Labyrinthes

Marc Lorenzi - 7 juin 2018

In [1]:
```
import matplotlib.pyplot as plt
%matplotlib inline
import random
```

Dans ce notebook nous allons répondre à deux questions :

- Comment créer un labyrinthe "aléatoire" ?
- Comment trouver son chemin dans un labyrinthe ?

Nous allons voir que les réponses à ces deux questions sont identiques.

1. Créer un labyrinthe

1.1 Le rectangle

Soient M et N deux entiers non nuls. Considérons l'ensemble des points à coordonnées entières du rectangle $[0, M[\times[0, N[$. Nous noterons S ce rectangle. Chaque point du rectangle (nous dirons **sommet**) possède en général 4 voisins, les sommets au-dessus, au-dessous, à gauche et à droite du point en question. Bien entendu, les points au bord du rectangle n'ont que 3 voisins, et les coins du rectangle n'en ont que deux. Nous appellerons **arête** tout couple (p, q) tel que p et q soient voisins. Nous noterons A l'ensemble des arêtes. Le couple $G = (S, A)$ est ce que l'on appelle un **graphe**.

La fonction `voisins` renvoie la liste des voisins d'un sommet. Elle calcule les 4 voisins de celui-ci et renvoie ceux qui sont admissibles, c'est à dire dont les coordonnées sont respectivement dans $[0, M[$ et $[0, N[$.

In [2]:
```
def voisins(p, M, N):
    s = [gauche(p), droit(p), bas(p), haut(p)]
    return [v for v in s if admissible(v, M, N)]
```

In [3]:
```
def gauche(p):
    x, y = p
    return (x - 1, y)

def droit(p):
    x, y = p
    return (x + 1, y)

def bas(p):
    x, y = p
    return (x, y - 1)

def haut(p):
    x, y = p
    return (x, y + 1)
```

In [4]:
```
def admissible(v, M, N):
    x, y = v
    return x >= 0 and x < M and y >= 0 and y < N
```

In [5]:
```
voisins((3, 2), 10, 10)
```

Out[5]:
```
[(2, 2), (4, 2), (3, 1), (3, 3)]
```

In [6]:
```
voisins((9, 2), 10, 10)
```

Out[6]:
```
[(8, 2), (9, 1), (9, 3)]
```

In [7]:
```
voisins((0, 0), 10, 10)
```

Out[7]:
```
[(1, 0), (0, 1)]
```

1.2 C'est quoi un labyrinthe ?

Un chemin dans un graphe G d'un sommet p à un sommet q est une liste d'arêtes reliant p à q. Évidemment, dans le cas de notre rectangle, il existe des tas de chemins reliant un sommet à un autre. Des questions rigolotes se posent (du genre "combien de chemins ?") mais ce n'est pas à l'ordre du jour.

Qu'appellerons-nous un "labyrinthe" ? Je vais choisir une définition qui, bien que restrictive, est intéressante.

Définition : Un sous-graphe de $G = (S, A)$ est un graphe $G' = (S, A')$ où S est l'ensemble des sommets de G et $A' \subset A$ est une partie de l'ensemble des arêtes de G.

Remarquons que G et G' ont les mêmes sommets. Notre définition de sous-graphe n'est pas la "vraie" définition, il y en a de plus générales, mais elle nous conviendra ici.

Définition : un labyrinthe est un sous-graphe L de G tel que pour tous sommets p et q il existe un **unique** chemin de p à q dans L.

Dit autrement, un labyrinthe est un graphe dans lequel il est difficile de trouver son chemin : il n'y en a qu'un seul !

En fait :

- L'existence d'un chemin entre tout couple de sommets est la notion de **connexité**.
- L'unicité du chemin est la notion d'**acyclicité**.

En résumé, un labyrinthe est un graphe connexe sans cycles. Un tel graphe porte un nom : cela s'appelle un **arbre**.

Un labyrinthe est un sous-graphe du rectangle qui est un arbre.

1.3 Création

Alors, comment créer un labyrinthe ? On part d'un point p_0 du rectangle et on explore le rectangle à partir de ce point. Comment ? En visitant les voisins de p_0 et en explorant le rectangle à partir de ces voisins. Bref, "de proche en proche". Voici la définition de la fonction `creer_labyrinthe` . Je l'expliquerai après.

