Le jeu de la vie : rapport de projet

Joffrey Cottin, Clarine Azinmedem, Christophe Ravier

27 mai 2022

Table des matières

1 Des recherches à la première feuille de route

Nous sommes partis sur l'idée de travailler autour du jeu de la vie, un thème déjà bien étudié mais sur lequel on trouve encore régulièrement des nouvelles découvertes aussi bien mathématiques qu'algorithmiques. Dans le cadre de notre projet, ce choix offre l'avantage de devoir penser les algorithmes aussi bien pour gérer les données "spatiales" (la grille, théoriquement infinie) que "temporelles" (l'évolution, sur un nombre de générations potentiellement très grand) et donc de devoir travailler pour obtenir des algorithmes efficaces. Le sujet permet aussi d'introduire naturellement une interface graphique, tout d'abord pour offrir une visualisation des évolutions de la population au fil des générations mais aussi pour offrir à l'utilisateur des possibilités d'interaction afin de faire ses propres expérimentations (placements dans la grille, sauvegardes ...)

Afin de mieux cerner le sujet et pour réfléchir au contenu à intégrer, nous avons débuté le travail par quelques recherches bibliographiques autour des automates cellulaires (introduits par Von Neumann alors qu'il travaillait sur des systèmes autoréplicatifs), leur rôle, les questions scientifiques à leur sujet, voire métaphysiques – certains envisageant l'univers comme un automate cellulaire simple dont les lois physiques seraient des propriétés émergentes ; *cf* bibliographie (page 11).

À partir de là, une première réflexion à propos du contenu minimal du projet (faire tourner le code du jeu de la vie dans une console Python) a permis d'arriver à une première feuille de route, présentant les étapes vers ce contenu minimal ainsi que ce qui pourrait l'étoffer pour obtenir un projet complet. Le travail en équipe sur la feuille de route a été l'occasion d'apprendre à utiliser Github, ce qui a été bien utile par la suite au moment de partager le code.

Cette feuille de route "idéale" (page 10) a été utile pour garder à l'esprit les différents objectifs et les hiérarchiser. D'un point de vue de la réalisation pratique, la vie privée et professionnelle (travail à temps plein en dehors du DU) a beaucoup pesé sur les ambitions initiales, générant une importante frustration. Ceci ajouté au fait que, si le travail de groupe est toujours difficile à organiser, il l'est encore plus à distance entre trois personnes qui ne se connaissent pas et ne se croisent jamais. Pour palier aux difficultés de coordinations, nous avons utiliser GitHub est un site web et un service de cloud qui aide les développeurs à stocker et à gérer leur code, ainsi qu'à suivre et contrôler les modifications qui lui sont apportées.

2 Le code

2.1 Règles du jeu de la vie, version classique de John H. Conway

Le jeu se déroule sur une grille rectangulaire de cellules ayant un état binaire : elles sont soit vivantes, soit mortes. L'évolution temporelle se fait de façon discrète : le nouvel état de toutes les cellules de la grille est mis à jour simultanément en fonction de leur voisines à l'état précédent. La règle de transition est dite B3/S23 (pour Born 3, Stay 2 ou 3) qui signifie qu'une cellule morte devient vivante si elle possède exactement 3 voisines vivantes et qu'une cellule vivante reste en vie si elle possède 2 ou 3 voisines vivantes. Dans les autres cas, une cellule morte reste morte et une cellule vivante décède.

2.2 Implémentation de la version "Terminal"

Pour implémenter ce jeu, nous avons construit deux classes, l'une implémentant une cellule et l'autre une grille qui permettra de suivre l'évolution du jeu de la vie sur une succession de générations. Pour tout le projet, c'est la notation matricielle qui est utilisée pour se repérer sur la grille, `(i, j)` représente ainsi le numéro de ligne puis le numéro de colonne.

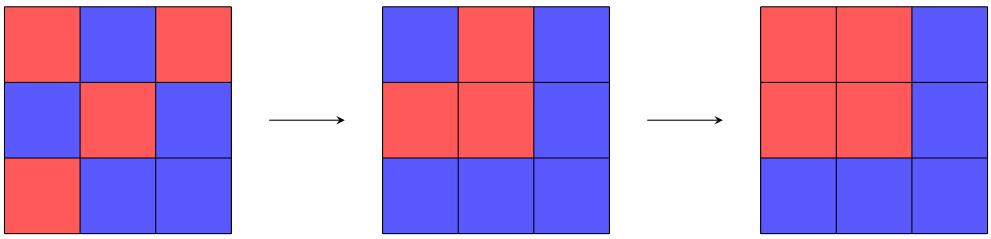

FIGURE 1 – Exemple d'une transition B3/S23 dans une configuration plane.

