Vendredi 6 janvier

Compte Rendu Info3A

Pouzin Pierre-Emmanuel

Poulet Alexandre

Table des matières

Partie 1

a) Pour répondre à cette première question, nous allons vous proposer » une implémentation de cette grille en utilisant une structure de données en forme de tableau a deux dimensions, ce tableau est une liste de liste.
Chaque élément du tableau représentera une cellule de la grille, et chaque cellule sera représenter par un dictionnaire contenant les informations suivantes :

- La position de la cellule (ligne, colonne)
- L'épaisseur du mur nord de la cellule
- L'épaisseur du mur sud de la cellule
- L'épaisseur du mur est de la cellule
- L'épaisseur du mur ouest de la cellule

```python
class Cellule:
    def __init__(self):
        self.lig = None
        self.col = None
        self.nord = None
        self.sud = None
        self.est = None
        self.ouest = None

    def modif(self, nord, sud, est, ouest):
        self.nord = nord
        self.sud = sud
        self.est = est
        self.ouest = ouest
```

Figure 1 class cellule

La fonction RemplirGrille parcourt chaque cellule de la grille et modifie ses côtés. Les côtés sont initialisés aléatoirement avec un nombre compris entre 1 et 5 (inclus) lorsque la cellule est sur un bord de la grille. Pour les autres cellules, les côtés sont laissés tels quels. Pour chaque cellule, la fonction modifie aussi les côtés de la cellule à sa droite et de la cellule en dessous. Si la cellule est sur un bord de la grille, elle ne modifie pas les côtés qui sont en dehors de la grille.

```python
class Grille:
    def __init__(self, lig, col):
        self.lig = lig
        self.col = col
        self.grille = [[Cellule() for i in range(col)] for j in range(lig)]


    def remplirGrille(self):
        for l in range(self.lig):
            for c in range(self.col):
                nord = random.randint(1,5)
                sud = random.randint(1,5)
                est = random.randint(1,5)
                ouest = random.randint(1,5)

                if l == 0:
                    if c == 0:
                        self.grille[l][c].modif(nord, sud, est, ouest)
                        self.grille[l][c+1].modif(self.grille[l][c+1].nord, self.grille[l][c+1].sud, self.grille[l][c+1].est, est)
                    elif c == self.col-1:
                        self.grille[l][c].modif(nord, sud, est, self.grille[l][c].ouest)
                    else:
                        self.grille[l][c].modif(nord, sud, est, self.grille[l][c].ouest)
                        self.grille[l][c+1].modif(self.grille[l][c+1].nord, self.grille[l][c+1].sud, self.grille[l][c+1].est, est)
                    self.grille[l+1][c].modif(sud, self.grille[l-1][c].sud, self.grille[l-1][c].est, self.grille[l-1][c].ouest)

                elif l == self.lig-1:
                    if c == 0:
                        self.grille[l][c].modif(nord, sud, est, ouest)
                        self.grille[l][c+1].modif(self.grille[l][c+1].nord, self.grille[l][c+1].sud, self.grille[l][c+1].est, est)
                    elif c == self.col-1:
                        self.grille[l][c].modif(nord, sud, est, self.grille[l][c].ouest)
                    else:
                        self.grille[l][c].modif(self.grille[l][c].nord, sud, est, self.grille[l][c].ouest)
                        self.grille[l][c+1].modif(self.grille[l][c+1].nord, self.grille[l][c+1].sud, self.grille[l][c+1].est, est)

                elif c == 0:
                    self.grille[l][c].modif(self.grille[l][c].nord, sud, est, ouest)
                    self.grille[l][c+1].modif(self.grille[l][c+1].nord, self.grille[l][c+1].sud, self.grille[l][c+1].est, est)
                    self.grille[l+1][c].modif(sud, self.grille[l+1][c].sud, self.grille[l+1][c].est, self.grille[l+1][c].ouest)

                elif c == self.col-1:
                    self.grille[l][c].modif(self.grille[l][c].nord, sud, est, self.grille[l][c].ouest)
                    self.grille[l+1][c].modif(sud, self.grille[l+1][c].sud, self.grille[l+1][c].est, self.grille[l+1][c].ouest)

                else:
                    self.grille[l][c].modif(self.grille[l][c].nord, sud, est, self.grille[l][c].ouest)
                    self.grille[l][c+1].modif(self.grille[l][c+1].nord, self.grille[l][c+1].sud, self.grille[l][c+1].est, est)
                    self.grille[l+1][c].modif(sud, self.grille[l+1][c].sud, self.grille[l+1][c].est, self.grille[l+1][c].ouest)

    def getCellule(self, lig, col):
        return self.grille[lig][col]
```