In [8]:
```
def creer_labyrinthe(p0, M, N):
    visites = set([p0])
    s = [p0]
    peres = {}
    while s != []:
        p = s.pop()
        vs = voisins(p, M, N)
        melanger(vs)
        for v in vs:
            if v not in visites:
                visites.add(v)
                s.append(v)
                peres[v] = p
    return peres
```

On crée

- Un ensemble `visites`. Chaque fois que l'on recontre un nouveau sommet lors de l'exploration, on l'ajoute à cet ensemble.
- Une liste s qui contient initialement le sommet p_0. Chaque fois que l'on visite un sommet on l'ajoute à s. La liste s est la liste des sommets à partir desquels une exploration doit être lancée.
- Un dictionnaire `peres`. À chaque fois que l'on découvre un voisin v d'un sommet p, on décrète que p est le père de v.

Que fait la boucle `while` ? À chaque itération,

- On extrait un sommet de la liste s.
- On récupère la liste de ses voisins et on la mélange (un peu de hasard, ça brise la monotonie)
- On fait des trucs aux voisins qui n'ont pas encore été visités. Quels trucs ?
  - On les ajoute aux sommets visités
  - On les ajoute à la liste du boulot à faire (la liste s)
  - On leur donne un père

Enfin, la fonction renvoie le dictionnaire `peres`. Chaque sommet du rectangle possède un et un seul père, sauf le sommet p_0 dont on est parti, qui n'en a pas.

La fonction `melanger` est intéressante en soi, mais je ne détaillerai pas son code. On utilise l'algorithme de Fisher-Yates, qui garantit un mélange uniforme sur toutes les permutations.

In [9]:
```
def melanger(s):
    n = len(s)
    for k in range(n):
        j = random.randint(k, n - 1)
        s[j], s[k] = s[k], s[j]
```

Testons sur un petit rectangle.

In [10]:
```
t = creer_labyrinthe((0, 0), 5, 5)
print(t)
```

```
{(1, 0): (0, 0), (0, 1): (0, 0), (0, 2): (0, 1), (1, 1): (0, 1), (2,
1): (1, 1), (1, 2): (1, 1), (1, 3): (1, 2), (2, 2): (1, 2), (2, 3):
(2, 2), (3, 2): (2, 2), (3, 1): (3, 2), (4, 2): (3, 2), (3, 3): (3,
2), (3, 4): (3, 3), (4, 3): (3, 3), (4, 4): (4, 3), (2, 4): (3, 4),
(1, 4): (2, 4), (0, 4): (1, 4), (0, 3): (0, 4), (4, 1): (4, 2), (4,
0): (4, 1), (3, 0): (4, 0), (2, 0): (3, 0)}
```

Théorème : dans un arbre, le nombre d'arêtes est égal au nombre de sommets - 1.

Il devrait donc y avoir 24 éléments dans le dictionnaire des pères.

