SAÉ 2.02

COMPARAISON D'ALGORITHMES
LE TOUR DU CAVALIER

HOAREAU – VERGER²

avril 2023

iut Lannion

SOMMAIRE

01 INTRODUCTION.

NOTRE CHOIX

Nous avons choisi d'étudier le problème du **Tour du Cavalier** car, parmi les sujets proposés, il s'agissait de celui qui nous semblait le plus accessible, aussi bien en termes de compréhension que de réalisation. Par ailleurs, ce problème étant particulièrement populaire, les sources d'informations à son sujet ne manquent pas.

PRÉSENTATION DU PROBLÈME

Le **Tour du Cavalier** est un problème dont la plus ancienne référence remonte à l'Inde du IXe siècle. Il s'agit d'un exercice mathématique qui consiste à déterminer si un cavalier, pièce du jeu d'échec dont le déplacement s'effectue toujours en L (*voir fig. 1*), en partant d'une case quelconque, peut parcourir toutes les cases d'un échiquier (dont la taille peut varier (*voir p.4*)), en ne passant qu'une seule fois sur chacune d'entre elles.

Il s'agit en réalité d'un exemple de **chemin hamiltonien** d'un **graphe non-orienté**, dans lequel chaque case du plateau représente un sommet et chaque arête le déplacement du cavalier d'une case à une autre. Si le sommet de départ est accessible à partir du dernier sommet visité, alors on parlera de **cycle** et on dira que le parcours est **fermé** (*voir page suivante*). Dans le cas contraire, on parlera de parcours **ouvert**.

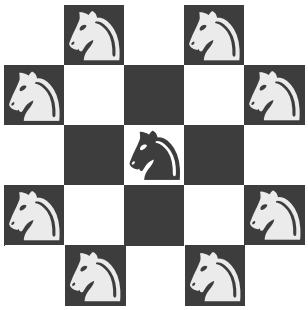


Fig.1 : Les déplacements possibles du cavalier

Un algorithme possible pour résoudre ce type de problème est celui du **parcours en profondeur** (*Depth-First Search / DFS*), qui consiste à partir de la racine d'un graphe (la case de départ choisie sur le plateau) et d'explorer un premier chemin jusqu'à rencontrer un sommet n'ayant plus de voisins non visités. En faisant alors appel à un algorithme de **retour sur trace** (*backtracking*), on remonte ainsi de niveau en niveau, à la recherche d'un nouveau chemin, que l'on parcourt à son tour. On procède ainsi jusqu'à ce que tout le graphe ait été parcouru ou que plus aucun chemin ne soit disponible.

Au fil des siècles, de nombreuses solutions à ce problème ont été proposées. On remarquera particulièrement les travaux du mathématicien **Leonhard Euler** (fin du XVIIIe siècle), s'appuyant notamment sur le principe des symétries, ou encore la méthode de **Warnsdorff** (début du XIXe siècle), consistant à choisir la prochaine case en fonction de son poids, et dont nos deux algorithmes s'inspirent.

EXEMPLES DE REPRÉSENTATIONS DE DIFFÉRENTS PARCOURS

...sur des plateaux de 64 cases (8*8)

00	15	26	21	02	17	46	55
25	22	01	16	45	56	03	18
14	27	24	61	20	47	54	57
23	34	29	44	59	62	19	04
28	13	60	33	48	43	58	53
35	30	37	40	63	52	05	08
12	39	32	49	10	07	42	51
31	36	11	38	41	50	09	06

Fig. 2 : Représentation numérique d'un parcours ouvert partant de la case (1,1)

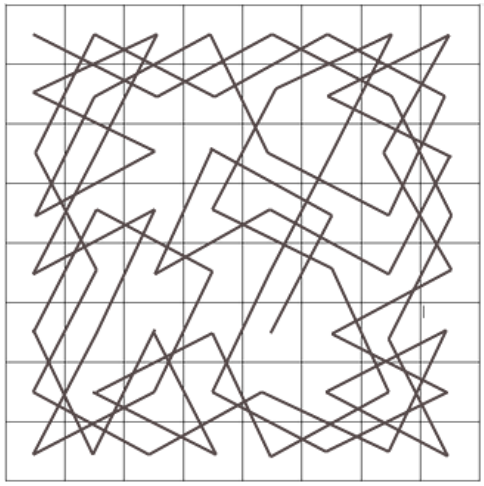


Fig. 3 : Représentation graphique du parcours ci-contre

00	25	18	13	20	23	38	41
17	12	63	24	37	40	35	22
26	01	14	19	28	21	42	39
11	16	27	62	47	36	29	34
02	61	04	15	30	59	48	43
07	10	53	60	51	46	33	58
54	03	08	05	56	31	44	49
09	06	55	52	45	50	57	32

Fig. 4 : Représentation numérique d'un parcours fermé partant de la case (1,1)

