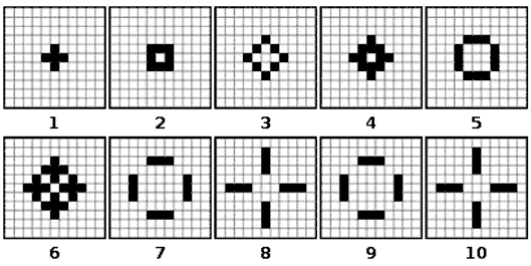


Chapitre 14

Automates cellulaires

14.1. Nouveaux thèmes abordés dans ce chapitre

- animation automatique
- tableaux bidimensionnels
- récursivité

Source :
Wikipédia
fr.wikipedia.org/wiki/Jeu_de_la_vie

14.2. Le jeu de la vie

En préambule, il faut préciser que le jeu de la vie n'est pas vraiment un jeu au sens ludique, puisqu'il ne nécessite aucun joueur ; il s'agit d'un automate cellulaire, un modèle où chaque état conduit mécaniquement à l'état suivant à partir de règles pré-établies.


John **Conway**
(1987-2020)

Le jeu de la vie fut inventé par John Horton **Conway** en 1970, alors qu'il était professeur de mathématiques à l'université de Cambridge, au Royaume-Uni.

Le premier contact que le grand public eut avec ces travaux se fit en 1970 à travers une publication dans *Scientific American* (et sa traduction française *Pour la Science*) dans la rubrique de Martin **Gardner** : « *Mathematical Games* » ou « Récréations mathématiques ».

Gardner écrivait dans ces colonnes que « le Jeu de la Vie rendit Conway rapidement célèbre mais il ouvrit aussi un nouveau champ de recherche mathématique, celui des automates cellulaires. En effet, les analogies du Jeu de la Vie avec le développement, le déclin et les altérations d'une colonie de micro-organismes, le rapprochent des jeux de simulation qui miment les processus de la vie réelle. »

D'après Gardner, Conway expérimenta plusieurs jeux de règles concernant la naissance, la mort et la survie d'une cellule avant d'en choisir un où la population des cellules n'explose pas (ce qui arrive souvent lorsque les conditions de naissances sont moins strictes) mais où des structures intéressantes apparaissent cependant facilement.

Voir les exercices 14.2 et 14.3

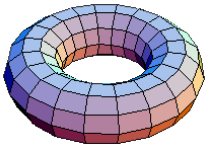
Plusieurs structures intéressantes furent découvertes, comme le « planeur », ou divers « canons » qui génèrent un flux sans fin de planeurs. Ces possibilités augmentèrent l'intérêt pour le jeu de la vie. De plus, arrivant à une époque où une nouvelle génération de mini-ordinateurs meilleur marché fut commercialisée, ce qui permettait de tester des structures pendant la nuit, lorsque personne d'autre ne les utilisait, sa popularité augmenta d'autant.

Vers la fin des années 1980, la puissance des ordinateurs fut suffisante pour permettre la création de programmes de recherche de structures automatiques efficaces ; couplés au développement massif d'Internet, ils conduisirent à un renouveau dans la production de structures intéressantes.

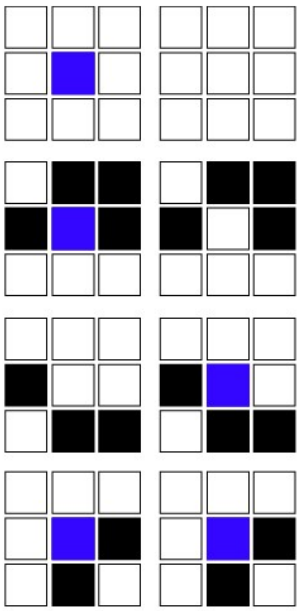
Règles

Le jeu se déroule sur une grille à deux dimensions, théoriquement infinie (mais de longueur et de largeur finies et plus ou moins grandes dans la pratique), dont les cases — qu'on appelle des « cellules », par analogie avec les cellules vivantes — peuvent prendre deux états distincts : « vivantes » ou « mortes ».