In [11]:
```
len(t)
```

Out[11]:
```
24
```

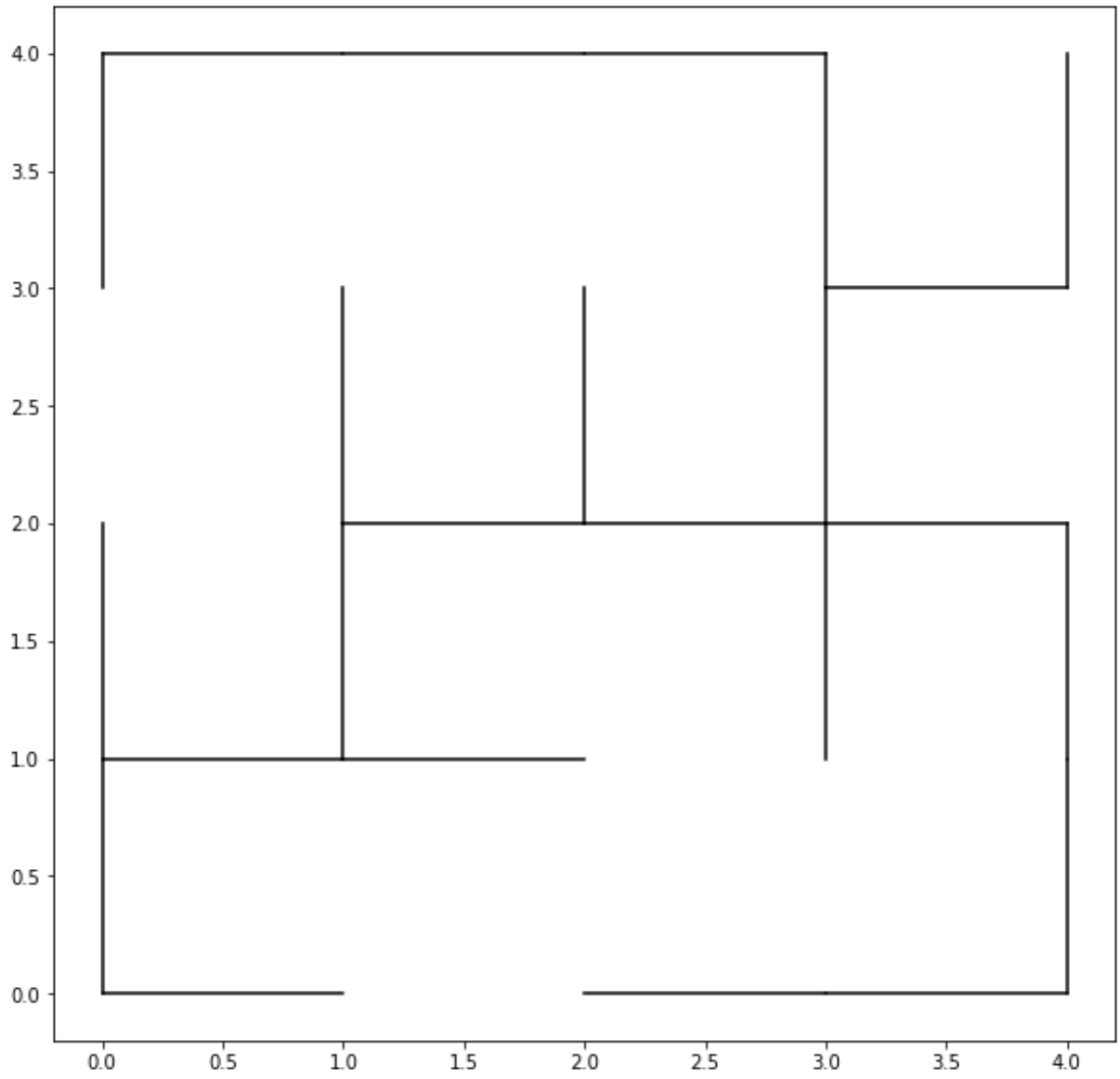
1.4 Dessiner l'arbre

Il n'y a qu'à tracer les arêtes de chaque sommet vers son père.

In [12]:
```
def plot_arbre(t):
    for (a, b) in t:
        c, d = t[(a, b)]
        plt.plot((a, c), (b, d), 'k')
    plt.show()
```

In [13]:
```
plt.rcParams['figure.figsize'] = (10, 10)
```

In [14]:
```
plot_arbre(t)
```



1.5 Dessiner le labyrinthe

La fonction `plot_laby` prend un dictionnaire de pères et deux dimensions M et N en paramètres. Que fait-elle ?