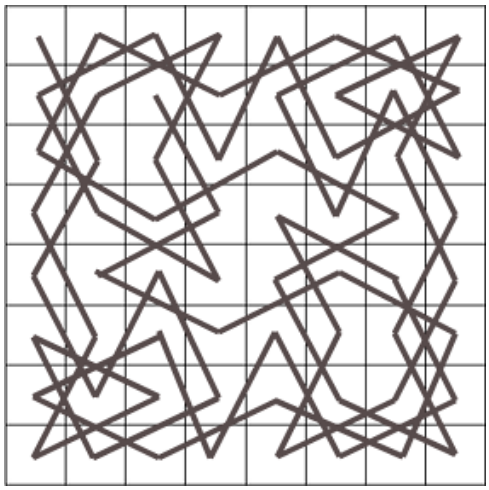


Fig. 5 : Représentation graphique du parcours ci-contre

Mais ces parcours ne représentent que deux solutions parmi tant d'autres.

En effet, pour un plateau de 64 cases, on estime à 26 534 728 821 064 le nombre de circuits fermés (cycles hamiltoniens) et à 19 591 828 170 979 904 celui de circuits ouverts !

... sur des plateaux moins conventionnels

00	23	12	17	06
13	18	07	22	11
08	01	24	05	16
19	14	03	10	21
02	09	20	15	04

*Fig. 6 : Représentation numérique d'un chemin (ouvert) partant de la case (1,1) sur un plateau de 5*5*

On notera que le Problème du Cavalier n'est pas réalisable sur des échiquiers carrés de moins de 25 cases (5*5).

00	27	24	05	18	03	20	09	12	15
25	06	29	02	23	08	17	14	21	10
28	01	26	07	04	19	22	11	16	13

*Fig. 7 : Représentation numérique d'un chemin (fermé) partant de la case (1,1) sur un plateau de 3*10*

76	57	44	55	40	59	46	49	18	51
43	72	75	58	45	54	17	52	47	32
74	77	56	41	60	39	48	31	50	19
71	42	73	78	63	36	53	16	33	30
84	79	64	61	12	87	38	35	20	15
65	70	85	88	37	62	11	14	29	34
80	83	66	03	86	13	28	23	92	21
69	00	81	98	89	10	91	06	27	24
82	97	02	67	04	95	08	25	22	93
01	68	99	96	09	90	05	94	07	26

*Fig. 8 : Représentation numérique d'un chemin (fermé) partant de la case (2,8) sur un échiquier de 10*10*

Il existe de nombreuses autres variantes, pouvant également modifier la forme du plateau, le nombre de joueurs, ou encore la façon dont le cavalier se déplace,...

Ce projet n'abordera néanmoins pas ces variantes plus en détails.

MISE EN APPLICATION DANS CETTE SAÉ

Nous cherchons donc à réaliser deux programmes capables de déterminer s'il existe ou non, pour un plateau de dimensions *m*n*, en partant d'une case quelconque, un chemin hamiltonien que notre cavalier pourrait emprunter. L'objectif final étant d'en comparer les performances.

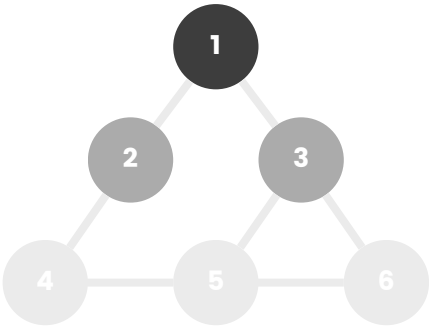
02
LES ALGORITHMES.
1/ PARCOURS ITÉRATIF

Le principe de cet algorithme est qu'il sauvegarde dans un dictionnaire chaque sommet visité (clé) et les voisins déjà visités qui lui sont associés (valeurs). Lorsque le bout d'un chemin a été atteint (le sommet courant a déjà visité tous ses voisins), alors l'algorithme revient sur ses pas jusqu'à trouver un sommet ayant encore des voisins non visités.

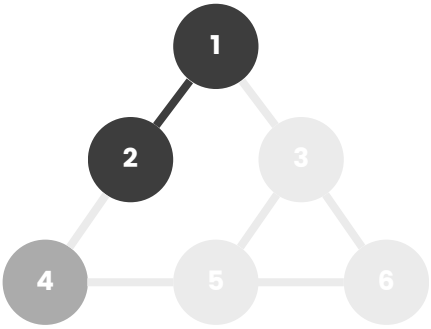
Interprétation algorithmique

```
chemin = [s]
sommets_visites = {}
tant que chemin n'est pas vide, faire :
     sommet_courant = chemin [-1]
     si longueur ( chemin ) = longueur ( graphe ), alors :
          retourner chemin
     sinon si sommet_courant a des voisins, alors :
          on ajoute un voisin à chemin
          sommets_visites [ sommet_courant ] = sommet_visites + voisin
     sinon
          sommet_temporaire = sommet_courant
          supprimer chemin [-1]
          sommet_courant = chemin [-1]
          si sommet_courant n'a plus de voisins, alors :
               supprimer sommets_visites [ sommet_courant ]
               supprimer chemin [-1]
               supprimer sommets_visites [ sommet_courant ]
retourner chemin
```

