UNIVESITE DE BOURGOGNE

RAPPORT INFO3A

La Grille Magique

RAMDANI Ferhat
AHMIM Mohamed

Tableau de matières

PARTIE 1

a. 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 :

```
grille = {
    (1, 1) : [1, 1, 5, 2],
    (1, 2) : [5, 2, 1, 3],
    #...
}
```

b. 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()**, c'est la fonction qui nous permet de dessiner une grille, prenant en paramètre une instance de turtle et une grille préalablement créée avec ses dimensions.

Globalement, cette fonction va commencer par dessiner tout les murs horizontaux de notre grille, par la suite, les murs verticaux.

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()** :
   Lors du dessin des murs horizontaux (verticaux resp.), on aura besoin de l'épaisseur du mur à dessiner, c'est cette fonction qui se charge de nous le retourner.
   Nous allons dans la suite parler de la i-ième ligne des murs, et de la i-ième colonne des murs, et aussi de la position de la tortue, la figure ci-dessous va clarifier ce que signifient ces mots.
   Voici un exemple d'une grille 3x4, les points gris montrent les positions que peut prendre la tortue, la direction dans cet exemple est horizontale:

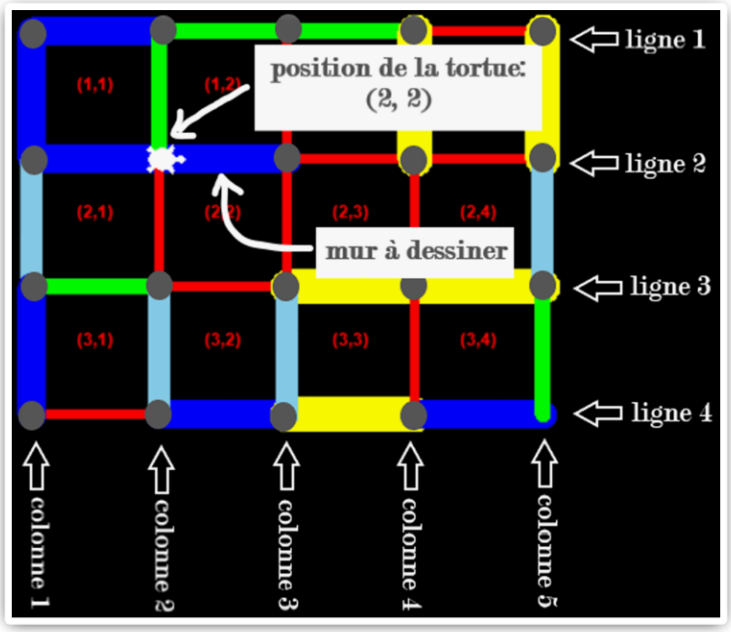


Figure montrant l'instant où la tortue se trouve à la position (2,2), ainsi que le mur à dessiner.

La fonction **Ep_mur()** prent en paramètre une grille, la position de la tortue à l'instant courant et la direction de la tortue qui peut être **horizontale** ou **verticale**, et retourne l'épaisseur du mur à dessiner.

Dans un premier temps, elle vérifie si la direction entrée est horizontale ou verticale, nous ne traitons que le cas horizonal, car identique au cas vertical.

Si la direction est horizontale, **deux possibilités** s'offrent à nous :

- Soit la tortue se trouve sur une des n premières lignes des mur, dans ce cas, il suffit de consulter le dictionnaire représentant notre grille à la position de la cellule qui se trouve juste au dessous du mur à dessiner (n, colonne de la tortue) et récupérer l'épaisseur du mur haut de cette cellule.
- Soit la tortue se trouve à la dérnière ligne des murs (n+1), (dans la figure ci-dessus, cela correspond à la ligne 4), auquels cas, nous retournons l'épaisseur du mur du bas de la cellule juste en dessus du mur à dessiner, cette cellule se trouve à la position (n, colonne de la tortue).

```
if dir == 'h':
    if pos[0] == n+1:
        return grille[(n, pos[1])]['b'][1]
    else:
        return grille[pos]['h'][1]
```

2. **Colorer_mur()** :

