"Tricher au jeu sans gagner est d'un sot." (*Voltaire*)

AGRANE Adam, RICHARD Balthazar

PRÉSENTATION DU PROJET

PRÉSENTATION GÉNÉRALE

Le jeu du **Lights Out**, ou "éteignez tout", est un jeu de réflexion qui a été commercialisé en 1995. Une grille de départ est générée et présentée au joueur qui devra éteindre toutes les cases. Cependant, l'appui d'une case entraîne son changement d'état (elle passe d'allumée à éteinte ou inversement), ainsi que le changement d'état de celles qui lui sont adjacentes (les cases situées à droite, à gauche, en haut et en bas, si elles existent...)

Notre projet est de recréer le jeu du **lights out** avec une interface graphique et de proposer une aide en indiquant la solution. Nous utilisons pour cela le langage de programmation Python . L'objectif est donc de faire un jeu avec le moins de bugs possible et 100% fonctionnel, bien que minimaliste.

Le projet nous est, dès le début, paru comme une évidence puisqu'il mixe nos spécialités (et même nos options) ! En effet, nous avons réussi à combiner les mathématiques de spécialité avec du dénombrement mais aussi des mathématiques "expertes" avec l'utilisation de matrices et leur inversion. Bien entendu, nous utilisons les connaissances de NSI pour appliquer la partie mathématiques au numérique et créer une interface graphique avec ***matplotlib***.

Les intérêts sont donc multiples puisque ce projet nous permet d'améliorer la cohésion à travers le travail d'équipe, de nous améliorer en programmation en découvrant de nouveaux modules tels que ***widgets*** ou ***ginput*** et en consultant leur documentation. Enfin, nous abordons quelques notions mathématiques du supérieur telles que les vecteurs du noyau d'une matrice.

ORGANISATION DU TRAVAIL

Vous vous demandez sûrement qui sont les deux membres de ce fabuleux binôme ? Nous allons vous répondre !

Nous sommes deux élèves du lycée Turgot : Adam AGRANE et Balthazar RICHARD.

Pour nous organiser, mon camarade Adam s'est chargé de l'interface graphique tandis que je travaillais sur la partie mathématique de la résolution, bien qu'il nous soit arrivé de nous aider mutuellement en échangeant les rôles quand l'inspiration venait à manquer.

Pour travailler en équipe, nous nous sommes vus au lycée et avons échangé sur les réseaux sociaux pendant les vacances afin de planifier le travail. Chaque weekend, nous avons consacré entre 2 et 4 heures au projet et durant les vacances, nous avons cumulé pas moins de 30 heures. Sans compter les heures en cours de NSI... Pour nous entraider, nous nous sommes envoyé des parties du code par mails.

La conception du projet s'est faite peu avant les vacances de février et s'est finie en mars.

LES ÉTAPES DU PROJET

Pour le réaliser, nous avons dû sectionner le projet en deux grands axes et deux sous-axes :

- L'interface graphique
- La résolution :
  - Les grilles "3×3"
  - Les grilles "4×4"

C'est sans conteste l'interface graphique qui a été la plus longue à façonner puisque nous avons rencontré pas mal de problèmes, que nous nous sommes efforcés de corriger ou de contourner.

Au final, le projet nous a pris pas moins d'une bonne centaine d'heures de travail, au lycée, chez nous et même dans le train !

FONCTIONNEMENT ET OPÉRATIONNALITÉ

 Il existe un problème dès lors que l'on utilise les modules ***ginput*** et ***widgets*** ensemble. Un clic sur le bouton **Solve** entraînait un clic sur la grille ce qui empêchait de jouer. Pour y remédier, nous avons donc décidé de jouer avec les deux clics : le droit pour la grille et le gauche pour les boutons.