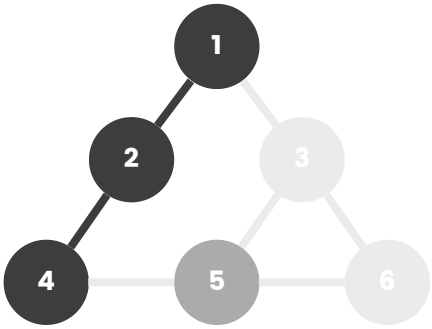
ILLUSTRATION AVEC UN GRAPHE (1)



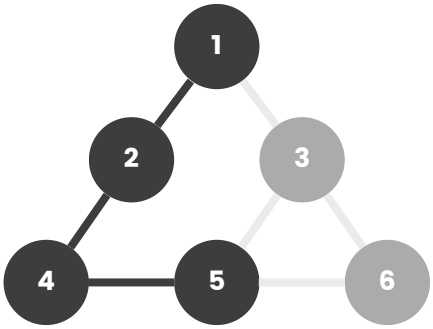

```
chemin = [1]
sommets_visites = {
     1  : [] }
voisins = [2,3]
```



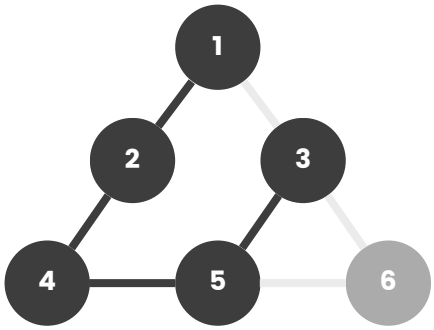

```
chemin = [1,2]
sommets_visites = {
     1 : [2],
     2 : [] }
voisins = [4]
```



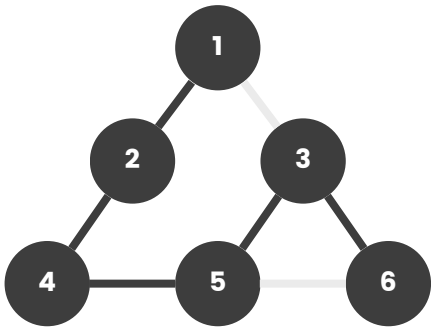

```
chemin = [1,2,4]
sommets_visites = {
     1 : [2],
     2 : [4] }
voisins = [5]
```




```
chemin = [1,2,4,5]
sommets_visites = {
     1 : [2],
     2 : [4],
     5 : [] }
voisins = [3,6]
```

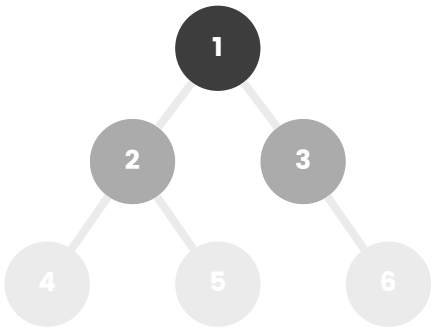



```
chemin = [1,2,4,5,3]
sommets_visites = {
     1 : [2],
     2 : [4],
     5 : [3],
     3 : [] }
voisins = [6]
```

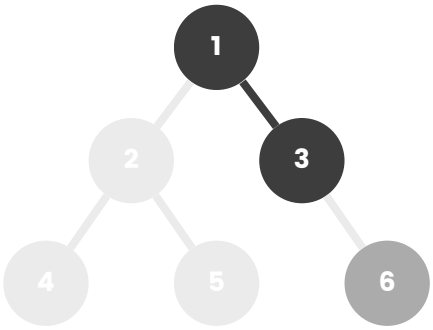



```
chemin = [1,2,4,5,3,6]
sommets_visites = {
     1 : [2],
     2 : [4],
     5 : [3],
     3 : [6],
     6 : [] }
voisins = []
```

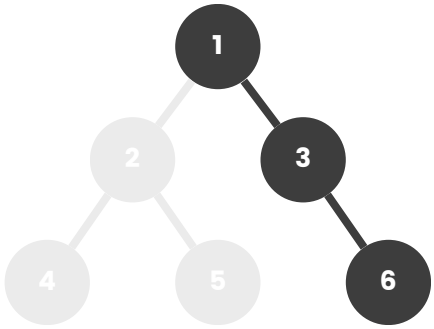
ILLUSTRATION AVEC UN GRAPHE (2)




```
chemin = [1]
sommets_visites = {
     1  : []
}
voisins = [3,2]
```

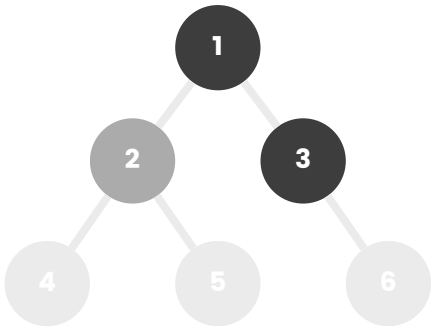



```
chemin = [1,3]
sommets_visites = {
     1 : [3],
     3 : []}
voisins = [6]
```