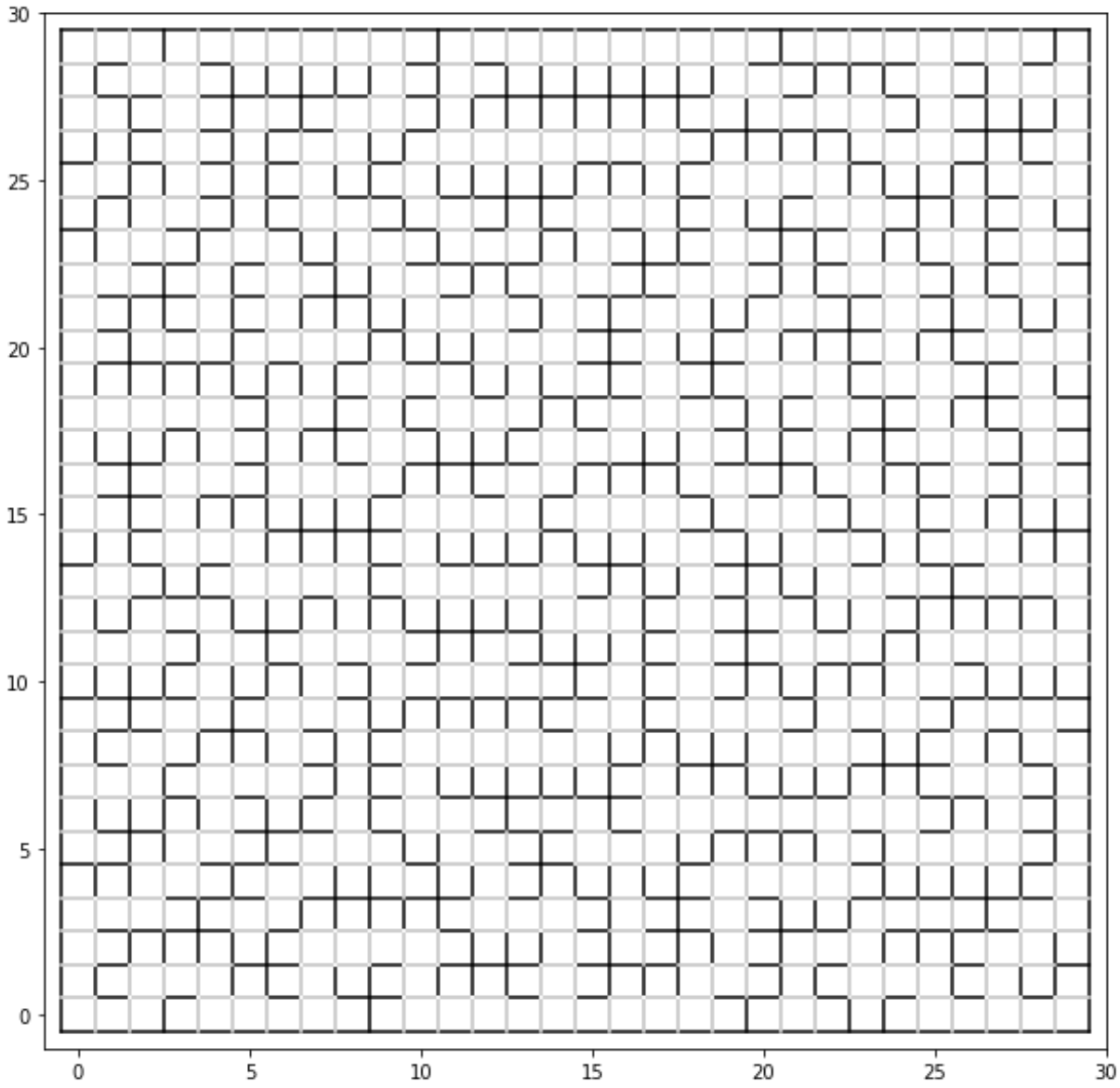
- Elle dessine les droites d'équations $x = i + \frac{1}{2}$ et $y = j + \frac{1}{2}$ pour $-1 \leq i < M$ et $-1 \leq j < N$. Bref, elle enferme les sommets du rectangle dans des petits carrés.
- S'il y a une arête d'un sommet vers un autre sommet (dictionnaire des pères !) elle efface le "mur" qui sépare ces deux sommets.

Le paramètre optionnel `lab` permet de tracer aussi l'arbre (en rouge).