Le message suivant permet de l'indiquer au joueur :

" ***Clic droit pour les cases*** "

Un autre problème est aussi survenu lorsqu'on quittait une partie pendant le "chargement" de la suivante (phase où s'affiche la grille vide avec le titre "Gagné ! ! !") car cela interrompait le processus du ***plt.pause()*** et relançait donc une autre fenêtre du jeu. Pour le résoudre, nous avons donc affiché le message suivant :

" ***Ne quittez pas maintenant*** "

 D'autre part, la taille des cercles étant fixée, ils rétrécissent dès lors que l'on change la taille de la fenêtre. On doit donc jouer dans la petite fenêtre pour profiter au mieux des graphismes.

 Enfin, on ne peut appuyer deux fois d'affilée sur un widget, ce qui implique que l'on ne peut pas afficher puis enlever la solution sans un clic sur une case entre temps.

Outre cette spécificité, le programme est donc aujourd'hui fini et fonctionne tout à fait normalement. Il est jouable et tous les boutons, bien que peu nombreux, réagissent correctement. Par exemple, un appui sur le bouton **Solve** alors que les solutions sont déjà affichées les effacera. Les solutions sont aussi calculables à tout moment du jeu, ce qui permet de gagner n'importe quand.

Pour tester son bon fonctionnement, le programme a été testé sur plusieurs ordinateurs avec des systèmes d'exploitation différents : **Windows**, **Ubuntu** et **Chrome OS**, et différents IDE tels que ***Spyder***, ***VS Code*** ou encore le ***Terminal***.

IDÉES D'AMÉLIORATIONS

Si nous devions l'améliorer, nous ajouterions peut-être d'autres tailles de grilles et nous referions de plus jolis graphismes (pourquoi pas des ampoules). Nous aurions aussi pu utiliser le module ***tkinter***, qui possède une meilleure intégration des widgets et aurait pu nous éviter certains problèmes graphiques, mais il ne rentre pas dans le cadre du programme scolaire.

Pour toucher un large public, nous pourrions organiser un concours de réflexion au sein de notre lycée (en enlevant l'accès aux solutions) et faire ainsi s'affronter des élèves sur une épreuve de temps. Le plus rapide remporterait un titre honorifique et notre projet serait popularisé, devenant - pourquoi pas - le nouveau jeu à la mode en salle des professeurs.

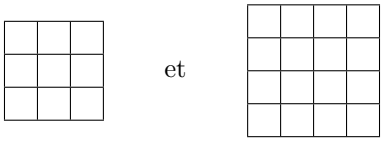
D'un point de vue critique, l'organisation et les fonctionnalités du projet nous satisfont amplement car elles remplissent le cahier des charges que nous nous sommes fixé. Par conséquent, le seul point qui devrait être amélioré, selon nous, est l'adaptabilité des cases du jeu en fonction de la taille de la fenêtre.

Ce projet est une très belle expérience qui, si elle était à refaire, le serait assurément !

DOCUMENTATION TECHNIQUE

Table des matières

Indication aux lecteurs : Dans tout ce document, nous parlons exclusivement de grilles **carrées**.



1 L'interface graphique

1.1 Explication théorique

Pour créer une interface graphique, on utilise la bibliothèque **matplotlib** et son module ***pyplot***. Celui-ci nous permet de former une grille que l'on remplit, de cases jaunes et grises, aléatoirement. La fonction ***ginput*** (de la même bibliothèque) nous permet d'obtenir les coordonnées du clic de la souris qu'il faut ensuite transformer en numéro de case afin de modifier celles concernées.

Deux boutons : **3×3** et **4×4** permettent de sélectionner la taille de la grille du jeu. Le bouton **Solve** permet de marquer d'une étoile rouge les cases à toucher afin de gagner et le bouton **Play !** permet de lancer une partie.

