Damien MEUR 1C1

Raphaël LE BOUEDEC

SAE 2.02 : Exploration algorithmique

Années : 2022 - 2023

**Sommaire :**

I - Introduction

1. Particularités
2. Difficultés
3. Ajouts

II - Description claire du type d’algorithmes

1. Points commun entre les algorithmes et explication générale
2. Premier algorithme
3. Second algorithme
4. L’interface graphique

III - Une comparaison argumentée

1. Explications
2. Tests effectués
3. Résultats

IV - Conclusion

1. Améliorations possibles / Solutions alternatives

**I - Introduction: Le problème du tour du cavalier**

I - a) Particularités :

Le problème du tour du cavalier consiste à savoir si à partir d’une case quelconque d’un échiquier la pièce du cavalier est capable de visiter toutes les cases sans jamais repasser deux fois par la même.

Pour résoudre ce problème, il faut être capable de trouver un chemin Hamiltonien et de faire un cycle Hamiltonien. L’utilisateur pourra donc choisir s' il souhaite faire un chemin Hamiltonien ou un cycle Hamiltonien. La seule chose qui change dans le cycle Hamiltonien c’est que le cavalier doit revenir à sa position de départ, toujours sans passer deux fois par la même case.

I - b) Difficulté(s) :

Le problème majeur que l’on a rencontré se trouve au niveau du cycle Hamiltonien. Pour réaliser ce cycle, nous avons besoin d’utiliser un algorithme de parcours en profondeur (DFS) avec backtracking or nous avons eu du mal à réaliser le backtracking. Ainsi, lorsque l’algorithme ne trouvait pas de solution, celui-ci ajoutait des solutions erronées au lieu de remonter le graphe.

I - c) Ajouts:

Pour faciliter la compréhension du Tour du Cavalier, nous avons décidé de créer une interface graphique qui modélise le parcours effectué par le cavalier. Ce parcours est modélisé coup à coup à l’aide de tkinter.(cf. II - d)

**II - Description claire du type d’algorithmes**

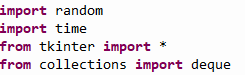
Au départ, nous étions partis pour réaliser notre projet seulement avec des fonctions itératives. Seulement avec le problème rencontré et énoncé précédemment, nous avons décidé d’utiliser un algorithme comprenant des fonctions récursives pour faire le cycle Hamiltonien.

II - a) Points commun entre les algorithmes et explication générale :

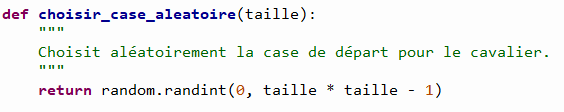
Pour commencer, je vais vous expliquer le fonctionnement général de notre algorithme et comment l’utiliser.

Dans notre algorithme, on commence par demander à l’utilisateur la taille de l’échiquier. Ensuite, on lui demande s’il veut placer lui-même son cavalier ou s' il veut que cela se fasse aléatoirement. S’il veut choisir où commence le cavalier, on lui demande de rentrer la case de départ, sinon, on appelle la fonction ”*choisir\_case\_aleatoire(taille)*” qui va la choisir aléatoirement. Une fois son choix fait, on lui demande s’il veut avoir un chemin Hamiltonien ou un cycle Hamiltonien avec comme point de départ, celui choisi précédemment. S’il veut faire un chemin Hamiltonien, on lancera la fonction *“trouver\_chemin(taille, graphe, case\_actuelle, plateau)”* sinon*, “trouver\_chemin\_boucle(taille, graphe, case\_actuelle, depart, plateau)”.* Pour finir, on lance la fonction *“créerInterface(taille, chemin)”* qui va modéliser l'échiquier avec les coups du cavalier avec le chemin généré par la fonction précédente.

