Bachelor en sciences et ingénierie, filière mathématiques

Mathématiques expérimentales

Semestre d'été 2020

Knights

Auteurs :
Léa Micard
Eva Ragazzini
Lara Suys

Superviseurs :
Prof. Dr. Gabor Wiese
Guendalina Palmirotta

Table des matières

Introduction **3**

1 Premières observations **4**

2 Adaptation aux matrices **8**
2.1 Cavalier classique . . . 8
2.1.1 Exemple particulier et résolution . . . 8
2.2 Généralisation . . . 9

3 Algorithmes **11**
3.1 Introduction aux graphes . . . 11
3.2 Algorithme de parcours en profondeur . . . 11
3.2.1 Concept . . . 11
3.2.2 Application . . . 12
3.3 Algorithme de parcours en largeur . . . 13
3.3.1 Concept . . . 13
3.3.2 Application . . . 14

4 Idées folles **16**
4.1 En trois dimensions . . . 16
4.1.1 Cas général . . . 16
4.1.2 Exemple particulier . . . 17
4.1.3 Relation entre la dimension et le nombre de mouvements possibles . . . 19
4.2 Cavalier snake . . . 21
4.2.1 Déplacement d'un cavalier sur un tore . . . 23
4.3 Plateaux spéciaux . . . 24
4.3.1 Plateau troué . . . 24
4.3.2 Plateau coeur . . . 26

Conclusion **27**

A Animation **28**

Bibliographie **31**

Introduction

Parmi les jeux mathématiques inspirés des échecs, on retrouve le tour du cavalier. Il s'agit d'un problème où un cavalier sur un plateau doit visiter toutes les cases du plateau sans passer deux fois par la même. Ici, nous nous intéressons à une variation de ce problème : trouver le chemin le plus court d'une case à une autre pour le cavalier.

Nos objectifs sont de programmer différentes variations du jeu d'échec : sur des plateaux arbitraires, dans d'autres dimensions, avec des cavaliers spéciaux ou même sur d'autres formes géométriques comme le tore.

Nous avons choisi ce sujet d'abord par intérêt pour les échecs mais aussi pour étudier les mathématiques sous un aspect ludique et créatif.

1 Premières observations

Dans un premier temps, nous avons réalisé des observations sur les déplacements d'un *cavalier classique* sur des plateaux carrés de taille $n \times n$, $n \in \{3, ..., 8\}$ en partant de la première case en bas à gauche. Un *cavalier classique* est un cavalier pouvant réaliser les mouvements suivants :

— 1 case vers la droite, 2 vers le haut,
— 1 case vers la gauche, 2 vers le haut,
— 1 case vers la droite, 2 vers le bas,
— 1 case vers la gauche, 2 vers le bas,
— 1 case vers le haut, 2 vers la droite,
— 1 case vers le bas, 2 vers la droite,
— 1 case vers le haut, 2 vers la gauche,
— 1 case vers le bas, 2 vers la gauche.

Nous illustrons différents mouvements sur un plateau 8×8 pour donner des exemples.

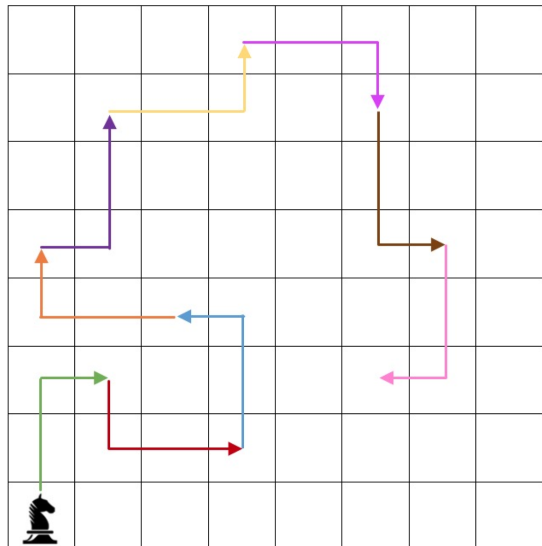


Figure 1 – Représentation de différents mouvements sur un plateau 8×8.

Deux cases vers le haut, une case vers la gauche.
Deux cases vers le haut, une case vers la droite.
Deux cases vers la droite, une case vers le bas.
Une case vers la droite, deux cases vers le haut.
Deux cases vers la gauche, une case vers le haut.
Deux cases vers le bas, une case vers la droite.
Deux cases vers la gauche, une case vers le haut.
Deux cases vers le bas, une case vers la gauche.
Une case vers le bas, deux cases vers la droite.

Figure 2 – Plateau 3×3.

5	2	3	2
2	1	4	3
3	4	1	2
♞	3	2	5

Figure 3 – Plateau 4×4.

2	3	2	3	4
3	2	3	2	3
2	1	4	3	2
3	4	1	2	3
♞	3	2	3	2

Figure 4 – Plateau 5×5.

3	4	3	4	3	4
2	3	2	3	4	3
3	2	3	2	3	4
2	1	4	3	2	3
3	4	1	2	3	4
♞	3	2	3	2	3

Figure 5 – Plateau 6×6.

4	3	4	3	4	5	4
3	4	3	4	3	4	5
2	3	2	3	4	3	4
3	2	3	2	3	4	3
2	1	4	3	2	3	4
3	4	1	2	3	4	3
♞	3	2	3	2	3	4

Figure 6 – Plateau 7×7.

5	4	5	4	5	4	5	6
4	3	4	3	4	5	4	5
3	4	3	4	3	4	5	4
2	3	2	3	4	3	4	5
3	2	3	2	3	4	3	4
2	1	4	3	2	3	4	5
3	4	1	2	3	4	3	4
♞	3	2	3	2	3	4	5

Figure 7 – Plateau 8×8.

Légende

On part du principe qu'on ne repasse jamais par une case visitée auparavant.

♞ représente la position de départ du cavalier.

1 représentent les endroits où le cavalier peut se rendre suite aux déplacements énumérés ci-dessus en partant de sa position de départ.

2 représentent les endroits où le cavalier peut se rendre suite aux déplacements énumérés ci-dessus en partant de toutes les cases 1 .

3 représentent les endroits où le cavalier peut se rendre suite aux déplacements énumérés ci-dessus en partant de toutes les cases 2 .

4 représentent les endroits où le cavalier peut se rendre suite aux déplacements énumérés ci-dessus en partant de toutes les cases 3 .

5 représentent les endroits où le cavalier peut se rendre suite aux déplacements énumérés ci-dessus en partant de toutes les cases 4 .

6 représentent les endroits où le cavalier peut se rendre suite aux déplacements énumérés ci-dessus en partant de toutes les cases 5 .

Nous pouvons à partir de là faire plusieurs observations :

- Nous observons que dès que la dimension est supérieure à 3, nous pouvons atteindre toutes les cases du plateau. Il n'y a qu'en dimension 3 qu'une case est impossible à atteindre, celle du milieu. Si nous partions de la case du milieu, nous ne pourrions atteindre aucune autre case.
- Nous observons qu'en dimension 4, le nombre maximal de mouvements pour atteindre une case est 5 mouvements.
- Nous observons qu'à partir de la dimension 5, le nombre maximal de mouvements pour atteindre une case est inférieur à la dimension du plateau. Par exemple, sur un plateau 8×8, il faut au maximum 6 mouvements et $6 \leq 8$.