Une grille 3×3 est donc représentée par un vecteur colonne :

1	2	3
4	5	6
7	8	9

peut aussi s'écrire $\begin{pmatrix} case_1 \\ case_2 \\ case_3 \\ case_4 \\ case_5 \\ case_6 \\ case_7 \\ case_8 \\ case_9 \end{pmatrix}$

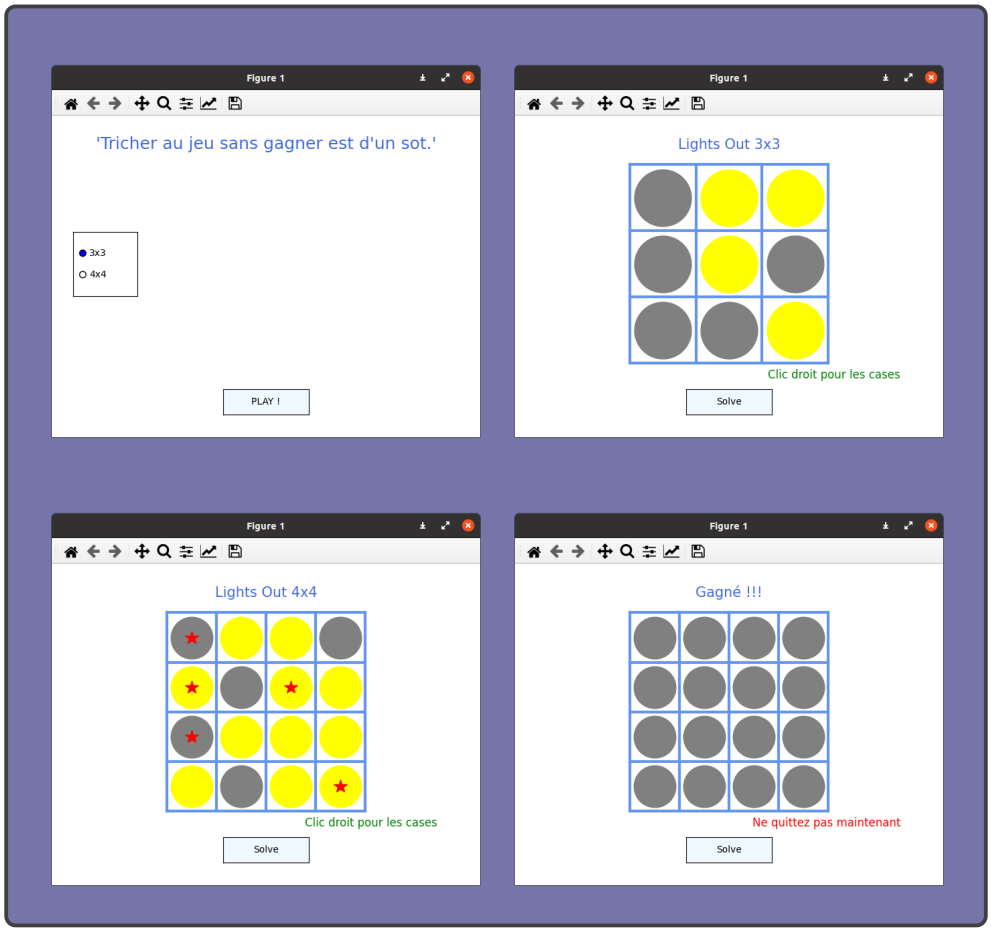
Pour générer une grille de taille 3×3, on place aléatoirement des 0 et des 1 dans une matrice en veillant toutefois à ce qu'elle ne soit pas nulle. En effet, toutes les grilles 3×3 sont solubles, contrairement aux grilles 4×4 qui présentent une difficulté supplémentaire. D'après un article [1], seulement $\frac{1}{16}$ de grilles 4×4 et $\frac{1}{4}$ de grilles 5×5 sont solubles (cela se démontre avec le calcul du déterminant [2]).

Il faut donc être certain que la grille proposée au joueur est bien soluble.

