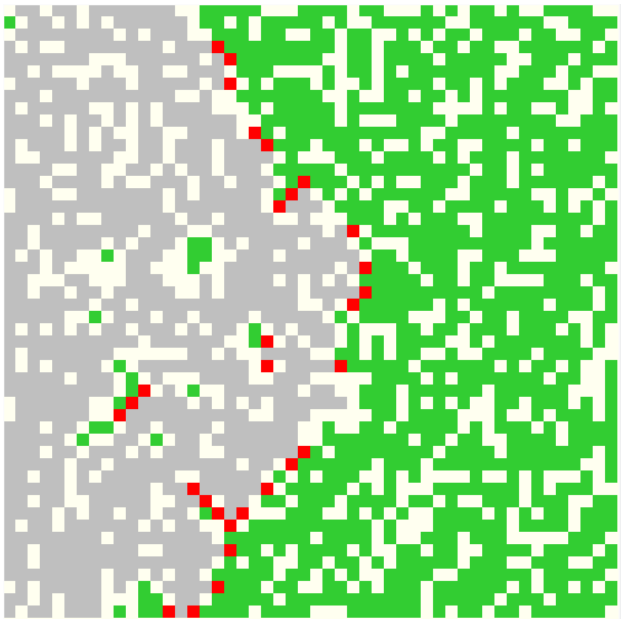
Pascal ORTIZ



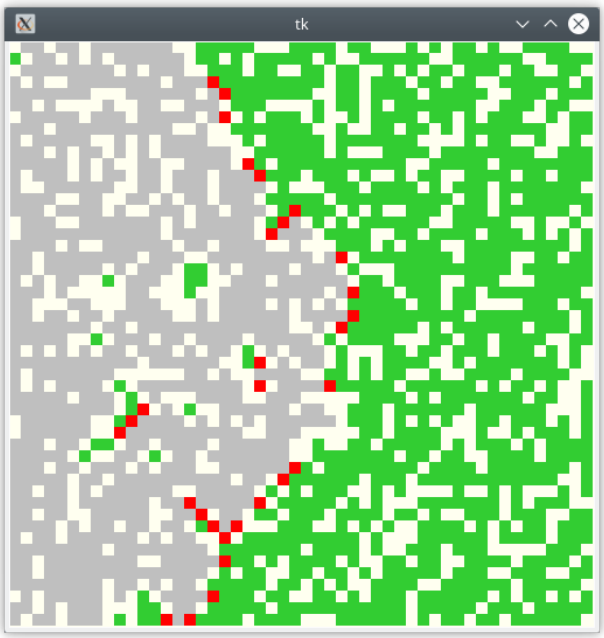
Simulation de feu de forêt

Version du 18 février 2020

Table des matières

L'essentiel de l'activité

L'objectif de l'activité est de construire une interface graphique en Tkinter simulant la propagation d'un feu de forêt. Dans la version simple, cela donnerait ceci :



Une animation véritablement interactive est proposée en fin d'activité.

La forêt est en fait plutôt une parcelle. Elle est représentée sous la forme d'une grille carrée de taille $n \times n$ contenant n^2 cellules. Ces dernières sont d'un des quatre types suivants :

Type de cellule	Couleur de la cellule	Code
Sol nu	ivoire	0
Arbre vivant	verte	1
Arbre qui brûle	rouge	2
Arbre en cendres	grise	3

Si N est le nombre d'arbres, on notera $p = N/(n \times n)$ la *proportion* d'arbres (ou encore la densité) dans la parcelle. En pratique, on se donnera n ainsi qu'une proportion, par exemple $p = 3/4$ et on en déduira la population N d'arbres de la parcelle.

L'évolution du feu de forêt dans l'interface graphique ne sera que le reflet de l'évolution de la grille dont les cellules changent d'état. Chaque cellule passe d'un état « courant » à un état « suivant » :

— si la cellule est un arbre en feu (code 2 ci-dessus), à l'état suivant, elle devient une cellule d'un arbre en cendres (code 3),

— si la cellule est un arbre en feu, et qu'une de ses voisines v dans les quatre directions est un arbre vivant (code 1), alors v devient une cellule d'un arbre en feu à l'état suivant.

Il s'agit donc de coder une succession de grilles jusqu'à ce qu'il n'y ait plus aucune cellule en feu dans la grille. Et puis de faire évoluer cette succession de grilles dans un canevas Tkinter.

Structure de données

Dans cette partie, on code la simulation d'abord **sans** tenir compte de la partie graphique. Il y aura deux étapes :

— on écrit une fonction `random_forest(n, p)` qui renvoie une forêt aléatoire ayant une densité donnée `p` ; cette forêt sera représentée par un tableau 2D de taille `n` × `n`, chaque élément valant un entier 0, 1, 2 ou 3 ;

— on écrit une fonction `update_states(states)` capable de mettre à jour la grille selon l'état de la forêt :

  — les cellules en feu deviendront des cendres,

  — les cellules vertes (un arbre) voisines d'une cellule rouge (feu) deviendront elle-mêmes rouges.

Jemploierai parfois le terme de *matrice des états* pour désigner la grille `states`.

Voici une fonction qui génère une forêt aléatoire :

foret_alea.py

```
1  from random import sample
2
3  def random_forest(p, n):
4      units=[(line,col) for col in range(n) for line in range(n)]
5      ntrees=int(n**2*p)
6      trees=sample(units,ntrees)
7      states=[[0]*n for _ in range(n)]
8      for (i,j) in trees:
9          states[i][j]=1
10     return states
11
12 states = random_forest(0.6, 10)
13 print('\n'.join(' '.join(map(str, L)) for L in states))
```

```
0 0 0 0 0 1 0 1 1 1
0 1 0 1 0 0 1 1 1 1
1 1 1 0 1 0 0 0 0 1
0 1 0 1 1 1 1 1 0 1
1 0 0 1 1 0 1 1 0 0
1 1 1 1 1 1 1 1 0 0
1 0 1 1 1 1 1 0 1 0
1 1 1 1 1 1 0 1 1 0
1 0 1 0 0 1 0 1 1 0
0 0 0 1 1 1 0 1 0 1
```

— Ligne 12 : on a généré une forêt 10 x 10 avec une proportion de 60 % d'arbres et on l'affiche (lignes 13 et 14-23)

— Ligne 5 : le nombre d'arbres est `ntrees` = $n^2 \times p$.

— Ligne 6 : avec la fonction `sample` du module standard `random`, on tire au hasard l'échantillon de `ntrees` arbres parmi les n^2 emplacements possibles.

— Ligne 7 : `states` est la grille représentant l'état de chaque cellule de la forêt, initialement sans arbre (uniquement du sol, code 0) ; on place les arbres tirés au hasard ligne 6.

— Lignes 8-9 et les lignes de sortie : on obtient donc une matrice formée uniquement de 0 et de 1.

Passons à la fonction de mise à jour `updates_states(states)`. Cette fonction va examiner l'état de chaque cellule de la grille courante `states`. En particulier, si la cellule correspond à des flammes, la fonction va regarder s'il faut propager les flammes à une cellule directement voisine.

Une cellule a au plus 4 cellules voisines, celles qui se trouvent au nord, à l'ouest, au sud ou à l'est. Il est donc utile de disposer d'une fonction qui, étant donné une cellule connue par sa position `ligne x colonne` dans une grille carrée de côté `n`, renvoie la liste des cellules voisines. Voici une telle fonction :

voisins.py

```
1 def voisins(n, i, j):
2     return [(a,b) for (a, b) in
3             [(i, j+1),(i, j-1), (i-1, j), (i+1,j)]
4             if a in range(n) and b in range(n)]
5
6 print(voisins(5, 2, 1))
7 print(voisins(5, 2, 4))
```

```
8 [(2, 2), (2, 0), (1, 1), (3, 1)]
9 [(2, 3), (1, 4), (3, 4)]
```

J'ai utilisé des listes en compréhension emboîtées. Noter que cette fonction n'a pas besoin de la matrice des états `states`.