2 Adaptation aux matrices

2.1 Cavalier classique

Nous considérons, ici, un cavalier classique. Les couples de mouvements qui lui sont donc possibles peuvent être représentés sous forme de vecteurs, en considérant que l'échiquier est un repère orthonormé. Les mouvements possibles sont les suivants :

$$\begin{pmatrix}1\\2\end{pmatrix},\begin{pmatrix}1\\-2\end{pmatrix},\begin{pmatrix}-1\\2\end{pmatrix},\begin{pmatrix}-1\\-2\end{pmatrix},\begin{pmatrix}2\\1\end{pmatrix},\begin{pmatrix}2\\-1\end{pmatrix},\begin{pmatrix}-2\\1\end{pmatrix},\begin{pmatrix}-2\\-1\end{pmatrix}.$$

On note par $\begin{pmatrix}x\\y\end{pmatrix}$ la position de départ du cavalier et par $\begin{pmatrix}x'\\y'\end{pmatrix}$ sa position d'arrivée, avec $x, x', y, y' \in \mathbb{N}$. Pour atteindre la position d'arrivée, il faudra utiliser une certaine combinaison des vecteurs possibles. On note a, b, c, d, e, f, g et h les nombres entiers non-nuls de mouvements de chaque type nécessaires (a-mouvements (1, 2), b-mouvements (1, -2), etc.). En résolvant un système de type AX = B, avec :

$$A = \begin{pmatrix}1 & 1 & -1 & -1 & 2 & 2 & -2 & -2\\ 2 & -2 & 2 & -2 & 1 & -1 & 1 & -1\end{pmatrix},\ X = \begin{pmatrix}a\\b\\c\\d\\e\\f\\g\\h\end{pmatrix}\ et\ B = \begin{pmatrix}x'-x\\y'-y\end{pmatrix}.$$

Cela correspond à réduire, grâce à la méthode du pivot de Gauss, une matrice de type :

$$\left(\begin{array}{cccccccc|c}1 & 1 & -1 & -1 & 2 & 2 & -2 & -2 & x'-x\\ 2 & -2 & 2 & -2 & 1 & -1 & 1 & -1 & y'-y\end{array}\right).$$

Les combinaisons possibles pour atteindre une case sont les solutions telles que les coefficients a, b, c, d, e, f, g et h appartiennent à $\mathbb{N}$. Si aucune solution entière positive n'existe, cela veut dire qu'on ne peut pas atteindre la case. En se plaçant dans $\mathbb{Z}$, on peut se ramener à l'étude de :

$$\left(\begin{array}{cccc|c}1 & 1 & 2 & 2 & x'-x\\ 2 & -2 & 1 & -1 & y'-y\end{array}\right).$$

2.1.1 Exemple particulier et résolution

Prenons un exemple particulier. Nous voulons ici savoir combien de mouvements il nous faut pour aller de la case $(3, 5)$ à la case $(1, 1)$. Ici, nous voulons donc résoudre le système

précédent avec B = $\begin{pmatrix} 2 \\ 4 \end{pmatrix}$. On voit ici sur Cocalc une solution particulière de cet exemple :

```
A=matrix([[1,1,2,2],[2,-2,1,-1]])
A

B=matrix([[2],[4]])
B

C=A\B
C
[ 1  1  2  2]
[ 2 -2  1 -1]
[2]
[4]
[2]
[0]
[0]
[0]
```

Figure 8 – Code sage pour la résolution d'un système.

Pour atteindre la case, il faut donc deux mouvements $(1, 2)$. Il s'agit ici cependant d'une solution particulière et non pas du système de toutes les solutions.

Dans l'idéal, nous devrions trouver un moyen d'obtenir le système de toutes les solutions de l'équation. Il faudrait être capable de trouver la solution entière la plus "courte" dans cet espace affine. La taille du vecteur est ici définie par la somme des modules des coefficients du vecteur.

2.2 Généralisation

On peut généraliser la méthode de la résolution d'un système à tout cavalier qui a un mouvement de type $a - b$:

- a case vers la droite, b vers le haut,
- a case vers la gauche, b vers le haut,
- a case vers la droite, b vers le bas,
- a case vers la gauche, b vers le bas,
- a case vers le haut, b vers la droite,
- a case vers le bas, b vers la droite,
- a case vers le haut, b vers la gauche,
- a case vers le bas, b vers la gauche,

.

Nous avons alors dans $\mathbb{N}$ un système de type :

$$\left(\begin{array}{cccccccc|c} a & a & -a & -a & b & b & -b & -b & x'-x \\ b & -b & b & -b & a & -a & a & -a & y'-y \end{array}\right),$$

et dans $\mathbb{Z}$ on a :

$$\left(\begin{array}{cccc|c} a & a & b & b & x'-x \\ b & -b & a & -a & y'-y \end{array}\right).$$

3 Algorithmes

3.1 Introduction aux graphes

Pour comprendre plus simplement le principe des algorithmes que nous utilisons, nous devons d'abord définir ce qu'est un graphe.

Définition 3.1. Comme défini par [1], un graphe est un schéma contenant des points, nommés "sommets", qui sont reliés ou non entre eux par des segments, nommés "arêtes".

Nous pouvons donc considérer tous les mouvements possibles d'un cavalier sur un plateau comme sur un graphe. Le graphe du cavalier est d'ailleurs un élément assez connu de la théorie des graphes, et peut être représenté par le graphe suivant, que nous pouvons retrouver dans [2] :

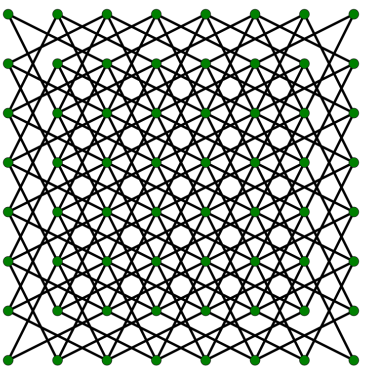


Figure 9 – "Knight's graph".

Cette image représente pour chaque sommet qui est représenté par un point vert (donc chaque case), tous les autres sommets auxquels il est relié (donc toutes les cases atteignables en partant d'une case).

En sachant maintenant ce qu'est un graphe, nous pouvons maintenant comprendre les algorithmes de parcours de graphes.

3.2 Algorithme de parcours en profondeur

3.2.1 Concept

Pour appréhender ce qu'est un algorithme de parcours en profondeur, nous étudions sa définition ainsi que celle de la récursivité, que nous utilisons dans sa structure.

Un algorithme de parcours en profondeur s'applique généralement sur un graphe, et se décrit naturellement de manière récursive. Son application la plus basique est de déterminer l'existence d'un chemin d'un sommet à un autre. Le principe est le suivant : on part d'un sommet et on parcourt un chemin jusqu'à atteindre un cul-de-sac ou bien un sommet déjà visité. Si c'est le cas, on revient alors au dernier endroit où on pouvait suivre un autre chemin. On liste à chaque fois les sommets déjà visités. L'exploration s'arrête lorsqu'on a visité tous les sommets que l'on peut atteindre depuis le sommet de départ.

En informatique, la récursivité définit généralement un algorithme qui fait appel à lui-même dans sa définition. D'un point de vue mathématique, nous pouvons le voir comme une fonction définie en fonction d'elle-même, comme c'est le cas de $F(n) = F(n-2) + F(n-1)$, avec $F : \mathbb{N} \to \mathbb{N}$, représentant la suite de Fibonacci vue comme une fonction.

Pour plus de détails sur ces concepts, nous pouvons nous référer à [3] et [4].

3.2.2 Application

Nous pouvons adapter la méthode du parcours en profondeur à notre problème, afin de déterminer s'il est possible pour le cavalier d'atteindre son but sur un plateau de taille $n \times m$, avec $n, m \in \mathbb{N}$.