Notre méthode consiste à calculer le produit scalaire des 4 vecteurs du noyau [3] de la matrice résolution et du vecteur représentant la grille de jeu. Ainsi, si les quatre sont congrus à 0 [2], la grille est soluble.

1. Information tirée de la thèse *Lights Out : Determining solvability on rectangular boards* de Tamar Elise Wilson datant de 2009.
2. Voir l'**annexe 4.1 L'algorithme du calcul du déterminant**.
3. Information tirée de la thèse *The game of Lights out* de Rebecca S. Meyer datant de 2013.

1.2 Quelques photos du jeu



1. Page d'accueil du jeu.
2. Partie de jeu avec une grille 3×3.
3. Partie de jeu avec une grille 4×4 et l'aide.
4. Affichage lors d'une victoire.

2 Le cas des grilles "3×3"

2.1 Explication théorique

Pour trouver les solutions d'une grille de taille 3, il faut s'aider de matrices. Le principe est le suivant : nous devons résoudre une équation de la forme $AX = B$, avec les matrices :

$$A = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix} \quad X = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{pmatrix} \quad B = \begin{pmatrix} case_1 \\ case_2 \\ case_3 \\ case_4 \\ case_5 \\ case_6 \\ case_7 \\ case_8 \\ case_9 \end{pmatrix}$$

Pour créer la matrice résolution, on commence par simuler un appui sur chaque case d'une grille vide de taille 3. Une case affectée par un appui prend la valeur 1. On met ensuite toutes ces grilles sous forme de matrices colonnes. Enfin, on juxtapose les matrices colonnes obtenues pour former une matrice carré d'ordre 9 : c'est la matrice A.

On peut donc modéliser la matrice résolution avec un code couleur : en rouge les cases touchées et en orange les cases affectées par un appui (chaque colonne représentant le numéro de la case) :

Exemple d'un appui sur la 1ère case :

$$A = \left(\begin{array}{ccccccccc} 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{array}\right|$$

×	×	0
×	0	0
0	0	0

$$appui_1 = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

X représente les cases (inconnues) à toucher afin de résoudre l'équation, et donc le jeu et B représente les cases de la grille de départ du jeu.

Il faut donc inverser la matrice A et la multiplier par B pour trouver les cases permettant de gagner, soit résoudre : $X = A^{-1}B$ en [2] car on travaille en binaire. En effet, un appui fait passer de 1 à 0 et inversement, et deux appuis sur la même case ne produisent aucun changement. On travaille donc avec les règles suivantes :

$$0 + 0 = 0 \qquad 0 + 1 = 1 \qquad 1 + 1 = 0.$$

2.2 L'algorithme du pivot de Gauss

On utilisera la fonction ***pivot*** qui est la version numérique du pivot de Gauss pour inverser la matrice résolution.

Pour la réaliser, on commence par créer 3 fonctions essentielles, puis on exécute le tout dans une autre fonction nommée ***resolution*** :

```
def echange_ligne(A, i, j):
    '''Fonction qui prend en argument une matrice, une ligne et une colonne et
       qui échange la case_i,j et la case_j,i.'''
    A[i], A[j] = A[j], A[i]
    return A

def substitutions_lignes(A, i, j, mu):
    '''Fonction qui prend en argument une matrice, une ligne, une colonne et
       une constante et qui effectue les substitutions.'''
    for z in range (len(A[i])):
        A[i][z] = round(A[i][z] + mu*A[j][z])%2
    return A

def recherche_pivot(A, i):
    '''Fonction qui prend en argument une matrice, une ligne et qui cherche le
       coefficient avec la valeur absolue la plus grande.'''
    liste = list()
    for z in range(i, len(A)):
        liste.append([abs(A[z][i]), z])
    liste = max(liste)
    j = liste[1]
    return j

def resolution(Y0, A0):   # Mise sous forme de matrice colonne
    '''Fonction qui prend en argument deux matrice, inverse la seconde puis les
       multiplie. On utilise pour ça le pivot de Gauss (même si nous somme dans
       le corps de Galois GF(2)).'''
    if len(Y0) > 1:   # Permet de passer d'une liste de listes à une liste
        y0 = list()
        for i in Y0:
            y0 += i
        Y0 = y0

    I = [[0 for k in range(len(A0))] for k in range(len(A0))]   # Matrice unité
    for i in range(len(I)):
        I[i][i] = 1
    indexe, p = list(), list()

    for i in range(len(A0)):   # Échange des lignes
        P = recherche_pivot(A0, i)
        A0 = echange_ligne(A0, i, P)
        I = echange_ligne(I, i, P)
```

