corrigé

Parcours dans un labyrinthe

1. Parcours de labyrinthe

Question 1. Le parcours du labyrinthe se poursuit tant que la case d'arrivée, de coordonnées $(p-1, q-1)$, n'a pas été vue. À chaque étape est examiné le sommet (i, j) de la pile : s'il existe parmi les cases voisines accessibles au moins une case inexplorée, celle-ci intègre la pile (on avance dans le labyrinthe) ; dans le cas contraire le sommet (i, j) est supprimé (on recule).

```
def explorer(lab):
    pile = Pile()
    dejavu = [[False for j in range(lab.q)] for i in range(lab.p)]
    pile.push((0, 0))
    dejavu[0][0] = True
    while not dejavu[lab.p-1][lab.q-1]:
        (i, j) = pile.top()
        if lab.tab[i][j].S and not dejavu[i+1][j]:
            pile.push((i+1, j))
            dejavu[i+1][j] = True
        elif lab.tab[i][j].E and not dejavu[i][j+1]:
            pile.push((i, j+1))
            dejavu[i][j+1] = True
        elif lab.tab[i][j].N and not dejavu[i-1][j]:
            pile.push((i-1, j))
            dejavu[i-1][j] = True
        elif lab.tab[i][j].W and not dejavu[i][j-1]:
            pile.push((i, j-1))
            dejavu[i][j-1] = True
        else:
            pile.pop()
    return pile
```

Lorsque cette fonction se termine, la pile contient les cases à parcourir pour rejoindre la sortie, la case d'arrivée se trouvant au sommet.

2. Génération de labyrinthes

Génération par exploration exhaustive

Question 2. Le processus débute par le choix d'une case arbitraire déposée dans la pile, puis, tant que la pile n'est pas vide, on dépile une case (i, j) et on recherche les cases inexplorées parmi ses voisines. S'il en existe au moins deux, la case (i, j) réintègre la pile (une case ne quitte la pile qu'une fois toutes ses voisines explorées). L'une des cases voisines inexplorée est choisie au hasard et intègre le sommet de la pile tandis que le mur correspondant est percé.

```
def creation1(lab):
    pile = Pile()
    dejavu = [[False for j in range(lab.q)] for i in range(lab.p)]
    i, j = random.randrange(lab.p), random.randrange(lab.q)
    pile.push((i, j))
    dejavu[i][j] = True
    while not pile.empty():
        (i, j) = pile.pop()
        v = []
        if j < lab.q-1 and not dejavu[i][j+1]:
            v.append('E')
```

```
        if i > 0 and not dejavu[i-1][j]:
            v.append('N')
        if j > 0 and not dejavu[i][j-1]:
            v.append('W')
        if i < lab.p-1 and not dejavu[i+1][j]:
            v.append('S')
        if len(v) > 1:
            pile.push((i, j))
        if len(v) > 0:
            c = v[random.randrange(len(v))]
            if c == 'N':
                lab.tab[i][j].N = True
                lab.tab[i-1][j].S = True
                pile.push((i-1, j))
                dejavu[i-1][j] = True
            elif c == 'W':
                lab.tab[i][j].W = True
                lab.tab[i][j-1].E = True
                pile.push((i, j-1))
                dejavu[i][j-1] = True
            elif c == 'S':
                lab.tab[i][j].S = True
                lab.tab[i+1][j].N = True
                pile.push((i+1, j))
                dejavu[i+1][j] = True
            else:
                lab.tab[i][j].E = True
                lab.tab[i][j+1].W = True
                pile.push((i, j+1))
                dejavu[i][j+1] = True
```

Question 3. Au début de l'algorithme, le labyrinthe possède pq composantes connexes ; chaque étape lui en fait perdre une donc on creuse exactement $pq-1$ murs avant d'en avoir fini (il ne reste plus alors qu'une seule composante connexe, le labyrinthe). Initialement il y avait $p(q-1)+q(p-1)$ murs internes donc quand le labyrinthe est créé il en reste $pq-p-q+1=(p-1)(q-1)$.

Génération par fusion des chemins

Question 4. On commence par créer une structure d'union-find à partir des pq cases, formant chacune un singleton de la partition des cases. D'après la question précédente, nous devons réunir $pq-1$ classes distinctes pour ne plus en obtenir qu'une seule.
Pour réunir deux classes, on tire au hasard une case et une de ses voisines. Si ces deux cases appartiennent à deux classes différentes, les deux classes sont fusionnées et le mur entre ces deux cases percé.

```
def creation2(lab):
    uf = UnionFind([(i, j) for i in range(lab.p) for j in range(lab.q)])
    for _ in range(lab.p * lab.q - 1):
        while True:
            i, j = random.randrange(lab.p), random.randrange(lab.q)
            k, l = random.choice([(i+1, j), (i-1, j), (i, j+1), (i, j-1)])
            if 0 <= k < lab.p and 0 <= l < lab.q and uf.find((i, j)) != uf.find((k, l)):
                break
        uf.union((i, j), (k, l))
        if k == i + 1:
            lab.tab[i][j].S = True
            lab.tab[k][l].N = True
        elif k == i - 1:
            lab.tab[i][j].N = True
```

```
            lab.tab[k][l].S = True
        elif l == j + 1:
            lab.tab[i][j].E = True
            lab.tab[k][l].W = True
        else:
            lab.tab[i][j].W = True
            lab.tab[k][l].E = True
```

