Python
Le Jeu de la Vie

Pidash Angelina
Lentali Thomas
Université de Bordeaux

Décembre 2015

1 Introduction

Ce projet a pour but de créer un programme sur le Jeu de la Vie de John Horton Conway en langage Python. Nous allons travailler dans un environnement composé de:

- un temps discret,
- un terrain de jeu - le plan, qui est constitué de cellules, chaque cellule a huit voisins,
- de cellules qui peuvent avoir deux états distincts: mort ou vivant. La totalité des cellules vivantes est appelée génération.

Nous allons appliquer les règles classiques suivantes aux cellules:

- chaque génération est calculée à partir de la précédente,
- une cellule devient vivante si elle a exactement 3 cellules vivantes comme voisins,
- une cellule vivante continue à vivre, si elle a 2 ou 3 voisins vivants.

2 Implémentation

2.1 Modèle de base

Le fonctionnement de base du jeu de la vie est défini comme suit:

1. Créer un tableau à deux dimensions de valeurs booléennes, où "1" pour signifier la présence d'une cellule vivante, et "0" pour signifier l'état mort de la cellule,
2. Générer un nouveau tableau,
3. Pour chaque cellule, il faut déterminer le nombre de voisins vivant qui l'entoure et définir si une cellule commence à vivre pour la nouvelle génération ou non,
4. Le nouveau tableau est écrit à la place de l'ancien,
5. Revenir à l'étape 2.

L'algorithme présente plusieurs inconvénients:

- le temps d'exécution linéaire par rapport à la taille du champ.
- La double consommation de mémoire (nécessaire pour stocker la génération précédente + nouvelle).
- Il exige beaucoup de code à mettre en œuvre.

Ceci nous a poussé à repenser cette idée de réalisation et l'avons optimisé en éliminant les inconvénients cités précédemment.

2.2 Amélioration du modèle

Pour augmenter le confort d'utilisation de notre implémentation du Jeu de la vie, nous avons utilisé la bibliothèque graphique ***Tkinter***.

```
from Tkinter import Tk, Canvas, Button, Frame, BOTH, NORMAL, HIDDEN
```

On crée une fênetre:

```
root = Tk()
```

On définit la hauteur et la largeur de la fenêtre:

```
win_width = 500
win_height = 500
config_string = "{0}x{1}".format(win_width, win_height + 32)
```

A l'aide de la méthode ***geometry()***, nous définissons la taille notre fenêtre:

```
root.geometry(config_string)
```

Nous définissons la largeur d'une cellule:

```
cell_size = 20
```

On crée l'objet Canvas dans la fenêtre root dans lequel nous allons directement dessiner les cellules.

```
canvas = Canvas(root, height=win_height)
canvas.pack(fill=BOTH)
```

On détermine la taille du champ selon les cellules:

```
field_height = win_height // a
field_width = win_width // a
```

On crée le tableau unidimensionnel afin de garder les cellules. On crée un Frame afin de garder les boutons:

```
frame = Frame(root)
btn1 = Button(frame, text='Next generation', command = step)
frame.pack(side='bottom')
```

Ici nous permettons une relation entre mouvement de la souris (en cliquant) avec le Canvas grâce à la fonction ***interpret***:

```
canvas.bind('', interpret)
root.mainloop()
```

2.3 Partie Fonctionnelle

La fonction ***interpret()*** récupère les coordonnées de la souris lorsque l'utilisateur clique une fois sur une case du Canvas ou, elle récupère une série de coordonnées lorsque l'utilisateur bouge sa souris sur le Canvas en maintenant le bouton gauche de la souris. Ces actions donne vie à une cellule morte.

```
def interpret(e):
    ii = (e.y-3) // cell_size
    jj = (e.x-3) // cell_size
    canvas.itemconfig(cell_matrix[addpoint(ii, jj)], state=NORMAL, tags='vis')
```

La fonction ***addpoint()*** convertit les deux dimensions de coordonnées dans l'adresse de notre un tableau en une dimension simple.

```
def addpoint(ii,jj):
    if(ii < 0 or jj < 0 or ii >= field_height or jj >= field_width):
        return len(cell_matrix)-1
    else:
        return ii * (win_width // cell_size) + jj
```

Le rôle de la fonction ***nextGeneration()*** est de:

- mettre à jour l'image,
- calcul le nombre de voisins,
- définit l'état d'une cellule en fonction du nombre de ses voisins .

Le statut de vie ou de mort de la cellule est géré avec des tags dans le canvas, "to vis" correspond à une cellule vivante, et "to hid" à une cellule morte:

```
   if(vie[k]=="1"):
      canvas.itemconfig(cell_matrix[addpoint(i, j)], tags=(current_tag, 'to_vis'))
   else:
      canvas.itemconfig(cell_matrix[addpoint(i, j)], tags=(current_tag, 'to_hid'))
```

La fonction ***repaint()*** a pour objectif de mettre à jour le Canvas avec les nouveaux états des cellules:

```
def repaint():
  for i in range(field_height):
        for j in range(field_width):
                if (canvas.gettags(cell_matrix[addpoint(i, j)])[1] == 'to_hid'):
                        canvas.itemconfig(cell_matrix[addpoint(i, j)], state=HIDDEN, tags=('hid','0'))
                if (canvas.gettags(cell_matrix[addpoint(i, j)])[1] == 'to_vis'):
                        canvas.itemconfig(cell_matrix[addpoint(i, j)], state=NORMAL, tags=('vis','0'))
```

Lorsque l'on souhaite générer et afficher la nouvelle génération de cellules, nous appelons la fonction ***step()*** qui elle même appelle les fonctions ***nextGeneration()*** et ***repaint()***.