2.2.1 La classe `Cellule`

La classe `Cellule` du fichier `cellule.py` contient trois attributs : l'état actuel de la cellule, son état futur et la liste des cellules voisines. La nécessité d'avoir deux attributs pour les états vient du fait que le changement d'état de toutes les cellules d'une grille devant être simultané, on ne peut pas modifier l'état d'une cellule avant de l'avoir utilisé sur ses voisines à venir.

```
13      def __init__(self : Cellule) -> None:
14          """ Constructeur de la classe Cellule """
15          self.actuel = False
16          self.futur = False
17          self.voisins = []
```

FIGURE 2 – Le constructeur de la classe `Cellule` de la version "Terminal"

La méthode `calcule_etat_futur()` parcourt la liste des voisins de la cellule et, en fonction du nombre de voisins vivants, modifie l'état futur de la cellule considérée.

```
51      def calcule_etat_futur(self: Cellule) -> None:
52          """ implémente les règles d'évolution du jeu de la vie en \
53          préparant l'état futur à sa nouvelle valeur """
54          # on compte le nombre de voisins vivants
55          nbre_voisins_vivants = 0
56          for voisin in self.voisins:
57              if voisin.est_vivant():
58                  nbre_voisins_vivants += 1
59          # on applique les règles d'évolution
60          if (nbre_voisins_vivants != 2) and (nbre_voisins_vivants != 3):
61              self.mourir()
62          elif nbre_voisins_vivants == 3:
63              self.naitre()
64          else:
65              self.futur = self.actuel
```

FIGURE 3 – La méthode `calcule_etat_futur()` de la version "Terminal"

Puis c'est la méthode `basculer()` qui, appelée en temps voulu, modifie l'état actuel de la cellule.

```
39      def basculer(self: Cellule) -> None:
40          """ affecte l'état futur de la cellule à l'état actuel """
41          self.actuel = self.futur
```

FIGURE 4 – La méthode `basculer()` de la version "Terminal"

Les autres méthodes de cette classe ne sont pas détaillées ici.
La fin de cette classe consiste en une série de tests unitaires afin de s'assurer de la correction des méthodes de la classe.

2.2.2 La classe `Grille`

La classe `Grille` du fichier `grille.py` contient quatre attributs : le nombre de lignes (`nbLignes`) ; de colonnes (`nbColonnes`) ; `matrix`, un tableau à deux dimensions contenant des cellules (toutes mortes à la création de la grille) et enfin `typeg` qui définit le type de grille (plane ou torique).

```
19      def __init__(self: Grille, nbLignes: int, nbColonnes: int,\
20                               typeg: str) -> None:
21          """ Constructeur de la classe Grille """
22          self.nbLignes = nbLignes
23          self.nbColonnes = nbColonnes
24          self.matrix = [[Cellule() for j in range(self.nbColonnes)]\
25                                    for i in range(self.nbLignes)]
26          self.type_grille = typeg
```

FIGURE 5 – Le constructeur de la classe `Grille` de la version "Terminal"

La méthode `remplir_alea(taux)` permet de passer aléatoirement selon un taux moyen donné certaines cellules à l'état vivant et la méthode `affecte_voisins()` va construire pour chaque cellule sa liste de voisins.

```
91      def remplir_alea(self, taux: int):
92          """ remplit aléatoirement la Grille avec un certain taux de \
93          Cellule vivantes """
94          for i in range(self.nbLignes):
95              for j in range(self.nbColonnes):
96                  if random() <= (taux / 100):
97                      cellule = self.getXY(i, j)
98                      cellule.naitre()
99                      cellule.basculer()
```

FIGURE 6 – La méthode `remplir_alea(taux)` de la classe `Grille`

Une fois la grille prête à débuter, le déroulement des générations successives du jeu de la vie se fait en deux étapes, ainsi qu'expliqué plus haut.