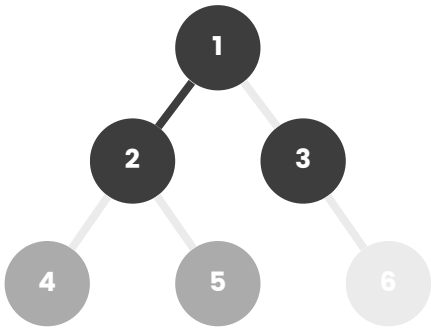



```
chemin = [1,3,6]
sommets_visites = {
     1 : [3],
     3 : [6],
     6 : []}
voisins = []
```

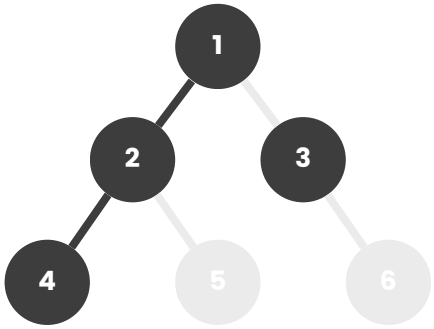
Le sommet 6 n'ayant plus de voisins non visités, on le retire du *chemin*, mais on le conserve dans une variable. Comme le sommet 3 est dans la même configuration, on supprime les voisins visités de 3, puis on supprime aussi les voisins visités de 6, qu'on avait stockés dans une variable.



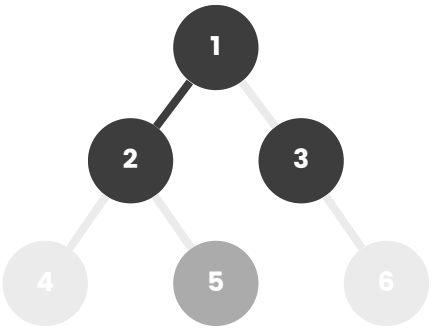

```
chemin = [1]
sommets_visites = {
     1 : [3],
     3 : [],
     6 : [] }
voisins = [2]
```



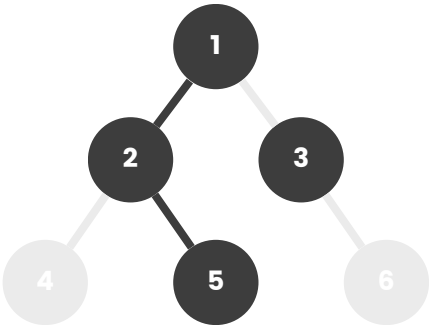

```
chemin = [1,2]
sommets_visites = {
     1 : [3,2],
     3 : [],
     6 : [],
     2 : [] }
voisins = [4,5]
```



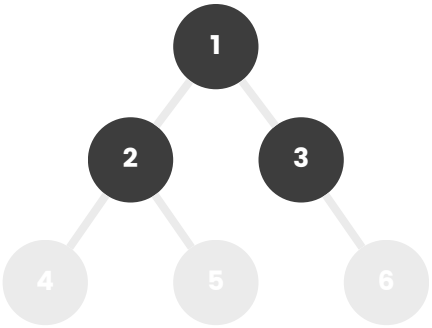

```
chemin = [1,2,4]
sommets_visites = {
     1 : [3,2],
     3 : [],
     6 : [],
     2 : [4],
     4 : [] }
voisins = []
```




```
chemin = [1,2]
sommets_visites = {
     1 : [3,2],
     3 : [],
     6 : [],
     2 : [4],
     4 : [] }
voisins = [5]
```




```
chemin = [1,2,5]
sommets_visites = {
     1 : [3,2],
     3 : [],
     6 : [],
     2 : [4,5],
     4 : []}
voisins = []
```




```
chemin = [1]
sommets_visites = {
     1 : [3,2],
     3 : [],
     6 : [],
     2 : [],
     4 : []}
voisins = []
```

Le sommet 1 ayant déjà les sommets 2 et 3 dans la liste de ses sommets visités, on le retire de la liste *chemin*.


```
chemin = []
```

La boucle prend fin car le chemin ne contient plus aucun sommet.

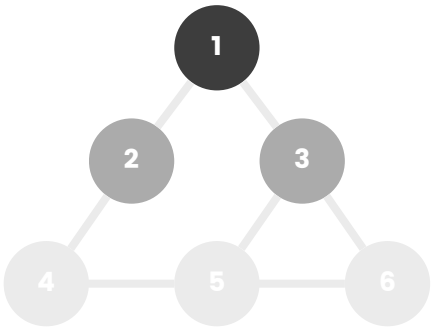
2/ PARCOURS RÉCURSIF

Ce second algorithme permet, grâce à la récursivité, de retourner en arrière lorsqu'il rencontre une impasse et de continuer sur les sommets qui étaient possibles. Pour chaque sommet voisin du sommet courant, l'algorithme appellera la fonction *parcours()*, qui ajoute le sommet de départ au chemin, puis, tant que le cavalier n'a pas parcouru l'intégralité du plateau, crée une liste des voisins du sommet courant. Si ce dernier n'a pas de voisins non visités, on supprime le dernier élément de chemin.