On peut désormais écrire une fonction qui détermine le nouvel état de la grille en fonction de son état précédent. Et en testant, on en profitera pour mettre le feu à un arbre et observer les trois premières étapes de l'incendie :

transition.py

```
1 from random import sample
2
3 def random_forest(p, n):
4     units=[(line,col) for col in range(n) for line in range(n)]
5     ntrees=int(n**2*p)
6     trees=sample(units,ntrees)
7     states=[[0]*n for _ in range(n)]
8     for (i,j) in trees:
9         states[i][j]=1
10    return states
11
12 def voisins(n, i, j):
13    return [(a,b) for (a, b) in
14            [(i, j+1),(i, j-1), (i-1, j), (i+1,j)]
15            if a in range(n) and b in range(n)]
16
17 def update_states(states):
18    n=len(states)
19    to_fire=[]
20    for line in range(n):
```

```
        for col in range(n):
            if states[line][col]==2:
                states[line][col]=3
                for (i, j) in voisins(n, line, col):
                    if states[i][j]==1:
                        to_fire.append((i, j))
    for (line,col) in to_fire:
        states[line][col]=2

states = random_forest(0.6, 10)
states[0][0]=2
print('\n'.join([' '.join(map(str, L)) for L in states]))
print('------------------')

for i in range(3):
    update_states(states)
    print('\n'.join([' '.join(map(str, L)) for L in states]))
    print('------------------')
```

```
2 1 0 1 1 0 1 1 1 1
1 0 0 1 1 1 1 1 0 1
1 0 0 0 1 1 1 0 1 1
1 0 1 0 0 1 0 1 1 1
1 1 1 0 1 1 1 1 0 1
1 0 1 0 0 1 0 0 1 0
1 0 1 0 0 0 0 0 0 1
1 1 0 0 0 1 0 1 1 1
1 0 1 1 1 1 0 1 1 0
0 0 1 1 0 1 0 1 1 0
------------------
3 2 0 1 1 0 1 1 1 1
2 0 0 1 1 1 1 1 0 1
1 0 0 0 1 1 1 0 1 1
1 0 1 0 0 1 0 1 1 1
1 1 1 0 1 1 1 1 0 1
1 0 1 0 0 1 0 0 1 0
1 0 1 0 0 0 0 0 0 1
1 1 0 0 0 1 0 1 1 1
1 0 1 1 1 1 0 1 1 0
0 0 1 1 0 1 0 1 1 0
------------------
3 3 0 1 1 0 1 1 1 1
3 0 0 1 1 1 1 1 0 1
2 0 0 0 1 1 1 0 1 1
1 0 1 0 0 1 0 1 1 1
1 1 1 0 1 1 1 1 0 1
1 0 1 0 0 1 0 0 1 0
1 0 1 0 0 0 0 0 0 1
1 1 0 0 0 1 0 1 1 1
```

```
1 0 1 1 1 1 0 1 1 0
0 0 1 1 0 1 0 1 1 0
------------------
3 3 0 1 1 0 1 1 1 1
3 0 0 1 1 1 1 1 0 1
3 0 0 0 1 1 1 0 1 1
2 0 1 0 0 1 0 1 1 1
1 1 1 0 1 1 1 1 0 1
1 0 1 0 0 1 0 0 1 0
1 0 1 0 0 0 0 0 0 1
1 1 0 0 0 1 0 1 1 1
1 0 1 1 1 1 0 1 1 0
0 0 1 1 0 1 0 1 1 0
------------------
```

— Lignes 30-31 : on crée une forêt 10 x 10 (ligne 30) et on met le feu dans le coin en haut à gauche (ligne 31).

— Lignes 50-82 : on laisse l'incendie se propager sur trois étapes. A chaque étape, on visualise l'état de la forêt. On voit que des cendres (code 3, lignes 50 ou 61) et du feu (code 2, ligne 63 par exemple) apparaissent.

— Lignes 20-26 : on parcourt la grille (ligne 20-21) : une cellule en feu peut être changée en cendres (lignes 22-23). Toutefois, si la cellule courante est en feu et qu'une celllule voisine est un arbre, il ne faut pas mettre à jour tout de suite cette cellule (en la mettant en feu) sinon on risque de modifier le voisinage de certaines cellules non encores examinées. On place donc dans une liste temporaire `to_fire` (ligne 19) et on met à jour la grille une fois qu'elle a été parcourue (ligne 27-28).

Partie graphique

Il faut maintenant écrire une fonction `fill(states)` qui à partir de la grille des états 0, 1, 2 ou 3, dessine dans un canevas Tkinter la forêt correspondante.

Chaque cellule est un carré de côté, disons `unit=10` pixels et de couleur appropriée que l'on va choisir dans la liste suivante :

```
COLORS=["ivory", "lime green", "red", "gray75"]
```

J'ai utilisé ce nuancier pour choisir les couleurs.

Noter que l'indice dans cette liste de la couleur d'une cellule est justement le code (0, 1, 2 ou 3) qui est placé pour cette cellule dans le tableau `states`. Par exemple, le code 2 correpond au feu et justement `COLORS[2]="red"`.

D'autre part, un carré sera dessiné avec la méthode `create_rectangle` du widget `Canvas`. Ecrivons donc une fonction `fill_cell` qui dessine juste la cellule en position `line x col` :

```
def fill_cell(states, line, col):
        A=(unit*col, unit*line)
        B=(unit*(col+1), unit*(line+1))
        state=states[line][col]
        color=COLORS[state]
        cnv.create_rectangle(A, B, fill=color, outline='')
```

Un point essentiel est que la fonction lit le plateau abstrait des états (`states`) pour savoir comment dessiner (lignes 4-6). Pour alléger, je n'ai pas placé le canevas `cnv` ni l'unité `unit` en paramètres de la fonction car ils ne changeront pas tout le long du programme.

On en déduit une fonction qui dessine toute la forêt définie par le tableau `states` :

```
def fill(states):
    n=len(states)
    for line in range(n):
        for col in range(n):
            fill_cell(states, line, col)
```

Il suffit de parcourir la grille et d'appeler la fonction `fill_cell` de dessin cellule par cellule.

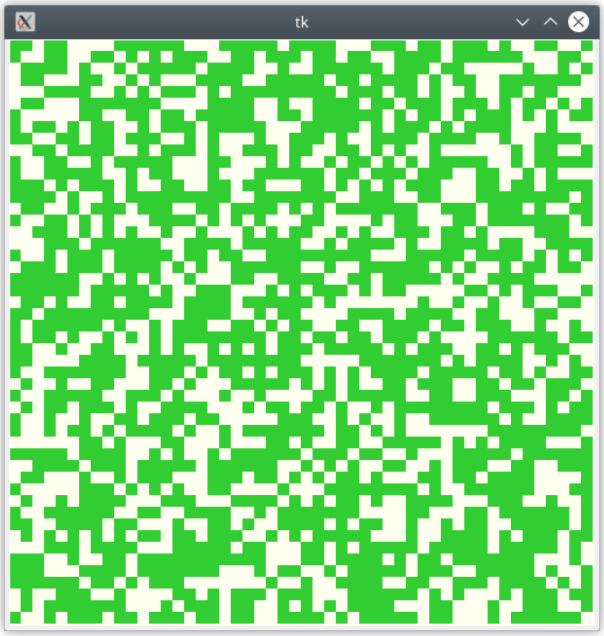
Voici un code complet qui affiche une forêt sans flamme :

foret_statique.py

```
from random import sample
from tkinter import Tk, Canvas

COLORS=["ivory", "lime green", "red", "gray75"]

def random_forest(p, n):
    units=[(line,col) for col in range(n) for line in range(n)]
    ntrees=int(n**2*p)
    trees=sample(units,ntrees)
    states=[[0]*n for _ in range(n)]
    for (i,j) in trees:
        states[i][j]=1
    return states

def fill_cell(states, line, col):
    A=(unit*col, unit*line)
    B=(unit*(col+1), unit*(line+1))
    state=states[line][col]
    color=COLORS[state]
    cnv.create_rectangle(A, B, fill=color, outline='')

def fill(states):
    n=len(states)
    for line in range(n):
        for col in range(n):
            fill_cell(states, line, col)

p=0.60
n=50
unit=10

# Fenêtre et canevas
root = Tk()
cnv = Canvas(root, width=unit*n, height=unit*n, background="ivory")
```

```
cnv.pack()

# Forêt aléatoire
states=random_forest(p, n)

# Plateau dessiné
fill(states)

root.mainloop()
```

qui produit



L'animation

Pour faire une animation de l'incendie, il faut :

— mettre le feu quelque part,
— faire tourner la fonction `update_states` mettant à jour la grille,
— afficher sur le canevas l'évolution de la grille.

Pour mettre le feu, on se contente d'allumer une cellule au milieu du bord gauche de la forêt :

```
i=n//2
j=0
states[i][j]=2
```

comme ça on verra l'indendie progresser de la gauche vers la droite.
Passons à la fonction `propagate` qui va gérer l'animation dans Tkinter :

```
def propagate():
    update_states(states)
```

```
3       cnv.delete("all")
4       fill(cnv, states, unit)
5       cnv.after(150, propagate)
```