On commence par entrer les valeurs pour la taille du plateau, la case de départ, la case à atteindre et les paramètres du mouvement du cavalier. Nous marquons toutes les cases comme étant non-visitées, sauf la case de départ. A partir de là, nous utilisons deux fonctions :

- La fonction `valide(i,j)` permet de savoir si un mouvement vers une case est valide sous deux conditions : il faut que la case soit comprise dans le plateau, et qu'elle n'ait pas déjà été visitée auparavant. Elle prend comme argument les coordonnées du cavalier. Elle retourne ensuite une valeur booléenne indiquant si la case est valide ou non.
- La fonction `possiblePath(i,j,steps)` nous retourne une valeur booléenne (vrai ou faux), qui indique si la case d'arrivée est atteignable ou non depuis la case de départ. Elle prend comme argument les coordonnées du cavalier. Nous commençons par marquer la case de départ comme visitée. Les paires de mouvements du cavalier sont indiquées dans deux listes, une avec les mouvements horizontaux et l'autre avec les mouvements verticaux. La première action de la fonction est de vérifier si les coordonnées qu'elle prend comme argument sont celles de la case d'arrivée. Si ce n'est pas le cas, nous rentrons dans une boucle "for" de la longueur des listes de mouvements. Dans cette boucle, nous créons en premier lieu des valeurs "next" qui représentent les coordonnées `(i,j)` auxquelles on ajoute le mouvement d'index `(0)` dans la liste. C'est à ce moment qu'on vérifie si le mouvement suivant est valide. On entre alors dans la récursion, qui appelle à nouveau la fonction initiale. Si les coordonnées ne sont pas valides, on retourne au dernier endroit où elles l'étaient et on suit un nouveau chemin.

Pour mieux comprendre, prenons un exemple simple : nous considérons un plateau de taille 8×8, avec un cavalier ayant $1-2$ comme paramètres. Nous partons de la case $(1, 1)$ et nous voulons atteindre la case $(2, 3)$. La case de départ est tout de suite listée comme visitée. Nous vérifions s'il s'agit de la case que nous voulons atteindre : c'est faux, donc nous entrons dans la boucle "for". Le premier mouvement testé de la liste est $(-1, -2)$. Appliqué aux coordonnées de la case de départ, nous obtenons des coordonnées non-valides : $(0, -1)$. Nous retournons donc dans la boucle et testons le mouvement suivant : $(-2, -1)$. Nous répétons alors ce processus jusqu'à ce que la case suivante soit valide. C'est le cas pour le $4e$ mouvement, qui est $(-1, 2)$. Comme la case résultant de ce mouvement, $(0, 3)$, est valide, nous entrons dans la partie récursive de la fonction, qui fait appel à elle-même sur ces nouvelles coordonnées. Si nous trouvons un chemin vers la case d'arrivée dans cette récursion, la valeur "True" est retournée. Sinon, on liste la case comme étant non-visitée, et on reprend un autre chemin à partir de la dernière case valide, ici $(0, 3)$.

Dans notre programme [1], l'algorithme s'arrête une fois qu'on atteint la case désirée, et pas uniquement après avoir visité tous les sommets comme c'est généralement le cas dans un algorithme de parcours en profondeur. En effet, si le cavalier devait parcourir toutes les cases, cela poserait un problème au niveau de la mémoire utilisée par l'algorithme lors de la récursion.

```
1  def valide(i, j):
2      if ((i>=0) and (i<N1) and (j>=0) and j<N2) and (not visited[i][j]): #vé
       rifie si la case est dans le plateau et n'a pas déjà été visitée
3          return True
4      return False
5
6  def possiblePath(i,j):
7      visited[i][j] = True #marque la case de départ comme visitée
8      mvmt_x = [-a, -b, -b, -a, a, b, b, a] #mouvements horizontaux possibles
9      mvmt_y = [-b, -a, a, b, -b, -a, a, b] #mouvements verticaux possibles
10     if i == x_ar and j == y_ar: #test si on est arrivés à la case de
       destination
11         return True
12
13     for k in range(0,8): #boucle de la longueur du nombre de mouvements
       possibles
14         next_x = i + mvmt_x[k] #position horizontale: position de départ +
       longueur du mouvement d'indice k
15         next_y = j + mvmt_y[k] #position verticale: position de départ +
       longueur du mouvement d'indice k
16         if valide(next_x, next_y): #teste la validité de la nouvelle case
17             if possiblePath(next_x,next_y): #récursion de la fonction
18                 return True
19             visited[next_x][next_y] = False; #permet retour à la dernière case
        qui était valide
20     return False
```

Listing 1 – Parcours en profondeur.

L'inconvénient de ce type d'algorithme de recherche est que, même s'il trouve un chemin possible, il ne s'agit pas forcément du plus court. La récursion peut également finir par être trop grande pour être supportée par la mémoire de l'ordinateur, dans le cas d'un grand plateau par exemple, ou si l'algorithme commence dès le départ à explorer des chemins inutiles.

3.3 Algorithme de parcours en largeur

3.3.1 Concept

Un algorithme de parcours en largeur s'applique en général à un graphe. Il permet le parcours du graphe en partant d'un sommet, dont il explore ensuite tous les voisins, puis les voisins de ces voisins, etc. Il permet donc de calculer la distance entre le sommet de départ et tous les autres sommets. Il diffère de l'algorithme de parcours en profondeur par l'utilisation d'une file d'attente dans laquelle il place d'abord le premier sommet puis tous ses voisins non encore explorés. Les sommets sont donc explorés par distance croissante du sommet de départ. Il est généralement utilisé pour des problèmes où il est nécessaire de connaître la plus petite distance entre deux sommets.

1. Tous les programmes listés dans ce rapport se trouvent dans le dépôt GitHub [6].

Nous nous référons à [5] pour plus de détails et d'exemples d'algorithmes de parcours en largeur.

3.3.2 Application

En utilisant ce type d'algorithme, nous pouvons non seulement savoir si une case est atteignable mais aussi connaître le nombre de mouvements minimal qu'il faut pour l'atteindre. Comme précédemment, nous entrons les valeurs pour la taille du plateau, la case de départ, la case d'arrivée et les paramètres du mouvement du cavalier. Nous marquons toutes les cases comme non-visitées avec la valeur booléenne "False". Nous utilisons à nouveau deux fonctions :

- La fonction `valide(i,j)` est la même que dans le parcours en profondeur et a le même but : vérifier la validité d'une case.
- La fonction `shortest(i,j)` nous retourne une valeur numérique, correspondant à la longueur du plus court chemin entre deux cases (ou -1 si la case n'est pas atteignable). Elle prend comme argument la case de départ du cavalier. Nous commençons ensuite par créer une liste vide Q, qui représente la file. Nous plaçons dans cette file la case de départ et le nombre de mouvements nécessaires pour l'atteindre (soit 0 mouvement nécessaire). Nous marquons cette case comme étant déjà visitée. Nous définissons ensuite les mouvements possibles du cavalier à nouveau sous forme de liste. Nous entrons ensuite une boucle "while" qui s'arrête soit lorsque l'objectif est atteint, soit quand on a exploré tous les chemins possibles sans atteindre la case d'arrivée (ce qui correspond à une case que l'on ne pourra jamais atteindre depuis notre case de départ avec les paramètres de départ choisis). En utilisant la méthode "pop", qui enlève et retourne les éléments d'un certain index dans une liste, nous commençons par enlever et renvoyer les éléments en première position dans la liste (qui correspond à l'indice 0). Nous vérifions d'abord s'il s'agit de la case d'arrivée, et si ce n'est pas le cas nous entrons dans une boucle qui explore toutes les cases atteignables à partir de cette case. Si elle est valide, chaque case est marquée comme explorée et est ajoutée à la file. On continue donc à explorer tant que la file n'est pas vide (ce qui correspond à l'exploration de toutes les cases atteignables du plateau depuis la position de départ), soit lorsqu'on a atteint la case voulue. Si on atteint la case voulue, on retourne le nombre de mouvements qu'il a fallu au cavalier.

