TP 13

Implémentation et parcours de graphes

Déplacement d'un cavalier sur un échiquier

Un cavalier se déplace, lorsque c'est possible, de 2 cases dans une direction verticale ou horizontale, et de 1 case dans l'autre direction (le trajet dessine une figure en L).

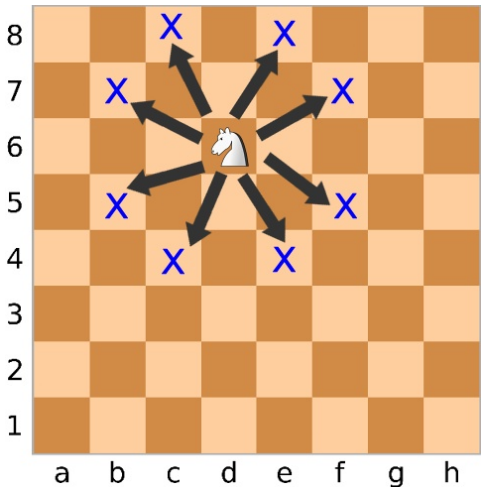

FIGURE 1 – Illustration du mouvement d'un cavalier sur un échiquier

Dans un premier temps, les cases de l'échiquier sont représentées par des tuples : le couple `(i,j)` désigne la case d'abscisse `i` et d'ordonnée `j`. Un échiquier possède 8 colonnes et 8 lignes, donc `i` et `j` seront compris entre 0 et 7.

Question 1 *Écrire une fonction* `estDansEch(i:int, j:int)->bool` *qui renvoie* `True` *si* `(i, j)` *correspond à une case valide de l'échiquier et* `False` *sinon.*

Question 2 *Écrire une fonction* `mvtsPossibles(i:int, j:int)->list` *qui renvoie la liste des cases où le cavalier peut se déplacer à partir de la case* `(i,j)` *à l'ordre près.*

Question 3 *Vérifier que :*

- `mvtsPossibles(0,0)` renvoie `[(1, 2), (2, 1)]`,
- `mvtsPossibles(3,5)` renvoie bien `[(1, 4), (1, 6), (2, 3), (2, 7), (4, 3), (4, 7), (5, 4), (5, 6)]`,
- `mvtsPossibles(7,7)` renvoie bien `[(5, 6), (6, 5)]`.

Tous ces résultats sont à l'ordre près.

Question 4 *Créer un graphe* `G` *sous la forme d'un dictionnaire d'adjacence avec pour sommets les différentes cases de l'échiquier et les arêtes qui correspondent à un mouvement possible du cavalier.*

Question 5 *Vérifiez que vous avez un graphe avec 64 sommets et 168 arêtes.*

Le codage des cases d'échiquier se fait classiquement par une chaine de caractères comprenant : une lettre minuscule pour l'abscisse (de `a` à `h`) et un chiffre pour l'ordonnée (de 1 à 8). La case en bas à gauche de l'échiquier est donc de code `'a1'`.

R `ord(c)` retourne le codage Unicode correspondant au caractère `c` et `chr(n)` renvoie le caractère dont le codage Unicode est `n`.

Question 6 *Écrire une fonction* `codage(i:int, j:int)->str` *qui renvoie le code correspondant à la case* `(i, j)`.

Question 7 *Créer un dictionnaire d'adjacence* `G2` *comme défini précédemment avec les cases maintenant nommées d'après leur code.*