Ici, nous travaillerons sur un tore : il n'y aura pas de bords. Sur notre grille, la première colonne sera voisine de la dernière et la première ligne sera voisine de la dernière.



À chaque étape, l'évolution d'une cellule (la case bleue ci-dessous) est entièrement déterminée par l'état de ses huit voisines de la façon suivante (les cellules vivantes sont les cases noires) :

	Si une cellule a moins de deux voisines vivantes, elle mourra à l'étape suivante. Si une cellule a plus de trois voisines vivantes, elle mourra à l'étape suivante. Si une cellule morte a exactement trois cellules vivantes comme voisines, elle renaîtra à l'étape suivante. Si une cellule (morte ou vivante) a exactement deux voisines vivantes, elle restera dans son état actuel à l'étape suivante.

14.3. Code du programme

jeudelavie.py

```
# Jeu de la vie de Conway

from tkinter import *
from random import randrange

haut = 30  # hauteur du tableau
larg = 30  # largeur du tableau
cote = 15  # côté d'une cellule
vivant = 1
mort = 0

# Créer les matrices
cell = [[0 for row in range(haut)] for col in range(larg)]
etat = [[mort for row in range(haut)] for col in range(larg)]
temp = [[mort for row in range(haut)] for col in range(larg)]

# Calculer et dessiner le prochain tableau
```

```
def tableau():
    calculer()
    dessiner()
    fenetre.after(100, tableau)

# Données initiales
def init():
    for y in range(haut):
        for x in range(larg):
            etat[x][y] = mort
            temp[x][y] = mort
            cell[x][y] = canvas.create_rectangle((x*cote, y*cote,
                           (x+1)*cote, (y+1)*cote), outline="gray", fill="white")
    # placer au hasard environ 25% de cellules vivantes
    for i in range(larg*haut//4):
        etat[randrange(larg)][randrange(haut)] = vivant

# Appliquer les 4 règles
def calculer():
    for y in range(haut):
        for x in range(larg):
            # Règle 1 - Mort de solitude
            if etat[x][y] == vivant and nb_voisins < 2:
                temp[x][y] = mort
            # Règle 2 - Toute cellule avec 2 ou 3 voisins survit.
            if etat[x][y] == vivant and (nb_voisins == 2 or nb_voisins == 3):
                temp[x][y] = vivant
            # Règle 3 - Mort par asphyxie
            if etat[x][y] == vivant and nb_voisins > 3:
                temp[x][y] = mort
            # Règle 4 - Naissance
            if etat[x][y] == mort and nb_voisins == 3:
                temp[x][y] = vivant
    for y in range(haut):
        for x in range(larg):
            etat[x][y] = temp[x][y]

# Compter les voisins vivants - tableau torique
def voisins_vivants_tore(a,b):
    nb_voisins = 0
    if etat[(a-1)%larg][(b+1)%haut] == 1:
        nb_voisins += 1
    if etat[a][(b+1)%haut] == 1:
        nb_voisins += 1
    if etat[(a+1)%larg][(b+1)%haut] == 1:
        nb_voisins += 1
    if etat[(a-1)%larg][b] == 1:
        nb_voisins += 1
    if etat[(a+1)%larg][b] == 1:
        nb_voisins += 1
    if etat[(a-1)%larg][(b-1)%haut] == 1:
        nb_voisins += 1
    if etat[a][(b-1)%haut] == 1:
        nb_voisins += 1
    if etat[(a+1)%larg][(b-1)%haut] == 1:
        nb_voisins += 1
    return nb_voisins

# Dessiner toutes les cellules
def dessiner():
    for y in range(haut):
        for x in range(larg):
            if etat[x][y]==0:
                coul = "white"
            else:
                coul = "blue"
            canvas.itemconfig(cell[x][y], fill=coul)

# Lancement du programme
fenetre = Tk()
fenetre.title("Le jeu de la vie de Conway")
canvas = Canvas(fenetre, width=cote*larg, height=cote*haut, highlightthickness=0)
canvas.pack()
init()
```

```
tableau()
fenetre.mainloop()
```

14.4. Analyse du programme

```
from random import randrange
```

La fonction `randrange()` correspond à un tirage au hasard dans la liste d'entiers qui serait générée par la fonction `range` avec les mêmes paramètres. Par exemple `range(4)` génère la liste [0, 1, 2, 3] et `randrange(4)` tire un élément au hasard dans la liste [0, 1, 2, 3].