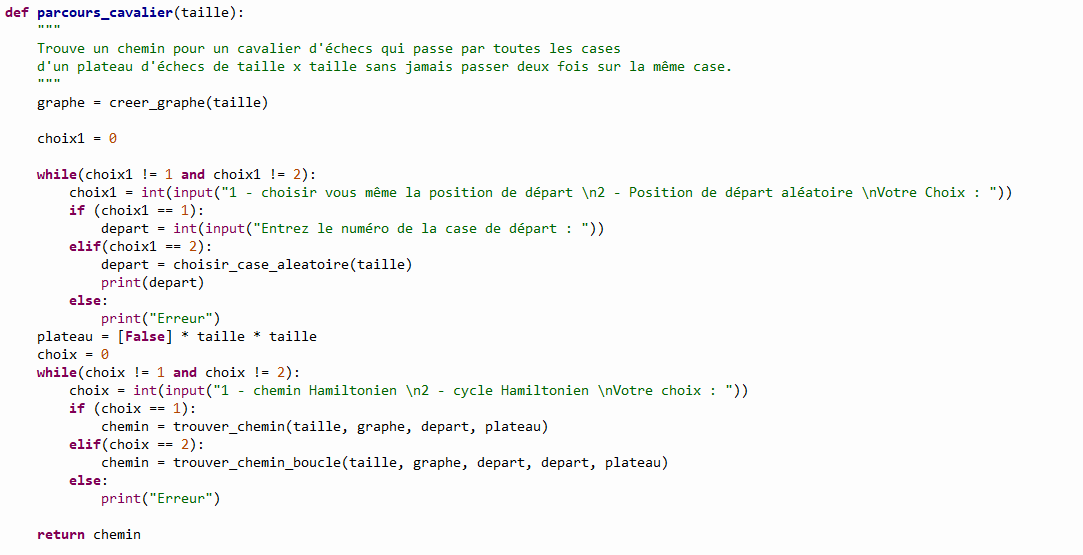
Les librairies que nous avons dû utiliser sont “random” pour l’aléatoire, “deque” de “collection” pour les listes, tkinter pour l’interface graphique et time pour le temps d’exécution.



Ensuite, nous avons créé une fonction ”*choisir\_case\_aleatoire(taille)*” qui nous permet de tirer une case au hasard en fonction de la taille du plateau qui est donnée en paramètre. Par exemple, si la taille est 4, on aura un chiffre entre 0 et 15 qui représentera la case.



Une fois le graphe et la case de départ déterminés, on a créé une fonction *“parcours\_cavalier(taille)”* qui initialise un graphe, demande comment l’utilisateur veut choisir sa case de départ, lui demande si il veut faire un cycle ou un chemin hamiltonien puis exécute les fonctions en conséquence.

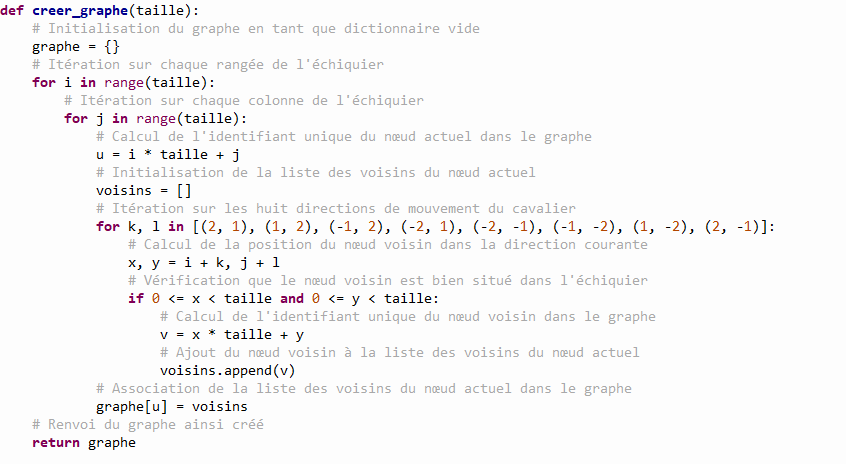


II - b) Premier algorithme :