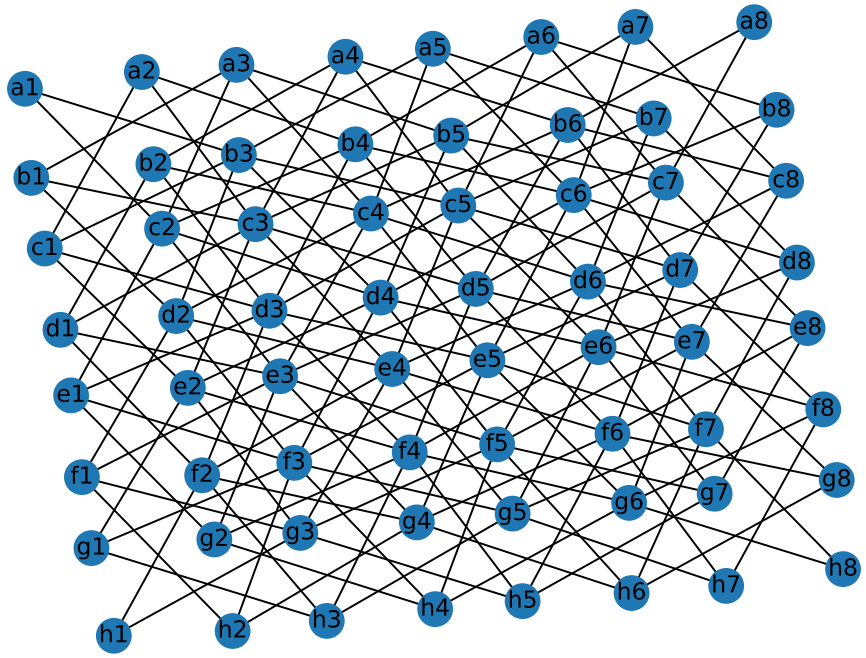

FIGURE 2 – Représentation graphique du graphe `G2`

Question 8 *Vérifier que :*

- `G2['a1']` *renvoie* `['b3', 'c2']`,
- `G2['d6']` *renvoie bien* `['b5', 'b7', 'c4', 'c8', 'e4', 'e8', 'f5', 'f7']`,
- `G2['h8']` *renvoie bien* `['f7', 'g6']`.

Tous ces résultats sont à l'ordre près.

Question 9 *Écrire une fonction* `largeur_dist(G:dict, dep:tuple)->dict:` *qui prend en entrée un graphe codé par un dictionnaire d'adjacence* `G` *et un sommet de départ dep et renvoie un dictionnaire de distance à partir du sommet* `dep`. *Pour ce faire, vous vous inspirerez du parcours en largeur fourni. Si un sommet n'est pas atteignable depuis le depuis* `dep`, *la valeur associée doit être de* `-1`.

```
def bfs(G:dict, s) -> None:
    """
    G : graphe sous forme de dictionnaire d'adjacence
    s : sommet du graphe objet non mutable quelconque.
    """
    visited = {}
    for sommet,voisins in G.items():
        visited[sommet] = False
    # Le premier sommet àvisiter entre dans la file
```

```
    file = deque([s])
    while len(file) > 0:
        # On visite la tête de file
        tete = file.pop()
        # On vérifier qu'elle n'a pas été visitée
        if not visited[tete]:
            # Si on ne l'avait pas visité, maintenant c'est le cas :)
            visited[tete] = True
            # On met les voisins de tete dans la file
            for v in G[tete]:
                file.appendleft(v)
```

Question 10 *Vérifier que vous obtenez le même résultat que sur la figure suivante en affichant les différentes valeurs de distance depuis* `dep = (0, 0)` *et* `dep = (4, 3)` *. La fonction print peut ne pas revenir à la ligne si on précise un argument optionnel end différent de* `n` *.*

5	4	5	4	5	4	5	6		4	3	2	3	2	3	2	3
4	3	4	3	4	5	4	5		3	2	3	2	3	2	3	2
3	4	3	4	3	4	5	4		2	3	4	1	2	1	4	3
2	3	2	3	4	3	4	5		3	2	1	2	3	2	1	2
3	2	3	2	3	4	3	4		2	3	2	3	0	3	2	3
2	1	4	3	2	3	4	5		3	2	1	2	3	2	1	2
3	4	1	2	3	4	3	4		2	3	4	1	2	1	4	3
0	3	2	3	2	3	4	5		3	2	3	2	3	2	3	2

FIGURE 3 – Distances depuis `(0, 0)` et `(4, 3)`

Question 11 *Pourquoi le parcours en profondeur n'est pas adapté à la résolution de ce problème ?*

Snakes and Ladders - Partie 2

La première partie a déjà été traitée et elle porte sur l'algorithme glouton et les dictionnaires. Le sujet et la correction sont disponibles sur le site de la classe.

Présentation du jeu

Le jeu *serpents et échelles* est un jeu de société où on espère monter les échelles en évitant de trébucher sur les serpents. Il provient d'Inde et est utilisé pour illustrer l'influence des vices et des vertus sur une vie.

Le plateau

- Le plateau comporte 100 cases numérotées de 1 à 100 en boustrophédon [1] : le 1 est en bas à gauche et le 100 est en haut à gauche ;
- des serpents et échelles sont présents sur le plateau : les serpents font descendre un joueur de sa tête à sa queue, les échelles font monter un joueur du bas de l'échelle vers le haut.