3 Intéraction Utilisateur-Programme

Nous avons enrichi notre programme avec plusieurs options qui permettent à l'utilisateur d'agir sur le Jeu de la Vie (en plus d'être sensible aux clique de souris dans le Canvas).

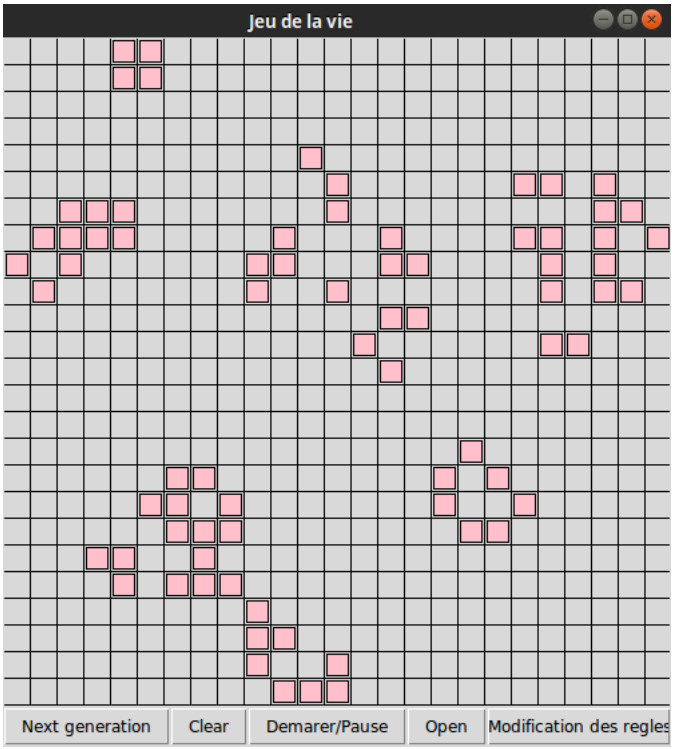

Figure 1: Interface graphique du Jeu de la Vie

L'utilisateur peut notamment:

- afficher la nouvelle génération de cellules en appuyant sur le bouton ***Next Generation***,
- lancer de façon automatique les nouvelles générations en appuyant sur le bouton ***Automatique/Pause*** afin que l'utilisateur n'ai pas à cliquer sur ***Next Generation*** pour générer une nouvelle génération à la fois,
- appuyer sur ce même bouton ***Automatique/Pause*** permet de mettre l'exécution automatique en pause, il suffit d'appuyer à nouveau sur ce même bouton pour relancer l'exécution automatique (principe "play/pause"),
- effacer les cellules vivantes sur le canvas en cliquant sur ***Clear***,
- ouvrir une configuration du canvas enregistré sur un fichier au format .txt composé d'une matrice dont les "1" représentent les cellules vivantes et les "0" les cellules mortes, via le bouton ***Open a File***. En sélectionnant un fichier dans un dossier, on efface la configuration courante sur le canvas qui affiche la nouvelle configuration. Nous proposons un choix de configurations originales comme, par exemple, différentes tailles de vaisseaux:

```
def file_to_open():
  file_path = askopenfilename()
  clear()
  try:
      ofi = open(file_path, 'rb')
  except:
      tkMessageBox.showwarning(
          "Open file", "Cannot open this file: (%s)\n" % file_path)
  compteur1 = 0
```

```
for text in ofi:
    elt = text.split(' ')
    compteur2 = 0
    for x in range(len(elt)):
        if int(elt[int(x)]) == 1:
            canvas.itemconfig(cell_matrix[addpoint(compteur1,compteur2)],state=NORMAL,tags='vis')
        compteur2 += 1
    compteur1 += 1
```

Exemple de fichier .txt lu par notre programme:

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

- changer les règles de vie et de mort des cellules en cliquant sur le bouton ***Modification des règles***. En cliquant, une petite fenêtre s'ouvre et demande à l'utilisateur d'entrer les nouvelles règles au format binaire. Une fois les règles sont tapé dans la entry box, l'utilisateur doit cliquer sur le bouton ***Changer les règles***. Un second bouton, appellé ***Règles de base*** permet de réinitialiser les règles originale du jeu et un troisième bouton, ferme la fenêtre. Les erreurs sont gérées au niveau de la saisie de la nouvelle règle.

```
def new_rules():
  global  e1, e2, new_vie, new_mort
  ask_rules = Tkinter.Toplevel(root)
  ask_rules.title('New Rules')
  ask_rules.geometry('450x80')

  l1 = Tkinter.Label(ask_rules).grid(row=0,column=0)
  e1 = Tkinter.Entry(ask_rules)
  e1.grid(row = 0, column = 1)

  l2 = Tkinter.Label(ask_rules).grid(row=1,column=0)
  e2 = Tkinter.Entry(ask_rules)
  e2.grid(row = 1, column = 1)

  Button(ask_rules,command=ask_rules.destroy).grid(row=2, column= 1)
  Button(ask_rules,command=default_values).grid(row=2, column= 2)
  Button(ask_rules,command=change_values).grid(row=2, column= 0)

  root.mainloop()
```

Conclusion

Ce projet nous a permis de pousser plus loin nos compétences en programmation.
Que ce soit du point de vu algorithmique avec l'amélioration de la structure de base du jeu de la vie et aussi du point de vu technique en créant une interface graphique qui peut notamment interagir directement avec les mouvements de la souris de l'utilisateur ainsi que la création de multiples fonctions utilisant différentes bibliothèques.