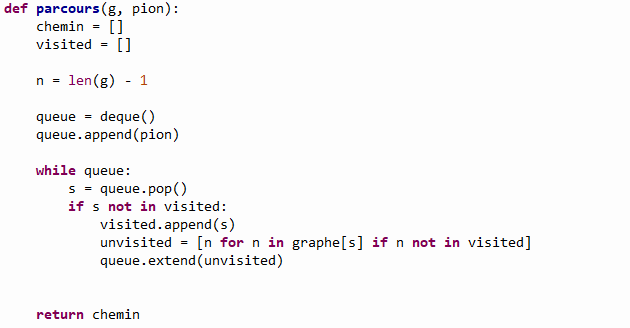
Pour ce premier algorithme, nous avions deux fonctions qui étaient différentes:

* la fonction “*creer\_graphe”* qui a été remplacée par la nouvelle vue précédemment.
* la fonction “*trouver\_chemin”* qui était nommée *“parcours(g, pion)”.*
* la fonction “*trouver\_chemin\_boucle*” qui était nommée “*tour(s, g)*”.

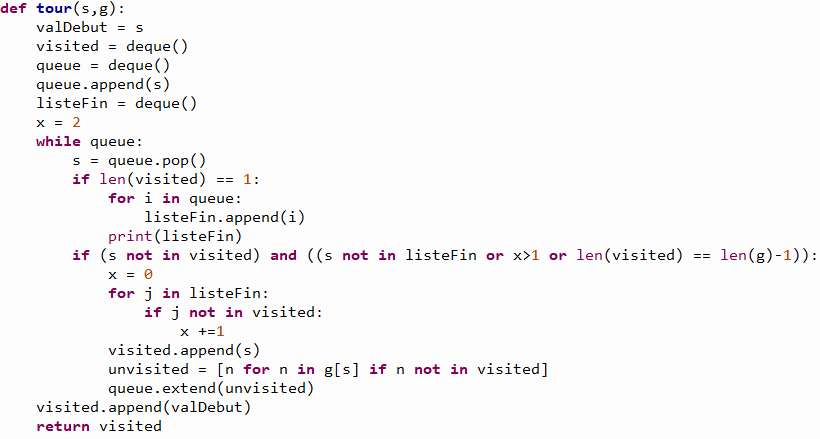
Pour la fonction creer\_graphe(taille), elle crée un dictionnaire *graphe* puis pour chaque case associe un numéro qui va de 0 à taille²-1. Chacune des clés(case) du dictionnaire sont associées à leurs cases voisines en fonction du mouvement du cavalier si ces cases sont à des coordonnées possibles. Puis renvoi le dictionnaire entièrement compléter.



La fonction “*parcours(g, pion)*”, il y a seulement un parcours en profondeur et on s’est retrouvé bloqué avec le problème de backtracking ce qui nous a fait faire un parcours récursif.



La fonction “*tour(s,g)*” est basée sur “*parcours(g, pion)”* et grâce à l’ajout de quelque condition était censé permettre de renvoyer le cycle Hamiltonien.



II - c ) Second algorithme :

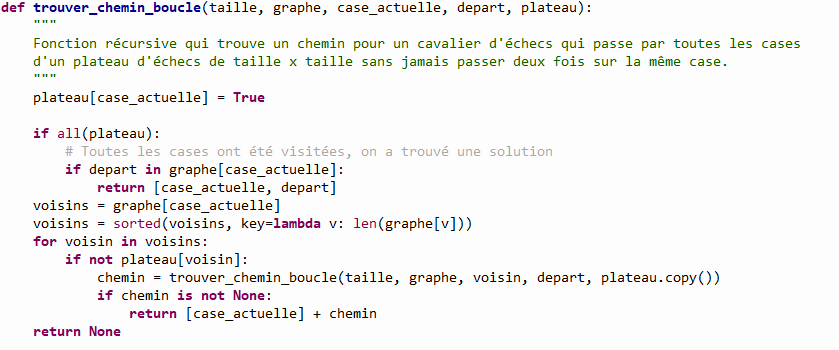
Cet algorithme est la version finale de notre code. Dans cet algorithme, on retrouve une fonction qui résout le problème au départ avec une fonction récursive.