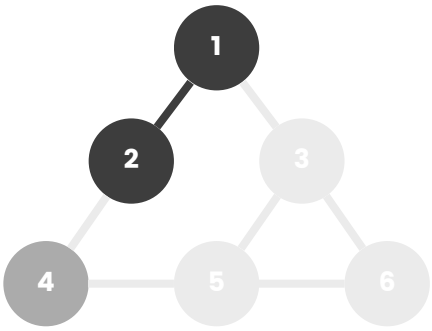
Interprétation algorithmique

```
trouve_chemin ( sommet_depart )
     chemin = [ ]
     parcours ( sommet_depart )
          chemin = ajout ( sommet_depart )
          si longueur chemin = longueur graphe, alors :
               gagne = vrai
          sinon :
               gagne = faux
               créer une liste de voisins
               pour chaque voisin,
                    si gagne = vrai
                         finir
                    sinon
                         gagne = parcours ( un voisin )
               si gagne = faux
                    supprimer le dernier sommet de chemin
          retourne gagne
     retourne chemin
```

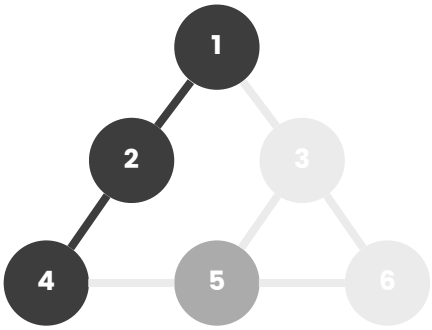
ILLUSTRATION AVEC UN GRAPHE (1)



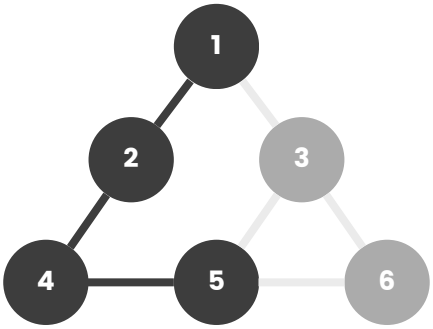

```
chemin = [1]
voisins = [2,3]
```



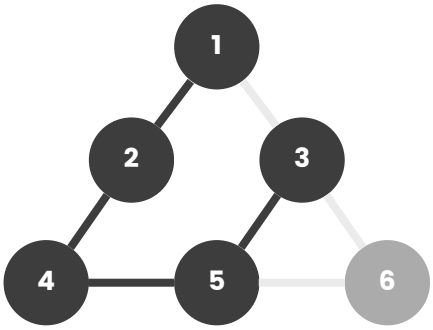

```
chemin = [1,2]
voisins = [4]
```



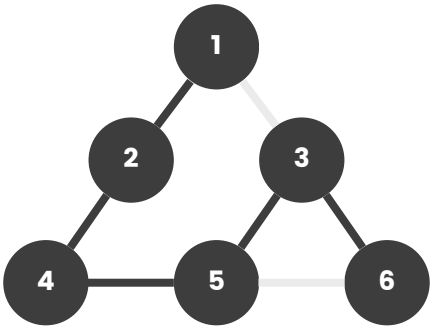

```
chemin = [1,2,4]
voisins = [5]
```




```
chemin = [1,2,4,5]
voisins = [3,6]
```

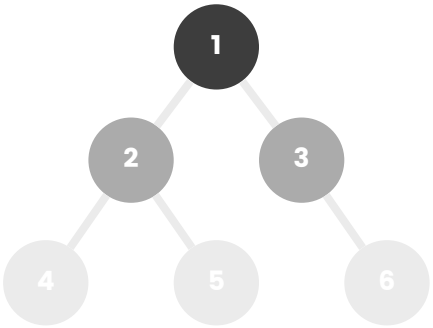



```
chemin = [1,2,4,5,3]
voisins = [6]
```

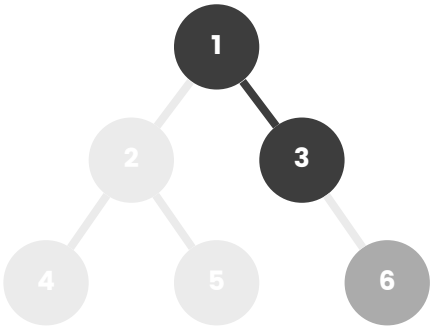



```
chemin = [1,2,4,5,3,6]
voisins = []
```