Cett fonction est appelée automatiquement toutes les 150 millisecondes (ligne 5). Elle met à jour la grille des états (ligne 2) et elle redessine la forêt (ligne 4) ; pour cela, elle retire tous les items du canevas (ce qu'on appelle un « rafraîchissement », ligne 3) et elle dessine la forêt dans son nouvel état. D'où le code complet suivant :

anim.py

```
1  from random import sample
2  from tkinter import Tk, Canvas
3
4  COLORS=["ivory", "lime green", "red", "gray75"]
5
6  def random_forest(p, n):
7      units=[(line,col) for col in range(n) for line in range(n)]
8      ntrees=int(n**2*p)
9      trees=sample(units,ntrees)
10     states=[[0]*n for _ in range(n)]
11     for (i,j) in trees:
12         states[i][j]=1
13     return states
14
15 def voisins(n, i, j):
16     return [(a,b) for (a, b) in
17             [(i, j+1),(i, j-1), (i-1, j), (i+1,j)]
18             if a in range(n) and b in range(n)]
19
20 def update_states(states):
21     n=len(states)
22     to_fire=[]
23     for line in range(n):
24         for col in range(n):
25             if states[line][col]==2:
26                 states[line][col]=3
27                 for (i, j) in voisins(n, line, col):
28                     if states[i][j]==1:
29                         to_fire.append((i, j))
30     for (line,col) in to_fire:
31         states[line][col]=2
32
33 def fill_cell(states, line, col):
34         A=(unit*col, unit*line)
35         B=(unit*(col+1), unit*(line+1))
36         state=states[line][col]
37         color=COLORS[state]
38         cnv.create_rectangle(A, B, fill=color, outline='')
39
```

```
40 def fill(states):
41     n=len(states)
42     for line in range(n):
43         for col in range(n):
44             fill_cell(states, line, col)
45 
46 def propagate():
47     update_states(states)
48     cnv.delete("all")
49     fill(states)
50     cnv.after(150, propagate)
51 
52 p=0.62
53 n=50
54 unit=10
55 
56 # Fenêtre et canevas
57 root = Tk()
58 cnv = Canvas(root, width=unit*n, height=unit*n, background="ivory")
59 cnv.pack()
60 
61 # Forêt aléatoire
62 states=random_forest(p, n)
63 
64 i=n//2
65 j=0
66 states[i][j]=2
67 
68 # Plateau dessiné
69 fill(states)
70 propagate()
71 
72 root.mainloop()
```

Pour que l'animation se lance, il faut lancer directement (ligne 70) la fonction `propagate` ; ensuite, cette fonction est relancée automatiquement (cf. ligne 50).

D'où l'animation annoncée.

Dans la suite de ce document, l'interface graphique sera rendue plus riche et plus interactive en permettant de

— monitorer l'évolution de l'incendie,

— lancer une nouvelle animation avec un bouton,

— cliquer directement sur une cellule pour incendier,

— modifier la densité de la forêt avec un curseur.

Automate cellulaire, percolation

L'activité proposée est directement inspirée de cette page du Cormas.

Simuler un incendie revient à créer ce qu'on appelle un *automate cellulaire* :

— automate car le système évolue par transitions successives à partir d'un état initial,

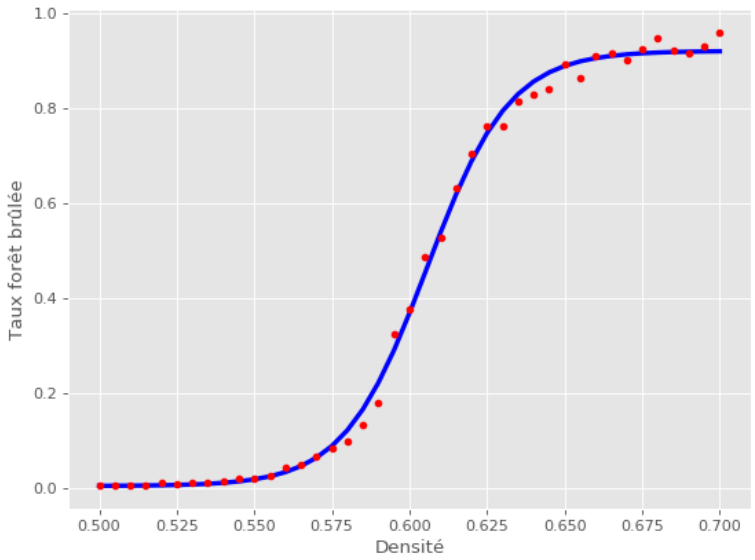
— cellulaire : les états sont enregistrés dans des cellules.

La propagation du feu va dépendre essentiellement de la densité p des arbres dans la parcelle. Les arbres sont supposés répartis aléatoirement selon une loi uniforme. Il est concevable que les dégâts sont plus étendus si la densité est élevée. En fait, en jouant avec la densité, on se rend compte qu'elle admet une valeur critique, dite *seuil de percolation* :

— en deçà, l'incendie reste très limité,

— au-delà, et presque immédiatement, la propagation est beaucoup plus importante.

Une simulation pourrait produire la courbe suivante :

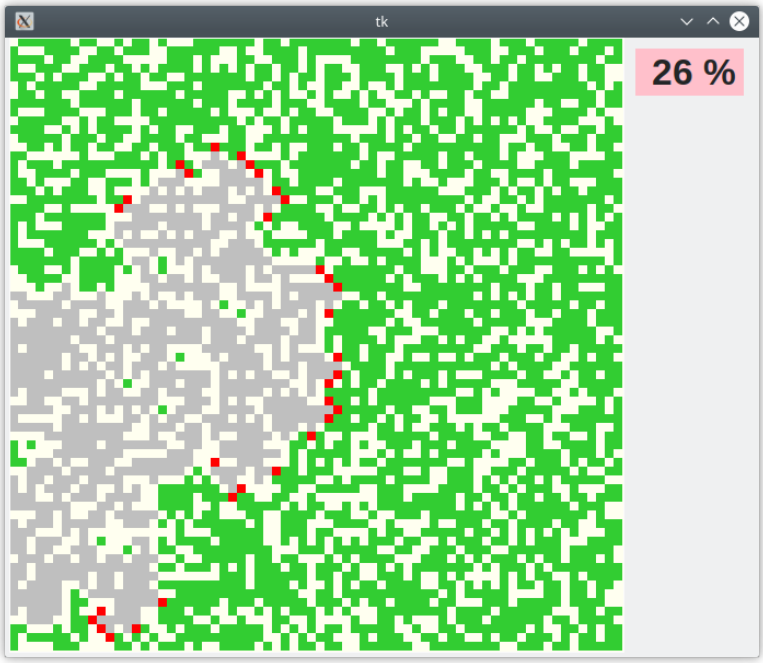


en abscisses, on lit la densité de la forêt, en ordonnées le pourcentage d'arbres qui ont été enflammés. Cette courbe sera construite dans la dernière partie de l'activité.

Le chercheur Mitchel Resnick a établi (1994) que la densité critique est d'environ 59 %. Plus précisément, d'après les simulations de l'auteur, ce seuil discrimine les incendies qui partant d'un bord se propagent jusqu'au bord opposé.

Monitorer les dégâts de l'incendie

On souhaite connaître à chaque instant le pourcentage de forêt brûlé :



On va reprendre le code précédent. Le nombre initial d'arbres est connu, il vaut $n^2 \times p$ où `n` est la longueur du côté du carré et `p` est la densité.

Ensuite, pour connaître le nombre total d'arbres incendiés, il suffit de tenir à jour un compteur (disons `cpt`) incrémenté par le nombre d'arbres en feux sur le plateau à chaque fin de transition (disons `nfires`). Ce nombre pourrait s'obtenir en modifiant la fonction de mise-à-jour de `states` mais on ne va chercher à optimiser et on va se contenter de reparcourir la matrice des états `states` :

```
nfires=sum(states[i][j]==2 for i in range(n) for j in range(n))
```

On peut alors calculer le pourcentage de forêt brûlée et il ne reste plus qu'à l'afficher avec un label. Cet affichage étant en temps réel, il doit être déterminé dans la fonction d'animation `propagate`.

D'où le code suivant :

pourcent.py

```
from random import sample
from tkinter import Tk, Canvas, Label

COLORS=["ivory", "lime green", "red", "gray75"]

def random_forest(p, n):
    units=[(line,col) for col in range(n) for line in range(n)]
    ntrees=int(n**2*p)
    trees=sample(units,ntrees)
    states=[[0]*n for _ in range(n)]
    for (i,j) in trees:
        states[i][j]=1
    return states
```

```
14
15 def voisins(n, i, j):
16     return [(a,b) for (a, b) in
17             [(i, j+1),(i, j-1), (i-1, j), (i+1,j)]
18             if a in range(n) and b in range(n)]
19
20 def update_states(states):
21     n=len(states)
22     to_fire=[]
23     for line in range(n):
24         for col in range(n):
25             if states[line][col]==2:
26                 states[line][col]=3
27                 for (i, j) in voisins(n, line, col):
28                     if states[i][j]==1:
29                         to_fire.append((i, j))
30     for (line,col) in to_fire:
31         states[line][col]=2
32
33
34 def fill_cell(states, line, col):
35         A=(unit*col, unit*line)
36         B=(unit*(col+1), unit*(line+1))
37         state=states[line][col]
38         color=COLORS[state]
39         cnv.create_rectangle(A, B, fill=color, outline='')
40
41 def fill(states):
42     n=len(states)
43     for line in range(n):
44         for col in range(n):
45             fill_cell(states, line, col)
46
47 def propagate():
48     global cnt
49     update_states(states)
50     nfires=sum(states[i][j]==2 for i in range(n) for j in range(n))
51     cnt+=nfires
52     percent = int(cnt/ntrees*100)
53     cnv.delete("all")
54     fill(states)
55     lbl["text"]="%3s %%" %percent
56     cnv.after(150, propagate)
57
58 p=0.62
59 n=70
60 unit=8
61 ntrees=int(n*n*p)
62
```

```
63 # Fenêtre et canevas
64 root = Tk()
65 cnv = Canvas(root, width=unit*n, height=unit*n, background="ivory")
66 cnv.pack(side="left")
67
68 lbl=Label(root,text="  0 %",  font='Arial 25 bold', bg='pink', width=5)
69 lbl.pack(padx=10, pady=10)
70
71 # Forêt aléatoire
72 states=random_forest(p, n)
73
74 i=n//2
75 j=0
76 states[i][j]=2
77 cnt=1
78
79 # Plateau dessiné
80 fill(states)
81 propagate()
82
83 root.mainloop()
```

Les principales modifications par rapport au code de `anim.py` sont les suivantes :

— on initialise le compteur `cnt` à 1 (ligne 77) et non pas à 0 puisqu'il y a un foyer initial (ligne 76) ;

— on calcule le nombre total `ntrees` (ligne 61) d'arbres ;

— on créé un label pour afficher le pourcentage (lignes 68-69) ; la couleur de fond du label est rose (pink).