La fonction s’appelle *“trouver\_chemin\_boucle(taille, graphe, case\_actuelle, depart, plateau)”* et retourne soit la liste des cases que doit traverser le cavalier pour réaliser le tour, soit rien s’il n’y a pas de solution.

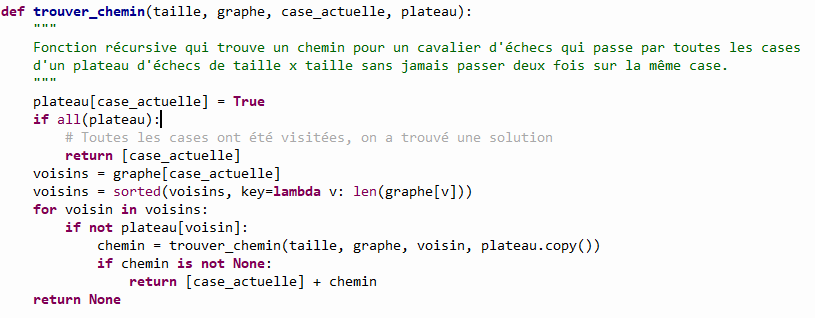
Pour cela la fonction à besoin de plusieurs paramètres:

* la *taille* de l’échiquier (n)
* le *graphe* qui comporte tous les coups possible à partir de chaque position
* la *case\_actuelle* (qui correspond à la case que la fonction test)
* le *depart* afin de pouvoir revenir au point de départ pour compléter le tour
* le *plateau* qui sert à stocker toutes les cases qui ont déjà été visité par la fonction

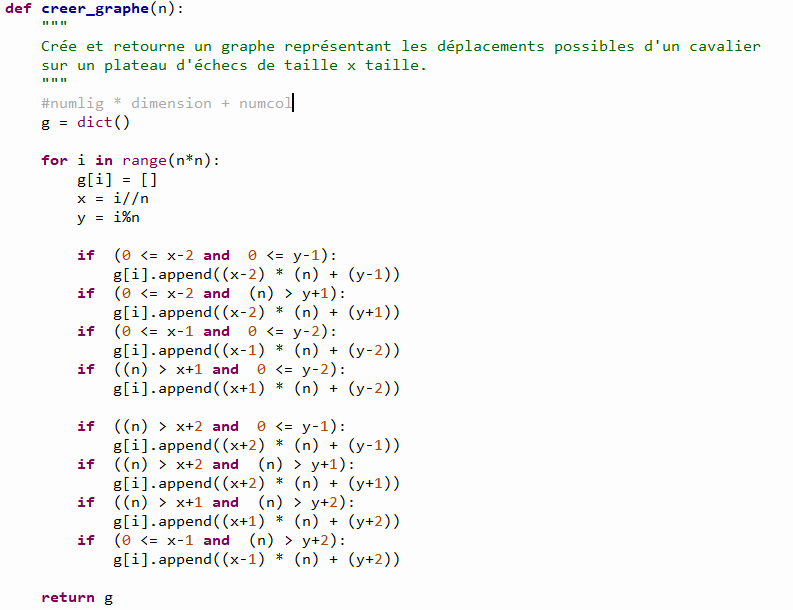
A chaque récursion l’algorithme ajoute la *case\_actuelle* au *plateau* puis regarde si le *plateau* est rempli, si oui alors il vérifie si le *depart* fait partie des “voisins” de *case\_actuelle* et si toutes ces conditions sont réunies alors cela veut dire qu’il a trouvé une solution. Autrement il récupère tous les “voisins” de *case\_actuelle* dans *voisins* puis regarde si le premier voisin fait partie ou non des cases déjà visitées, etc. S' il trouve une case non visitée alors il lance une nouvelle récursion.