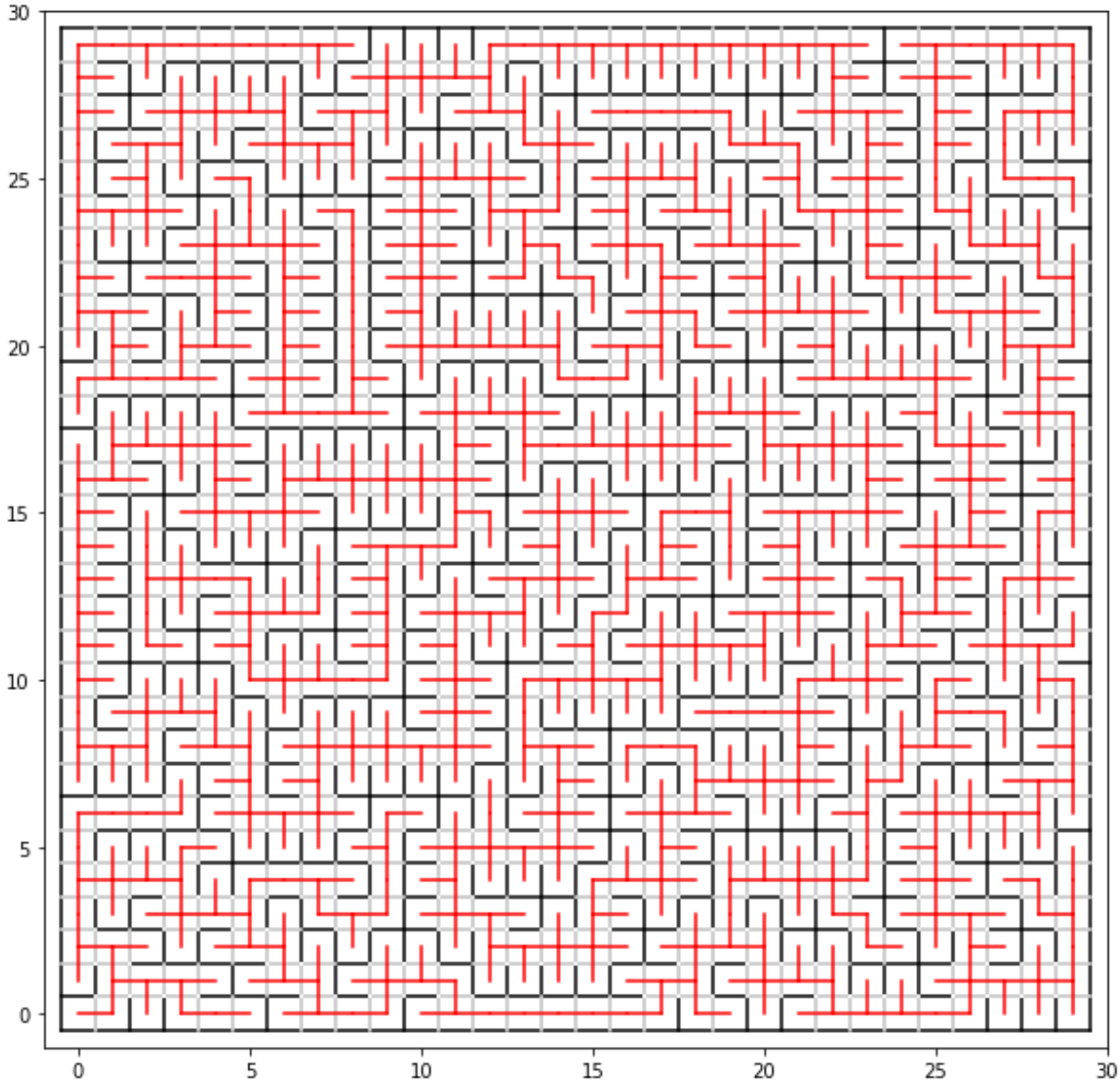
Mea Culpa : je m'excuse à l'avance pour la lenteur du tracé. `matplotlib` n'aime pas dessiner des milliers de petits segments.

In [15]:
```
def plot_labyrinthe(t, M, N, lab=False):
    d = 0.5
    plt.axis([-1, M, -1, N])
    for x in range(M + 1):
        plt.plot((x - d, x - d), (-d, N - d), 'k')
    for y in range(N + 1):
        plt.plot((-d, M - d), (y - d, y - d), 'k')
    for (x, y) in t:
        (a, b) = t[(x, y)]
        if a == x + 1: plt.plot((x+d,x+d),(y-d,y+d),'w')
        elif a == x - 1: plt.plot((x-d,x-d),(y-d,y+d),'w')
        elif b == y + 1: plt.plot((x-d,x+d),(y+d,y+d),'w')
        elif y == b + 1: plt.plot((x-d,x+d),(y-d,y-d),'w')
        if lab: plt.plot((x,a), (y,b), 'r')
```

In [16]:
```
N = 30
t = creer_labyrinthe((N // 2,N // 2), N, N)
plot_labyrinthe(t, N, N)
```



In [17]:
```
N = 30
t = creer_labyrinthe((N // 2,N // 2), N, N)
plot_labyrinthe(t, N, N, lab=True)
```



2. Retrouver son chemin

2.1 La structure de données "graphe"

Jusqu'à maintenant, le dictionnaire des pères convenait parfaitement à nos besoins. Le problème c'est que dans la suite il nous faudra une structure "inverse" : nous devrons, pour chaque sommet du rectangle, accéder à ses éventuels fils dans le labyrinthe. Pour cela, nous allons représenter un graphe par un dictionnaire G. Pour chaque sommet p, $G[p]$ est la liste des voisins de p dans le graphe. G est ce que l'on appelle la représentation du graphe par des **listes d'adjacence**.

La fonction `peres_vers_graphe` fait le travail. Partant d'un dictionnaire G vide, elle ajoute progressivement des arêtes en parcourant le dictionnaire des pères.