Considérons le même exemple que précédemment pour comprendre la différence entre les deux algorithmes. Nous avons un plateau 8×8, un cavalier avec $1-2$ comme paramètres, nous partons de la case $(1,1)$ et nous voulons atteindre la case $(2,3)$. La première étape réalisée dans la fonction est d'ajouter la case de départ à la file en indiquant qu'il ne faut pas réaliser de mouvements supplémentaires pour l'atteindre et de marquer la case comme visitée. Nous entrons alors dans la boucle "while" qui s'arrêtera quand la file sera vide ou quand nous aurons atteint l'arrivée. La première vérification dans la boucle est de prendre le premier élément de la file et de savoir si on a atteint l'arrivée : ce n'est pas le cas, donc nous passons à la boucle "for". Dans cette boucle, nous testons toutes les cases atteignables à partir de $(1,1)$, nous les marquons, et nous les ajoutons à la file, en précisant le nombre de mouvements pour les atteindre, soit 1 : ce sont les cases $(3,0),(3,2),(2,3)$ et $(0,3)$. Comme la file n'est pas vide, nous retournons dans la boucle "while". Cette fois-ci, lorsqu'on prend l'élément $(2,3,1)$ dans la file, la fonction nous retourne la valeur "True", car nous avons atteint la case d'arrivée, et 1 qui est le nombre minimal de mouvements nécessaires.

```
def valid(i, j): #vérifie si la case est dans le plateau et n'a pas déjà été visitée
    if  i>=0 and i<N1 and j>=0 and j<N2 and (not visited[i][j]):
        return True
    return False

def shortest(i,j):
    Q = [] #cree file
    Q.append((i, j, 0)) #ajoute case de départ à la file
    visited[i][j] = True #marque la case de départ comme visitée

    x_move = [-a, -b, -b, -a, a, b, b, a] #mouvements horizontaux
    y_move = [-b, -a, a, b, -b, -a, a, b] #mouvements verticaux

    while len(Q)!=0: #tant que la file n'est pas vide
        i, j, min_moves = Q.pop(0) #on prend le premier tuple de la liste
        if i == x_ar and j == y_ar: #teste si la case d'arrivée est atteinte
            print("Le nombre minimal de mouvements est:")
            return min_moves
        for k in range(8): #boucle de la longueur du nombre de mouvements possibles
            next_x = i+x_move[k] #mouvement horizontal suivant
            next_y = j+y_move[k] #mouvement vertical suivant
            if valid(next_x, next_y): #test si nouvelle position valide
                visited[next_x][next_y] = True #marque case d'après comme visitée
                Q.append((next_x, next_y, min_moves+1)) #ajoute à la file la nouvelle position et le nombre minimal de mouvements pour l'atteindre
    return -1 #case non-atteignable
```

Listing 2 – Parcours en largeur.

Contrairement à l'algorithme de parcours en profondeur utilisé précédemment, nous pouvons donc être sûrs que la première fois que la case d'arrivée apparaît, nous avons parcouru la plus petite distance nécessaire.

Dans les deux algorithmes, nous pourrions incorporer une liste dans laquelle nous ajoutons toutes les cases visitées, pour voir combien de cases ont été visitées avant d'atteindre notre but.

4 Idées folles

Nous pouvons maintenant considérer des versions plus originales et créatives du jeu d'échecs, comme une version en trois dimensions et des plateaux spéciaux.

4.1 En trois dimensions

Nous commençons par étudier un jeu d'échecs en trois dimensions, dans un cube de taille $n \times n \times n$, $n \in \mathbb{N}$.

4.1.1 Cas général

Considérons maintenant un jeu d'échecs dans un cube : nous adaptons le jeu à la 3eme dimension, où chaque case devient un cube. Un cavalier peut donc alors se déplacer vers la gauche et la droite, ainsi que vers le haut et le bas. Nous l'appelons alors cavalier $a-b-c$. Pour chaque arrangement de $a-b-c$, il y a 8 mouvements de cavalier possibles. Nous listons alors les mouvements possibles pour la combinaison (a, b, c) :

— $a-$mouvements vers la droite, $b-$mouvements vers le haut, $c-$mouvements vers l'avant,

— $a-$mouvements vers la droite, $b-$mouvements vers le bas, $c-$mouvements vers l'avant,

— $a-$mouvements vers la droite, $b-$mouvements vers le haut, $c-$mouvements vers l'arrière,

— $a-$mouvements vers la droite, $b-$mouvements vers le bas, $c-$mouvements vers l'arrière,

— $a-$mouvements vers la gauche, $b-$mouvements vers le haut, $c-$mouvements vers l'avant,

— $a-$mouvements vers la gauche, $b-$mouvements vers le bas, $c-$mouvements vers l'avant,

— $a-$mouvements vers la gauche, $b-$mouvements vers le haut, $c-$mouvements vers l'arrière,

— $a-$mouvements vers la gauche, $b-$mouvements vers le bas, $c-$mouvements vers l'arrière.

Les permutations possibles sont :

- (a, b, c)
- (b, a, c)
- (c, a, b)
- (b, c, a)
- (c, b, a)
- (a, c, b)

Cela représente donc 48 mouvements possibles.

En modifiant les algorithmes utilisés précédemment, nous pouvons alors déterminer si une case du cube est atteignable. On modifie la taille de la boucle, les listes de mouvements possibles, et on rajoute les coordonnées dans l'axe z. Ici, nous allons à nouveau utiliser l'algorithme de parcours en largeur, qui nous permet à la fois de voir si la case est atteignable, mais également le nombre de mouvements minimum.

```
"""Version 3D avec cavalier mouvement (a,b,c)"""

def valid(i, j, h): #on voit qu'on rajoute une coordonnée dans l'axe des z
    if  i>=0 and i<N1 and j>=0 and j<N2 and  h>=0 and h<N3 and (not visited[i][j][h]): #verifie si le cube dans le "plateau" est valide et n'a pas été visité
        return True
    return False

def shortest(i,j,h):
    Q = [] #queue pour utiliser BFS
    Q.append((i, j, h, 0))
    visited[i][j][h] = True #marque premier cube visité comme vrai

    #liste des mouvements possibles dans chaque direction
    x_move = [a, a, a, a, -a, -a, -a, -a, b, b, b, b, -b, -b, -b, -b, c, c, c, c, -c, -c, -c, -c, b, b, b, b, -b, -b, -b, -b, c, c, c, c, -c, -c, -c, -c, a, a, a, a, -a, -a, -a, -a]
    y_move = [b, -b, b, -b, b, -b, b, -b, a, -a, a, -a, a, -a, a, -a, a, -a, a, -a, a, -a, a, -a, c, -c, c, -c, c, -c, c, -c, b, -b, b, -b, b, -b, b, -b, c, -c, c, -c, c, -c, c, -c]
    z_move = [c, c, -c, -c, c, c, -c, -c, c, c, -c, -c, c, c, -c, -c, b, b, -b, -b, b, b, -b, -b, a, a, -a, -a, a, a, -a, -a, a, a, -a, -a, a, a, -a, -a, b, b, -b, -b, b, b, -b, -b]

    while len(Q)!=0: #début du Breadth-First Search: continue tant que la longueur de queue est diff de 0
        i, j, h, min_moves = Q.pop(0) #enleve eléments d'indice 0
        if i == x_ar and j == y_ar and h == z_ar: #teste si on est arrivés à la position voulue
            print("Le nombre minimal de mouvements est:")
            return min_moves
        for k in range(48): #boucle 48 car longueur du nb de mouvements possibles en 3 dimensions
            next_x = i+x_move[k]
            next_y = j+y_move[k]
            next_z = h+z_move[k]
            if valid(next_x, next_y, next_z): #vérifie validité
                visited[next_x][next_y][next_z] = True #marque cube comme visitée
                Q.append((next_x, next_y, next_z, min_moves+1)) #ajoute dans queue
    return -1 #rend -1 si impossible à atteindre
```

Listing 3 – Parcours en largeur dans un cube.