La fonction “*trouver\_chemin(taille, graphe, case\_actuelle, depart, plateau)”* fonctionne exactement comme la fonction ci-dessus à l’exception près que l’on ne regarde pas si la dernière case du chemin est une voisine de la case de départ ce qui fait que le chemin est ouvert.



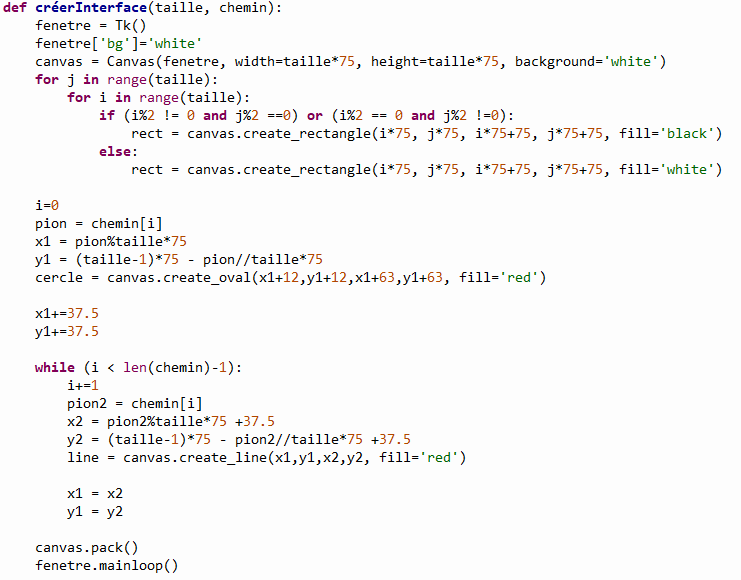
Ainsi, nous avons décidé de créer un graphe pour représenter l’échiquier. Pour cela nous avons fait une fonction nommée *“creer\_graphe(n)”* qui en fonction des dimensions “n” passés en argument crée un dictionnaire où les clés correspondent à un numéro de 0 à n²-1. Ce numéro représente une case de l'échiquier. Les valeurs de ces clés sont les cases sur lesquelles le cavalier peut aller en étant sur une certaine case(la clé concernée).



II - d) L’interface graphique :

L’interface graphique réalisée dans la fonction “*creerInterface(taille, chemin)*” utilise la bibliothèque tkinter de python.

Tout d’abord la fonction crée une fenêtre en fonction des dimensions de l’échiquier (75 pixels par case). Ensuite toutes les cases sont créées grâce à 2 boucles “for” en alternant case blanches et case noires.La fonction commence par créer un point là ou le pion commence en récupérant la valeur de la première case du *chemin*. Enfin entre chaque case du *chemin* on trace des lignes. Les coordonnées *x* et *y* de chaques cases sont calculées en fonction soit du reste de la division euclidienne entre la valeur de la case et la *taille* pour *x* ou bien la division entière entre la valeur de la case et la *taille* pour *y*.

****

**III - Une comparaison argumentée**

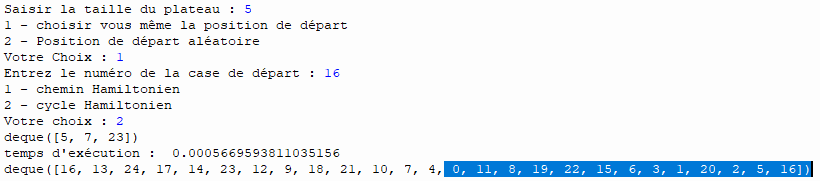
III - a) Explications:

Ci-dessous, vous pouvez retrouver différents tests de performances effectués entre les deux algorithmes. Les tests ont été effectués sur des échiquiers allant d’une taille 5 par 5 à 8 par 8 grâce à la bibliothèque “time”.