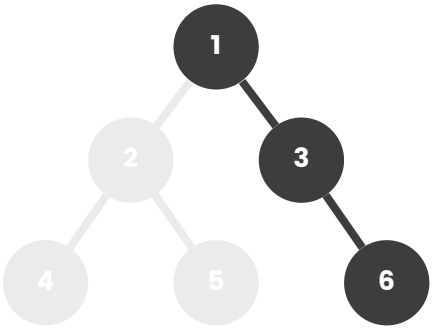
ILLUSTRATION AVEC UN GRAPHE (2)




```
chemin = [1]
voisins = [3,2]
```

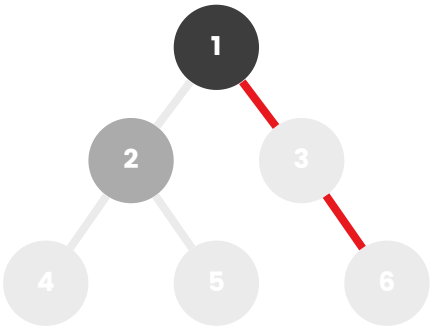



```
chemin = [1,3]
voisins = [6]
```

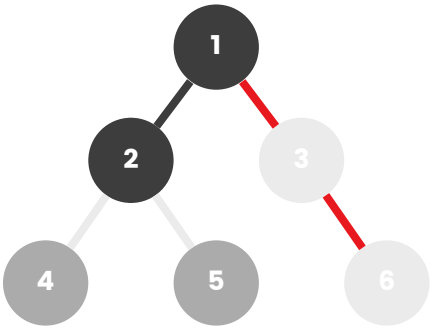



```
chemin = [6]
voisins = []
```

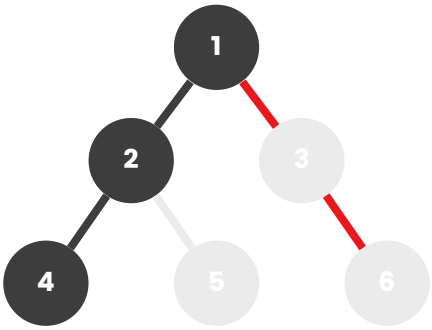
Comme le sommet 6 n'a plus de voisins et qu'on n'a pas encore parcouru tous les sommets, on retourne à l'appel précédent (récursion), en l'occurrence ici, au sommet 1.




```
chemin = [1]
voisins = [2]
```

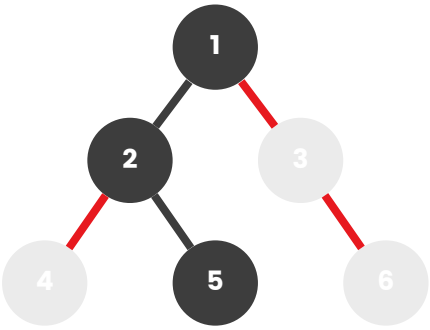



```
chemin = [1,2]
voisins = [4,5]
```

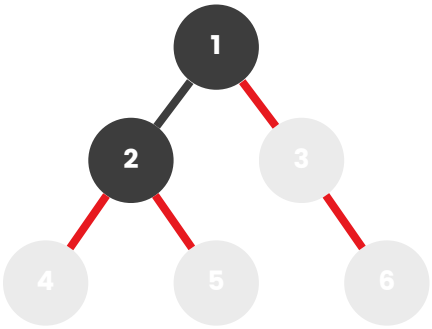



```
chemin = [1,2,4]
voisins = []
```

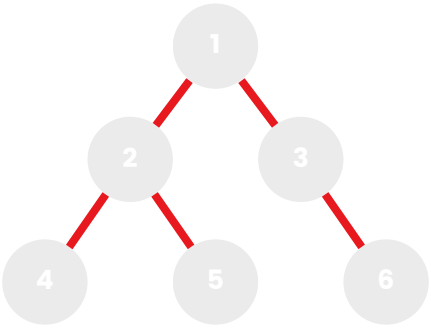
Comme le sommet 4 n'a plus de voisins et qu'on n'a pas encore parcouru tous les sommets, on retourne à l'appel précédent (récursion), en l'occurrence ici, au sommet 2.




```
chemin = [1,2,5]
voisins = []
```




```
chemin = [1,2]
voisins = []
```




```
chemin = []
voisins = []
```

La fonction récursive a tout depilé. LOL

DIFFICULTÉS RENCONTRÉES

Cette SAÉ a été difficile à appréhender. La compréhension du cours de graphe n'a pas toujours été évidente et réaliser nos programmes en Python fut une difficulté supplémentaire, car beaucoup de notions nous échappaient ou avaient été oubliées.

Durant la phase de développement, l'une des difficultés majeures a été de réussir à faire en sorte de ne pas repasser plusieurs fois par le même sommet/chemin. Par la suite, une fois nos programmes (plus ou moins) fonctionnels, nous avons rencontré des problèmes de capacité d'exécution : en effet, nos algorithmes implémentés ne fonctionnaient pas pour des plateaux dont les dimensions excédaient les 7 par 7. Ce problème, depuis résolu, était dû au fait qu'initialement, nos programmes choisissaient le sommet suivant de manière aléatoire.