Comparaison des deux méthodes

Question 5. Pour cette dernière question nous avons besoin d'une fonction qui calcule la hauteur d'une pile, ce qui correspond à la longueur du chemin qui relie l'entrée à la sortie :

```
def longueur(p):
    s = 0
    while not p.empty():
        p.pop()
        s += 1
    return s
```

Le script suivant détermine la longueur moyenne de la solution pour un labyrinthe créé par la première méthode :

```
N = 100
s = 0

for i in range(N):
    lab = Labyrinthe(50, 50)
    creation1(lab)
    p = explorer(lab)
    s += longueur(p)

print('Longueur moyenne pour {} essais : {}'.format(N, s / N))
```

Pour la première méthode on obtient :

```
Longueur moyenne pour 100 essais : 590.44
```

et pour la seconde :

```
Longueur moyenne pour 100 essais : 175.42
```

On trouvera figures 1 et 2 deux exemples de labyrinthes créés par chacune de ces deux méthodes dans une grille 20×30.

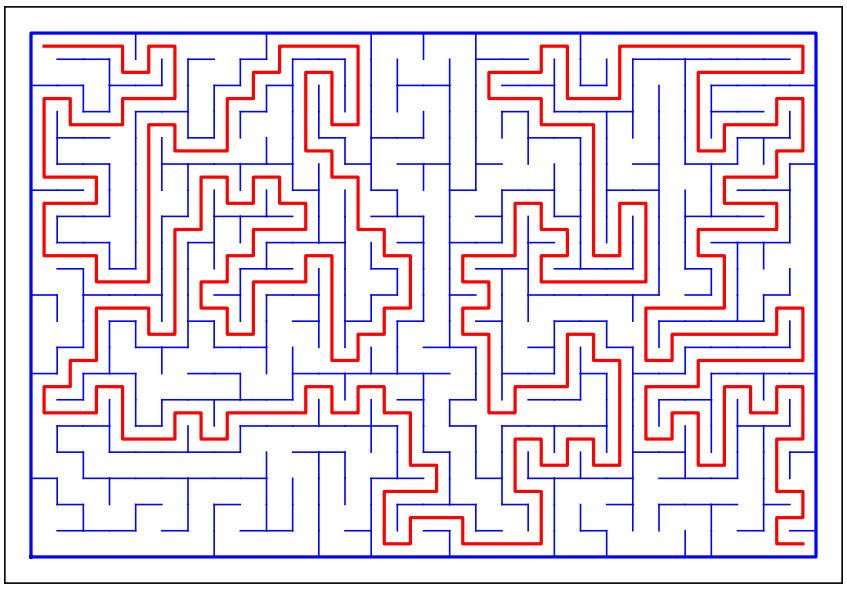

Figure 1 – Un labyrinthe 20×30 généré par exploration exhaustive.

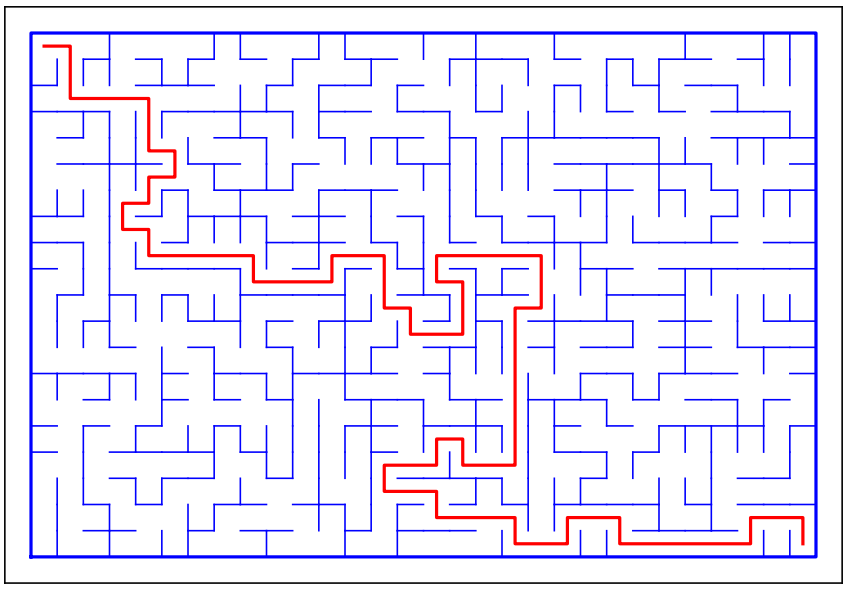

Figure 2 – Un labyrinthe 20×30 généré par fusion des chemins.