In [18]:
```
def peres_vers_graphe(peres):
    G = {}
    for x in peres:
        if x in G: G[x].append(peres[x])
        else: G[x] = [peres[x]]
        if peres[x] in G: G[peres[x]].append(x)
        else: G[peres[x]] = [x]
    return G
```

In [19]:
```
t = creer_labyrinthe((2,2), 5, 5)
G = peres_vers_graphe(t)
print(G)
```

```
{(3, 2): [(2, 2)], (2, 2): [(3, 2), (1, 2), (2, 1), (2, 3)], (1, 2):
[(2, 2)], (2, 1): [(2, 2)], (2, 3): [(2, 2), (1, 3), (3, 3), (2, 4)]
, (1, 3): [(2, 3)], (3, 3): [(2, 3)], (2, 4): [(2, 3), (1, 4), (3, 4
)], (1, 4): [(2, 4)], (3, 4): [(2, 4), (4, 4)], (4, 4): [(3, 4), (4,
3)], (4, 3): [(4, 4), (4, 2)], (4, 2): [(4, 3), (4, 1)], (4, 1): [(4
, 2), (4, 0), (3, 1)], (4, 0): [(4, 1)], (3, 1): [(4, 1), (3, 0)], (
3, 0): [(3, 1), (2, 0)], (2, 0): [(3, 0), (1, 0)], (1, 0): [(2, 0),
(0, 0), (1, 1)], (0, 0): [(1, 0)], (1, 1): [(1, 0), (0, 1)], (0, 1):
[(1, 1), (0, 2)], (0, 2): [(0, 1), (0, 3)], (0, 3): [(0, 2), (0, 4)]
, (0, 4): [(0, 3)]}
```

In [20]:
```
def nombre_sommets(G): return len(G)
```

In [21]:
```
nombre_sommets(G)
```

Out[21]: 25

In [22]:
```
def nombre_aretes(G):
    s = 0
    for p in G: s = s + len(G[p])
    return s
```

In [23]:
```
nombre_aretes(G)
```

Out[23]: 48

Mais pourquoi 48 ? Cela ne devrait-il pas être 24 ? Eh non. Car si p et q sont voisins dans G, q apparaît dans la liste $G[p]$ ET p apparaît dans la liste $G[q]$. Donc, tout va bien puisque $2 \times 24 = 48$. Je rappelle que dans un arbre, nombre d'arêtes = nombre de sommets -1.

2.2 Explorer un graphe

Comment explorer un graphe G à partir d'un sommet p_0 ? Il suffit de faire un copier-coller de la fonction qui crée un labyrinthe ! Techniquement, cette fonction va faire un parcours **en profondeur** du graphe (il existe d'autres types de parcours, je n'en parlerai pas ici).

In [24]:
```
def explorer(G, p0):
    visites = set([p0])
    s = [p0]
    peres = {}
    while s != []:
        p = s.pop()
        vs = G[p]
        for v in vs:
            if v not in visites:
                visites.add(v)
                s.append(v)
                peres[v] = p
    return peres
```

In [25]:
```
t = creer_labyrinthe((2,2), 5, 5)
peres = explorer(peres_vers_graphe(t), (4,3))
print(peres)
```

```
{(4, 2): (4, 3), (3, 3): (4, 3), (4, 4): (4, 3), (3, 4): (4, 4), (2,
4): (3, 4), (1, 4): (2, 4), (0, 4): (1, 4), (1, 3): (1, 4), (0, 3):
(1, 3), (0, 2): (0, 3), (0, 1): (0, 2), (0, 0): (0, 1), (1, 0): (0,
0), (4, 1): (4, 2), (3, 1): (4, 1), (2, 1): (3, 1), (3, 0): (3, 1),
(4, 0): (3, 0), (2, 2): (2, 1), (1, 1): (2, 1), (2, 0): (2, 1), (1,
2): (2, 2), (2, 3): (2, 2), (3, 2): (2, 2)}
```

2.3 Trouver l'unique chemin entre deux sommets

Soient p et q deux sommets d'un graphe connexe G. Comment trouver un chemin de p vers q ?

- On explore G à partir du sommet p, ce qui nous donne un dictionnaire `peres` .
- Le chemin cherché est q, le père de q, le père du père de q, ... jusqu'à p.

In [26]:
```
def chemin(G, p, q):
    if p == q: return [p]
    else:
        peres = explorer(G, p)
        c = [q]
        while peres[q] != p:
            c.append(peres[q])
            q = peres[q]
        c.append(p)
        return c
```

In [27]:
```
G = peres_vers_graphe(creer_labyrinthe((2,2), 10, 10))
print(chemin(G, (5,5), (1,5)))
```