Déroulement

- Chaque joueur a un pion sur le plateau. Plusieurs pions peuvent être sur une même case. Les joueurs lancent un dé à tour de rôle et ils avancent du nombre de cases marqués sur le dé. S'ils atterrissent sur un bas d'échelle ou une tête de serpent, ils vont directement à l'autre bout ;
- les joueurs commencent sur une case 0 hors du plateau : la première case où mettre leur pion correspond donc au premier lancer de dé ;

1. à la manière du bœuf traçant des sillons, avec alternance gauche-droite et droite-gauche

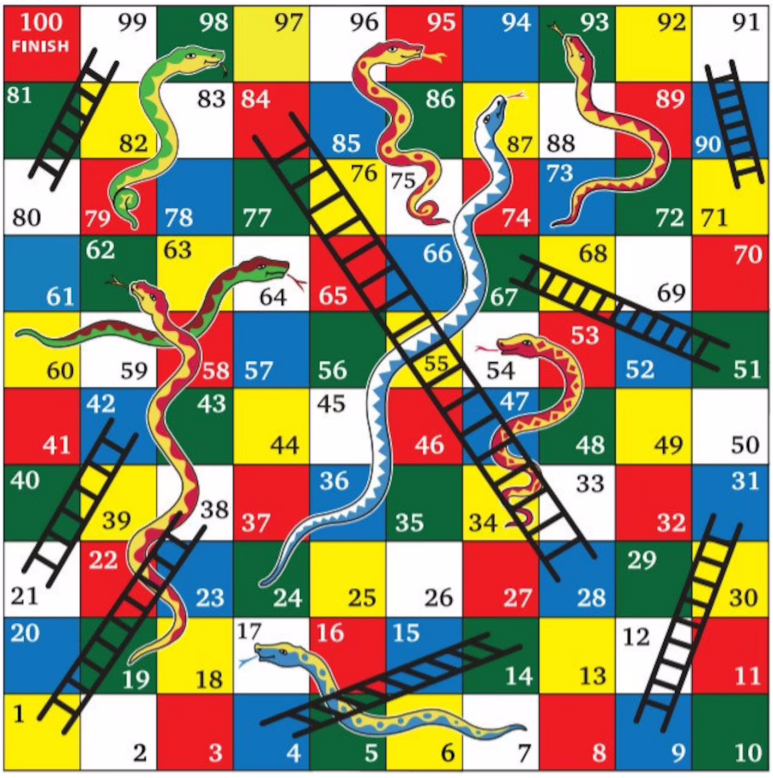

FIGURE 4 – Exemple d'un plateau de serpents et échelles

- le premier joueur à arriver sur la case 100 a gagné ;
- il existe 3 variantes quand la somme de la case actuelle et du dé dépasse 100 :
  - le rebond : on recule d'autant de cases qu'on dépasse ;
  - l'immobilisme : on n'avance pas du tout si on dépasse :
  - la fin rapide : on va à la case 100 quoi qu'il arrive.

On utilisera les notations suivantes pour les complexités : N_{cases}, le nombre de cases du plateau (100), et N_{SeE} la somme du nombre de serpents et du nombre d'échelle (16 dans notre exemple). Ces variables ne sont pas déclarées dans le script.

Étude du graphe correspondant

Élimination des doublons

Question 12 *Écrire une fonction* `eliminationDoublon_naif(L: [int]) -> [int]` *: qui renvoie une liste ayant exactement les mêmes éléments, mais sans répétition. Vous n'utiliserez ni dictionnaire, ni tri.*

Question 13 *Prouver que cette fonction est de complexité quadratique en la longueur de la liste dans le pire des cas.*

Question 14 *Écrire une fonction* `eliminationDoublon_tri(L: [int]) -> [int]` *: ayant une complexité en* $O\big(len(L)\log(len(L))\big)$ *en utilisant un tri sur* `L`.

Question 15 *Proposer une fonction* `eliminationDoublon_dict(L: [int]) -> [int]` *: ayant une complexité linéaire en la longueur de la liste. Vous pourrez vous aider d'un dictionnaire.*

Construction d'un graphe

On souhaite créer un graphe à partir du jeu : chaque case stable est un sommet et il existe un arc de la case `case1` à la case `case2` si un lancer de dé permet d'aller de `case1` à `case2`. On notera que la case 100 n'a pas de successeurs (puisque le jeu s'arrête). De plus, une case 0 sera utilisée pour prendre en compte le départ. Une représentation de ce graphe est donnée sur la figure 5.