— après chaque transition (ligne 49), on calcule le nombre d'arbres encore en feu (ligne 50) et on incrémente le compteur (ligne 51) ce qui permet de calculer le pourcentage (lignes 52) ;

— on met à jour le label (ligne 55) ; j'ai utilisé la syntaxe classique des chaînes formatées. En outre, avec la syntaxe classique, le signe `%` est traduit par les deux caractères `%%`.

Il resterait à améliorer un point : quand l'incendie est éteint, l'animation continue à tourner ; pour s'en convaincre, il suffit d'écrire une instruction du genre

```
print(randrange(42))
```

(après avoir importé `randrange`) à la fin de la fonction `propagate` (après la ligne 56) et de constater que, malgré la fin de l'incendie, des valeurs continuent à s'afficher dans la console, signe que l'animation tourne :

Il est assez simple de remédier à ce problème ; en effet, s'il n'y a plus aucun nouveau foyer, l'incendie est terminé (il n'y a pas de reprise de feu). Il suffit donc, lorsque `nfires` vaut 0 (ligne 50), de faire un `return` dans la fonction `propagate`, juste avant la relance de la fonction `propagate` (avant ligne 56) :

```
lbl["text"]="%3s %%" %percent
if nfires == 0:
    return
cnv.after(150, propagate)
```

D'où le code complet :

anim_complet.py

```
1  from random import sample
2  from tkinter import Tk, Canvas, Label
3  from random import randrange
4
5  COLORS=["ivory", "lime green", "red", "gray75"]
6
7  def random_forest(p, n):
8      units=[(line,col) for col in range(n) for line in range(n)]
9      ntrees=int(n**2*p)
10     trees=sample(units,ntrees)
11     states=[[0]*n for _ in range(n)]
12     for (i,j) in trees:
```

```
        states[i][j]=1
    return states

def voisins(n, i, j):
    return [(a,b) for (a, b) in
            [(i, j+1),(i, j-1), (i-1, j), (i+1,j)]
            if a in range(n) and b in range(n)]

def update_states(states):
    n=len(states)
    to_fire=[]
    for line in range(n):
        for col in range(n):
            if states[line][col]==2:
                states[line][col]=3
                for (i, j) in voisins(n, line, col):
                    if states[i][j]==1:
                        to_fire.append((i, j))
    for (line,col) in to_fire:
        states[line][col]=2


def fill_cell(states, line, col):
        A=(unit*col, unit*line)
        B=(unit*(col+1), unit*(line+1))
        state=states[line][col]
        color=COLORS[state]
        cnv.create_rectangle(A, B, fill=color, outline='')

def fill(states):
    n=len(states)
    for line in range(n):
        for col in range(n):
            fill_cell(states, line, col)

def propagate():
    global cnt
    update_states(states)
    nfires=sum(states[i][j]==2 for i in range(n) for j in range(n))
    cnt+=nfires
    percent = int(cnt/ntrees*100)
    cnv.delete("all")
    fill(states)
    lbl["text"]="%3s %%" %percent
    if nfires == 0:
        return
    cnv.after(150, propagate)
    print(randrange(42))
```

```
61
62 p=0.62
63 n=70
64 unit=8
65 ntrees=int(n*n*p)
66
67 # Fenêtre et canevas
68 root = Tk()
69 cnv = Canvas(root, width=unit*n, height=unit*n, background="ivory")
70 cnv.pack(side="left")
71
72 lbl=Label(root,text="  0 %",  font='Arial 25 bold', bg='pink', width=5)
73 lbl.pack(padx=10, pady=10)
74
75 # Forêt aléatoire
76 states=random_forest(p, n)
77
78 i=n//2
79 j=0
80 states[i][j]=2
81 cnt=1
82
83 # Plateau dessiné
84 fill(states)
85 propagate()
86
87 root.mainloop()
```

Note que l'instruction `print(randrange(42))` à la fin de la fonction `propagate` permettent de surveiller que la fin de l'animation est effective.

Mettre le feu avec la souris

Plutôt que d'imposer un départ du feu au milieu du bord gauche du plateau, on souhaite pouvoir choisir l'emplacement d'où va démarrer le feu. Le plus simple est de cliquer sur un arbre pour provoquer l'incendie :

On va donc devoir associer (*bind*) le clic de souris sur le canevas à une fonction de rappel qu'on appelera `fire`, ce qui se fera comme ceci :

```
cnv.bind("<Button>", fire)
```

ce qui permettra de calculer l'indice de ligne et de colonne de la cellule qui a été cliquée. On va reprendre le code de `anim.py`. Il suffit donc de remplacer `i` et `j` dans le code ci-dessous

```
1 i=n//2
2 j=0
3 states[i][j]=2
```

pour que `(i, j)` soit la position du clic de souris (s'il est sur un arbre).

Par ailleurs dans le code `anim.py`. (lignes 68-72) on avait :

```
# Plateau dessiné
fill(states)
propagate()

root.mainloop()
```

ce qui doit désormais être supprimé car l'animation doit être lancée par le clic de souris et donc dans la fonction `fire`. D'où le code suivant :

feu_clic.py

```
from random import sample
from tkinter import Tk, Canvas

COLORS=["ivory", "lime green", "red", "gray75"]

def random_forest(p, n):
    units=[(line,col) for col in range(n) for line in range(n)]
    ntrees=int(n**2*p)
    trees=sample(units,ntrees)
    states=[[0]*n for _ in range(n)]
    for (i,j) in trees:
        states[i][j]=1
    return states

def voisins(n, i, j):
    return [(a,b) for (a, b) in
            [(i, j+1),(i, j-1), (i-1, j), (i+1,j)]
            if a in range(n) and b in range(n)]

def update_states(states):
    n=len(states)
    to_fire=[]
    for line in range(n):
        for col in range(n):
            if states[line][col]==2:
                states[line][col]=3
                for (i, j) in voisins(n, line, col):
                    if states[i][j]==1:
                        to_fire.append((i, j))
    for (line,col) in to_fire:
        states[line][col]=2


def fill_cell(states, line, col):
        A=(unit*col, unit*line)
        B=(unit*(col+1), unit*(line+1))
        state=states[line][col]
        color=COLORS[state]
        cnv.create_rectangle(A, B, fill=color, outline='')
```

```
40
41 def fill(states):
42     n=len(states)
43     for line in range(n):
44         for col in range(n):
45             fill_cell(states, line, col)
46
47 def propagate():
48     update_states(states)
49     cnv.delete("all")
50     fill(states)
51     cnv.after(400, propagate)
52
53 def fire(event):
54     i, j=event.y//unit, event.x//unit
55     if states[i][j]==1:
56         states[i][j]=2
57         fill_cell(states, i, j)
58         fill(states)
59         propagate()
60
61 p=0.62
62 n=50
63 unit=10
64
65 # Fenêtre et canevas
66 root = Tk()
67 cnv = Canvas(root, width=unit*n, height=unit*n, background="ivory")
68 cnv.pack()
69
70 cnv.bind("<Button>", fire)
71
72 # Forêt aléatoire
73 states=random_forest(p, n)
74
75 # Plateau dessiné
76 fill(states)
77
78 root.mainloop()
```

— Lignes 53-59 : la fonction de rappel du clic de souris sur le canevas.

— Ligne 70 : liaison du clic sur le canevas et de la fonction `fire`.

— Ligne 54 : on récupère la position du clic, on en déduit les indices de ligne et de colonne de la cellule cliquée ; on met à jour `states` (ligne 57) et on lance la fonction d'animation (ligne 59) si on a bien cliqué sur un arbre (ligne 55).

Le code ci-dessus a néanmoins un problème si on reclique sur un arbre : cela va lancer deux animations concurrentes du même plateau ce qui va fausser l'animation. Pour éviter ce problème mais accepter malgré tout la possibilité que plusieurs foyers puissent être créées par l'utilisateur, il suffit de créer un drapeau (ci-dessous `running`, ligne 2 et 9) qui surveille l'exécution de

l'animation :

```
1 def fire(event):
2     global running
3     i, j=event.y//unit, event.x//unit
4     if states[i][j]==1:
5         states[i][j]=2
6         fill_cell(states, i, j)
7         fill(states)
8         if not running:
9             running=True
10            propagate()
```

Si on reclique sur le plateau et qu'une animation est en cours, il suffit de ne pas relancer l'animation (ligne 8) mais d'accepter la mise à jour de la matrice des états (ligne 5).