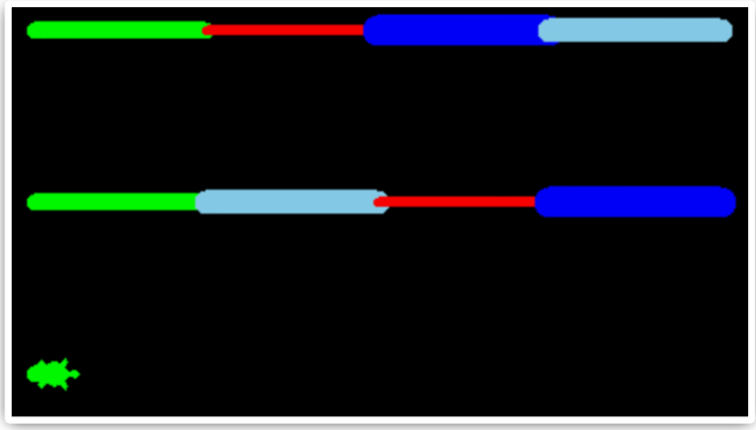
Cette fonction décide de la couleur du mur à dessiner.

Elle prend en paramètre une instance de turtle et une épaisseur entre 1 et 5, puis modifie la couleur de la tortue en fonction de l'épaisseur rentrée (ex : Jaune pour une épaisseur de 5 et bleue 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 11.

Enfin, à l'aide de deux boucles for, une première pour les lignes des murs, et une deuxième pour les m premières position sur la ligne des mur courante, la tortue récupère l'épaisseur du mur horizontal à dessiner en faisant appel à **Ep_mur()**. Par la suite, on fait appel à la fonction **colorer_mur()**, vue juste au dessus, pour régler la couleur du traçage, en fonction de l'épaisseur récupérée. Ensuite, la tortue avance 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 toute la ligne des murs courante.

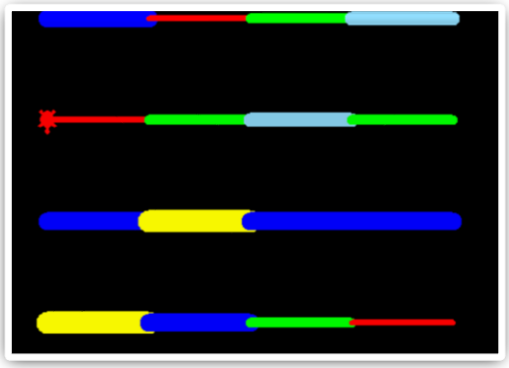
Ensuite, on répète cela pour les autres lignes sauf que la tortue doit se repositionner à la bonne place afin de dessiner les murs de la deuxième ligne, ect…



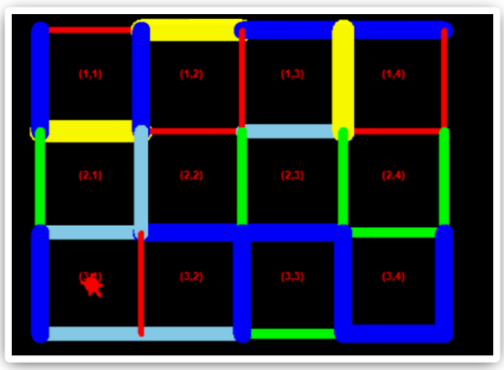
Pour ce faire lorsque la tortue aura fini de dessiner les épaisseurs des murs de la première ligne c'est-à-dire quand elle sera positionné sur la dernière colonne on va la faire reculer du pas qu'on l'a fait avancer fois le nombre de colonnes afin qu'elle puisse revenenir à sa position puis une fois 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 fois que 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.

Ci-dessus 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.



La tortue remonte pour démarrer le dessin des murs verticaux



La tortue a fini de dessiner tous les murs, horizontaux et verticaux.

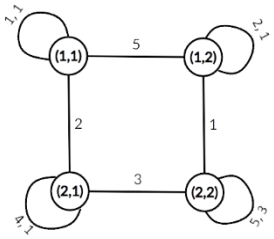
Une fois que la construction des murs horizontaux et verticaux réalisée, on fait appel a une fonction nommée **ajoute_coor()** qui ajoutera les coordonées à chaque cellule, pour ce faire, nous appliquons quelques méthodes sur Turtle, telles que : **write()**, **forwrard()**, ect ... (cf. **DessinerGrille.py** ligne 48-63).

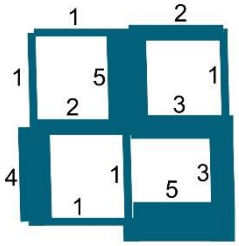
PARTIE 2

a. Structure de données

La 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 :





b. 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.