La méthode `jeu()` parcourt les cellules et calcule pour chacune leur état futur, le résultat étant stocké dans l'attribut correspondant de la cellule.

```
101     def jeu(self: Grille) -> None:
102         """ passe en revue toutes les Cellule de la Grille, calcule \
103         leur état futur """
104         for i in range(self.nbLignes):
105             for j in range(self.nbColonnes):
106                 cellule = self.getXY(i, j)
107                 cellule.calcule_etat_futur()
```

FIGURE 7 – La méthode `jeu()` de la classe `Grille`

Dans un second temps, lorsque tous les états futurs ont été calculés, la méthode `actualise()` reparcourt la grille et réalise la bascule pour chaque cellule afin d'obtenir la nouvelle génération.

```
109     def actualise(self: Grille) -> None:
110         """ bascule toutes les Cellule de la Grille dans leur état \
111         futur """
112         for i in range(self.nbLignes):
113             for j in range(self.nbColonnes):
114                 cellule = self.getXY(i, j)
115                 cellule.basculer()
```

FIGURE 8 – La méthode `actualise()` de la classe `Grille`

Cette classe se termine comme la précédente avec un jeu de tests unitaires afin de tester les différentes méthodes.

2.2.3 Le fichier `main.py`

Le fichier est le fichier à exécuter via la commande `python3 main.py` dans un `Terminal`

2.3 Implémentation de l'interface graphique

On utilise tout d'abord deux classes `CelluleFrame` et `GrilleFrame` implémentant les versions graphiques des classes `Cellule` et `Grille` aisément grâce à l'héritage multiple permis par `Python`.

La figure suivante représente une capture d'écran de léinterface graphique du jeu de la vie à travers laquelle on peut voir les différentes fonctionnalités implémentées.

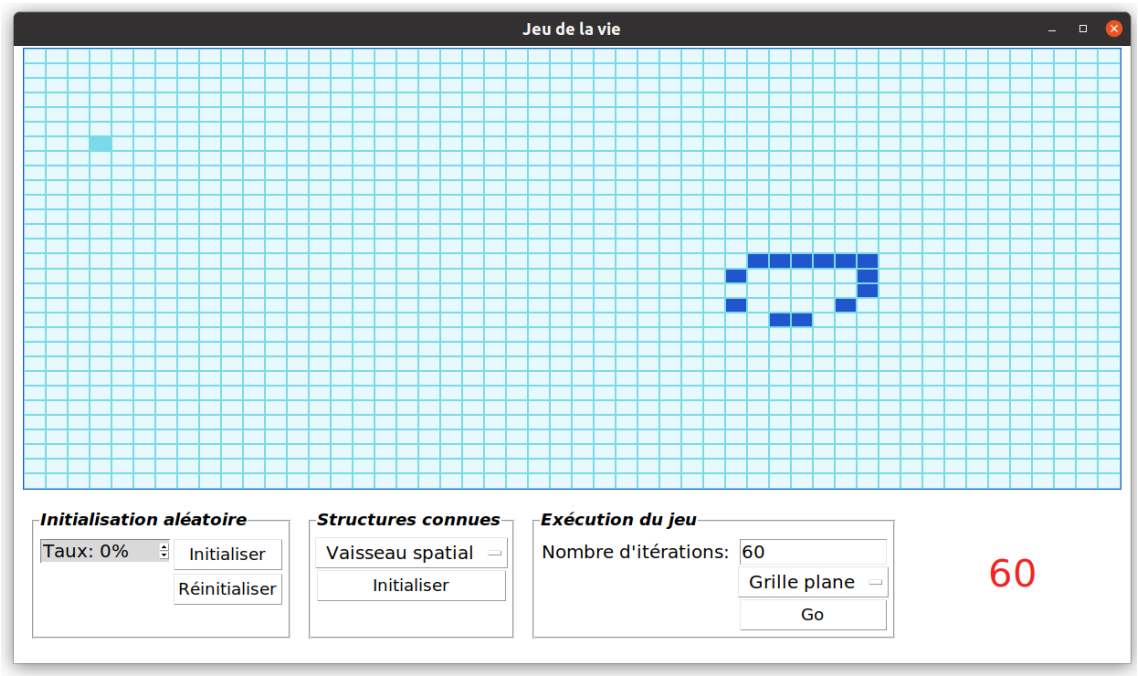

FIGURE 9 – Capture d'écran de l'interface graphique

L'interface graphique réalisée dans ce projet met à la disposition de l'utilisateur un certain nombre de fonctionnalités énumérées ci après :