Figure 2 Classe Grille, fonction remplir grille

b) Nous avons fait un affichege console qui était obligatoire,le □ represente le centre de la case

```
Taille de la grille : 5
Grille initiale :
□ 5 □ 2 □ 5 □ 2 □ 4
5 - 5 - 3 - 1 - 3 -
□ 3 □ 1 □ 1 □ 5 □ 5
3 - 1 - 4 - 2 - 5 -
□ 5 □ 4 □ 5 □ 1 □ 1
1 - 4 - 2 - 5 - 2 -
□ 5 □ 3 □ 2 □ 3 □ 5
1 - 3 - 1 - 1 - 1 -
□ 4 □ 2 □ 4 □ 1 □ 1
3 - 3 - 5 - 4 - 5 -
```

Figure 3 Affichage console

```python
def __str__(self):
    chemin, cout = self.chemin_min()
    s = 0
    print("Grille initiale :")
    for i in range(self.lig):
        for j in range(self.col):
                print("□ "+str(self.getCellule(i, j).est)+" ", end="")
        print("")
        for k in range(self.col):
            print(str(self.getCellule(i, k).sud)+" - ", end="")
        print("")
```

Figure 4 Code pour l'affiche, fonction str

Partie 2

a) On peut utiliser une structure de données telle qu'un graphe pondéré pour représenter l'ensemble des chemins passant par toutes les cellules d'une grille. Dans ce graphe, chaque cellule de la grille peut être modélisée comme un nœud et chaque mur percé entre deux cellules adjacentes peut être modélisé comme une arête connectant ces deux nœuds. L'épaisseur de chaque mur peut être enregistrée comme un poids associé à l'arête correspondante. Ainsi, en parcourant le graphe, il est possible de trouver tous les chemins possibles à travers la grille en utilisant l'algorithme de parcours de graphe approprié (par exemple, l'algorithme de parcours en profondeur ou en largeur).

b) Un dictionnaire dans lequel pour chaque nœud on associe des couple nœuds poids selon leur adjacence. Cette représentation est plus efficace en termes d'espace de stockage lorsque le graphe est peu dense, car elle ne stocke que les arêtes réelles du graphe. Cependant, elle peut être moins efficace en termes de temps de calcul lorsqu'il faut parcourir tous les nœuds du graphe, car il faut parcourir chaque liste d'adjacence individuellement.

```
def cellules_voisines(self,p,i,j):
    L=[]
    if j>0:
        L.append((i,j-1))
    if i>0:
        L.append((i-1,j))
    if j<p-1:
        L.append((i,j+1))
    if i<p-1:
        L.append((i+1,j))
    return L

def tab_to_graph(self):
    p= self.lig
    G={}
    for i in range(p):
        for j in range(p):
            CV=self.cellules_voisines(p,i,j)
            G[p*i+j]=[]
            for coord in CV:
                x, y = coord
                if(p*i+j == p*x+y+1):
                    G[p*i+j].append((p*x+y, self.getCellule(x,y).est))
                elif(p*i+j == p*x+y+3):
                    G[p*i+j].append((p*x+y, self.getCellule(x,y).sud))
                elif(p*i+j == p*x+y-1):
                    G[p*i+j].append((p*x+y, self.getCellule(x,y).ouest))
                elif(p*i+j == p*x+y-3):
                    G[p*i+j].append((p*x+y, self.getCellule(x,y).nord))
    return G
```

Figure 5 fonctions cellule voisine et tab_to_graph

La méthode cellule voisine retourne un tableau dont les couples contiennent les coordonnées de leur voisin. La fonction tab_to_graph permet de construire un graphe à partir de la grille. Le graphe est représenté sous la forme d'un dictionnaire G dont chaque clé est un sommet du graphe et chaque valeur est une liste de tuples. Chaque tuple correspond à un voisin d'un sommet et contient l'identifiant du voisin et la distance entre le sommet et ce voisin.