Suite du code :

```
    for i in range(len(A0)):   # Indexe des substitutions à effectuer
        for j in range(len(A0)-1):
            v=[k for k in range(len(A0))]
            del v[i]
            l = [v[j], i]
            indexe.append(l)

    for i in indexe:   # Substitutions
        p = A0[i[1]][i[1]]
        mu = -((A0[i[0]][i[1]])/p)
        A0 = substitutions_lignes(A0, i[0], i[1], mu)
        I = substitutions_lignes(I, i[0], i[1], mu)
    for i in range(len(A0)):
        mu = (1/A0[i][i])
        for j in range(len(A0)):
            A0[i][j] = round(A0[i][j]*mu)
            I[i][j] = round(I[i][j]*mu)

    m = len(I)   # Produit A^-1 x B :
    n2 = len(I)
    p = 1
    X = list()
    for i in range(m):
        for j in range(p):
            S = 0
            for k in range(n2):
                S += (I[i][k] * Y0[k])
            X.append(round(S)%2)
    liste_cases = list()
    x1 = X[:3]
    x2 = X[3:6]
    x3 = X[6:9]
    for i in range(3):
        if x1[i] == 1:
            liste_cases.append([0, i])
        if x2[i] == 1:
            liste_cases.append([1, i])
        if x3[i] == 1:
            liste_cases.append([2, i])
    liste_cases.sort()
    return liste_cases
```

La sortie du programme nous retourne la matrice A^{-1} dans le corps de Galois $GF(2)$ [4], soit en binaire.

4. Information tirée de la thèse *Lights Out : Determining solvability on rectangular boards* de Tamar Elise Wilson datant de 2009.

3 Le cas des grilles "4×4"

3.1 Explication théorique

Pour ce qui est des grilles de taille 4 ou 5, la matrice résolution n'est pas inversible [5], ce qui implique que toutes les configurations n'ont pas forcément de solutions et l'on doit donc chercher un autre moyen de les trouver.

La méthode pour résoudre les grilles de taille 4 que nous avons mise en place est la méthode par **force-brute**. Cela consiste à tester toutes les possibilités en touchant une case, puis deux et ce jusqu'à n s'il le faut (pour une grille de taille n).
Au maximum, on cherche la somme des coefficients binomiaux : $\sum_{p=1}^{n} \binom{n}{p}$.

Cependant, la méthode est plutôt coûteuse en ressources car le nombre de parties d'un ensemble E à n éléments est égal à $card\,(\mathscr{P}(E)) = 2^n$. Plus exactement, nous devons tester $2^n - 1$ combinaisons dans le pire des cas car nous ne commençons pas à 0 mais à 1 appui (on ne prend pas en compte l'ensemble vide $\emptyset$).

Pour une grille de taille 4, cela fait $2^{16} - 1 = 65535$ combinaisons, ce qui est encore calculable par **Python**. Mais pour une grille de taille 5, cela en fait $2^{25} - 1 = 33554431$ ce qui est bien trop conséquent.

La complexité est donc au moins exponentielle, c'est un $\mathcal{O}(2^n)$.

Pour ce faire, une fonction ***appui*** simule le toucher de chaque case d'une grille. Deux fonctions, ***brute_force*** et ***compilation*** (qui est récursive), relancent le programme tant qu'une solution n'a pas été trouvée.