Voici un exemple illustrant cette structure :

```
grille = {
    (1, 1): {'g': [(1, 1), 4], 'h': [(1, 1), 4], 'd': [(1, 2), 5], 'b': [(2, 1), 2]},
    (1, 2): {'g': [(1, 1), 2], 'h': [(1, 2), 1], 'd': [(1, 3), 5], 'b': [(2, 2), 2]},
    (1, 3): {'g': [(1, 2), 1], 'h': [(1, 3), 4], 'd': [(1, 4), 2], 'b': [(2, 3), 5]},
    (1, 4): {'g': [(1, 3), 5], 'h': [(1, 4), 3], 'd': [(1, 4), 1], 'b': [(2, 4), 5]},
    (2, 1): {'g': [(2, 1), 1], 'h': [(1, 1), 1], 'd': [(2, 2), 3], 'b': [(3, 1), 3]},
    (2, 2): {'g': [(2, 1), 4], 'h': [(1, 2), 2], 'd': [(2, 3), 1], 'b': [(3, 2), 3]},
    (2, 3): {'g': [(2, 2), 2], 'h': [(1, 3), 1], 'd': [(2, 4), 4], 'b': [(3, 3), 4]},
    (2, 4): {'g': [(2, 3), 5], 'h': [(1, 4), 3], 'd': [(2, 4), 5], 'b': [(3, 4), 1]},
    (3, 1): {'g': [(3, 1), 5], 'h': [(2, 1), 4], 'd': [(3, 2), 3], 'b': [(3, 1), 5]},
    (3, 2): {'g': [(3, 1), 2], 'h': [(2, 2), 2], 'd': [(3, 3), 1], 'b': [(3, 2), 3]},
    (3, 3): {'g': [(3, 2), 3], 'h': [(2, 3), 4], 'd': [(3, 4), 1], 'b': [(3, 3), 2]},
    (3, 4): {'g': [(3, 3), 2], 'h': [(2, 4), 5], 'd': [(3, 4), 3], 'b': [(3, 4), 5]}
}
```

Pour les choix d'implémentation, nous avons pensé à créer une fonction 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 une grille de 3 lignes et 4 colonnes, l'appel à cette fonction nous retourne la grille de dessus.

Pour ce faire, on va remplir la grille cellule après cellule, en suivant un certain nombre de règles. Si c'est la cellule de la première ligne et première colonne, alors elle crée des épaisseurs aléatoires pour ces 4 murs adjacents, tandis que pour les cellules qui sont déjà bordées par une cellule qui a déjà été créée telle que la cellule en position (2,2), elle va récupérer l'épaisseur du mur bas de la cellule (1,2) qui aura déjà été créée et l'affecter à la valeur du mur 'h' de la cellule (2,2), car l'épaisseur du mur est commune aux deux cellules. Idem pour la valeur 'g' qui sera

récupérée via sa cellule gauche voisine ayant la position (2,1). Par ailleurs, les épaisseurs des murs 'd' et 'b' non encore créées seront gérées aléatoirement, puisque les cellules (2,3) et (3,2) ne sont pas encore créées. C'est lorsque la cellule (2,3) sera créée 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, chaque cellule du tableau aura pour mot-clé sa position dans le dictionnaire représentant la grille, tant dis que les murs qui bordent la cellule concernés auront les mots-clés 'g', 'h', 'd' et 'b', qui représenteront les cellules voisines de la cellule concerné avec des valeurs qui auront été soit générées aléatoirement, soit récupérés à l'aide des cellules voisines.

c. Choix d'un algorithme adapté pour le chemin optimal

On définit le coût comme la somme des épaisseurs des murs percés lors du parcours de la grille, il correspond donc au poids du chemin entre le sommet du départ (1,1) et celui de l'arrivé (n,m) dans le graph associé à la grille.

C'est pour cela que nous avons choisi d'utiliser l'algorithme de **Dijkstra**. En effet, cet algorithme est bien adapté à notre problème, vu que la recherche du parcours avec un coût minimal revient à retrouver un plus cours chemin entre le sommet (1, 1) et (n,m) dans le graph associé.

Fonctionnement de l'algorithme :

Le but de l'algorithme de Dijkstra avec prédécesseurs est de nous retourner :

1. Le dictionnaire D de distances entre chaque cellule de la grille et la cellule initiale (1,1), les positions des cellules de la grille seront les mots-clés de ce dictionnaire, les valeurs seront les distances.
2. Un dictionnaire P qui associe à chaque sommet du graph, son prédécesseur dans le chemin trouvé entre le sommet (1,1) et ce sommet.

