# Compte rendu

## SAE 2.02 Exploration algorithmique

Proposé par Mattéo Gaillard, Timéo Tribotté et Pierre Huger

# Table des matières

# 1.0 Présentation du problème

Nous avons choisi le problème du tour du cavalier car il a comme particularité d'être plutôt bien documenté ce qui nous permet de chercher facilement les informations sur le sujet. Il présente comme autre particularité celle que le cavalier ne peut pas repasser deux fois sur la même case. Cette particularité fait aussi la difficulté du problème puisqu'il faut revenir en arrière s'il se trouve dans une impasse.

**CE PROBLEME POSE DONC UNE QUESTION IMPORTANTE : EST-CE QUE NOUS SOMMES CAPABLE DE CODER UN ALGORITHME QUI RESOUT LE PROBLEME SUR UN ECHIQUIER D'AU MOINS 5X5 ?**

Les principales difficultés qui ne pourront rencontrés sont les suivantes : parcourir tout l'échiquier en suivant le parcours du cavalier suppose de créer le paterne du cavalier. Il implique aussi donc de regarder à chaque fois s'il est possible de placer un pion dans les mouvements du cavalier en vérifiant que nous ne sommes jamais passé sur celui-ci. Enfin, une des difficultés principales sera de proposer un algorithme avec un faible temps d'exécution et une faible consommation en ressource (mémoire utilisé, espace de stockage) qui permet de résoudre le problème sur un échiquier d'au moins 5 par 5.

# 2.0 Description des algorithmes

## 2.1 Description générale

Nos algorithmes utilisent la même base : une classe "Chess" permettant de modéliser l'échiquier sous forme d'une matrice ou d'un graphe.

La classe "Chess" contient :

- Un constructeur pour définir un échiquier d'une taille MxN et le type de l'échiquier (graphe ou matrice (appelé LIST dans le programme))
- Des méthodes pour définir l'échiquier sous forme d'un graphe ou d'une matrice, ajouter un point, ajouter un point de départ, afficher l'échiquier, prédire le paterne du cavalier, résoudre l'échiquier à l'aide d'un algorithme passé en paramètre, récupérer les possibles coups en suivant le paterne du cavalier, etc...

Cette classe est essentielle dans le fonctionnement de l'algorithme puisqu'elle permet d'avoir toutes les informations essentielles et utiles à l'utilisation de l'échiquier qui a été modélisé comme la position initiale du cavalier ou l'affichage de l'échiquier.

Pour modéliser un graphe, la classe “Graphe” déjà codé auparavant dans la ressource R2.07 Graphes, a été utilisé.

La matrice est modélisée par une liste de liste déjà présente en python.

## 2.2 Algorithme n°1

**CET ALGORITHME UTILISE LES MATRICES**

L’algorithme consiste en partant d’un point de départ soit saisi par l’utilisateur dans le programme soit par un point générer aléatoirement par le programme, de parcourir tout le tableau en suivant le paterne du cavalier en stockant dans un tableau les points où il est déjà passé pour ne pas y repasser. Une fois que le chemin est trouvé retourne le chemin sous forme d’une liste. Si aucune solution n’est trouvée alors l’algorithme renvoie “aucune solution trouvée”.

### 2.2.1 Étapes de réalisation

1. Le cavalier est sur une case, on le marque dans le chemin ce qui permettra de savoir s'il est déjà visité ou non.
2. Si la taille du chemin est égale à la taille de l’échiquier alors on renvoi le chemin.
3. On regarde les voisins de la case
4. Ensuite, on tri les voisins de la case de la plus petite possibilité à la plus grandes.
5. Une fois le tri fait, on parcourt les voisins de la plus petite possibilité à la plus grande.
6. Pour chaque voisin parcouru, nous vérifions le chemin fait avec le voisin (refaire étape 1 avec le voisin).
7. Si aucun chemin n’est accessible alors retirer la dernière case du chemin et retourner faux.

### 2.2.2 Algorithme en pseudo-code

```
procédure resoudre(echiquier: Echiquier, case: (x, y), chemin: liste = []) c'est
début
    chemin.ajouter(case);
    si taille(chemin) == echiquier.taille() alors
        retourner chemin;
    sinon
        voisins = cases_voisines_de(position);
        voisins_possibles = [];
        pour voisin de voisins_possibles faire
            voisins_possibles.ajouter([voisin, taille(cases_voisins_de(voisin)]);
        finfaire
        voisins_possibles.trier_par_taille();
        pour voisin de voisins_possibles faire
            retourner resoudre(echiquier, voisin, chemin);
        finfaire
    retourner faux
fin
```

### 2.2.3 Avantages et complexité temporelle

#### 2.2.3.1 Avantages

- L'algorithme permet de vérifier si le pion passe déjà sur une case ou non puisqu'il enregistre le passage du pion et donc il sait déjà à l'avance si l'utilisateur est déjà passé sur la case.
- Le fait de trier les chemins par celui qui a le moins de voisin permet de réduire le temps d'exécution puisqu'il essaie de faire le plus petit chemin.