L'algorithme va donc essayer de résoudre le jeu par un appui sur chaque case, puis deux, et ceux jusqu'à 16 appuis au maximum (pour une grille de taille 4).

Ainsi à chaque itération i, le programme renvoie $\binom{16}{i}$ grilles représentant i appuis simulés sur la grille de départ.
Cela est du au fait qu'une des propriétés du **Lights Out** stipule que l'ordre des cases touchées n'importe pas. Si l'on appui sur les cases 1 puis 2 par exemple, alors il n'est pas nécessaire de générer une grille ou l'on appui sur les cases 2 puis 1 car cela engendrerait un doublon.

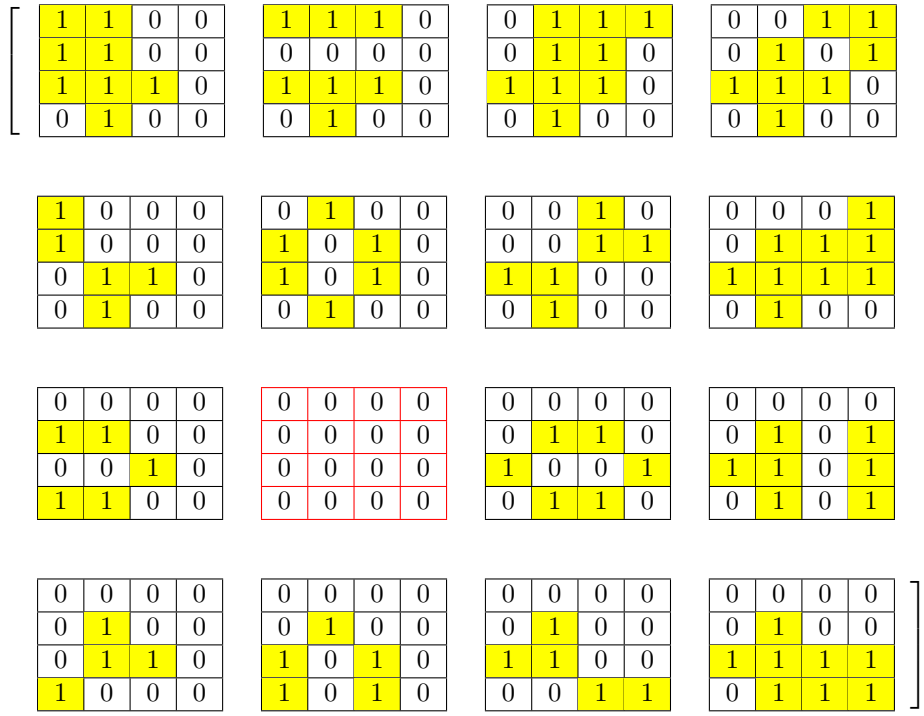
Pour mieux comprendre, considérons la grille de départ suivante :

0	0	0	0
0	1	0	0
1	1	1	0
0	1	0	0

et simulons un appui sur chaque case, soit l'exécution du programme à l'itération $i = 1$.

5. Voir l'**annexe 4.1 L'algorithme du calcul du déterminant**.

La sortie sera donc une liste contenant les $\binom{16}{1} = 16$ grilles :



(On remarque qu'on à ici une solution puisqu'un appui sur la $10^{\text{ème}}$ case entraîne la résolution du jeu).