Attention ! Les méthodes `randrange()` et `randint()` ont un paramétrage différent :
`randrange(8)` = `randint(0,7)`

```
# Créer les matrices
cell = [[0 for row in range(haut)] for col in range(larg)]
etat = [[mort for row in range(haut)] for col in range(larg)]
temp = [[mort for row in range(haut)] for col in range(larg)]
```

Trois matrices (tableaux bidimensionnels) seront utilisées : `cell` mémorisera les carrés qui représentent les cellules, `etat` mémorisera les statuts des cellules (mort ou vivant), `temp` contiendra la nouvelle matrice d'états.

On pourrait se demander si l'on ne pourrait pas déclarer un tableau bidimensionnel de manière plus simple :

```
cell = [[0] * haut] * larg
```

Eh bien non ! Surtout pas ! En faisant cela, Python va créer une liste A contenant des 0 et ensuite il va créer une liste B qui contient elle-même plusieurs fois la liste A. Le problème c'est que c'est la référence de la liste A qui est copiée et les éléments de B correspondent donc en réalité à la même liste (voir § 5.4). Cela a pour effet que lorsque l'on modifie une ligne du tableau, on modifie en réalité toutes les lignes du tableau de la même manière.

Pourquoi faut-il deux matrices pour calculer les changements d'états ? Parce que tous les changements sont simultanés. Si l'on utilisant que la matrice `etat`, les éventuels changements des cellules seront influencés pas les changements des cellules précédentes : l'ordre des changements d'états aurait de l'importance. Avec deux matrices, on n'a plus ce problème : on calcule la nouvelle grille en regardant l'ancienne, sans modifier cette dernière. Une fois que tous les statuts ont été mis à jour, on recopie la matrice `temp` dans la matrice `etat` (voir procédure `calculer()`).

```
# Calculer et dessiner le prochain tableau
def tableau():
    calculer()
    dessiner()
    fenetre.after(100, tableau)
```

Une nouveauté se trouve tout à la fin de la fonction `tableau()` : vous y noterez l'utilisation de la méthode `after()`. Elle déclenche l'appel d'une fonction après qu'un certain laps de temps se soit écoulé. Ainsi par exemple, `fenetre.after(100,qqc)` déclenche pour le widget `fenetre` un appel de la fonction `qqc()` après une pause de 100 millisecondes.

Dans notre script, la fonction qui est appelée par la méthode `after()` est la fonction `tableau()` elle-même. Nous utilisons donc ici pour la première fois une technique de programmation très puissante, que l'on appelle récursivité. Pour faire simple, nous dirons que la récursivité est ce qui se passe lorsqu'une fonction s'appelle elle-même. On obtient bien évidemment ainsi un bouclage, qui peut se perpétuer indéfiniment si l'on ne prévoit pas aussi un moyen pour l'interrompre.

Voyons comment cela fonctionne dans notre exemple.

La fonction `tableau()` est invoquée une première fois à l'avant-dernière ligne du programme. Elle effectue son travail. Ensuite, par l'intermédiaire de la méthode `after()`, elle s'invoque elle-même après une petite pause. Elle repart donc pour un second tour, puis s'invoque elle-même à nouveau, et ainsi de suite, indéfiniment... Nous aurons donc une boucle infinie que l'on ne pourra

stopper qu'en fermant la fenêtre.

La suite du programme ne présente pas de difficultés.

Exercice 14.1

Ajoutez au programme du § 14.3 quatre boutons permettant de :

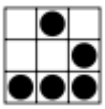
- démarrer la simulation
- stopper la simulation
- suivre l'animation pas à pas (il faudra presser sur ce bouton pour que l'animation passe à l'étape suivante)
- quitter le programme

Indication

Pour stopper ou recommencer la simulation, il faudra utiliser une variable globale `flag` qui vaudra 0 pour arrêter la simulation, 1 pour la démarrer, et 2 pour ne faire qu'une étape.