#### 2.2.3.2 Complexité

VOICI COMMENT SERA CALCULER LA COMPLEXITE TEMPORELLE POUR LES ALGORITHMES :

- Une effectuation, l'appel de fonction et la comparaison ont une complexité de $O(1)$.
- Un calcul a une complexité de $O(2)$ puisque nous avons deux opérations : une opération $O(1)$ et une affectation $O(1)$.
- Une boucle (for, while) a une complexité algorithmique de $O(n)$.
- Les algorithmes utilisent un tri rapide qui est de complexité $O(n \cdot \log(n))$.

La complexité temporelle de l'algorithme dépend de plusieurs facteurs, tels que la taille de l'échiquier et la position de départ. Supposons que la taille de l'échiquier est **N x N** et que la position de départ est **(x, y)**.

VOICI LE CALCUL RASSEMBLANT L'ENSEMBLE DES OPERATIONS DE L'ALGORITHME : $O(1) + O(1) + O(1) + O(1) + O(1) + O(n) + O(1) + O(n \cdot \log(n)) + O(n) + O(n) + O(1)$
$= O(7) + 2O(n) + O(n \cdot \log(n)) + O(n^2) = O(n \cdot \log(n))$

*Ce calcul a été réalisé en additionnant toutes les opérations réalisées dans l'algorithmique.*

En conclusion, dans le pire des cas l'algorithme a une complexité algorithmique d'environ $O(n \cdot \log(n))$.

# 2.3 Algorithme n°2

**CET ALGORITHME UTILISE LES GRAPHES**

L'algorithme consiste à partir soit d'un point de départ d'une saisie par l'utilisateur dans le programme soit par un point générer aléatoirement par le programme. Nous allons ensuite trouver le chemin le plus rapide dans le graphe pour l'afficher. Pour cela nous allons utiliser le parcours en profondeur et faire du backtracking comme dans l'algorithme précédent. Pour réduire le temps d'exécution nous allons trier par le chemin le plus court et ne peut visiter les voisins qui ont tout leur voisin visité. Si aucune solution n'est trouvée alors nous retournons "aucune solution trouvée".

## 2.3.1 Étapes de réalisation

1. On ajoute le sommet au chemin ce qui permettra de savoir s'il a été visité.
2. Si la taille du chemin est égale à la taille de l'échiquier alors on retourne le chemin
3. On trie par le voisin avec le moins de sommet et on regarde les voisins qui n'ont pas été visitées
4. Si le voisin que l'on visite à déjà tous ses voisins visités alors on l'ignore.
5. Si ce n'est pas le cas alors on appel à nouveau l'étape 1 avec le voisin.
6. Si aucun voisin n'est trouvé alors on retire le dernier sommet du chemin puis on recommence.

## 2.3.2 Algorithme en pseudo-code

```
fonction solve(chess, sommet, path=None):
    si path est None:
        path = []

    path.append(sommet)
    si len(path) != chess.taille():
        retourner path

    voisins = [voisin pour voisin in chess.get_data().aretes(sommet) si voisin pas dans chemin]
    voisins = trier(voisins, key=lambda voisin: len([x pour x in chess.get_data().aretes(voisin) si x pas dans chemin]))
    pour voisin in voisins:
        si len(path) < taille(chess) - 1 et len([x pour x in chess.get_data().aretes(voisin) si x n'est pas dans path]) == 0:
            continuer  # si tous les voisins ont été visités, sauter ce voisin
        retourner solve(chess, voisin, path)
    path.pop()
    retourner Faux
```

## 2.3.3 Avantages et complexité temporelle

### 2.3.3.1 Avantages

- On peut aisément retourner sur le sommet précédent afin de changer de parcours.
- Le tri des chemins permet de réduire le temps d'exécution.
- L'élagage des possibilités permet de réduire le temps d'exécution.

### 2.2.3.2 Complexité

Nous n'allons pas rappeler ici les règles fixées dans le premier algorithme, pour voir les règles avant de regarder le calcul cliquer ici.