4 Annexes

4.1 L'algorithme du calcul du déterminant

Ci-dessous, les codes des algorithmes qui calculent le déterminant d'une matrice carrée d'ordre $n \times n$:

```
def Aij(A, i, j):
    '''Fonction qui prend en entrée une matrice A, une ligne i et une
       colonne j et qui supprime la ligne et la colonne.'''
    B = []
    for z in range(len(A)):
        B.append(A[z][:])
    del B[i]
    for colone in B:
        del colone[j]
    return B
```

```
def det(A):
    '''Fonction qui prend en entrée une matrice A, la réduit jusqu'à
       obtenir une matrice carrée d'ordre 2 × 2 dont le déterminant
       est facilement calculable.'''
    n = len(A)
    S = 0
    if n == 2:
        S += A[0][0]*A[1][1] - A[0][1]*A[1][0]
    else :
        for j in range(n):
            if  A[0][j] != 0 :
                S += A[0][j] * (-1)**j * det(Aij(A, 0, j))
    return S
```

4.2 L'algorithme du calcul du produit scalaire

Ci-dessous, le code de l'algorithme permettant de calculer le produit scalaire de la grille 4×4 avec les vecteurs du noyau de la matrice résolution :

```
vecteurs_noyaux = \
        [[0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0], \
         [1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0], \
         [1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0], \
         [1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1]]

def produit_scalaire(vecteurs_noyaux, matrice):
    '''Fonction qui prend en entrée une grille de départ , mise sous forme de
       vecteur, et les 4 vecteurs du noyau de la matrice résolution 4 × 4 et qui
       effectue le produit scalaire de chaque vecteurs avec la grille de départ.'''
    A = list ()
    for i in range (4):
        A += matrice[i]
    s = 0
    validite = True
    for vecteurs in vecteurs_noyaux :
        for lignes in range (16):
            s += (vecteurs[lignes] * A[lignes])
        if s%2 != 0:
            validite = False
    return validite
```

Les vecteurs du noyau [6] de la matrice résolution de taille 4×4 sont :

$$\begin{pmatrix}0\\1\\1\\1\\1\\0\\1\\0\\1\\1\\0\\0\\1\\0\\0\\0\end{pmatrix} \quad \begin{pmatrix}1\\1\\0\\1\\0\\0\\0\\1\\1\\1\\1\\0\\0\\1\\0\\0\\0\end{pmatrix} \quad \begin{pmatrix}1\\0\\1\\1\\1\\0\\0\\0\\0\\1\\1\\1\\0\\0\\1\\0\\0\end{pmatrix} \quad \begin{pmatrix}1\\1\\1\\0\\0\\1\\0\\1\\0\\0\\0\\1\\1\\0\\0\\0\\1\end{pmatrix}$$

la matrice résolution des grilles 4×4 étant :

$$A_4 = \begin{pmatrix}
1&1&0&0&1&0&0&0&0&0&0&0&0&0&0&0\\
1&1&1&0&0&1&0&0&0&0&0&0&0&0&0&0\\
0&1&1&1&0&0&1&0&0&0&0&0&0&0&0&0\\
0&0&1&1&0&0&0&1&0&0&0&0&0&0&0&0\\
1&0&0&0&1&1&0&0&1&0&0&0&0&0&0&0\\
0&1&0&0&1&1&1&0&0&1&0&0&0&0&0&0\\
0&0&1&0&0&1&1&1&0&0&1&0&0&0&0&0\\
0&0&0&1&0&0&1&1&0&0&0&1&0&0&0&0\\
0&0&0&0&1&0&0&0&1&1&0&0&1&0&0&0\\
0&0&0&0&0&1&0&0&1&1&1&0&0&1&0&0\\
0&0&0&0&0&0&1&0&0&1&1&1&0&0&1&0\\
0&0&0&0&0&0&0&1&0&0&1&1&0&0&0&1\\
0&0&0&0&0&0&0&0&1&0&0&0&1&1&0&0\\
0&0&0&0&0&0&0&0&0&1&0&0&1&1&1&0\\
0&0&0&0&0&0&0&0&0&0&1&0&0&1&1&1\\
0&0&0&0&0&0&0&0&0&0&0&1&0&0&1&1
\end{pmatrix}$$

6. Information tirée de la thèse *The game of Lights out* de Rebecca S. Meyer datant de 2013.