La fonction commence par parcourir chaque cellule de la grille et ajoute un sommet au graphe pour chaque cellule. Pour chaque sommet, la fonction utilise la méthode cellules_voisines pour obtenir les coordonnées de toutes les cellules voisines de la cellule correspondante au sommet. Pour chaque cellule voisine, la fonction ajoute un tuple dans la liste des voisins du sommet. Le tuple contient l'identifiant du voisin, qui est obtenu en multipliant la ligne de la cellule par le nombre de colonnes de la grille et en ajoutant la colonne de la cellule, et la distance entre les deux sommets, qui est la valeur du mur entre les deux cellules. Enfin, la fonction retourne le graphe.

c) L'algorithme de Dijkstra permet de trouver le plus court chemin entre un sommet de départ ‘s’ et tous les autres sommets d'un graphe pondéré et orienté. L'algorithme utilise une table

de distances ‘d’ qui stocke la distance minimale connue entre le sommet de départ et chaque autre sommet du graphe. L'algorithme fonctionne en itérant sur les sommets du graphe et en mettant à jour les distances minimales connues pour chaque voisin d'un sommet k si le chemin passant par ce sommet est plus court que le chemin connu précédemment. Lorsqu'un sommet a été traité, il est copié dans une table de distances finales D et supprimé de la table de distances d. L'algorithme s'arrête lorsque la table de distances d est vide. Enfin, l'algorithme retourne également une table de prédécesseurs P qui permet de reconstituer le chemin entre le sommet de départ et chaque autre sommet.

L'algorithme de Dijkstra est particulièrement adapté dans ce cas car il permet de trouver le plus court chemin entre deux sommets d'un graphe pondéré, ce qui correspond parfaitement à la situation décrite dans l'énoncé du problème. En effet, dans la grille, chaque mur entre deux cellules adjacentes correspond à une arête du graphe pondérée par l'épaisseur du mur. De plus, l'algorithme de Dijkstra est particulièrement efficace lorsque les poids des arêtes sont positifs, ce qui est le cas ici car les épaisseurs des murs sont toujours positives.

d)

```
def chemin_min(self):
    p=self.lig
    G3=self.tab_to_graph()
    L,P=self.algo_dijkstra(G3)
    print("chemin : ",end="")
    c=p*p-1
    lst=[(c//p, c%p)]
    while c!=0:
        lst=[(P[c]//p, P[c]%p)]+lst
        c=P[c]
    print(lst)
```

Figure 6 fonction chemin_min

La fonction affiche le chemin en parcourant la table de prédécesseurs P. Elle commence par ajouter la cellule d'arrivée à une liste de coordonnées « lst » et stocke l'identifiant de la cellule dans la variable c. Ensuite, la fonction entre dans une boucle qui s'exécute tant que la cellule courante n'est pas la cellule de départ. A chaque itération, la fonction ajoute les coordonnées du prédécesseur de la cellule courante à la liste « lst « et met à jour la variable c avec l'identifiant du prédécesseur. A la fin de la boucle, la liste « lst » contient les coordonnées de toutes les cellules du chemin dans l'ordre de départ à arriver. La fonction affiche finalement la liste.

e) Le code rajouter dans la partie 2 :

```
def minimum(self,dico):
    m=float('inf')
    for k in dico:
        if dico[k] < m:
            m=dico[k]
            i=k
    return i
def cellules_voisines(self,p,i,j):
    L=[]
    if j>0:
        L.append((i,j-1))
    if i>0:
        L.append((i-1,j))
    if j<p-1:
        L.append((i,j+1))
    if i<p-1:
        L.append((i+1,j))
    return L
def tab_to_graph(self):
    p= self.lig
    G={}
    for i in range(p):
        for j in range(p):
            CV=self.cellules_voisines(p,i,j)
            G[p*i+j]=[]
            for coord in CV:
                x, y = coord
                if(p*i+j == p*x+y+1):
                    G[p*i+j].append((p*x+y, self.getCellule(x,y).est))
                elif(p*i+j == p*x+y+self.lig):
                    G[p*i+j].append((p*x+y, self.getCellule(x,y).sud))
                elif(p*i+j == p*x+y-1):
                    G[p*i+j].append((p*x+y, self.getCellule(x,y).ouest))
                elif(p*i+j == p*x+y-self.lig):
                    G[p*i+j].append((p*x+y, self.getCellule(x,y).nord))
    return G
```