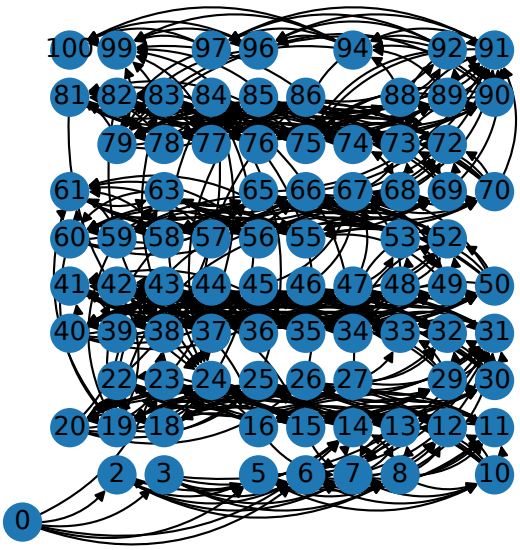

FIGURE 5 – Graphe correspondant au plateau de la figure 4

On rappelle que les éléments ci-dessous avaient été donnés :

```
dSeE = {1: 38, 4: 14, 9: 31, 17: 7, 21: 42, 28: 84, 51: 67, 54: 34,
        62: 19, 64: 60, 71: 91, 80: 99, 87: 24, 93: 73, 95: 75, 98: 79}

def casesAccessibles(case: int) -> [int]:
    return [avanceCase(case, de, "q") for de in range(1,7)]
```

Question 16 *En vous aidant des fonctions* `casesAccessibles()` *et* `eliminationDoublon_dict()`*, construire un graphe orienté* `G` *sous la forme d'un dictionnaire d'adjacence (dictionnaire de listes).*

Question 17 *Pourquoi a-t-on besoin d'éliminer des doublons ? Prenez un exemple pour illustrer.*

Question 18 *Quel est le nombre de sommets de ce graphe en fonction de* N_{cases} *et* N_{seE}*.*

Afin de résoudre le problème du parcours à nombre de coups minimum, on a mis au point une fonction qui trouve si un chemin existe entre un sommet de départ et un sommet d'arrivée :

```
def chemin(G, depart, arrivee):
    predecesseur = {}
    file = deque([depart])
    predecesseur[depart] = None
    while file:
        sommet = file.popleft()
        for successeurDeSommet in G[sommet]:
            if successeurDeSommet not in predecesseur.keys():
                file.append(successeurDeSommet)
                predecesseur[successeurDeSommet] = sommet
    return arrivee in predecesseur
```

Question 19 *Sur quel type de parcours est basée la fonction* `chemin` *? Justifiez cette réponse. Précisez l'intérêt d'utiliser ce parcours ?*

Pour l'instant, la fonction renvoie un booléen, mais on souhaiterait qu'elle renvoie l'ensemble des cases par lesquelles on est passé pour passer de `depart` à `arrivee` avec, comme premier élément de la liste retournée `depart` et comme dernier élément de la liste retournée `arrivee`. La fonction renverra une liste vide si aucun chemin n'a été trouvé.

Question 20 *Remplacer la ligne 11 par un ensemble de lignes permettant de répondre à cette exigence.*

Question 21 *Écrire une fonction* `partieOptimale() -> [int]` *qui renvoie une liste de case à longueur minimum partant de 0 et arrivant à 100 en vous aidant de la fonction mise en place précédemment.*

Cette dernière fonction nous renvoie `[0, 38, 39, 45, 67, 91, 94, 100]`. Pour ce plateau, elle ne nous donne pas une partie plus courte que l'algorithme glouton, et un chemin qui n'est pas fondamentalement différent.

Dessiner le plateau

On souhaite faire une représentation schématique du plateau (voir figure 6) avec le code suivant :

```
for case in range(1, 101): # pour les cases 1 à 100
    i, j = position(case)
    plt.text(i, j, str(case),
        horizontalalignment='center', verticalalignment='center')

for caseD, caseA in dSeE.items(): # ligne 6
    iD, jD = position(caseD)
    iA, jA = position(caseA)
    if caseA > caseD : # ligne 9
        couleur = 'b'
    else:
        couleur = 'r'
    plt.plot(iA, jA, '^', color=couleur)
    plt.plot(iD, jD, 'o', color=couleur)
    plt.plot([iA, iD], [jA, jD], color=couleur)
plt.axis("equal")
plt.show()
```

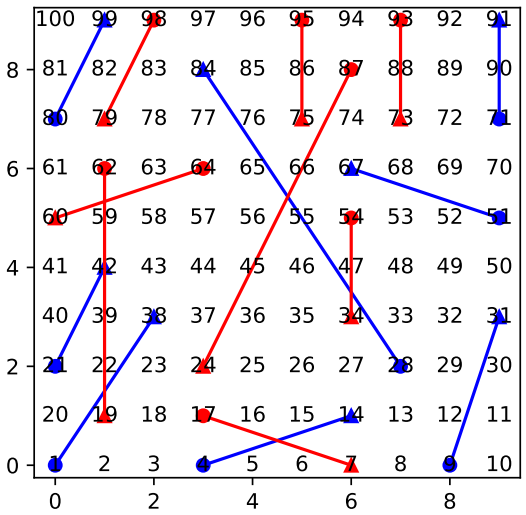

FIGURE 6 – Représentation schématisée du plateau de la figure 4

Question 22 *Écrire une fonction* `position(case: int) -> (int, int)` *qui renvoie les coordonnées de la case numérotée* `case` *sous las forme d'un couple d'entier. On doit avoir* `position(1)` *qui renvoie* `(0, 0)` *(coin en base à gauche) et* `position(91)` *qui renvoie* `(9, 9)` *(coin en haut à droite).*