- **Initialisation aléatoire** : Le menu défilant *Taux* (instance de la classe `ttk.Spinbox`) permet à l'utilisateur de fixer le pourcentage de cellules qui seront vivantes dans la configuration initiale. Une fois le taux fixé, l'utilisateur devra cliquer sur `Initialiser` (instance de la classe `ttk.Button`) appelant ainsi la méthode `alea_init_config()` qui initialise la grille avec le taux choisi. Il est également possible de réinitialiser la grille à travers un click sur le boutton `Réinitialiser` qui fait appel à la méthode `Reset_grid()`.
- **Structures connues** : ici, un menu déroulant (instance de la classe `ttk.OptionMenu`) permet à l'utilisateur de choisir une des neuf structures que nous avons mis à sa disposition. Le bouton `Initialiser` permet ici d'obtenir une grille avec la configuration de base de la structure choisie.
- **Configuration manuelle** : en cliquant sur une cellule quelconque , l'utilisateur peut la rendre vivante ou morte (faisant ainsi appel à la méthode `on_cell_click()` de la classe `GrilleFrame`). Cette fonctionnalité peut par conséquent lui permettre de créer manuellement sa propre configuration initiale.
- **Exécution du jeu** : après avoir crée une configuration initiale à l'aide des fonctionnalités énumérées précédemment, l'utilisateur à la possibilité d'insérer dans le champ de texte nombre d'itérations (instance de la classe `ttk.Entry`) le nombres de génération à simuler. Il peut également choisir le type de grille(torique ou plane) et ensuite cliquer sur le boutton `Go` pour lancer la simulation. La géneration courante est en permanence visible en rouge sur l'écran en bas à droite.

La classe `CelluleFrame` ajoute simplement des informations de couleur de la case et seule la méthode `basculer()` est surchargée.

```
14      def __init__(self: CelluleFrame, master, height, width) -> None:
15          """ Constructeur de la classe CelluleFrame """
16          self.DEATH_CELL_BG = "#E8F9FD"
17          self.LIVING_CELL_BG = "#2155CD"
18          self.HOVER_CELL_BG = "#79DAE8"
19          super().__init__(master=master, height=height, width=width,\
20                           bg=self.DEATH_CELL_BG,\
21                           highlightbackground=self.HOVER_CELL_BG,\
22                           highlightthickness="1")
23          Cellule.__init__(self)
24          self.bg = self.DEATH_CELL_BG
```

FIGURE 10 – Le constructeur de la classe `CelluleFrame`

Pour la classe `GrilleFrame`, c'est essentiellement l'attribut `matrix` qui est modifié car, en plus d'être dessinées et colorées, les cases de la grille doivent pouvoir réagir à l'utilisateur. Cette classe contient aussi trois méthodes correspondant au clic de souris, au survol d'une case ou au contraire lorsque le pointeur quitte une case.

```
15      def __init__(self: GrilleFrame, master, nbLignes: int,\
16                   nbColonnes: int, typeg: str) -> None:
17          """ Constructeur de la classe CelluleFrame """
18          super().__init__(master=master, bg="#E8F9FD",
19                           highlightbackground="#2155CD",
20                           highlightthickness="1")
21          Grille.__init__(self, nbLignes, nbColonnes, typeg)
22          self.HOVER_CELL_BG = "#79DAE8"
23
24          # Initialisation des cellules
25          self.matrix = []
26          for i in range(self.nbLignes):
27              self.grid_rowconfigure(i, weight=1)
28              row = []
29              for j in range(self.nbColonnes):
30                  self.grid_columnconfigure(j, weight=1)
31                  cell = CelluleFrame(master=self,
32                         height=self.winfo_height() // self.nbLignes,
33                         width=self.winfo_width() // self.nbColonnes)
34                  cell.grid(row=i, column=j, sticky="NSEW")
35                  # Associer le click gauche de la souris sur la cellule
36                  # à un changement d'etat de la cellule
37                  cell.bind("<Button-1>", self.on_cell_click)
38                  # Associer le survol de la souris sur la cellule à un
39                  # changement de couleur de fond
40                  cell.bind("<Enter>", self.on_cell_hover)
41                  # Quand on arrete de survoler une cellule, sa couleur
42                  # de fond est rétablie
43                  cell.bind("<Leave>", self.on_cell_leave)
```

FIGURE 11 – Le constructeur de la classe `GrilleFrame`

Le fichier `jeuVie.py` est le fichier principal du programme. C'est lui qui construit la fenêtre d'interface grâce à `tkinter`. L'attribut `gridFrame` est une instance de la classe `GrilleFrame` et les différentes méthodes fonctionnent à peu près toutes sur le même principe : récupérer les données entrées par l'utilisateur et modifier l'attribut `gridFrame` pour prendre en compte les vœux de l'utilisateur. Par exemple la méthode `alea_init_config()` utilise d'abord `self.get_alea_rate()` pour obtenir le taux entré par l'utilisateur puis l'utilise en argument de la méthode `remplir_alea()` sur l'attribut `gridFrame`.