Dernier détail : on observera que si on clique à la frontière du canevas en bas et à droite qu'on obtient un débordement d'indice : on a cliqué sur le bord d'une cellule et l'indice est un de plus que l'indice autorisé. Pour régler ce problème, il suffit de rogner légèrement sur la taille du canevas, et lui retirer 2 pixels (dernière ligne ci-dessous) :

```
n=50
unit=10

# Fenêtre et canevas
root = Tk()
cnv = Canvas(root, width=unit*n-2, height=unit*n-2, background="ivory")
```

Le code complet est le suivant :

feu_clic_var.py

```
1 from random import sample
2 from tkinter import Tk, Canvas
3
4 COLORS=["ivory", "lime green", "red", "gray75"]
5
6 def random_forest(p, n):
7     units=[(line,col) for col in range(n) for line in range(n)]
8     ntrees=int(n**2*p)
9     trees=sample(units,ntrees)
10    states=[[0]*n for _ in range(n)]
11    for (i,j) in trees:
12        states[i][j]=1
13    return states
14
15 def voisins(n, i, j):
16    return [(a,b) for (a, b) in
17            [(i, j+1),(i, j-1), (i-1, j), (i+1,j)]
18            if a in range(n) and b in range(n)]
19
20 def update_states(states):
```

```
    n=len(states)
    to_fire=[]
    for line in range(n):
        for col in range(n):
            if states[line][col]==2:
                states[line][col]=3
                for (i, j) in voisins(n, line, col):
                    if states[i][j]==1:
                        to_fire.append((i, j))
    for (line,col) in to_fire:
        states[line][col]=2


def fill_cell(states, line, col):
        A=(unit*col, unit*line)
        B=(unit*(col+1), unit*(line+1))
        state=states[line][col]
        color=COLORS[state]
        cnv.create_rectangle(A, B, fill=color, outline='')

def fill(states):
    n=len(states)
    for line in range(n):
        for col in range(n):
            fill_cell(states, line, col)

def propagate():
    update_states(states)
    cnv.delete("all")
    fill(states)
    cnv.after(400, propagate)

def fire(event):
    global running
    i, j=event.y//unit, event.x//unit
    if states[i][j]==1:
        states[i][j]=2
        fill_cell(states, i, j)
        fill(states)
        if not running:
            running=True
            propagate()

p=0.62
n=50
unit=10
running=False

```

```
69 # Fenêtre et canevas
70 root = Tk()
71 cnv = Canvas(root, width=unit*n, height=unit*n, background="ivory")
72 cnv.pack()
73
74 cnv.bind("<Button>", fire)
75
76 # Forêt aléatoire
77 states=random_forest(p, n)
78
79 # Plateau dessiné
80 fill(states)
81
82 root.mainloop()
```

Curseur pour modifier la densité des arbres

On souhaite ajouter un peu d'interactivité en permettant à l'utilisateur de modifier la densité de la forêt, ce qui lui permet de mieux se rendre compte du seuil de percolation. La modification se fera à l'aide d'un curseur (le widget `Scale`) pour obtenir le résultat suivant :

On va repartir du code `foret_statique.py` qui affiche une vue statique de la forêt :

`foret_statique.py`

```
from random import sample
from tkinter import Tk, Canvas

COLORS=["ivory", "lime green", "red", "gray75"]

def random_forest(p, n):
    units=[(line,col) for col in range(n) for line in range(n)]
    ntrees=int(n**2*p)
```

```
    trees=sample(units,ntrees)
    states=[[0]*n for _ in range(n)]
    for (i,j) in trees:
        states[i][j]=1
    return states

def fill_cell(states, line, col):
    A=(unit*col, unit*line)
    B=(unit*(col+1), unit*(line+1))
    state=states[line][col]
    color=COLORS[state]
    cnv.create_rectangle(A, B, fill=color, outline='')

def fill(states):
    n=len(states)
    for line in range(n):
        for col in range(n):
            fill_cell(states, line, col)

p=0.60
n=50
unit=10

# Fenêtre et canevas
root = Tk()
cnv = Canvas(root, width=unit*n, height=unit*n, background="ivory")
cnv.pack()

# Forêt aléatoire
states=random_forest(p, n)

# Plateau dessiné
fill(states)

root.mainloop()
```

Le curseur va lire un pourcentage (la densité de la forêt), ce qui correspond à p dans le code ci-dessus. Ce pourcentage va être traité par une fonction de commande que l'on appelera `make_forest(percent)`. Cette fonction devra se charger de construire un plateau `states` avec la bonne densité et faire comme ci-dessus pour dessiner la forêt sur le canevas.

Créons un curseur vertical :

```
curseur = Scale(root, orient = "vertical", command=make_forest, from_=100,
                to=0, length=200)
curseur.set(100*p)
curseur.pack(side='right')
```

Le curseur sera initialisé à une valeur p par défaut placée dans le code (ci-dessous, `p=0.5`). Lorsqu'on clique sur le curseur, il appelle une fonction `make_forest`. Voici le code :

curseur.py

```
from random import sample, randrange
from tkinter import Tk, Canvas, Scale

COLORS=["ivory", "lime green", "red", "gray75"]

def random_forest(p, n):
    units=[(line,col) for col in range(n) for line in range(n)]
    ntrees=int(n**2*p)
    trees=sample(units,ntrees)
    states=[[0]*n for _ in range(n)]
    for (i,j) in trees:
        states[i][j]=1
    return states

def fill(states):
    n=len(states)
    for line in range(n):
        for col in range(n):
            A=(unit*col, unit*line)
            B=(unit*(col+1), unit*(line+1))
            state=states[line][col]
            color=COLORS[state]
            cnv.create_rectangle(A, B, fill=color, outline='')

def init(p):
    global states
    states=random_forest(p, n)
    fill(states)

def make_forest(percent):
    cnv.delete("all")
    p=float(percent)/100
    init(p)

p=0.50
n=80
unit=8

# Fenêtre et canevas
root = Tk()
cnv = Canvas(root, width=unit*n, height=unit*n, background="ivory")
cnv.pack(side="left")

trees=init(p)
fill(states)
curseur = Scale(root, orient = "vertical", command=make_forest, from_=100,
                to=0, length=200)
```

```
48 curseur.set(100*p)
49 curseur.pack(side='right')
50
51 init(p)
52
53 root.mainloop()
```

— On crée une fonction `init(p)` (ligne 25) qui, connaissant la densité `p`, crée la matrice des états initiaux (ligne 27) et dessine la forêt correspondante (ligne 28). La fonction `init` sert à montrer un premier plateau (ligne 44) et à recréer un plateau si le curseur est déplacé (ligne 33).

— La fonction `make_forest` lit sur le curseur la valeur `percent` indiquée par l'utilisateur. Cette valeur qui est une chaîne de caractères est convertie en un flottant représentant la densité de la forêt (ligne 32). Le plateau est effacé (ligne 31) pour permettre la création du nouveau plateau qui est généré par un appel à la fonction `init` (ligne 33).