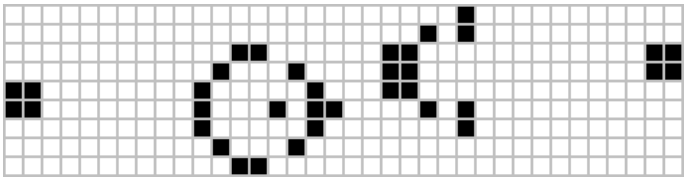
Exercice 14.2

Modifiez le programme de l'exercice 14.1 pour observer le comportement de ce « vaisseau » :



Exercice 14.3

Modifiez le programme de l'exercice 14.1 pour observer le comportement de ce « canon » :



Prenez une grille assez grande (60 x 60 par exemple) et mettez une « marge » de deux carrés autour du canon.

Exercice 14.4

Le petit script ci-dessous vous montre comment détecter un clic sur le bouton gauche de la souris `<Button-1>`, et comment récupérer les coordonnées *x* et *y* du pointeur.

clic.py

```
from tkinter import *

def rond(event):
    x, y = event.x, event.y
    can.create_oval(x-3,y-3,x+3,y+3, fill='red')

fen = Tk()
can = Canvas(fen, width=200, height=200, bg="white")
can.bind("<Button-1>", rond)
can.pack()
fen.mainloop()
```

Modifiez le programme de l'exercice 14.1 pour que l'on puisse placer les cellules vivantes grâce à la souris, en vous inspirant du script ci-dessus. Plus précisément, quand on cliquera sur une case du tableau, on veut qu'elle change d'état : de morte elle deviendra vivante et vice-versa.

Exercice 14.5

Modifiez l'exercice 14.4 pour observer le « *Replicator* ». Les règles de cet automate sont : une cellule naît ou reste vivante si, parmi ses huit cellules voisines, un nombre impair sont vivantes. Autrement, elle ne naît pas, ou meurt.

Afin de bien voir ce qui se passe, prenez un tableau 75 x 75 avec des côtés de cellule de longueur 10. Prévoyez aussi un nouveau bouton pour tout effacer.

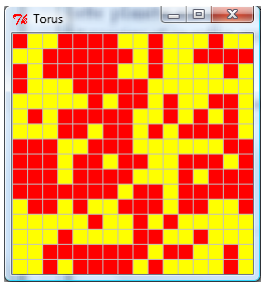
Dessinez un motif au milieu du tableau et lancez la simulation.

14.5. Évolution des opinions sur la planète Torus

Cet automate est décrit dans un article de A. Dewdney : **Le hasard simulé**, Pour la Science - Dossier Hors Série, Avril 1996, pp. 88-90

Sur la planète *Torus*, planète qui, comme son nom l'indique, est en forme de tore, 256 personnes occupe chacune un territoire. Les habitants sont partagés entre deux courants politiques, le parti rouge et le parti jaune. Au début, les opinions politiques sont réparties au hasard sur la planète.

Une particularité des habitants de cette planète est qu'ils sont très influençables. Ainsi, chaque jour, un habitant va trouver un de ses huit voisins et se rallie à ses convictions politiques.