Figure 7 Partie 1 du code classe grille

```python
def algo_dijkstra(self,G,s=0):
    D={}   #tableau final des distances minimales
    d={k: float('inf') for k in G}   #distances initiales
    d[s]=0   #sommet de départ
    P={}   #liste des prédécesseurs
    while len(d)>0:   #fin quand d est vide
        k=self.minimum(d)   #sommet de distance minimale pour démarrer une étape
        for i in range(len(G[k])):   #on parcourt les voisins de k
            v, c = G[k][i]   #v voisin de k, c la distance à k
            if v not in D:   #si v n'a pas été déjà traité
                if d[v]>d[k]+c:   #est-ce plus court en passant par k ?
                    d[v]=d[k]+c
                    P[v]=k   #stockage du prédécesseur de v
        D[k]=d[k]   #copie du sommet et de la distance dans D
        del d[k]   #suppression du sommet de d
    return D, P   #on retourne aussi la liste des prédécesseurs
def chemin_min(self):
    p=self.lig
    G3=self.tab_to_graph()
    L,P=self.algo_dijkstra(G3)
    print("chemin : ",end="")
    c=p*p-1
    lst=[(c//p, c%p)]
    while c!=0:
        lst=[(P[c]//p, P[c]%p)]+lst
        c=P[c]
    print(lst)
```

Figure 8 Partie 2 code classe Grille

La transformation du tableau en graphe est une méthode très efficace qua nous avion déjà vu en TP, c'est pourquoi nous l'avons utilisé ici, de même pour l'algorithme de Dijkstra qui s'utilise parfaitement dans notre situation. Pour le chemin le plus court nous nous sommes référés au TP et au cours.

Partie 3

```python
def chemin_min(self):
    p=self.lig
    G3=self.tab_to_graph()
    L,P=self.algo_dijkstra(G3)
    c=p*p-1
    lst=[(c//p, c%p)]
    while c!=0:
        lst=[(P[c]//p, P[c]%p)]+lst
        c=P[c]
    cout = 0
    for i in range(len(lst)-1):
        if lst[i+1][0] - lst[i][0] == 1:
            cout += self.getCellule(lst[i][0], lst[i][1]).sud
        elif lst[i+1][1] - lst[i][1] == 1:
            cout += self.getCellule(lst[i][0], lst[i][1]).est
    return lst, cout
```

Figure 9 fonction chemin_min modifier

On a modifié chemin_min et ajouter un « for » pour récupérer cout et l'avoir dans l'affichage demandé dans la question.

```python
def __str__(self):
    chemin, cout = self.chemin_min()
    s = 0
    print("Grille initiale :")
    for i in range(self.lig):
        for j in range(self.col):
                print("□ "+str(self.getCellule(i, j).est)+" ", end="")
        print("")
        for k in range(self.col):
            print(str(self.getCellule(i, k).sud)+" - ", end="")
        print("")

    print("\nGrille percée :")
    for i in range(self.lig):
        for j in range(self.col):
            if (i,j) == chemin[s]:
                print("X "+str(self.getCellule(i, j).est)+" ", end="")
                s+=1
            else:
                print("□ "+str(self.getCellule(i, j).est)+" ", end="")
        print("")
        for k in range(self.col):
            print(str(self.getCellule(i, k).sud)+" - ", end="")
        print("")

    print("\nChemin : "+str(chemin))

    print("\nCout : "+str(cout))
    return ""
```

Figure 10 Fonction str en entière

On a modifier notre affichage pour le rendre plus propre et lisible , et qu'il puisse aficcher le chemin minimum et percer les murs.

```
In [16]: runfile('C:/Users/pepou/Downloads/Grille.py', wdir='C:/Users/pepou/Downloads')
Taille de la grille : 4
Grille initiale :
□ 1 □ 1 □ 4 □ 5
1 - 3 - 3 - 4 -
□ 1 □ 3 □ 2 □ 2
1 - 4 - 2 - 4 -
□ 1 □ 4 □ 5 □ 1
4 - 4 - 2 - 2 -
□ 4 □ 5 □ 2 □ 4
1 - 5 - 2 - 4 -

Grille percée :
X 1 □ 1 □ 4 □ 5
1 - 3 - 3 - 4 -
X 1 □ 3 □ 2 □ 2
1 - 4 - 2 - 4 -
X 1 X 4 X 5 □ 1
4 - 4 - 2 - 2 -
□ 4 □ 5 X 2 X 4
1 - 5 - 2 - 4 -

Chemin : [(0, 0), (1, 0), (2, 0), (2, 1), (2, 2), (3, 2), (3, 3)]

Cout : 11
```

Figure 11 Affichage complet du terminal