A noter : Par défaut chaque valeur (sommet-départ) est initialisée à $+\infty$ mise à part le sommet de départ qui sera rentré en paramètre et qui aura la valeur 0.

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 tous 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 1-7)

Ensuite pour les sommets adjacents nous récupèrerons les cellules voisines correspondantes ainsi que son épaisseur, puis si celle-ci n'a pas été traitée, nous la 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.

```
def dijkstra_pred(G,s):
    D={}
    d={k: float('inf') for k in G}
    d[s]=0
    P={}
    while len(d)>0:
        k=minimum(d)
        for mur in G[k]:
            cel_v = G[k][mur][0]
            ep = G[k][mur][1]
            # cel_v, ep = G[k][mur]
            if cel_v not in D:
                if d[cel_v]>d[k]+ep:
                    d[cel_v]=d[k]+ep
                    P[cel_v]=k
        D[k]=d[k]
        del(d[k])
```

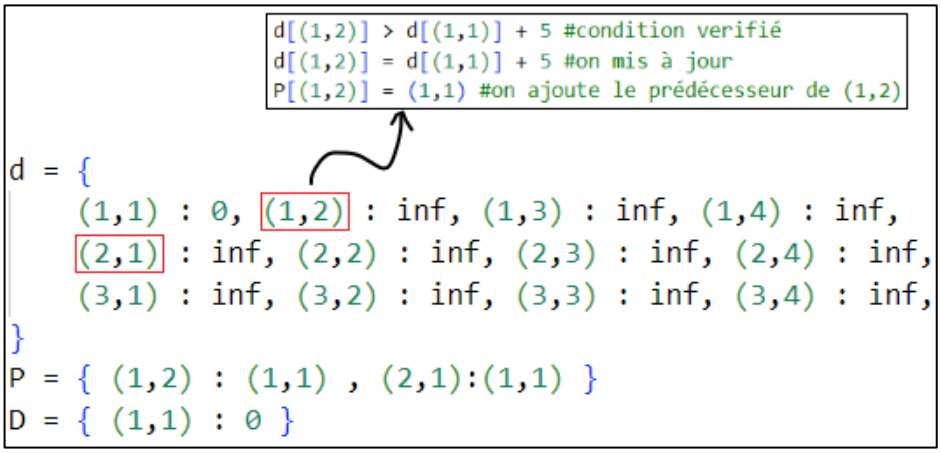
Code pour compréhension de l'exemple ci-dessous

Exemple : Pour notre sommet de Départ (1,1) on obtient le dictionnaire correspondant :

```
d = {
    (1,1) : 0, (1,2) : inf, (1,3) : inf, (1,4) : inf,
    (2,1) : inf, (2,2) : inf, (2,3) : inf, (2,4) : inf,
    (3,1) : inf, (3,2) : inf, (3,3) : inf, (3,4) : inf,
}
```

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), c'est donc les cellules (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 20) ,** on a donc trouvé un chemin plus court, dans ce cas on remplace dans notre dictionnaire d[(1,2)] par la nouvelle valeur qui est égale à 5 **(ligne 21).** Puis on affecte dans notre dictionnaire de prédécesseur la valeur (1,1) à la clé (1,2) **(ligne 22)** qui mentionnera que le prédécesseur de (1,2) est (1,1). Après avoir traité tous les sommets voisins (cellules voisines) de notre sommet considéré (dans notre cas le sommet de départ), on va supprimer le sommet considéré du dictionnaire, car on vient de passer par celui-là.

Exemple :



On 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 :

```
d = {
    (1,1) : 0, (1,2) : 5, (1,3) : 7, (1,4) : 9,
    (2,1) : 2, (2,2) : 5, (2,3) : 6, (2,4) : 9,
    (3,1) : 5, (3,2) : 8, (3,3) : 9, (3,4) : 10,
}
P = { (1,2) : (1,1), (2,1) : (1,1), (2,2) : (2,1),
    (3,1) : (2,1), (1,3) : (2,3), (2,3) : (2,2),
    (3,2) : (2,2), (2,4) : (2,3), (3,3) : (3,2),
    (1,4) : (1,3), (3,4) : (3,3)
}
```

d. 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 choisi d'utiliser l'algorithme de Bellman-Ford, ce choix est dû au fait que nous cherchons une solution optimale, or Dijkstra, étant un algorithme glouton, n'est pas en mesure de retourner un résultat optimal à tous les cours, à l'inverse de l'algorithme de Bellman-Ford. Cet algorithme nous retourne, comme Dijkstra, le dictionnaire D des distances entre chaque sommet et le sommet (1,1), ainsi que le dictionnaire P des prédécesseurs de chaque sommet, provenant du plus cours chemin entre ce sommet et le sommet (1,1).