Carte géopolitique de la planète Torus

14.6. Code du programme

torus.py

```
# Evolution des opinions politiques sur la planète Torus

from tkinter import *
from random import randrange

haut = 16  # hauteur du tableau
larg = 16  # largeur du tableau
cote = 15  # côté d'une cellule
rouge = 0
jaune = 1
nb_rouges = haut*larg
nb_jaunes = 0

# Créer les matrices
cell = [[0 for row in range(haut)] for col in range(larg)]
parti = [[rouge for row in range(haut)] for col in range(larg)]

# Dessiner toutes les cellules
def dessiner():
    for y in range(haut):
        for x in range(larg):
            if parti[x][y]==rouge:
                coul = "red"
            else:
                coul = "yellow"
            canvas.itemconfig(cell[x][y], fill=coul)

# Données initiales
def init():
```

```
    global nb_rouges, nb_jaunes
    for y in range(haut):
        for x in range(larg):
            parti[x][y] = rouge   # tout le monde est rouge au départ
            cell[x][y] = canvas.create_rectangle((x*cote, y*cote,
                                (x+1)*cote, (y+1)*cote), outline="gray", fill="red")
    # placer au hasard 50% d'opinions jaunes
    while nb_jaunes<nb_rouges:
        x = randrange(larg)
        y = randrange(haut)
        if parti[x][y] == rouge:
            parti[x][y] = jaune
            nb_jaunes += 1
            nb_rouges -= 1
    dessiner()

# Choisir l'opinion d'un des 8 voisins de la cellule (a,b) - tableau torique
def opinion_voisin(a,b):
    voisin = randrange(8)
    if voisin==1:
        opinion = parti[(a-1)%larg][(b+1)%haut]
    elif voisin==2:
        opinion = parti[a][(b+1)%haut]
    elif voisin==3:
        opinion = parti[(a+1)%larg][(b+1)%haut]
    elif voisin==4:
        opinion = parti[(a-1)%larg][b]
    elif voisin==5:
        opinion = parti[(a+1)%larg][b]
    elif voisin==6:
        opinion = parti[(a-1)%larg][(b-1)%haut]
    elif voisin==7:
        opinion = parti[a][(b-1)%haut]
    else:
        opinion = parti[(a+1)%larg][(b-1)%haut]
    return opinion

# Appliquer la règle
def calculer():
    global nb_rouges, nb_jaunes
    x = randrange(larg)
    y = randrange(haut)
    nouvelle_opinion = opinion_voisin(x,y)
    if parti[x][y] != nouvelle_opinion:
        if nouvelle_opinion == rouge:
            nb_rouges += 1
            nb_jaunes -= 1
        else :
            nb_rouges -= 1
            nb_jaunes += 1
        parti[x][y] = nouvelle_opinion
        if parti[x][y] == rouge:
            coul = "red"
        else:
            coul = "yellow"
        canvas.itemconfig(cell[x][y], fill=coul)

# Calculer et dessiner le prochain tableau
def tableau():
    calculer()
    fenetre.after(1, tableau)

# Lancement du programme
fenetre = Tk()
fenetre.title("Torus")
canvas = Canvas(fenetre, width=cote*larg, height=cote*haut, highlightthickness=0)
canvas.pack()
init()
tableau()
fenetre.mainloop()
```

14.7. Analyse du programme

Ce programme est directement inspiré du programme du § 14.3. Nous allons seulement nous attarder sur la procédure `init()`.

```
def init():
    global nb_rouges, nb_jaunes
    for y in range(haut):
        for x in range(larg):
            parti[x][y] = rouge   # tout le monde est rouge au départ
            cell[x][y] = canvas.create_rectangle((x*cote, y*cote,
                                (x+1)*cote, (y+1)*cote), outline="gray", fill="red")
    # placer au hasard 50% d'opinions jaunes
    while nb_jaunes<nb_rouges:
        x = randrange(larg)
        y = randrange(haut)
        if parti[x][y] == rouge:
            parti[x][y] = jaune
            nb_jaunes += 1
            nb_rouges -= 1
    dessiner()
```

Pour avoir au début de la simulation autant d'habitants rouges que de jaunes, nous avons d'abord attribué la couleur rouge à tout le monde, puis nous avons peint en jaune certains rouges pris au hasard, jusqu'à ce qu'il y ait autant de jaunes que de rouges (à un près si le nombre de cases est impair).

Si l'on n'a vraiment pas de chance, cette méthode peut durer un certain temps... En effet, si l'on veut peindre en jaune un habitant jaune, il ne se passera rien.

Exercice 14.6

Imaginez et programmez une autre manière de faire pour que la population de Torus soit partagée en deux partis politiques avec une répartition moitié rouge moitié jaune.