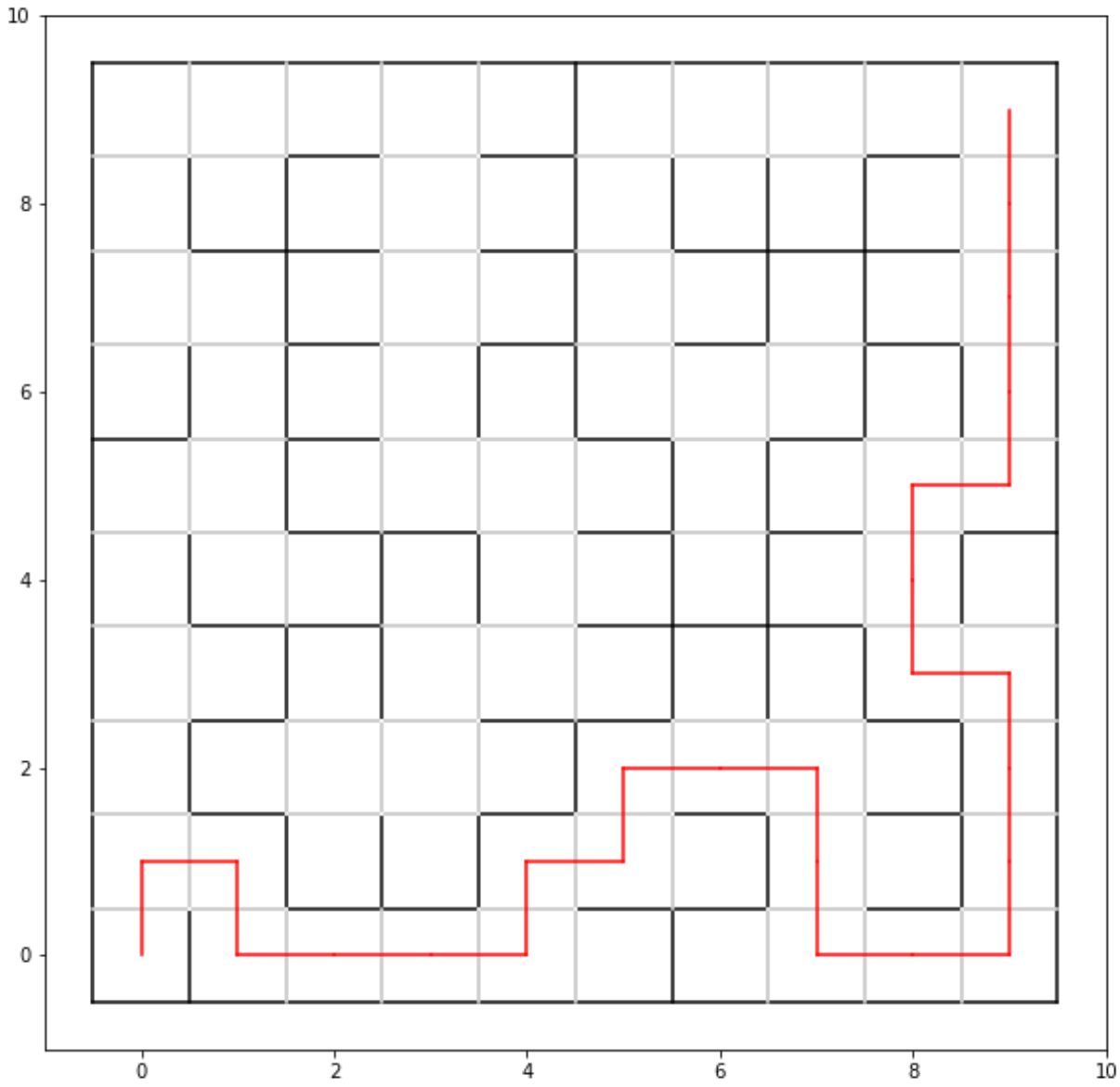
```
[(1, 5), (0, 5), (0, 6), (0, 7), (0, 8), (1, 8), (2, 8), (3, 8), (3,
7), (4, 7), (4, 6), (4, 5), (5, 5)]
```

2.4 Afficher le chemin dans le labyrinthe

Euh, eh bien on affiche le labyrinthe et puis on affiche le chemin.

