des murs ‘d’ et ‘b’ non encore crées seront gérés aléatoirement puisque les cellules (2,3) et (3,2) ne sont pas encore crées. C’est lorsque la cellule (2,3) sera crée qu’elle récupèrera la valeur de son mur adjacent gauche via sa cellule voisine (2,2) Il en va de même pour la cellule (3,2) qui récupérera son mur haut via la valeur du mur ‘bas’ de la cellule (2,2). Enfin pour chaque cellule de notre Tableau de la cellule (1,1) jusqu’à la cellule (m,n) aura pour mot clé sa position dans le dictionnaire grille puis pour les épaisseurs de leurs murs qui bordent la cellule concernés seront des mots clés ‘g’ ‘h’ ‘d’ ‘b’ représenteront les cellules voisines de la cellule concerné avec des valeurs qui auront été soient générés aléatoirement soient récupérés à l’aide des cellules voisines.

## Choix d’un algorithme adapté pour le chemin optimal

Pour ce qui est du choix d’un algorithme afin de calculer le chemin le plus optimal (c’est-à-dire avec un coût minimal), nous avons utilisé l’algorithme de Dijkstra.

**Fonctionnement de l’algorithme :**

* Le but de l’algorithme de Dijkstra avec les prédécesseurs et dans un premier temps d’initialiser un dictionnaire avec comme clé tous les sommets de notre graphe dans notre cas il s’agit de chaque cellule de notre grille, et comme valeur la distance **sommet-départ** qui nous permettra de connaitre la distance de n’importe quel sommet par rapport au sommet de départ.

**A noter** : Par défaut chaque valeur (sommet-départ) est initialisée à + **∞** mise à part le sommet de départ qui sera rentré en paramètre et qui lui par défaut est à 0.

* Puis enfin une liste P faisant référence au prédécesseur d’un sommet car chaque sommet possède un prédécesseur.

Dans un premier temps nous avons une fonction qui prend en paramètres une liste de sommets G ainsi que le sommet de départ et qui nous retournera la liste des distances les plus courtes ainsi que la liste des prédécesseurs. Nous initialisons d[s] correspondant à la valeur du sommet de départ dans le dictionnaire à 0 puis tous les autres sommets ont pour valeur + **∞**. Une fois cela fait nous allons prendre tout les sommets qui sont adjacents à notre sommet de départ dans un premier lieu, puis par la suite le sommet qui sera choisi proviendra de la fonction **minimum()** qui récupérera dans notre dictionnaire d juste en dessous le sommet ayant la distance la plus minimal avec celui de départ. (cf. classe **PlusCoursChemin.py** ligne 17-23)

Ensuite pour les sommets adjacents nous récupèrerons les cellules voisines correspondantes ainsi que son épaisseur, puis si celle-ci n’a pas été traité nous le traitons.

Pour ce qui est du traitement on va vérifier si la valeur dans notre dictionnaire d (correspondant aux distances depuis sommet-départ) de la cellule voisine en question est supérieur à la distance du sommet considéré ajouté à l’épaisseur en question entre ces deux cellules et si c’est le cas cela montrera qu’on a trouvé un chemin plus court.

Une image contenant texte

Description générée automatiquement**Code Pour Compréhension de L’Exemple ci-dessous :**

**Exemple :**

Une image contenant texte

Description générée automatiquementPour notre **Sommet de Départ (1,1)** on obtient le dictionnaire correspondant :

Ici la fonction **minimum()** récupèrera (1,1) puisqu’il s’agit de la distance la plus courte de notre dictionnaire, puis nous allons récupérer les cellules adjacentes à la cellule (1,1) tels que (1,2) et (2,1) avec leurs épaisseur respectives 5 et 2. Puis dans le code inséré juste au-dessus on obtiendra que **d[(1,2)]> d[(1,1)]+5 (ligne 56)** donc on a trouvé un chemin plus court et dans ce cas on remplace donc dans notre dictionnaire d[(1,2)] par la nouvelle valeur qui est égale à 5 **(ligne 57).** Puis on affecte dans notre dictionnaire de prédécesseur la valeur (1,1) à la clé (1,2) **(ligne 58)** qui mentionnera que le prédécesseur de (1,2) est (1,1). Après avoir traité tous les sommets voisins (cellules voisines) de notre sommet considérer (dans notre cas le sommet de départ) on va supprimer le sommet considéré du dictionnaire, car on vient de Une image contenant texte

Description générée automatiquementpasser par celui-là.

Une image contenant texte

Description générée automatiquementOn répète ce processus qui nous permettra de renvoyer la liste des distances les plus courtes par rapport au sommet de départ ainsi que la liste des prédécesseurs. On obtient un résultat tel quel :

## Obtention d’un chemin optimal

Enfin pour obtenir le chemin optimal qui traverse la grille sous forme de couple représentant les cellules traversées dans l’ordre nous avons utilisés l’algorithme de Dijkstra présentée ci-dessus. En effet une fois l’algorithme finie il nous retourne le chemin avec le cout minimal ainsi que chaque prédécesseur de chaque sommet (cellules) sous deux listes qu’on notera D pour la distance et P pour la liste des prédécesseurs.