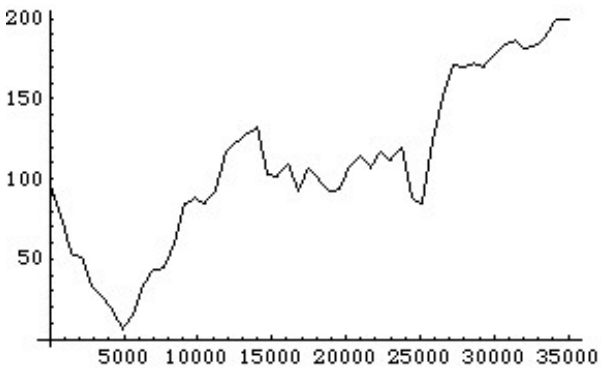
Exercice 14.7

Modifiez le programme du § 14.6 afin de pouvoir dessiner un graphe représentant l'évolution de la population rouge dans le temps. Pour cela, sauvegarder l'évolution au jour le jour dans un fichier.

Vous pourrez ensuite :

- soit utiliser un tableur pour lire ce fichier et faire un graphique
- soit écrire en Python un programme qui fera un graphique à partir des données lues dans ce fichier.

Le graphe devra ressembler à ceci :



On aura en abscisse les jours et en ordonnée la population des rouges.

Exercice 14.8

Faites des expériences pour réfléchir aux questions suivantes. Que se passe-t-il si...

- on change les valeurs de `larg` et `haut` (largeur et hauteur du tableau) ?
- on change le pourcentage de rouges dans l'état initial ?
- on modifie la répartition géographique des couleurs au départ ?
- on prend plus de deux couleurs au départ ?
- on change la fonction de voisinage ?
- on change la forme de la planète (rectangle avec bords, cylindre, ruban de Möbius, ...) ?

Exercice 14.9 - Modèle de Schelling pour la ségrégation urbaine

Dans une ville composée de plusieurs groupes sociaux ou communautés (ethniques, religieuses, économiques...), le modèle de **Schelling** montre comment un comportement de ségrégation spatiale peut apparaître sans être forcément le résultat de comportements individuels délibérément ségrégatifs ou racistes. En effet, il montre que même si tous les individus ont un seuil élevé de tolérance concernant la présence « d'étrangers à leur groupe » dans leur voisinage, on voit néanmoins émerger, avec le temps, une séparation, une ségrégation socio-spatiale, qui se traduit par l'apparition de quartiers beaucoup plus homogènes que la tolérance individuelle ne le laissait supposer.

C'est donc un exemple à la fois simple et significatif de la notion d'émergence dans un système complexe. Même si la réalité est très différente, ce modèle montre quand même, sans forcément tomber dans un fatalisme malsain, que le tout (c'est-à-dire le comportement collectif) ne se déduit pas simplement des règles du comportement individuel.

Thomas C. **Schelling** (1921-2016)

Prix Nobel d'économie en 2005

Description

Les cellules (au nombre de 10'000) représentent chacune une habitation de la ville, elle-même représentée par un damier 100 x 100. La ville est composée de trois communautés notées *A* (en rouge), *B* (en bleu) et *C* (en vert). On a donc trois états possibles pour une cellule habitée : *A*, *B* ou *C*. Lorsqu'une maison n'est pas habitée, la cellule est dans l'état *L* (libre) de couleur grise.

La configuration initiale résulte d'un tirage aléatoire pour chaque cellule d'un état parmi les quatre états possibles (*A*, *B*, *C*, ou *L*) qui donne donc un poids quasi égal entre les communautés.

Les règles de transition sont très simples et au nombre de deux.

- **Règle n°1 :** si une cellule est libre, une famille appartenant à n'importe laquelle des trois communautés, *A*, *B* ou *C* peut s'y installer, avec une chance égale. L'installation d'une famille n'est pas liée à la libération d'une autre cellule, elle est considérée comme venant de l'extérieur. Néanmoins le modèle possède un seuil maximum de 99 % de remplissage de manière à garder un certain dynamisme au système (donc 100 cases libres au minimum).
- **Règle n°2 :** si une famille habitant une cellule donnée est entourée de plus de 70 % d'étrangers à son groupe, alors elle déménage et libère la cellule. Suivant son emplacement sur la grille, une case peut avoir 3, 5 ou 8 voisins. Le déménagement n'est pas lié à un emménagement immédiat dans une autre cellule (on peut donc considérer que la famille va à l'extérieur).