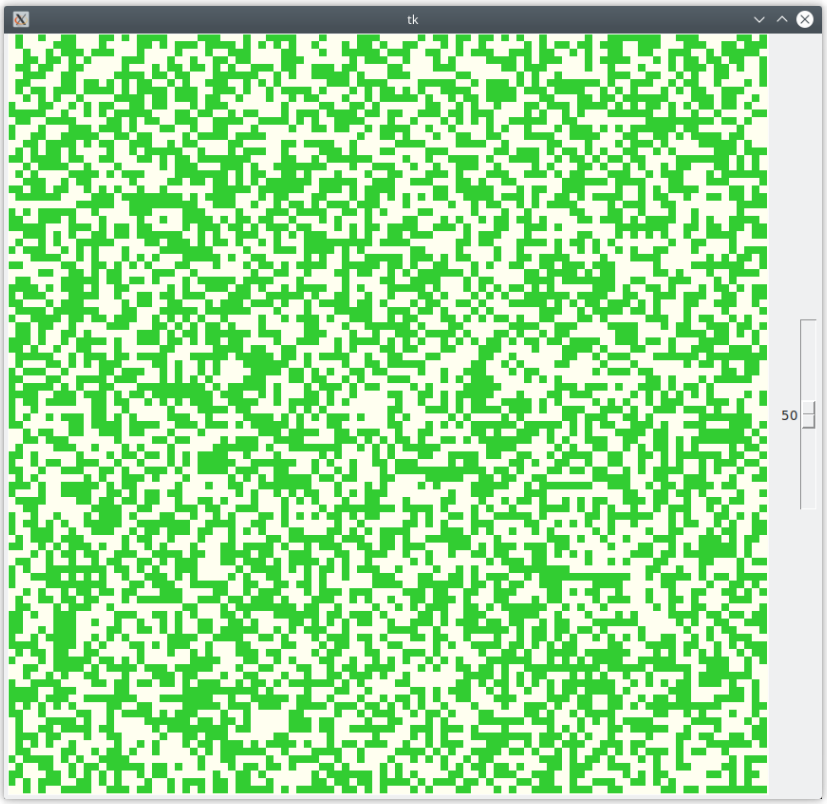
Variante

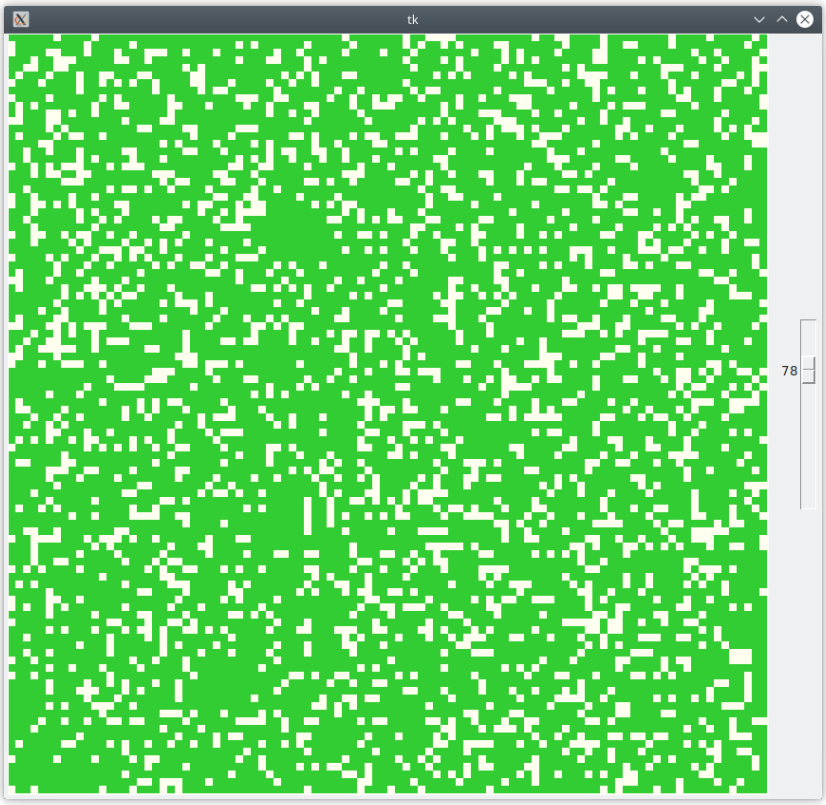
Dans le code précédent, lorque le curseur est modifié, la forêt est complètement renouvelée. On pourrait imaginer que la forêt est uniformément

— **complétée** si la densité demandée est plus forte que la densité de la forêt visible,

— **allégée** si la densité demandée est plus faible

ce qui donne un comportement plus lisse :





Il faut donc écrire une fonction `set_density` qui va modifier l'état du plateau pour l'ajuster à la nouvelle densité. Voici un code possible :

```
1  def set_density(states, p):
2      n=len(states)
3      trees= [(i,j) for i in range(n) for j in range(n) if states[i][j]==1]
4      nontrees=[(i,j) for i in range(n) for j in range(n) if states[i][j]!=1]
5      density=len(trees)/n**2
6      new_trees=round(n*n*p)
7      before=len(trees)
8      now=len(nontrees)
9      delta=abs(new_trees-before)
10     if new_trees>=before:
11         for (i, j) in sample(nontrees, delta):
12             states[i][j]=1
13     else:
14         for (i, j) in sample(trees, delta):
15             states[i][j]=0
```

— On recherche

  — la densité actuelle de la forêt (lignes 5)

  — on localise les cellules contenant des arbres (ligne 3) et on calcule le nombre d'arbres (ligne 7)

  — on localise les cellules où il y a le sol (ligne 4)

  - le nombre d'arbres de la future forêt (ligne 6)
- Selon les besoins, on supprime aléatoirement des arbres parmi les arbres existants ou on rajoute des arbres sur les emplacements libres.

D'où le code suivant :

curseur_progressif.py

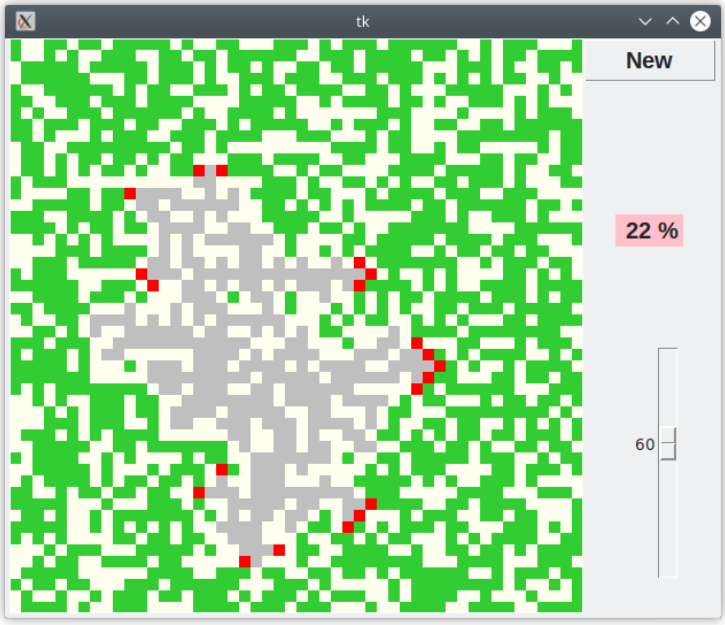
```
from random import sample, randrange
from tkinter import Tk, Canvas, Scale

COLORS=["ivory", "lime green", "red", "gray75"]

def random_forest(p, n):
    units=[(line,col) for col in range(n) for line in range(n)]
    ntrees=int(n**2*p)
    trees=sample(units,ntrees)
    states=[[0]*n for _ in range(n)]
    for (i,j) in trees:
        states[i][j]=1
    return states

def fill(states):
    n=len(states)
    for line in range(n):
        for col in range(n):
            A=(unit*col, unit*line)
            B=(unit*(col+1), unit*(line+1))
            state=states[line][col]
            color=COLORS[state]
            cnv.create_rectangle(A, B, fill=color, outline='')

def init(p):
    global states
    states=random_forest(p, n)
    fill(states)

def set_density(states, p):
    n=len(states)
    trees= [(i,j) for i in range(n) for j in range(n) if states[i][j]==1]
    nontrees=[(i,j) for i in range(n) for j in range(n) if states[i][j]!=1]
    density=len(trees)/n**2
    new_trees=round(n*n*p)
    before=len(trees)
    now=len(nontrees)
    delta=abs(new_trees-before)
    if new_trees>=before:
        for (i, j) in sample(nontrees, delta):
            states[i][j]=1
    else:
```

```
        for (i, j) in sample(trees, delta):
            states[i][j]=0

def make_forest(percent):
    cnv.delete("all")
    p=float(percent)/100
    set_density(states,p)
    fill(states)

p=0.5
n=100
unit=8

# Fenêtre et canevas
root = Tk()
cnv = Canvas(root, width=unit*n, height=unit*n, background="ivory")
cnv.pack(side="left")

trees=init(p)
fill(states)
curseur = Scale(root, orient = "vertical", command=make_forest, from_=100,
                to=0, length=200)
curseur.set(100*p)
curseur.pack(side='right')

root.mainloop()
```

Application complète

On veut désormais que l'animation possède toutes les fonctionnalités précédentes réunies :

- un curseur pour modifier la densité de la forêt
- enflammer par clics de souris
- le label de pourcentage
- un bouton pour relancer une animation.



La difficulté est que l'animation peut être initialisée plusieurs fois et donc qu'il faut écrire une fonction d'initilisation définissant certains paramètres de l'animation qui doivent être visibles d'autres fonctions.

Voici un code réalisant les fonctionnalités cités. Seules les fonctions véritablement nouvelles sont montrées :

```
# Code omis

def init():
    global states, cpt, ntrees, running

    p=int(curseur.get())/100
    running=False
    cpt=0
    lbl["text"]="%3s %%" %0
    curseur["state"]='normal'
    states=random_forest(p, n)
    ntrees=int(n*n*p)
    cnv.delete(ALL)
    fill(states)

def set_density(states, p):
    n=len(states)
    trees= [(i,j) for i in range(n) for j in range(n) if states[i][j]==1]
    nontrees=[(i,j) for i in range(n) for j in range(n) if states[i][j]!=1]
    density=len(trees)/n**2
    new_trees=int(n*n*p)
    before=len(trees)
    now=len(nontrees)
```

```
    delta=abs(new_trees-before)
    if new_trees>=before:
        for (i, j) in sample(nontrees, delta):
            states[i][j]=1
    else:
        for (i, j) in sample(trees, delta):
            states[i][j]=0

def make_forest(percent):
    global ntrees

    cnv.delete("all")
    p=float(percent)/100
    ntrees=int(n*n*p)
    set_density(states,p)
    fill(states)


def propagate():
    global cpt, running

    update_states(states)
    nfires=sum(states[i][j]==2 for i in range(n) for j in range(n))
    cpt+=nfires
    percent = int(cpt/ntrees*100)
    cnv.delete("all")
    fill(states)
    lbl["text"]="%3s %%" %percent
    if nfires==0:
        running=False
        return
    cnv.after(150, propagate)

def fire(event):
    global running, cpt

    i, j=event.y//unit, event.x//unit
    if states[i][j]==1:
        states[i][j]=2
        fill_cell(i, j)

        cpt+=1
        if not running:
            running=True
            curseur["state"]='disabled'
            propagate()

# Code omis
```

```
72
73 init()
74
75 root.mainloop()
```

Examinons comment s'effectue l'exécution de l'application.