3 Et après ?

Comme déjà mentionné, le manque de temps nous a amené à restreindre nos ambitions et à laisser de côté certaines idées initialement envisagées pour y revenir plus tard, une fois les examens passés par exemple. Parmi les pistes d'amélioration, agrandir la bibliothèque d'exemples (pourquoi pas un canon générateur de nombres premiers ?) et permettre à l'utilisateur de sauvegarder des configurations sous forme d'images. Un peu plus long à implémenter, offrir la possibilité de définir des règles du jeu différentes (en terme d'apparition et de mort des cellules en fonction du voisinage, voire en utilisant un voisinage différent comme celui de Von Neumann) pour étudier des variantes du jeu de la vie. A plus long terme, envisager d'autres variantes avec un nombre d'états des cellules différent (exemple du jeu de la vie quantique mentionné dans la bibliographie), et du point de vue algorithmique réfléchir à un programme de reconnaissance de motifs (permettant par exemple d'identifier des configurations stables ou périodiques dans une grille donnée), ou de recherche d'antécédents (travailler sur les jardins d'Eden) et enfin, essayer de comprendre et d'implémenter une version de l'algorithme Hashlife qui a permis dans les années 1980 de grands progrès sur le sujet grâce à des calculs extrêmement efficaces (mais qui dépasse largement le niveau licence !).

ANNEXE : La feuille de route ”idéale”

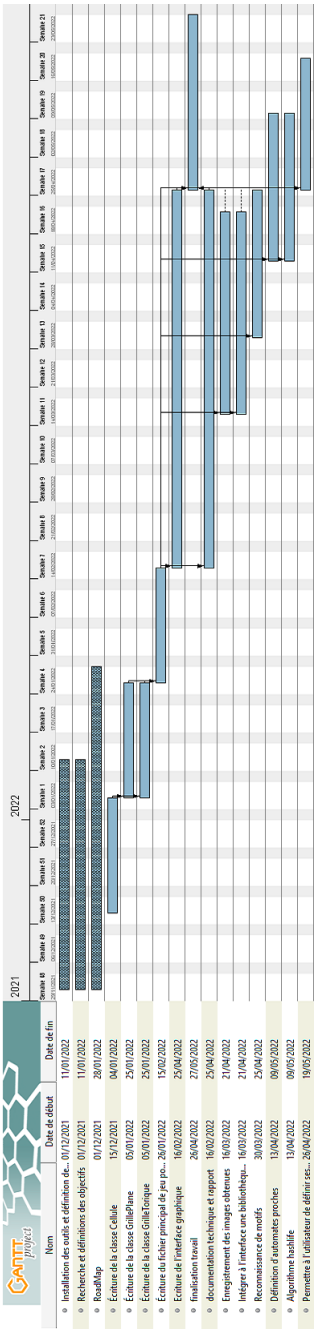

FIGURE 12 – la feuille de route.

Références

[1] Olivier POSTEL-VINAY. L'univers est-il calculateur ? les nouveaux démiurges. *La Recherche*, n° 360 :p. 34–37, janvier 2003.

[2] Mélanie MITCHELL. L'univers est-il calculateur ? quelques raisons de douter. *La Recherche*, n° 360 :p. 38–43, janvier 2003.

[3] Marianne DELORME et Jacques MAZOYER. La riche zoologie des automates cellulaires (consulté le 21/05/2022), 2007. https ://interstices.info/la-riche-zoologie-des-automates-cellulaires/.

[4] Michel BITBOL. Émergence, la théorie qui bouscule la physique - les lois physiques existent-elles ? *La Recherche*, n° 405 :p. 31–36, février 2007.

[5] Nazim FATÈS et Irène MARCOVICI. Automates cellulaires : la complexité dans les règles de l'art. *La Recherche*, n° 538 :p. 67–70, juillet-aout 2018.

[6] Pablo ARRIGHI et Jonathan GRATTAGE. Le monde est un ordinateur quantique. *La Recherche*, n° 467 :p. 72–75, septembre 2012.

[7] Jean-Paul DELAHAYE. Le royaume du jeu de la vie. *Pour la Science*, n° 378 :p. 86–91, avril 2009.

[8] La résurrection du jeu de la vie. https ://www.drgoulu.com/2009/03/29/la-resurrection-du-jeu-de-la-vie/#.YcSnHWjMKUk.

[9] Programme golly. http ://golly.sourceforge.net/ (consulté le 21/05/2022).