Remarques

Ce modèle est légèrement différent du modèle de Schelling original. En effet, le modèle de Schelling ne gère que deux types d'individus (noirs et blancs), de plus, il procède de manière synchrone (toutes les cellules changent en même temps) et enfin, chaque déménagement est immédiatement suivi d'un relogement ailleurs. Ici, la simulation est asynchrone aléatoire, c'est-à-dire que les cellules changent d'état les unes après les autres par un choix totalement aléatoire et indépendant des tirages précédents.

Travail

Programmez ce modèle.
Étudiez ensuite ce modèle en faisant varier le taux de tolérance d'étrangers autour de chez soi.

Exercice 14.10 - Le gasp

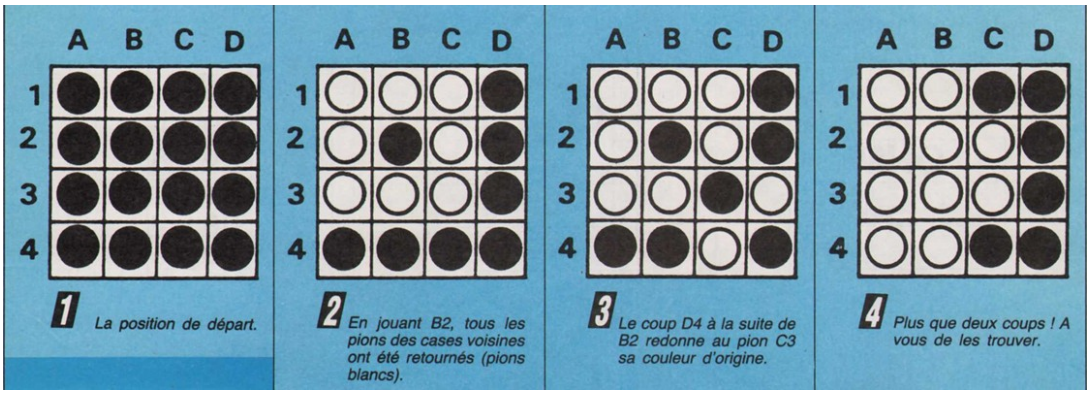
Le Gasp est un casse-tête apparu dans la revue *Jeux & Stratégie* n°38 (avril/mai 1986).

Sur un damier n x n, on place n^2 pions bicolores. Au départ, tous les pions sont noirs. Le but est de retourner ces pions pour qu'ils deviennent tous blancs.

Les cases centrales ont 8 voisines, celles du bord 5 voisines, tandis que celles des coins n'en ont que 3.

Pour retourner des pions, on désigne une case. Ses voisines changent de couleur (mais pas la case désignée).

Le schéma ci-dessous, tiré de *Jeux & Stratégie*, montre un début de partie.



Programmez ce casse-tête en Python. Inspirez-vous de l'exercice 14.4. On veut pouvoir désigner les cases avec le bouton gauche de la souris.

Essayez ensuite de résoudre un Gasp 4 x 4, 6 x 6, 8 x 8, 10 x 10, 12 x 12 ! Et pourquoi ne pas écrire un programme qui cherche ces solutions ?

Une solution possible pour un Gasp 4 x 4.
L'ordre des coups n'a pas d'importance.

14.8. Ce que vous avez appris dans ce chapitre

- Vous avez appris à créer un tableau bidimensionnel et à l'utiliser. Attention à la façon dont vous déclarez ce tableau !
- Vous avez vu pour la première fois une fonction récursive (§ 14.4). Nous nous attarderons un peu plus sur ce sujet au chapitre suivant. La récursivité, associée à la méthode `after()`, permet de créer une animation automatique.
- Bien que mus par des règles très simples, les automates cellulaires peuvent avoir des comportements complexes, et même plus ou moins modéliser la vie réelle : on peut les utiliser pour simuler un feu de forêt ou la formation d'un tas de sable, par exemple.
- Vous avez vu comment utiliser le bouton droit de la souris et récupérer les coordonnées de l'emplacement du curseur.