A tout instant, on peut relancer une nouvelle animation en cliquant sur le bouton `New`. On peut créer à la souris de nouveaux foyers **pendant** un incendie qui est déjà en train de se propager. Avant de déclencher l'incendie, on peut modifier la densité. Une fois l'incendie en cours, le curseur est désactivé (ligne 70). Pour le réactiver, il faut créer une nouvelle forêt, cf. ligne 10.

La fonction `init`

— La fonction `init` est appelée (ligne 73). Comme son nom l'indique, elle initialise le jeu et affiche une forêt statique.

— Certaines variables définies dans `init` sont indispensables à d'autres fonctions, par exemple `states` doit être connue de pratiquement toutes les fonctions dont on ne maîtrise pas le passage des arguments, telles que la fonction `fire`. Donc, avec ce découpage, on est **obligé** d'utiliser une variable globale pour `states`.

— On introduit une nouvelle variable `running` qui est un drapeau indiquant si l'animation tourne ou pas. Au départ, elle est à `False`. Cela permet de désactiver l'animation ou d'empêcher que deux animations aient lieu en même temps, cf. lignes 66-69.

— Une fois la fonction `init` exécutée, l'interface est en attente d'une activité de la souris (clic sur le canevas, sur le bouton ou déplacement du curseur).

— La densité ne doit pas être gravée dans la fonction `init` mais être lue depuis le curseur lui même (ligne 6) puisque si on réinitialise le canevas, il est attendu que la densité de la forêt corresponde à celle indiquée par le curseur.

— Comme la fonction `init` peut être appelée après une animation, il faut remettre à zéro le pourcentage de dégâts (ligne 8).

La fonction `make_forest`

Elle est appelée (ligne 32) suite à un mouvement du curseur. Comme la densité change, il faut mettre à jour le nombre d'arbres `ntrees` (ligne 37) dont on a besoin pour calculer le pourcentage. La variable `ntrees` doit être globale pour être connue par exemple de la fonction `propagate` (ligne 48).

Les fonctions `propagate` et `fire`

Elles ont assez peu changé.

— S'il n'y a aucun arbre enflammé (ligne 52), l'incendie s'arête et donc l'animation se termine, cf . ligne 54. Le drapeau `running` doit être mis à `False` (ligne 53). Cela oblige à placer `running` en `global` (ligne 43).

— Lorsqu'on clique sur une cellule pour l'enflammer, il faut penser à incrémenter le compteur, cf. ligne 65 (sinon, le calcul de pourcentage est faussé).

Voici le code complet :

feu_foret_final.py

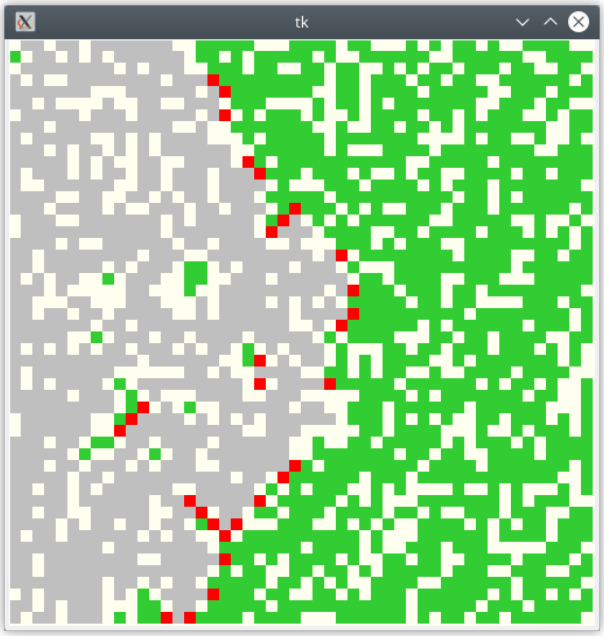
```
from random import sample, randrange
from tkinter import Tk, Canvas, Scale, Button, Label, N, ALL

COLORS=["ivory", "lime green", "red", "gray75"]

def random_forest(p, n):
    units=[(line,col) for col in range(n) for line in range(n)]
    ntrees=int(n**2*p)
    trees=sample(units,ntrees)
    states=[[0]*n for _ in range(n)]
    for (i,j) in trees:
        states[i][j]=1
    return states

def voisins(n, i, j):
    return [(a,b) for (a, b) in
            [(i, j+1),(i, j-1), (i-1, j), (i+1,j)]
            if a in range(n) and b in range(n)]

def fill_cell(states, line, col):
        A=(unit*col, unit*line)
        B=(unit*(col+1), unit*(line+1))
        state=states[line][col]
        color=COLORS[state]
        cnv.create_rectangle(A, B, fill=color, outline='')

def fill(states):
    n=len(states)
    for line in range(n):
        for col in range(n):
            fill_cell(states, line, col)

def update_states(states):
    n=len(states)
    to_fire=[]
    for line in range(n):
        for col in range(n):
            if states[line][col]==2:
                states[line][col]=3
                for (i, j) in voisins(n, line, col):
                    if states[i][j]==1:
                        to_fire.append((i, j))
    for (line,col) in to_fire:
        states[line][col]=2

def init():
    global states, cpt, ntrees, running
```

```
48
49      p=int(curseur.get())/100
50      running=False
51      cpt=0
52      lbl["text"]="%3s %%" %0
53      curseur["state"]='normal'
54      states=random_forest(p, n)
55      ntrees=int(n*n*p)
56      cnv.delete(ALL)
57      fill(states)
58
59  def set_density(states, p):
60      n=len(states)
61      trees= [(i,j) for i in range(n) for j in range(n) if states[i][j]==1]
62      nontrees=[(i,j) for i in range(n) for j in range(n) if states[i][j]!=1]
63      density=len(trees)/n**2
64      new_trees=int(n*n*p)
65      before=len(trees)
66      now=len(nontrees)
67      delta=abs(new_trees-before)
68      if new_trees>=before:
69          for (i, j) in sample(nontrees, delta):
70              states[i][j]=1
71      else:
72          for (i, j) in sample(trees, delta):
73              states[i][j]=0
74
75  def make_forest(percent):
76      global ntrees
77
78      cnv.delete("all")
79      p=float(percent)/100
80      ntrees=int(n*n*p)
81      set_density(states,p)
82      fill(states)
83
84
85  def propagate():
86      global cpt, running
87
88      update_states(states)
89      nfires=sum(states[i][j]==2 for i in range(n) for j in range(n))
90      cpt+=nfires
91      percent = int(cpt/ntrees*100)
92      cnv.delete("all")
93      fill(states)
94      lbl["text"]="%3s %%" %percent
95      if nfires==0:
96          running=False
```

```
97          return
98      cnv.after(150, propagate)
99
100 def fire(event):
101     global running, cpt
102
103     i, j=event.y//unit, event.x//unit
104     if states[i][j]==1:
105         states[i][j]=2
106         fill_cell(states, i, j)
107
108         cpt+=1
109         if not running:
110             running=True
111             curseur["state"]='disabled'
112             propagate()
113
114 n=50
115 unit=10
116
117 # Fenêtre et canevas
118 root = Tk()
119 cnv = Canvas(root, width=unit*n-2, height=unit*n-2, background="ivory")
120 cnv.grid(row=0, column=0, rowspan=4)
121
122 btn=Button(root,text="New",  font='Arial 15 bold', command=init, width=8)
123 btn.grid(row=0, column=1, sticky=N)
124
125 lbl=Label(root,text="%3s %%" %0,  font='Arial 15 bold', bg='pink', width=5)
126 lbl.grid(row=2, column=1, sticky=N)
127
128 # Clic qui met le feu
129 cnv.bind("<Button-1>", fire)
130
131 curseur = Scale(root, orient = "vertical", command=make_forest, from_=100,
132                 to=0, length=200)
133 curseur.set(50)
134 curseur.grid(row=3, column=1)
135
136 init()
137
138 root.mainloop()
```

Simulations numériques

Observons la propagation de l'incendie sur l'animation :



et en même temps, reprenons le code de la fonction de transition d'un état de la forêt à l'état suivant :

```
1 def update_states(states):
2     n=len(states)
3     to_fire=[]
4     for line in range(n):
5         for col in range(n):
6             if states[line][col]==2:
7                 states[line][col]=3
8                 for (i, j) in voisins(n, line, col):
9                     if states[i][j]==1:
10                        to_fire.append((i, j))
11    for (line,col) in to_fire:
12        states[line][col]=2
```

La double boucle `for` (lignes 4-5) parcourt **toute** la grille à la recherche de cellules en feu (ligne 6). Or, comme le montre l'animation, le **front de flammes** représente une partie **très réduite** de la zone parcourue et donc, il est bien plus efficace de mémoriser les positions des cellules en flammes puis de les passer en revue car c'est seulement d'elles que proviennent les nouveaux foyers.

