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 au moins une case inexplorée, la case (i, j) réintègre la pile ainsi que cette nouvelle case.

```
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.pop()
        if lab.tab[i][j].S and not dejavu[i+1][j]:
            pile.push((i, 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))
            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, 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))
            pile.push((i, j-1))
            dejavu[i][j-1] = True
    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

Question 2. Ma fonction utilise la fonction `randint` du module `numpy.random` : `randint(n)` renvoie une valeur arbitraire prise dans l'intervalle $[\![0, n-1]\!]$.
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 une, 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.

```
from numpy.random import randint

def creation(lab):
    pile = Pile()
    dejavu = [[False for j in range(lab.q)] for i in range(lab.p)]
    i, j = randint(lab.p), randint(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) > 0:
            pile.push((i, j))
            c = v[randint(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
```

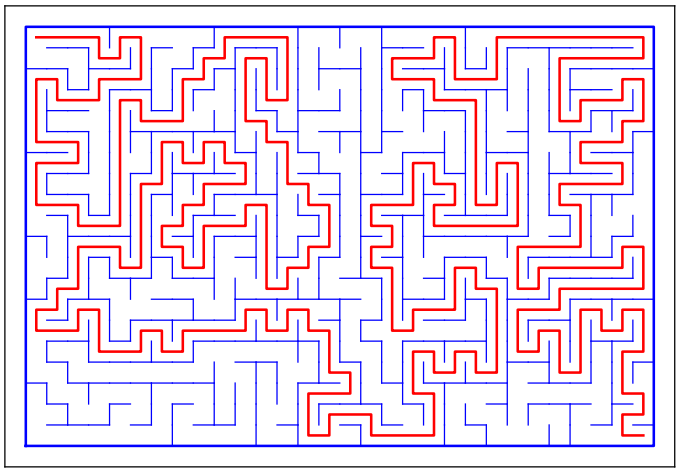

FIGURE 1 – Un labyrinthe 20×30 généré par exploration exhaustive.

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 $pq-p-q+1=(p-1)(q-1)$.