Nous voyons qu'il s'agit du même principe qu'en deux dimensions. La différence est le nombre de mouvements possibles.

4.1.2 Exemple particulier

Nous considérons un cavalier $1-2-3$, dans un cube de taille $4 \times 4 \times 4$. Les mouvements qu'il peut réaliser sont les suivants (dans le repère orthonormé (x, y, z)) :

- $1-$mouvement vers la droite, $2-$mouvements vers le haut, $3-$mouvements vers l'avant,
- $1-$mouvement vers la droite, $2-$mouvements vers le bas, $3-$mouvements vers l'avant,
- $1-$mouvement vers la droite, $2-$mouvements vers le haut, $3-$mouvements vers l'arrière,

— 1−mouvement vers la droite, 2−mouvements vers le bas, 3−mouvements vers l'arrière,
— 1−mouvement vers la gauche, 2−mouvements vers le haut, 3−mouvements vers l'avant,
— 1−mouvement vers la gauche, 2−mouvements vers le bas, 3−mouvements vers l'avant,
— 1−mouvement vers la gauche, 2−mouvements vers le haut, 3−mouvements vers l'arrière,
— 1−mouvement vers la gauche, 2−mouvements vers le bas, 3−mouvements vers l'arrière.

Les permutations possibles sont :

- $(1, 2, 3)$
- $(2, 1, 3)$
- $(3, 1, 2)$
- $(2, 3, 1)$
- $(3, 2, 1)$
- $(1, 3, 2)$

Nous réalisons une illustration de certains de ces mouvements sur *GeoGebra*. Pour plus de lisibilité, vous pouvez consulté le lien suivant : [7].

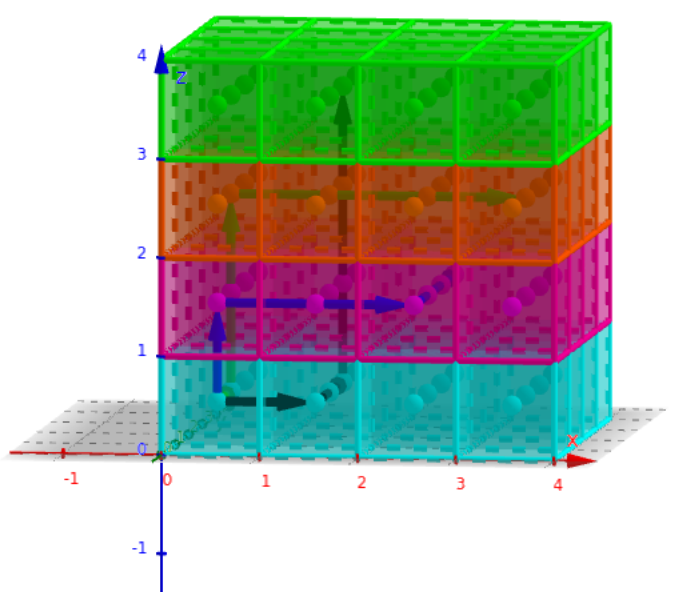


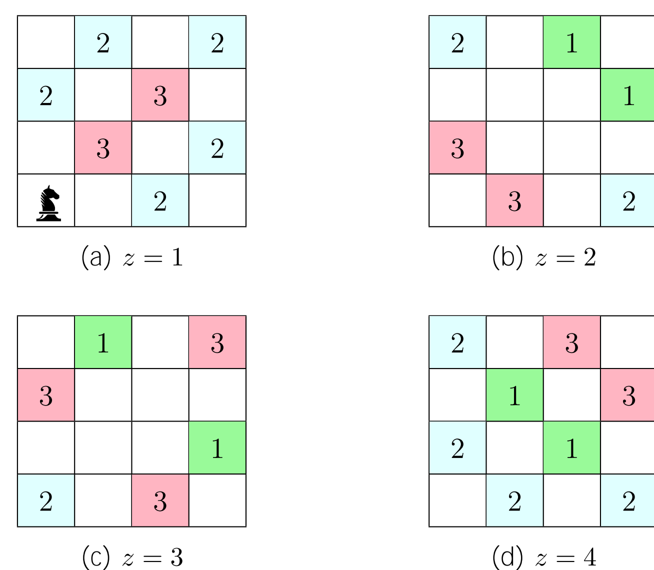
Figure 10 – Représentation de 3 mouvements dans un cube $4 \times 4 \times 4$.

Chaque couche colorée du cube représente une valeur z différente : bleu pour $z = 1$, violet pour $z = 2$, orange pour $z = 3$ et vert pour $z = 4$. Nous donnons un exemple de trois mouvements différents représentés par des flèches, partant tous du point $(0, 0, 0)$:

- Flèches bleues : 1−mouvement vers l'avant, 2−mouvements vers la droite et 3−mouvements vers le haut.

- ◦ Flèches vertes : 1−mouvement vers le haut, 2−mouvements vers l'avant et 3−mouvements vers la droite.
- ◦ Flèches noires : 1−mouvement vers la droite, 2−mouvements vers le haut et 3−mouvements vers l'avant.

Nous voulons ensuite savoir si, dans ce cube, nous pouvons atteindre toutes les cases avec notre cavalier. Pour illustrer plus facilement le nombre de mouvements nécessaires pour atteindre chaque case, on représente le cube couche par couche.



(a) $z = 1$ (b) $z = 2$ (c) $z = 3$ (d) $z = 4$

Figure 11 – Représentation des différentes couches d'un cube.

On remarque que beaucoup plus de cases sont inatteignables que dans le cas du plateau 4×4 en dimension 2.

4.1.3 Relation entre la dimension et le nombre de mouvements possibles

Nous constatons les résultats suivants en dimension 2 et 3, en admettant que chaque mouvement conduit à une case valide du plateau choisi :

- ⋄ Nombre de mouvements possibles en dimension 2 : $8 = 2^2 \cdot 2!$.
- ⋄ Nombre de mouvements possibles en dimension 3 : $48 = 2^3 \cdot 3!$.

Nous pouvons alors généraliser cette observation.

Lemme 4.1. Soit M_n le nombre de mouvements dans la dimension $n \in \mathbb{N}$. Alors

$$M_n = 2^n \cdot n!$$

En partant du principe que le mouvement d'un cavalier possède $n-$composantes, nous pouvons déduire que $n!$ représente le nombre de permutations possibles des $n-$composantes du mouvement. 2^n représenterait alors la puissance $n-$ième de 2, 2 étant le nombre de directions

dans lesquelles le cavalier peut réaliser un mouvement dans un axe. Par exemple, dans l'axe x, nous pouvons aller soit vers les positifs, soit vers les négatifs.

Démonstration. Nous pouvons prouver ce lemme par induction. Soit $n \in \mathbb{N}$.

- Au rang $n = 1$: le cavalier peut réaliser deux mouvements. Cela vérifie donc bien $M_1 = 2^1 \cdot 1! = 2$.
- Au rang $n+1$: on suppose que la proposition est vraie au rang n et on la montre au rang $n+1$. Comme on augmente la dimension de 1, on peut multiplier par 2, soit les deux directions possibles dans la nouvelle dimension, et par $n+1$, pour le nombre de permutations du mouvement à $(n+1)-$composantes. On a :

$$M_{n+1} = 2^n \cdot n! \cdot 2 \cdot (n+1).$$

  Soit :

$$M_{n+1} = 2^{n+1} \cdot (n+1)!.$$

- On peut donc conclure que la propriété est vraie par induction.