**Exemple ci-dessous** :

Graphical user interface, text, application, email

Description automatically generated

Enfin pour obtenir le cout minimal il suffit de récupérer la valeur de la dernière case de notre grille dans **le tableau D** retourné par l’algorithme de Dijkstra ici pour l’exemple la dernière case est (3,4) puis en récupérant la valeur dans le tableau retourné par Dijkstra on obtient le cout minimal qui est égal à 10.

Pour ce qui est du chemin sous forme d’une liste de couple indiquant les cellules traversées, puisque l’algorithme de Dijkstra nous retourne une liste de prédécesseur de remonter depuis le sommet d’arrivée c’est-à-dire (3,4) de prédécesseur en prédécesseur jusqu’à la cellule (1,1) puis une fois cela fait inverser la liste ce qui nous donnera le chemin optimal traversée. **(Voir Schéma – Exemple du dessus)**.

**Partie 3 :**

1. **Enrichissement de La Grille**

Pour ce qui est de l’enrichissement de la grille , nous avons utilisés une classe **Enrichir\_Grille.py** qui a pour but à partir du chemin optimal retourné par l’algorithme de Dijkstra de percer notre grille passant par le chemin optimal. Pour ce faire nous allons dans un premier temps à partir de la liste du chemin optimal retourné , ajouté dans notre grille aux cellules concernées par le chemin optimal une valeur booléenne **True** qui indiquera que cette cellule est concernée par le chemin optimal. **Cependant** il faut savoir quel est le mur de cette cellule qui est concerné par le chemin optimal .

Pour savoir quel est le mur concerné nous allons comparer la cellule **i** courante de la liste du chemin optimal avec la cellule **i+1**. A partir des coordonnées de ces deux cellules on pourra savoir quel est le mur concernée et par conséquent ajouter la valeur **True** au mur concerné. **(Voir Enrichir\_Grille.py ligne 15-30)**

Une image contenant texte

Description générée automatiquement

1. **Adaptation de la Méthode D’Affichage de La Grille**

Ensuite, une fois l’enrichissement de la grille finie , c’est-à-dire les attributs **True** ajoutées aux cellules qu’il faudra dessiner pour avoir le plus court chemin . Nous n’avons plus qu’à parcourir toute notre grille , puis pour chaque cellule de la grille , nous allons parcourir ses différents murs avec ses cellules voisines, et si la longueur du tableau du mur en question avec la cellule voisine est égale à 3 , cela indiquera que la valeur **True** à été ajouté, et que par conséquent le mur de la cellule en question fait partie du chemin optimal.

**Exemple**: Au-Dessus lors du parcours de la cellule (1,1) on voit que le mur **bas** fait partie du chemin optimal puisque la longueur du tableau de la clé **‘b’** est égale à 3.

On récupère ensuite l’épaisseur du mur entre la cellule en question et la cellule voisine. Puis à partir d’une fonction **general()** prenant en paramètre le mur en question ainsi que la cellule et **retournant 4 valeurs** qui seront utiles à notre instance de Turtle afin de dessiner le bon mur au bon endroit . **La première** indiquant la position du mur par rapport à la cellule voisine ce qui nous permettra de supprimer l’attribut **True** de la cellule voisine car sinon on dessinera deux fois le chemin entre ces deux cellules **, la deuxième valeur** indiquant l’angle pour notre instance de turtle et enfin **la position x et y** qui variera et qui indiquera ou l’instance doit se trouver.

On répète ensuite ce processus pour chaque mur d’une cellule avec la valeur **True**. Ce processus est répétée dans une fonction **adapter\_dessin()** que vous trouverez juste en-dessous ce qui permettra de dessiner le chemin optimal de notre Grille.

Une image contenant texte

Description générée automatiquement

**Partie 4 : Bonus**

Pour cette dernière partie , en vue de la consigne qui indique que nous devons mettre en point une version plus efficace mais pas forcément optimale , nous avons pensés à une stratégie gloutonne. Le but de cette stratégie va être de comparer les épaisseur de la cellule concernée avec ces cellules voisines gauche et bas afin d’avoir une suite de résultat localement optimal tout en espérant avoir un résultat globalement optimal.

Pour ce faire vous retrouverez une classe **Bonus.py** avec une fonction **efficace\_pas\_opti()** qui initialise une liste qui sera représentative du chemin parcouru avec les coordonnées des cellules et un cout initialisé à 0. Cette fonction a pour but de commencer de la première cellule (1,1) puis de comparer **localement** les épaisseur des mur avec la cellule voisine du **bas** et la cellule voisine **gauche** puis d’ajouter l’épaisseur avec la valeur la plus faible dans la liste et donc de passer par l’épaisseur la moins grosse et d’ajouter la valeur de cette épaisseur à notre variable cout. Cependant si on arrive sur la dernière ligne (resp dernière colonne) nous sommes obligés de nous déplacer sur l’épaisseur du mur droit (resp l’épaisseur du mur du bas ) On répète ce processus jusqu’à ce qu’on arrive à la dernière case (n.m) de notre grille puis nous retournons le chemin associée ainsi que le cout. **(Cf Bonus.py ligne 9-62)**

Par ailleurs vous retrouverez dans cette fonction que nous réitérons à titre comparatif le même processus en partant de la dernière case cette fois-ci et en remontant jusqu’à la première case **(Cf ligne 39-57)** . Le but de cela sera à la fin de la fonction de comparer le cout du chemin direct avec le chemin indirect et de retourner le chemin avec le meilleur cout. **( cf ligne 59-62) .**