In [28]:
```
def afficher_chemin(peres, M, N, chemin):
    plot_labyrinthe(peres, M, N)
    for k in range(len(chemin) - 1):
        (x, y) = chemin[k]
        (a, b) = chemin[k + 1]
        plt.plot((x, a), (y, b), 'r')
```

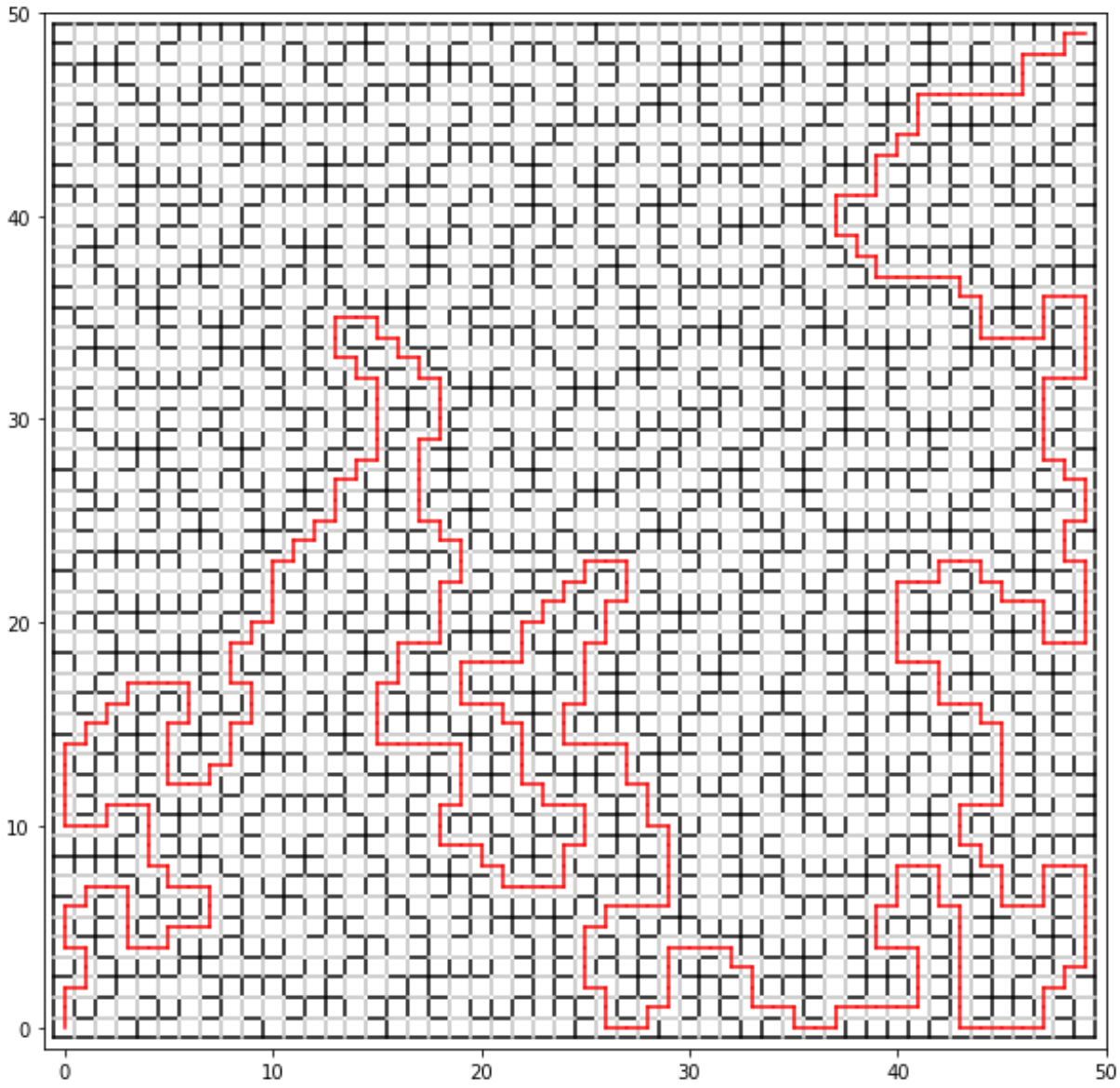
In [29]:
```
t = creer_labyrinthe((2,2), 10, 10)
G = peres_vers_graphe(t)
c = chemin(G, (0,0), (9,9))
afficher_chemin(t, 10, 10, c)
```



Allons, un dernier test, sur une grille de 2500 points. Encore une fois, l'affichage est TRÈS lent. Si vous commentez la ligne `afficher_chemin` , c'est instantané : nos algorithmes de création et de parcours s'exécutent donc en zéro seconde, ce qui est un temps acceptable :-).

In [31]:
```
M = 50
N = 50
t = creer_labyrinthe((M // 2, N // 2), M, N)
G = peres_vers_graphe(t)

p = (0, 0)
q = (M - 1, N - 1)
c = chemin(G, p, q)
afficher_chemin(t, M, N, c)
```



In [ ]:
```

```