Exemple :

```
d = {
    (1,1) : 0, (1,2) : 5, (1,3) : 7, (1,4) : 9,
    (2,1) : 2, (2,2) : 5, (2,3) : 6, (2,4) : 9,
    (3,1) : 5, (3,2) : 8, (3,3) : 9, (3,4) : 10,
}
P = { (1,2) : (1,1), (2,1) : (1,1), (2,2) : (2,1),
    (3,1) : (2,1), (1,3) : (2,3), (2,3) : (2,2),
    (3,2) : (2,2), (2,4) : (2,3), (3,3) : (3,2),
    (1,4) : (1,3), (3,4) : (3,3)
}
```

A travers cette liste on obtient :

- Le coût : d[(3,4)] = 10
- cel = (3,4)
- c = [(3,4), (3,3), (3,2), (2,2), (2,1), (1,1)]

Afin d'obtenir notre chemin à coût minimal, nous utilisons la fonction reverse(), qui retourne le chemin souhaité de la cellule (1,1) vers (3,4) puis, ainsi, notre fonction retourne :

- c = [(1,1), (2,1), (2,2), (3,2), (3,3), (3,4)]
- cout = 10

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 prédécesseurs, il suffit de remonter

depuis le sommet d'arrivé, en l'occurrence (3,4), de prédécesseur en prédécesseur jusqu'à la cellule (1,1), puis une fois cela fait, on inverse la liste, ce qui nous donnera le chemin optimal traversé. **(Voir Schéma – Exemple du dessus)**.

PARTIE 3 :

a. 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 concerné et par conséquent ajouter la valeur 'True' au mur concerné.
(Voir Enrichir_Grille.py ligne 3-21)

Liste Chemin optimal : [(1,1) ,(2,1) , (2,2) ..... (n,m)]

Cellule i **Cellule i+1**

Ici on peut voir que la comparaison entre les coordonnées de la première cellule et la deuxième cellule de la liste des chemins optimals indique que c'est au mur bas de la cellule (1,1), donc dans la clé 'b' qu'il faut rajouter la valeur **True**

Idem pour les cellules (2,1) et (2,2) c'est au mur droit de la cellule (2,1) , donc la clé 'd' qu'il faut rajouter la valeur **True**

On obtient un dictionnaire de la sorte :

(1,1) : { 'g' : [(1,1) , 4] , 'h': [(1,1) , 4] , 'd' : [(1,2), 5] , 'b' :[(2,1) , 2 , **TRUE**]
.....
.....
.....
(2,1) : { 'g' : [(2,1) , 4] , 'h': [(1,1) , 2 , **TRUE**] , 'd' : [(2,2), 3] , 'b' :[(3,1) , 3]

b. 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' a 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.

```
for cel in grille:
    for mur in grille[cel]:
        if len(grille[cel][mur]) == 3:

            # print("general : ", general(mur, pas, cel))
            pos, angle, x, y = general(mur, pas, cel)

            cel_rep = grille[cel][mur][0]
            del(grille[cel_rep][pos][2])
            ep = grille[cel][mur][1]

            t.width(6 + (ep-1)*4)
            t.seth(angle)

            t.goto(x, y)
            t.down()
            t.forward(pas/3)
            t.color("black")
            t.forward(pas/3)
            t.color(color)
            t.forward(pas/3)
```

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 pensé à une stratégie gloutonne. Le but de cette stratégie va être de comparer les épaisseurs de la cellule concernée avec ses cellules voisines gauche et bas afin d'avoir une suite de résultats localement optimaux 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 épaisseurs des murs avec la cellule voisine du bas et la cellule voisine gauche, puis d'ajouter l'épaisseur avec la valeur la plus petite dans la liste, 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 (la dernière colonne resp.) nous sommes obligés de nous déplacer sur l'épaisseur du mur droit (l'épaisseur du mur du bas resp.) On répète ce processus jusqu'à ce qu'on arrive à la dernière case (n,m) de notre grille puis nous retournons le chemin associé ainsi que le coût. **(Cf. Bonus.py ligne 1-51)**.

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, puis en remontant jusqu'à la première case **(Cf. ligne 29-46)**. Le but de cela sera de comparer, à la fin de la fonction, le coût du chemin direct avec le chemin indirect, et de retourner le chemin avec le meilleur cout. **(Cf. ligne 48-51)**.