ALGORITHME 1.

Pour ce premier algorithme, réussir à conserver le chemin et ne pas repasser sur les mêmes sommets n'a pas été chose aisée. En effet, ayant dans un premier temps opté pour un système de piles, la difficulté était de supprimer les bons sommets visités lors du retour sur trace. C'est pour cette raison que nous avons finalement redirigé notre choix vers un dictionnaire. Dans cette nouvelle configuration, chaque clé représente un sommet et ses valeurs la liste des sommets que ce dernier a visités. Notre réflexion s'est ensuite portée sur la façon dont nous pourrions procéder pour pouvoir conserver une suite de sommets afin de les revisiter plus tard. Ainsi, si ni le sommet courant, ni son parent n'ont ni l'un ni l'autre plus de voisins, on remonte au parent du parent et on vide la liste des sommets visités des deux sommets précédemment cités.

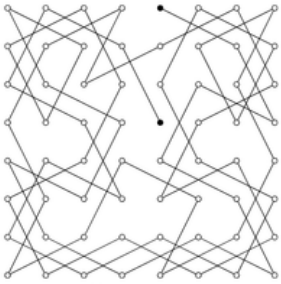
ALGORITHME 2.

Afin d'obtenir deux programmes suffisamment différents pour pouvoir en réaliser un comparatif intéressant, nous avons souhaité utiliser la récursivité pour ce second algorithme. Néanmoins, réussir à faire en sorte que la fonction ne boucle pas sur elle-même indéfiniment et continue à trouver de nouveaux sommets, même après avoir échoué sur un chemin, fut compliqué. Nous avons résolu ce problème en imbriquant deux fonctions - l'une pour trouver le chemin et l'autre pour l'utiliser. Ainsi, lorsque l'algorithme se retrouve dans une impasse, il peut retourner aux appels récursifs qui n'ont pas encore été exécutés et continuer à tester les chemins restants, tout en ayant repris le chemin qui était présent à cet appel. Il retournera donc à la case où il reste des voisins qu'il n'a pas encore essayés.

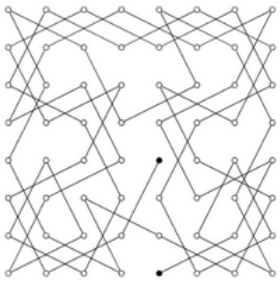
SOLUTIONS LOGIQUES

Nos deux algorithmes nous permettent d'obtenir un résultat répondant à notre problématique. Néanmoins, comme nous l'avons évoqué précédemment, il existe de nombreuses solutions à ce problème. Certaines sont d'ailleurs plutôt aisées à découvrir sans l'aide d'algorithmes, en appliquant des principes mathématiques basiques, telles que les symétries.

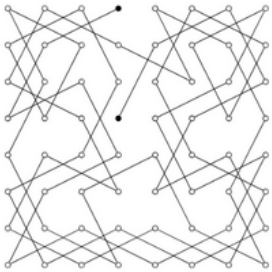
Prenons en exemple ce graphe, équivalent à un parcours sur un plateau de 64 cases :



À l'aide d'une **symétrie axiale horizontale** et **verticale**, on peut rapidement en déduire deux nouvelles solutions :

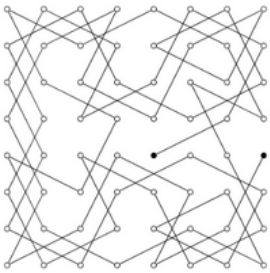


Symétrie axiale horizontale

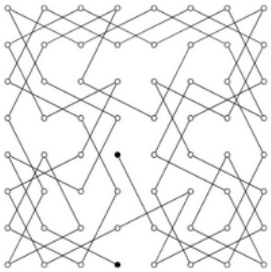


Symétrie axiale verticale

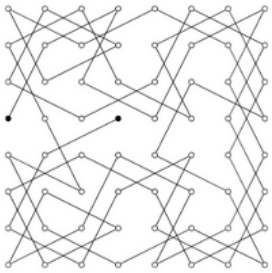
Une **rotation** par tranche de 90° (**symétrie centrale**) permet d'en trouver trois supplémentaires :