Pour pouvoir effectuer quelques simulations numériques, on va donc coder autrement la fonction de mise à jour `update_states` en sorte qu'elle renvoie la liste des cellules en feu à l'issue de la transition. Voici un code possible :

```
1 def update_states(states, fires):
2     n=len(states)
3     to_fire=[]
4     for (line, col) in fires:
```

```
5           for (i, j) in voisins(n, line, col):
6               if states[i][j]==1:
7                   to_fire.append((i, j))
8       for (line, col) in to_fire:
9           states[line][col]=2
10      for (line, col) in fires:
11          states[line][col]=3
12
13      return list(set(to_fire))
```

— Ligne 1 : la fonction a besoin des positions des cellules en feu, cf. le paramètre `fires`. En effet, cf. ligne 4, la fonction parcourt la liste des cellules en feu.

— Lignes 5-7 : si une cellule est en feu (ligne 4), on regarde si parmi ses voisines (ligne 5) se trouverait un arbre (ligne 6), auquel cas cet arbre va prendre feu et est donc inclus dans la liste des arbres nouvellement en feu (ligne 7).

— Sous peine de fausser l'obtention du nouvel état, on ne met pas à jour directement la matrice des états et on place dans une liste temporaire, ici `to_fire`, les nouveaux foyers ligne 7).

— Les changement d'états ne peuvent provenir que des cellules initialement en flammes. Donc une fois toutes les cellules en feu examinées (à partir de la ligne 8), la transition est achevée et il est possible de mettre à jour la plateau des cellules en cendres et des cellules inflammées (lignes 8-11).

— Ligne 13 : pour les besoins de futures itérations, la fonction doit renvoyer la liste des feux qui sont apparus. On ne peut renvoyer la liste `to_fire` car elle peut contenir des doublons. C'est pour les éliminer qu'on applique le constructeur `set` à la liste, suivant une technique bien connue.

Phase de test

Pour essayer d'approcher expérimentalement le seuil de percolation, on se donne une densité `p`, ci-dessous `p` variera entre 0.5 et 0.7 et on répète un incendie un nombre `repet` de fois, ci-dessous `repet=100`, sur une parcelle de taille `n` × `n` donnée, ci-dessous `n` = 150 et on calcule la moyenne des proportions de forêt brûlée. Voici le code :

`simulation.py`

```
1  from random import sample, randrange
2
3  def random_forest(p, n):
4      units = [(line, col) for col in range(n) for line in range(n)]
5      ntrees = int(n**2*p)
6      trees = sample(units, ntrees)
7      states = [[0]*n for _ in range(n)]
8      for (i, j) in trees:
9          states[i][j] = 1
10     return states, trees
11
12 def voisins(n, i, j):
13     return [(a, b) for (a, b) in
14             [(i, j+1), (i, j-1), (i-1, j), (i+1, j)]
15             if a in range(n) and b in range(n)]
```

```
16
17 def start_fire(states, trees):
18     i, j = trees[randrange(len(trees))]
19     states[i][j] = 2
20     return (i, j)
21
22 def update_states(states, fires):
23     n = len(states)
24     to_fire = []
25     for (line, col) in fires:
26         for (i, j) in voisins(n, line, col):
27             if states[i][j] == 1:
28                 to_fire.append((i, j))
29     for (line, col) in to_fire:
30         states[line][col] = 2
31     for (line, col) in fires:
32         states[line][col] = 3
33
34     return list(set(to_fire))
35
36 def test(n, p, repet):
37     stats = []
38     for _ in range(repet):
39         states, trees = random_forest(p, n)
40         i, j = start_fire(states, trees)
41         ntrees = len(trees)
42         nfire = 1
43         foyers = [(i, j)]
44         while True:
45             foyers = update_states(states, foyers)
46             fire = len(foyers)
47             nfire += fire
48             if fire == 0:
49                 break
50         stats.append(nfire/ntrees)
51     return sum(stats)/repet
52
53 n=150
54 repet=100
55 f=lambda p:test(n,p,repet)
56 P = [0.5+k/1000 for k in range(0, 201, 5)]
57 Q = list(map(f, P))
```

— Ligne 56 : on examine l'incendie pour les densités possibles entre 0.5 et 0.7 par tranches de 5/1000.

— Le traitement d'un seul incendie (lignes 39-49).

— Pour chaque densité p (ligne 36), on évalue le taux de forêt consumée (par rapport à la forêt intacte, cf. ligne 50) en répétant `repet` incendies (ligne 38), chaque taux étant placé dans la liste `stats` (lignes 37 et 51).

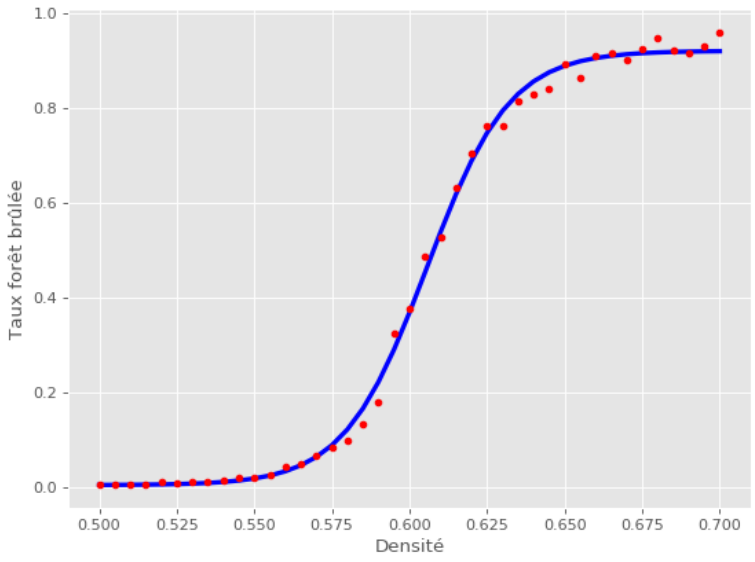
L'exécution donne sur un processeur i5 :

```
$ time python3 simulation.py
real    1m51,388s
user    1m51,304s
sys     0m0,017s
$
```

ce qui est déjà assez long. La simulation serait complètement irréalisable si on avait gardé la première méthode de traitement de la propagation de l'incendie, cf. le code `transition.py`. Pour accélérer les calculs, on peut utiliser l'interpréteur PyPy ce qui divise ici le temps d'exécution par plus d'un facteur 5. Il serait possible d'accélérer l'exécution en utilisant Numpy et en vectorisant le code, ou alors d'écrire le code en Cython ou encore avec Numpy + Numba ce qui donnerait sans doute de très bonnes performances vu le type de code.

Visualisation

Pour visualiser, il faut avoir installé les packages Numpy, SciPy et Matplotlib. Il serait plus commode de placer le code dans une feuille Jupyter Notebook. On peut observer la courbe fittée des résultats de la simulation précédente :



Voici le code correspondant :

```
from matplotlib.pyplot import figure
from scipy.optimize import curve_fit
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('ggplot')

from simulation import P as X, Q as Y

print("Data generated!")

```

```
11 def sigmoid(x, a, b, c, d):
12     return a/(b+np.exp(-c*x))+d
13
14 x = np.array(X)
15 y = np.array(Y)
16 popt, _ = curve_fit(sigmoid, x, y, maxfev=5000)
17
18 # Ordonnées de la courbe fittée
19 yy = sigmoid(x, *popt)
20
21 # Dessin
22 figure(num=None, figsize=(8, 6), dpi=80, facecolor='w', edgecolor='k')
23 plt.plot(x, yy, color="blue", linewidth=3)
24 plt.plot(x, y, 'o', markersize=4, color='red')
25 plt.xlabel('Densité')
26 plt.ylabel('Taux forêt brûlée')
27 plt.show()
```

— Ligne 7 : on a besoin du fichier `simulation.py` qui génère le nuages de points.

— lignes 2-3 : pour le calcul de l'ajustement, on a besoin de SciPy qui lui même utilise Numpy.

— Lignes 11-12 : Le nuage (cf. lignes 14-15) est fitté (cf. ligne 16) par une sigmoïde qu'il faut définir (lignes 11-12) et dépendant de 4 paramètres `a`, `b`, `c` et `d`.

— Lignes 14-15 : il faut convertir les listes en tableaux Numpy.

— Ligne 16 : les 4 paramètres sont calculés dans `popt` par la fonction `curve_fit` pour un ajustement par la méthode des moindres carrés ce qui va permettre de tracer la courbe fittée (en bleu sur le dessin).

— Ligne 27 : utile si on n'exécute pas le fichier dans une feuille Jupyter Notebook.