Question 23 *En commentant la ligne 6, dites à quoi correspondent les variables* `caseD` *et* `caseA` *par rapport au dictionnaire* `dSeE`.

Question 24 *Commenter le rôle des lignes 9 à 12 du code fourni.*

Annexe

Utilisation du module `random`

On vous donne les docstrings correspondant à deux fonctions du module `random` :

```
randint(a, b) method of random.Random instance
    Return random integer in range [a, b], including both end points.
```

```
choice(seq) method of random.Random instance
    Choose a random element from a non-empty sequence.
```

Complexité des opérations sur les listes et dictionnaires

Principales opérations sur les listes

`n`, longueur de la liste `L`, `k`, un indice valide en négatif (`1` à `n`).

Opération	Moyen
Longueur (`len(L)`)	$O(1)$
Accès en lecture d'un élément	$O(1)$
Accès en écriture d'un élément	$O(1)$
Copie (`L.copy()` ou `L[:]`)	$O(n)$
Ajout (`L.append(elt)` ou `L+=[elt]`)	$O(1)$
Extension (`L1.extend(L2)` ou `L1+=L2`)	$O(n_2)$
Concaténation (`L1 + L2`)	$O(n_1+n_2)$
Test de présence (`elt in L`)	$O(n)$
Désempiler dernier (`L.pop()`)	$O(1)$
Désempiler autre (`L.pop(-k)`)	$O(k)$
Maximum ou minimum (`max(L)` et `min(L)`)	$O(n)$
Tri (`L.sort()` ou `sorted(L)`)	$O(n\log(n))$

Principales opérations sur les dictionnaires

`n`, longueur du dictionnaire `d`, `k`, une clé du dictionnaire.

Opération	Moyen
Longueur (`len(d)`)	$O(1)$
Accès en lecture d'un élément (`x = d[k]`)	$O(1)$
Accès en écriture d'un élément (`d[k] = x`)	$O(1)$
Copie (`d.copy()`)	$O(n)$
Ajout (`d[k] = x` la première fois)	$O(1)$
Test de présence (`k in d`)	$O(1)$
Retrait d'un élément (`del d[k]` ou `d.pop(k)`)	$O(1)$

Utilisation du module `matplotlib`

`plt.text(x, y, s)` permet de placer la chaine de caractères `s` aux coordonnées `(x, y)`. Des arguments optionnels permettent la méthode de placement horizontal (`horizontalalignment` qui peut prendre les valeurs `'center'`, `'right'` ou `'left'`) et vertical (`verticalalignment` qui peut prendre les valeurs `'center'`, `'top'`, `'bottom'`, `'baseline'`, `'center_baseline'`).

`plt.plot(x, y, '^')` permet de placer un point aux coordonnées `(x, y)` sous la forme d'un triangle vers le haut. Beaucoup d'autres existent, en plus du triangle vers le haut, dont (liste non-exhaustive) : `'v'`, `'<'` et `'>'` pour des triangles orientés différemment ; `'.'` et `'o'` pour des disques plus ou moins gros ; `'s'`, `'p'` et `'h'` pour des figures régulières à 4, 5 ou 6 côtés.

`plt.plot(Sx, Sy)` permet de tracer une courbe en trait plein en reliant les points dans l'ordre des 2 séquences données en entrées (le point de coordonnées `(Sx[i], Sy[i])` est lié au point de coordonnées `(Sx[i+1], Sy[i+1])`.

Un argument de couleur peut être utilisé avec `plt.plot` : des raccourcis existent pour les couleurs le plus fréquentes : `'b'` pour bleu, `'r'` pour rouge, `'g'` pour vert, `'c'` pour cyan, `'m'` pour magenta, `'y'` pour jaune, `'k'` pour noir et `'w'` pour blanc.

`plt.axis('equal')` permet de contraindre un ratio de 1 entre l'échelle en abscisses et en ordonnées (repère orthonormé).

`plt.show()` crée la figure en cours dans une nouvelle fenêtre.