Rotation à 90°



Rotation à 180°



Rotation à 270°

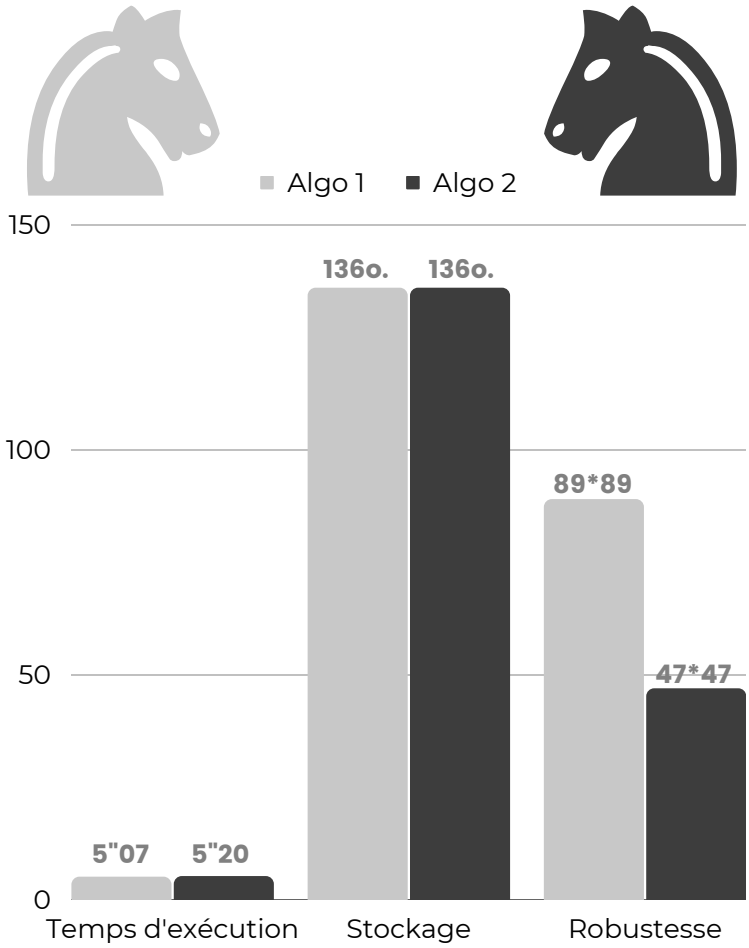
Couplées, trois nouvelles solutions nous apparaissent encore.

Pour toute solution trouvée, nous pouvons donc, uniquement par symétrie, déduire au moins huit solutions supplémentaires.

03 COMPARAISON DE PERFORMANCES.

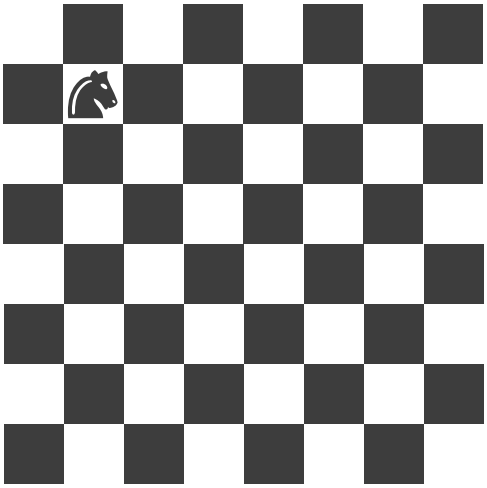
Ce comparatif se base sur trois critères :

- Le **temps d'exécution** (en secondes), calculé en itérant 50 fois sur un plateau de 25 * 25 ;
- Le **stockage** (en octets), calculé à l'aide de la fonction *getsizeof()* (de la bibliothèque *sys*) ;
- La **robustesse**, déterminée par la taille maximale du plateau que chaque programme est en mesure d'exécuter avec succès ;



Bien que les deux algorithmes aient des résultats quasiment identiques en termes de temps d'exécution et de stockage, on constate que le premier est finalement plus performant, car capable de parcourir un graphe plus de trois fois plus grand.

04 CONCLUSION.



Cette SAÉ nous a permis de découvrir le monde des graphes et certains algorithmes incontournables y étant liés, comme le **parcours en largeur**, le **parcours en profondeur** ou encore le **retour sur trace**.

Cet exercice, a été plus qu'un travail de programmation, il a été un travail de recherche en groupe, mais surtout un travail d'autonomie où chacun devait apporter sa pierre à l'édifice, dans l'objectif de rechercher des solutions.

Nos recherches nous ont donc menés à deux types de programmes vus plus tôt : la **récursivité** et **l'itération.**

Les différentes expériences réalisées dans le cadre du test de performance nous ont donc amenés à différentes modifications, améliorant grandement l'efficacité de nos programmes. La Méthode de Warnsdorff nous a notamment permis de résoudre un problème de capacité limitée d'exécution, en triant dans une liste les voisins en fonction du nombre de voisins qu'eux-même possédaient. Ce changement nous a permis de passer de deux programmes qui fonctionnaient uniquement pour des échiquiers de 7*7 maximum, à deux programmes fonctionnant pour l'un sur des échiquiers de 47*47 et l'autre sur des plateaux de jeu de 89*89. On peut donc en conclure, grâce à cet ajout, que le programme itératif a des performances bien supérieures au programme récursif, qui est un algorithme possédant un nombre d'appels limités contrairement à l'itération.

Lien vers le diaporama de soutenance :

https://www.canva.com/design/DAFfoIkVDkY/Y8PPwxhzRD5sMy-LVxomNA/edit?utm_content=DAFfoIkVDkY&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton