Proposition de corrigé du TPGraphes et labyrinthes, application à la porosité

1 Sortie d'un labyrinthe : parcours en profondeur

1. La case $(1,1)$ possède 3 voisins $[(1,2),(2,1),(0,1)]$. La case $(5,4)$ possède un seul voisin $(4,4)$.
2. Les voisins ne peuvent être que dans les 4 directions, reste à tester les cas en veillant à ne pas "sortir" du labyrinthe.

```
def voisins(case, A):
    i, j = case
    voisins_possibles = [(i-1, j), (i+1, j), (i, j-1), (i, j+1)]
    voisins_accessibles = [(x, y) for x, y in voisins_possibles if 0 <= x < len(A)
                           and 0 <= y < len(A[0]) and A[x][y] == 1]
    return voisins_accessibles
```

3. Il vient de suite :

```
def casedepart(A):
    for j in range(len(A[0])):
        if A[0][j] == 1:
            return (0, j)
    return None
```

4. On crée la matrice ligne par ligne :

```
def init_visite(A):
        matrice = []
        for k in range(len(A)):
                matrice.append([False] * len(A[0]))
        return(matrice)
```

5. On applique la définition de l'énoncé :

```
def est_visite(s, S_visites):
    i, j = s
    return S_visites[i][j]
```

6. On reprend ligne par ligne le pseudo-code en n'oubliant pas de faire appel aux propriétés des piles.

```
def parcoursProfondeur(A, s):
    S_visites = init_visite(A)
    pile = Creer()
    Empiler(s, pile)
    S_visites[s[0]][s[1]] = True

    while not EstVide(pile):
        s_courant = Depiler(pile)
        for voisin in voisins(s_courant, A):
            if not est_visite(voisin, S_visites):
                Empiler(voisin, pile)
                S_visites[voisin[0]][voisin[1]] = True

    return S_visites
```

7. Le code est semblable à la question précédente. On a déjà la fonction `casedepart`, on parcourt alors en profondeur en partant de cette case.Enfin, il suffit de tester si une case de la ligne du bas du labyrinthe a été visitée.


```
def parcoursLabyrinthe(A):
    s=casedepart(A)
    S_visites = parcoursProfondeur(A, s)
    # Vérifie s'il y a une case visitée sur la dernière ligne
    for case in S_visites[-1]:
        if case:
            return True

    # si on est arrivé jusque là, c'est qu'aucune sortie n'a été trouvée.
    return False
```

2 Application à la physique : la percolation

8. La fonction `creerMilieu` est donnée, il ne reste qu'à renvoyer un booléen, résultat de la fonction `parcoursLabyrinthe`.

```
def circulation(n, p):
    laby = creerMilieu(n, p)
    return parcoursLabyrinthe(laby)
```

9. Il suffit d'initialiser un compteur de succès et d'incrémenter à chaque fois que `circulation` renvoie `True`.

```
def simulationEcoulement(n, p, nb):
    count = 0
    for _ in range(nb):
        if circulation(n, p):
            count += 1
    print(f"Pour␣n={n},␣p={p},␣et␣{nb}␣simulations,␣{count␣/␣nb␣*␣100:.2f}%␣des␣simulations
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣ont␣abouti␣à␣un␣écoulement.")
```

10. On peut tester de dixième en dixième puis raffiner. On fait alors varier la taille du milieux et la porosité.

```
def rechercheporositeCritique(n, nb_simulations=200):
    porosites = np.linspace(0.55, 0.65, 10)
    results = []
    for p in porosites:
        count = 0
        for _ in range(nb_simulations):
            if circulation(n, p):
                count += 1
        results.append((p, count / nb_simulations))
    return results
```

11. On reprend quasiment le même code que précédemment, mais cette fois-ci il faut s'arranger pour renvoyer aussi la matrice des sommets visités.

```
def parcoursLabyrintheAvecCasesVisitees(A):
    s=casedepart(A)
    return parcoursProfondeur(A, s)
```

La fonction `Circulation2` s'en déduit directement :

```
def circulation2(n, p):
    laby = creerMilieu(n, p)
    cases_visitees = parcoursLabyrintheAvecCasesVisitees(laby)
    return cases_visitees, laby
```