□

4.2 Cavalier snake

On s'inspire du célèbre jeu du "snake", dans lequel il faut contrôler un serpent qui doit manger ce qui se trouve sur son chemin tout en évitant de se heurter à son propre corps. Dans certaines versions de ce jeu, lorsqu'il atteint un des quatre murs qui l'entoure, il se téléporte de l'autre côté du jeu. C'est de ce jeu dont nous nous inspirons pour cette version du cavalier.

Quand le cavalier atteint une extrémité du plateau, au lieu d'être limité dans ses mouvements, il peut se téléporter de l'autre côté et reprendre son parcours.

Pour rendre cette idée réalisable en pratique, il suffit de modifier la façon dont la fonction "shortest" de l'algorithme de parcours en largeur procède. Au moment de tester un des mouvements suivants "next", pour ne pas se heurter à des coordonnées en dehors du plateau, on utilise les congruences modulo (représentées par l'opérateur % en Python) la largeur/longueur du plateau pour obtenir une coordonnée valide. On voit dans l'extrait de code suivant la modification faite dans la fonction "shortest" :

```
def shortest(i,j):
    Q = [] #queue pour utiliser BFS
    Q.append((i, j, 0))
    visited[i][j] = True #marque première case visitée comme vraie

    x_move = [-a, -b, -b, -a, a, b, b, a] #liste des mouvements dans directions x et y
    y_move = [-b, -a, a, b, -b, -a, a, b]

    while len(Q)!=0: #début du Breadth-First Search: continue tant que la longueur de queue est diff de 0
        i, j, min_moves = Q.pop(0) #enleve elément d'indice 0
        if i == x_ar and j == y_ar: #teste si on est arrivés à la position voulue
            print("Le nombre minimal de mouvements est:")
            return min_moves
        for k in range(8): #boucle *8 car longueur du nb de mouvements possibles
            next_x = ((i+x_move[k])+N1)%N1 #utilisation des congruences pour obtenir coordonnée valide
            next_y = ((j+y_move[k])+N2)%N2
            if valid(next_x, next_y): #vérifie validité
                visited[next_x][next_y] = True #marque case comme visitée
                Q.append((next_x, next_y, min_moves+1)) #ajoute dans queue
    return -1 #rend -1 si impossible à atteindre
```

Listing 4 – Parcours en largeur version snake.

Prenons un exemple concret pour voir la différence avec un cavalier qui ne peut pas se téléporter. Au départ, notre cavalier se trouve sur la case $(1, 4)$. Pour cet exemple, nous utilisons un cavalier "classique", mais il est possible d'utiliser un cavalier avec un autre mouvement. Nous voulons que ce cavalier atteigne la case $(7, 3)$. En utilisant un cavalier qui ne peut pas se téléporter une fois qu'il a atteint l'extrémité du plateau, nous voyons qu'il est possible d'atteindre cette case en seulement trois mouvements (nous utilisons pour le savoir l'algorithme du parcours en largeur). Cependant, dans le cas où le cavalier se téléporte une fois qu'il atteint le bord du plateau, il peut atteindre la case en un seul mouvement. Par exemple, il peut s'agir d'un mouvement $(-1, -2)$: le cavalier se déplace d'une case vers le bas, puis de deux cases vers la gauche. Comme, directement à sa gauche, il a atteint le bord du plateau, il reprend son parcours de l'autre côté du plateau.

Nous voyons dans l'image suivante la position de départ du cavalier ainsi que la position d'arrivée en rouge.

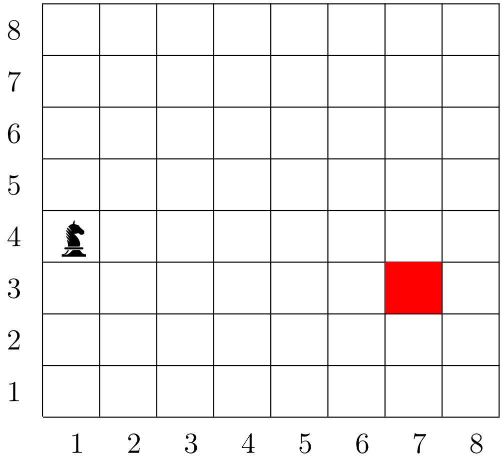


Figure 12 – Exemple d'un cavalier "Snake".

Dans le cas d'un tel cavalier, nous pouvons considérer qu'aucune case du plateau n'est particulière. Il n'y a plus de restrictions concernant le mouvement du cavalier quand il part d'une case (mis à part le fait de ne jamais repasser par la même case). Les 8 mouvements qu'il peut faire en théorie sont donc possibles peu importe la case de départ.

3	2	1	2	3	2	1	2
2	1	4	3	2	3	4	1
3	2	3	2	3	2	3	2
2	3	2	3	4	3	2	3
3	2	3	2	3	2	3	2
2	1	4	3	2	3	4	1
3	2	1	2	3	2	1	2
♞	3	2	3	2	3	2	3

Figure 13 – Plateau 8×8 version "Snake".

Nous voyons dans l'illustration ci-dessus qu'il peut dès le départ réaliser 8 mouvements.

En restant dans l'idée du jeu original, nous pourrions imaginer que le cavalier doit passer par certaines cases pour récupérer toute la nourriture qu'il peut trouver sur son chemin, tout en ayant un chemin le plus court possible.

4.2.1 Déplacement d'un cavalier sur un tore

En géométrie, un tore est un solide géométrique représentant un tube courbé refermé sur lui-même. Nous le représentons à l'aide de *GeoGebra*.

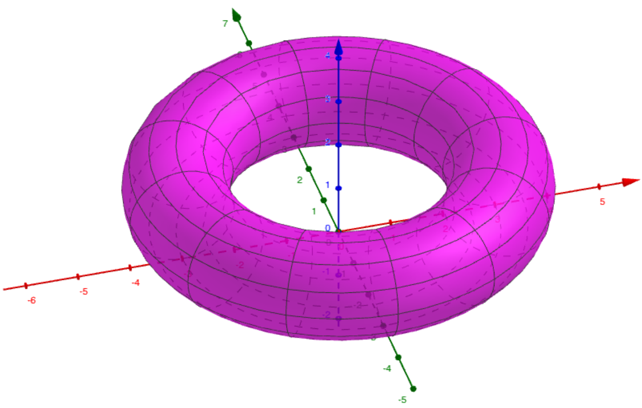


Figure 14 – Tore.

Si le cavalier peut se "téléporter", alors aucune case du plateau n'est spéciale, dans le sens où il n'a pas de limitations dans ses mouvements. Nous pourrions donc alors considérer ses mouvements comme ceux qu'il aurait sur un plateau en forme de tore. En effet, sur un tore, il peut revenir à son point de départ, peu importe la direction qu'il emprunte. Mais comment contraindre le cavalier à se mouvoir sur la surface d'un tore alors que notre plateau d'échecs est plat ?

Nous pouvons alors considérer le plateau comme un *tore carré plat*, un terme introduit par le mathématicien américain John Nash en 1953. Nous pouvons trouver plus de détails sur l'histoire de cette découverte à l'adresse suivante [8]. Lorsque l'on place ce tore carré plat dans l'espace tridimensionnel, cette surface devient alors un tore de révolution. Pour mieux comprendre, regardons une image de [8] qui explique le passage du tore carré plat au tore de révolution :

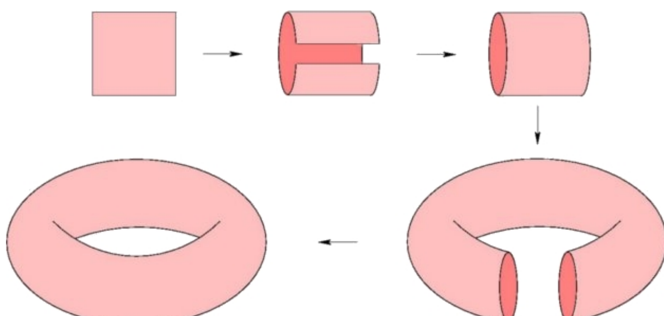


Figure 15 – Passage d'un tore carré plat à un tore de révolution.