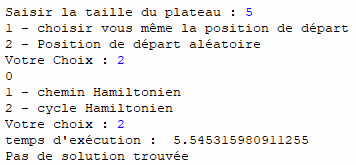
III - b) Tests effectués :

échiquier de taille 5x5(avec cycle):

*premier algorithme*

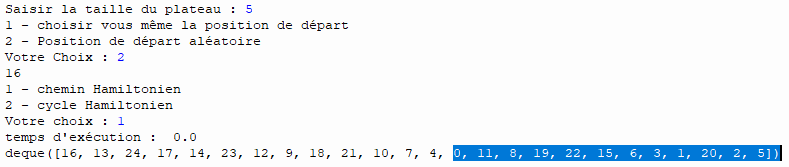
**

*second algorithme*

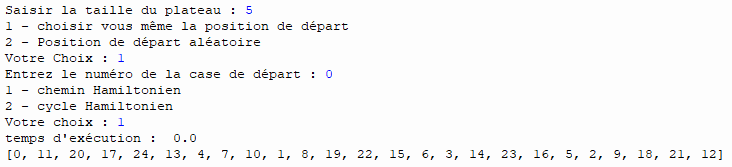


échiquier de taille 5x5(sans cycle):

*premier algorithme*

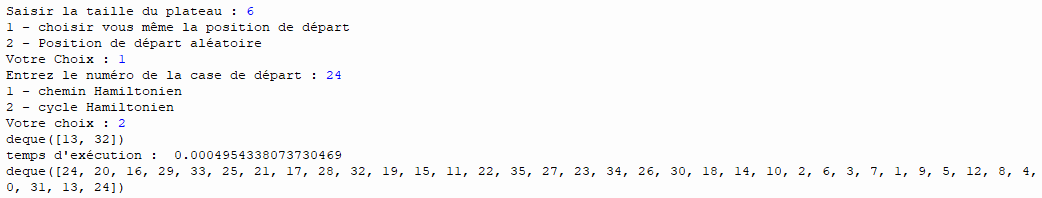
**

*second algorithme*

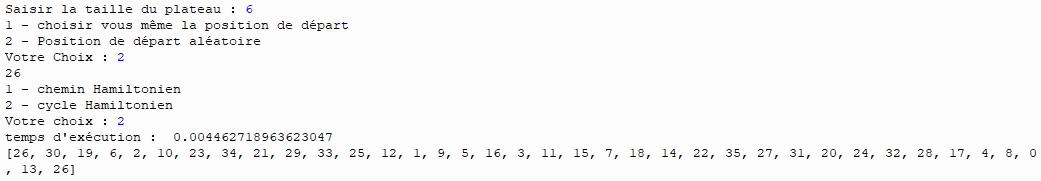


échiquier de taille 6x6(avec cycle):

*premier algorithme*

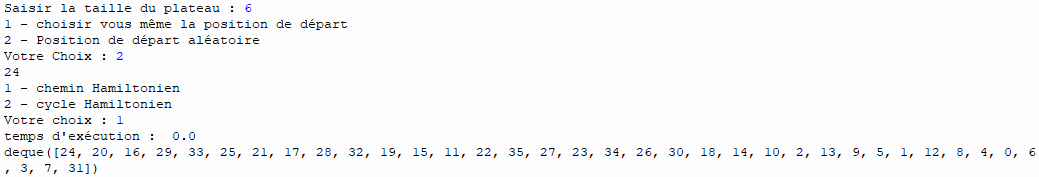


*second algorithme*

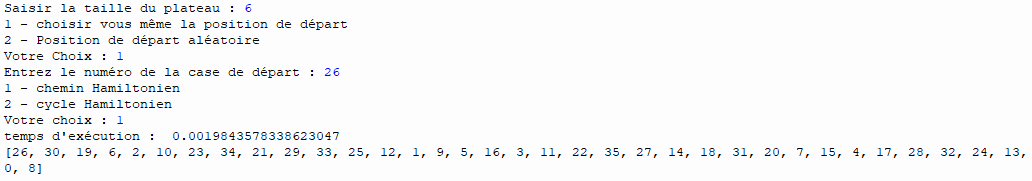
****

échiquier de taille 6x6(sans cycle):

*premier algorithme*

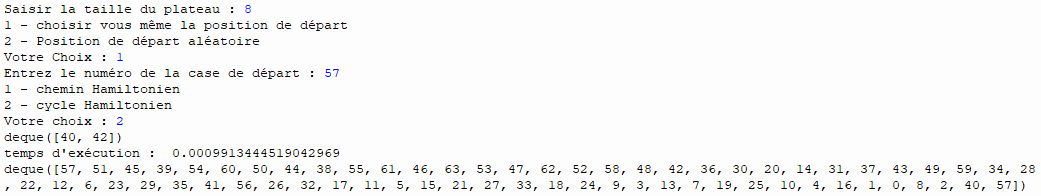
**

*second algorithme*

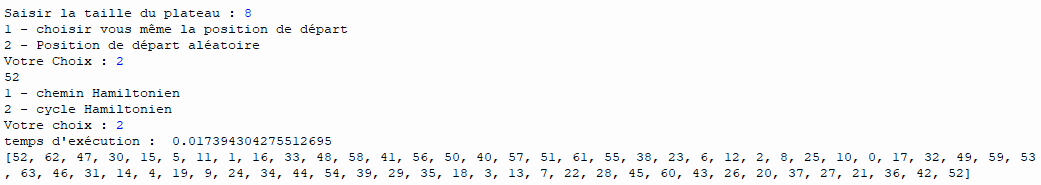
****

échiquier de taille 8x8(avec cycle):

*premier algorithme*

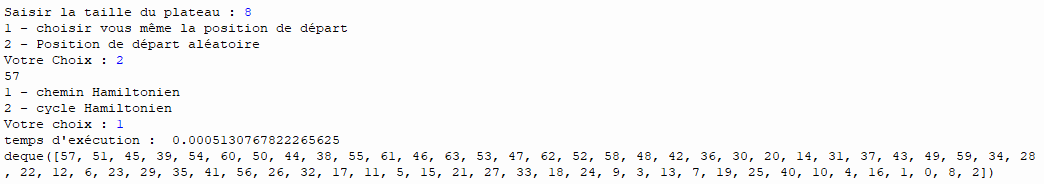
**

*second algorithme*

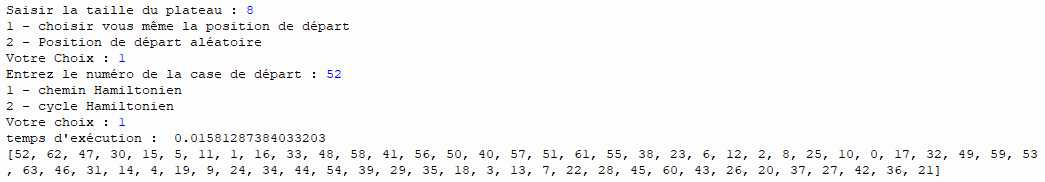
****

échiquier de taille 8x8(sans cycle):

*premier algorithme*

**

*second algorithme*

****

III - c) Résultats

En effet, on aurait pu s’en douter, le premier algorithme à un temps d'exécution nettement plus court que le deuxième mais cela est normal étant donné qu’il est buggé et itératif. D’autre part, le second à un temps d’exécution plutôt court lui aussi.

**IV - Conclusion**

IV - a) Améliorations possibles / Solutions alternatives

Le problème du tour du cavalier est résolu mais une amélioration peut être apportée. Les problèmes rencontrés ont été résolus pour la majorité. La fonction *“trouver\_chemin\_boucle(taille, graphe, case\_actuelle, depart, plateau)“* est améliorable car lorsque le pion est placé dans le coin en bas à gauche ou en bas à droite au départ, l’algorithme n’arrive pas à trouver de solution. Malheureusement, nous n’avons pas trouvé de solution pour remédier à ce problème.