Nous pouvons donc réutiliser le même programme que pour le cavalier "snake", en adaptant la dimension au besoin.

Ici, nous représentons le plateau sur le tore sur *GeoGebra*, avec le nombre de mouvements qu'il faut pour atteindre chaque case.

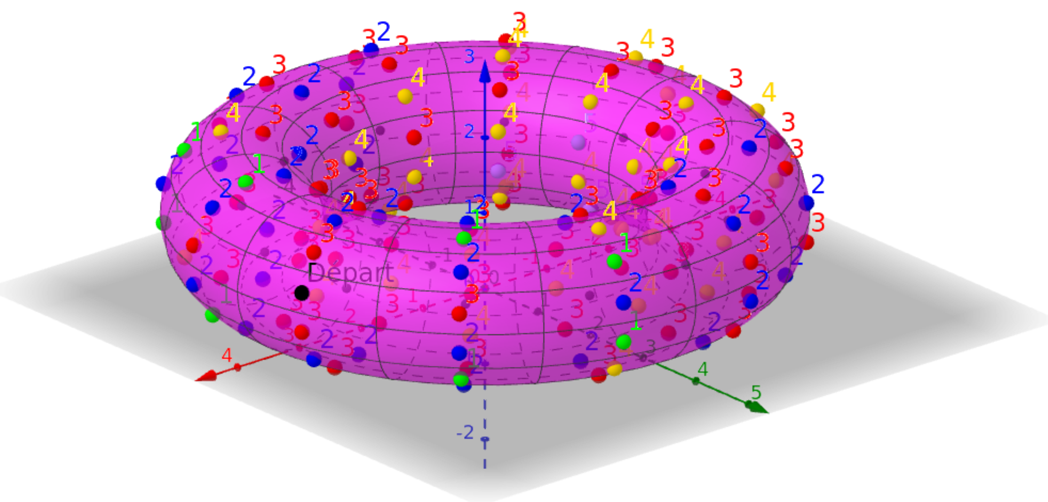


Figure 16 – Nombre de mouvements sur le tore.

Nous pouvons le retrouver ici [9] pour une meilleure visibilité.

4.3 Plateaux spéciaux

Sans changer la façon dont un cavalier se comporte, on peut changer le plateau sur lequel il se déplace en lui donnant une forme différente. Nous pouvons étudier différents exemples afin d'observer les résultats concernant le nombre de mouvements minimal.

4.3.1 Plateau troué

Nous considérons d'abord un plateau de taille 8×8 avec un trou 4×4 au milieu. Le cavalier part de la première case en bas à droite. Pour que le programme puisse considérer ce type de plateau, il suffit de marquer les cases que nous considérons comme "interdites" comme étant déjà visitées. Ainsi, lorsque la fonction "`valide(i,j)`" vérifie la validité d'une case, elle considère qu'elle est déjà visitée et donc le cavalier ne peut pas y aller.

Nous voyons un extrait du programme pour constater quelles cases sont interdites dans notre cas pour pouvoir former un "trou" de taille 4×4 :

```
visited[2][2]=True; visited[2][3]=True; visited[2][4]=True; visited[2][5]=
    True
visited[3][2]=True; visited[3][3]=True; visited[3][4]=True; visited[3][5]=
    True
visited[4][2]=True; visited[4][3]=True; visited[4][4]=True; visited[4][5]=
    True
visited[5][2]=True; visited[5][3]=True; visited[5][4]=True; visited[5][5]=
    True
```

Listing 5 – Plateau avec un trou de taille 4×4.

Pour créer ce plateau de taille 8×8 avec un trou 4×4 au milieu, il faut donc marquer les cases $(2,2)$, $(2,3)$, $(2,4)$, $(2,5)$, $(3,2)$, $(3,3)$, $(3,4)$, $(3,5)$, $(4,2)$, $(4,3)$, $(4,4)$, $(4,5)$, $(5,2)$, $(5,3)$, $(5,4)$ et $(5,5)$ comme interdites.

5	4	5	4	5	6	7	6
4	3	4	5	6	5	6	7
3	6					5	6
2	3					6	5
7	2					5	4
2	1					4	5
3	8	1	2	3	6	3	4
♞	3	2	7	2	3	4	5

Figure 17 – Plateau 8×8 avec un trou 4×4.

Nous rajoutons ici des éléments à la légende que nous avons utilisée précédemment.

Légende (suite)

7 représentent les endroits où le cavalier peut se rendre suite aux déplacements énumérés ci-dessus en partant de toutes les cases 6 .

8 représentent les endroits où le cavalier peut se rendre suite aux déplacements énumérés ci-dessus en partant de toutes les cases 7 .

4.3.2 Plateau coeur

Imaginons maintenant un plateau en forme de coeur de taille 9×9. Voyons un extrait du programme pour constater quelles cases sont interdites dans ce cas :

```
1 #créer la forme:coeur
2 visited[0][8]=True;visited[1][8]=True;visited[3][8]=True;visited[4][8]=True;
    visited[5][8]=True;visited[7][8]=True;visited[8][8]=True
3 visited[0][7]=True;visited[4][7]=True;visited[8][7]=True
4 visited[0][3]=True;visited[8][3]=True
5 visited[0][2]=True;visited[1][2]=True;visited[7][2]=True;visited[8][2]=True
6 visited[0][1]=True;visited[1][1]=True;visited[2][1]=True;visited[6][1]=True;
    visited[7][1]=True;visited[8][1]=True
7 visited[0][0]=True;visited[1][0]=True;visited[2][0]=True;visited[3][0]=True;
    visited[5][0]=True;visited[6][0]=True;visited[7][0]=True;visited[8][0]=
    True
```

Listing 6 – Création du plateau coeur 9×9.

Nous partons de la pointe du coeur :

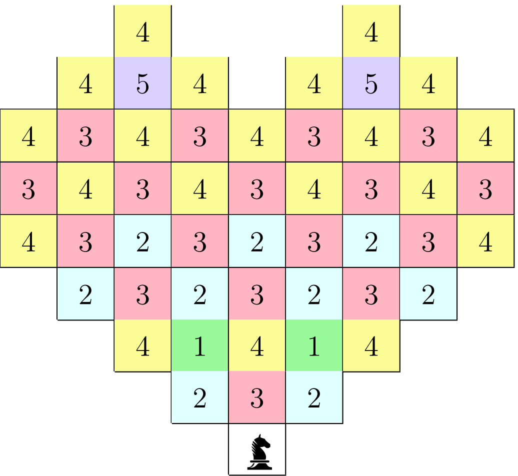


Figure 18 – Plateau coeur 9×9.

Nous pouvons imaginer encore toutes sortes de plateaux : la seule limite est notre imagination.

Conclusion

L'option *Mathématiques expérimentales* nous a permis de découvrir une nouvelle facette des mathématiques : un projet de recherche qui, en plus, laissait place à la créativité et à l'imagination. Nous avons découvert un côté moins traditionnel des mathématiques et plus ludique. Cette option nous a permis d'apprendre à travailler en groupe et de s'organiser autour d'un projet. Nous avons pu approfondir certaines connaissances, en particulier l'utilisation des algorithmes.

Durant ce projet, nous avons fait face à quelques difficultés, notamment au niveau de *Cocalc*. Nous avons du laisser de côté une partie que nous aurions pu approfondir pour obtenir plus de résultats. Il y a eu des idées que nous n'avons pas eu le temps d'exploiter faute de temps. Par exemple, nous avions imaginé réaliser des plateaux spéciaux en forme de lettres, et pourquoi pas même tout l'alphabet, où nous passerions de lettre en lettre. Notre imagination a, en effet, occupé une grande place parmi nos réflexions autour de ce projet. La partie "Idées folles" est celle que nous avons d'ailleurs préférée travailler et que nous aurions aimée approfondir davantage.

A Animation

Pour réaliser une animation d'un cavalier sur un plateau, nous pouvons générer des images en format *.svg*. Le format *.svg* correspond à *Scalable Vector Graphics*. Il s'agit d'un format de données ASCII basé sur XML, permettant de décrire des ensembles de graphiques vectoriels. L'avantage de ce format est de pouvoir redimensionner les images produites à volonté, sans aucune perte de qualité. Nous utilisons donc le code suivant pour générer un plateau avec un cavalier représenté par un disque coloré :

```
<?xml version="1.0" encoding="UTF-8"?>
<svg xmlns="http://www.w3.org/2000/svg" width="810"
height="810" viewBox="-.05 -.05 8.1 8.1">
<rect x="-0.5" y="-0.5" width="9" height="9"/>
<path fill="#FFF" d="M0,0H8v1H0zm0,2H8v1H0zm0
2H8v1H0zm0,2H8v1H0zM1,0V8h1V0zm2,0V8h1V0zm2
0V8h1V0zm2,0V8h1V0z"/>

<g id="knight">
    <circle cx=".5" cy=".5" r=".35" fill="blue"/>
    <animateTransform attributeName="transform"
    type="translate"
    from="0 0"
    to="0 2"
    begin="0s"
    dur="1s"
    />
    <animateTransform attributeName="transform"
    type="translate"
    from="0 2"
    to="1 2"
    begin="1s"
    dur="0.5s"
    />
    <animateTransform attributeName="transform"
    type="translate"
    from="1 2"
    to="3 2"
    begin="1.5s"
    dur="1s"
    />
    <animateTransform attributeName="transform"
    type="translate"
    from="3 2"
    to="3 3"
    begin="2.5s"
    dur="1s"
    />
</g>
```

Listing 7 – Animation d'un cavalier.

Chaque commande indiquant "transform" nous permet de déplacer le cercle en indiquant les coordonnées, le moment de départ du cercle relatif au début de l'animation et la durée du mouvement à accomplir.

En plaçant le fichier *.svg* dans un fichier *.html*, on peut utiliser le site de conversion suivant, `https://html5animationtogif.com/html5togif`, pour convertir ce fichier *.svg* en

fichier *.gif*, nous permettant d'avoir une animation.

En prenant une capture de chaque étape, nous obtenons le résultat suivant :

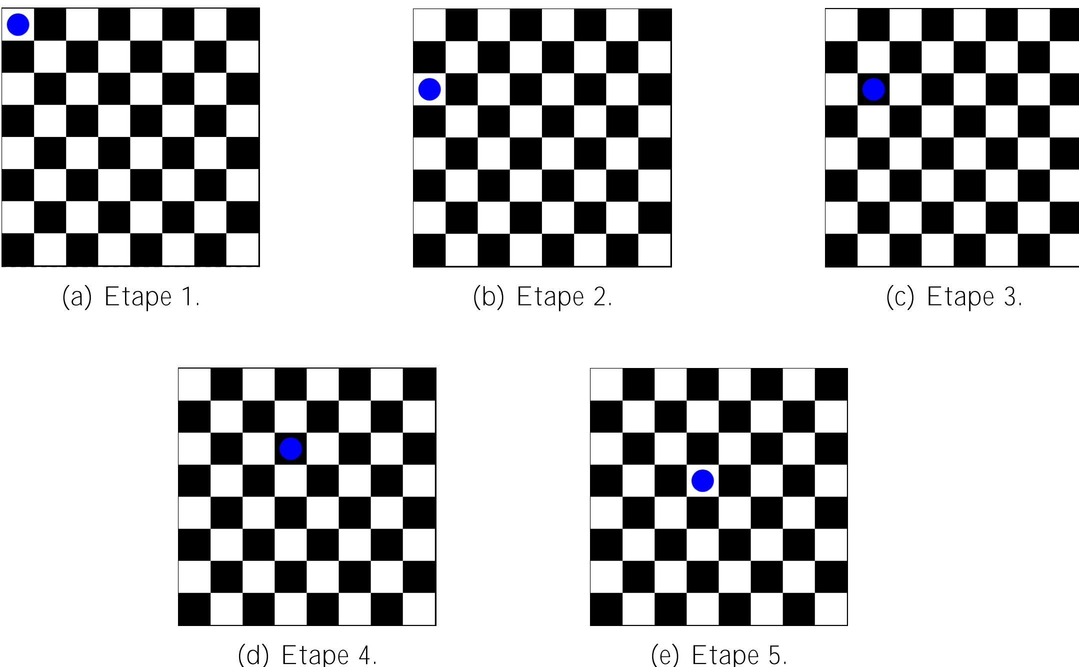


Figure 19 – Animation d'un parcours.

Table des figures

1 Représentation de différents mouvements sur un plateau 8×8. 4
2 Plateau 3×3. 5
3 Plateau 4×4. 5
4 Plateau 5×5. 5
5 Plateau 6×6. 5
6 Plateau 7×7. 6
7 Plateau 8×8. 6
8 Code sage pour la résolution d'un système. 9
9 "Knight's graph". 11
10 Représentation de 3 mouvements dans un cube $4 \times 4 \times 4$. 18
11 Représentation des différentes couches d'un cube. 19
12 Exemple d'un cavalier "Snake". 22
13 Plateau 8×8 version "Snake". 22
14 Tore. 23
15 Passage d'un tore carré plat à un tore de révolution. 23
16 Nombre de mouvements sur le tore. 24
17 Plateau 8×8 avec un trou 4×4. 25
18 Plateau coeur 9×9. 26
19 Animation d'un parcours. 29

Listings

1 Parcours en profondeur. 13
2 Parcours en largeur. 14
3 Parcours en largeur dans un cube. 17
4 Parcours en largeur version snake. 21
5 Plateau avec un trou de taille 4×4. 24
6 Création du plateau coeur 9×9. 26
7 Animation d'un cavalier. 28

Références

[1] Définition de Maxicours niveau lycée d'un graphe https://www.maxicours.com/se/cours/graphes-definitions-proprietes.

[2] Knight's graph, https://en.wikipedia.org/wiki/Knight's_graph, modifié pour la dernière fois le 27-11-2019.

[3] Algorithme de parcours en profondeur, https://fr.wikipedia.org/wiki/Algorithme_de_parcours_en_profondeur, modifié la dernière fois le 03-12-2019.

[4] Récursivité, https://fr.wikipedia.org/wiki/Récursivité#En_mathématique, modifié la dernière fois le 21-03-2020.

[5] Algorithme de parcours en largeur, https://fr.wikipedia.org/wiki/Algorithme_de_parcours_en_largeur, modifié la dernière fois le 20-01-2020.

[6] Dépôt Github des programmes créés, https://github.com/ergzz/maths_exp.

[7] Représentation GeoGebra des mouvements dans un cube, https://www.geogebra.org/classic/zcpk9py2.

[8] Gnash, un tore plat, https://images.math.cnrs.fr/Gnash-un-tore-plat.html.

[9] Plateau tore sur Geogebra, https://www.geogebra.org/classic/jgfdxhfh.