VOICI LE CALCUL RASSEMBLANT L'ENSEMBLE DES OPERATIONS DE L'ALGORITHME :

$$O(1) + O(1) + O(1) + O(n) + O(n * \log(n)) + (n) + O(1) + O(n) + o(1)$$
$$= O(5) + 2O(n) + O(n * \log(n)) = O(n \cdot \log(n))$$

*Ce calcul a été réalisé en additionnant toutes les opérations réalisées dans l'algorithmique.*

En conclusion, nous pouvons voir que l'algorithmique a une complexité algorithmique de $O(n \cdot \log(n))$.

# 3.0 Comparaison des algorithmes

## 3.1 Description générale et critères de comparaison

### 3.1.1 Description générale

Pour comparer les algorithmes nous avons créé une classe “Evaluator” permettant de :

1. Évaluer tous les algorithmes et de les ranger par temps d’exécution moyen sur un nombre d’exécution prédéfini (dans notre exemple sur 100 exécutions).
2. Évaluer le temps d’exécution entre deux algorithmes pour déterminer le plus rapide.
3. Évaluer le temps d’exécution d’un algorithme.

Cette classe nous permet d’évaluer facilement les algorithmes entre eux juste en les ajoutant dans la liste des algorithmes via une méthode prévu.

Les algorithmes sont aussi formattés d’une façon bien définie dans un dossier “algos”, où il faut pour chaque algorithme préciser si l’algorithme utilise les matrices ou bien s'il utilise les graphes ainsi que de coder dans une fonction “main” (fonction principale l’algorithme).

*Attention : cette classe ne permet pas de connaître la complexité algorithme mais seulement de mesurer le temps d’exécution d’un algorithme.*

### 3.1.2 Critères de comparaison et hypothèses

**C-1** : Les algorithmes sont comparés par le temps d’exécution en secondes.

**C-2** : Les algorithmes sont comparés par leur complexité algorithmique. (Voir complexité temporelle algo1) (Voir complexité temporelle algo2).

**C-3** : Les algorithmes seront exécutés sur la même taille d’échiquier.

**C-4** : Les algorithmes seront exécutés avec des positions initiales différentes.

**H-1** : Nous partirons du principe que les algorithmes seront exécutés sur le même environnement (la même machine).

## 3.2 Comparaison

### 3.2.1 Temps d'exécution

Nous avons pu remarquer que l'algorithme utilisant les Matrices est plus rapide en moyenne que l'algorithme utilisant les Graphes. Voici le résultat lorsque nous avons lancé l'évaluation des algorithmes sur 100 exécutions :

```
Classement des meilleures algorithmes par temps d'éxécution:
[1] • Algo de Algorithme Matrice : 0.0009s en moyenne
[2] • Algo de Algorithme Graphe : 0.0038s en moyenne
```

*Capture d'écran du programme d'évaluation des algorithmes.*

Nous avons aussi réalisé ce test entre les deux algorithmes et voici le résultat :

```
Algo de Algorithme Matrice est plus rapide.
```

*Capture d'écran du programme comparant le temps d'exécution des deux algorithmes.*

Nous pouvons constater que l'algorithme des matrices restent plus rapide.

Enfin, nous avons lancé le programme d'évaluation individuellement sur les deux algorithmes :

```
Temps d'execution pour Algo de Algorithme Matrice: 0.0087990760803222665
```

*Capture d'écran du programme affichant le temps d'exécution de l'algorithme utilisant les Matrices.*

```
Temps d'execution pour Algo de Algorithme Graphe: 0.010362863540649414s
```

*Capture d'écran du programme affichant le temps d'exécution de l'algorithme utilisant les Graphes.*

Ces deux captures d'écrans montrent que l'algorithme utilisant les Matrices est plus rapide que l'algorithme utilisant les Graphes.

En conclusion, l'algorithme utilisant les Matrices est plus rapide que l'algorithme utilisant les graphes.

### 3.2.2 Complexité algorithmique

La complexité algorithme de l'algorithme utilisant les Matrices est de $O(n \cdot \log(n))$.

La complexité algorithmique de l'algorithme utilisant les Graphes est de $O(n * \log(n))$.

En conclusion, les deux algorithmiques ont la même complexité temporelle.

## 3.3 Conclusion

Nous pouvons conclure que l'algorithme utilisant des Matrices à plus petit temps d'exécution que l'algorithme utilisant les Graphes. Malgré leur complexité algorithmique équivalente l'algorithme utilisant les Matrices est plus rapide.

Dans notre cas nous pouvons dire que l'algorithmique utilisant les matrices est plus rapide mais les deux algorithmes auront même sur un grand nombre de donnée, une grande rapidité. Il faut quand même noter que notre test peut différer selon la machine ou alors selon la position du cavalier.

# 4.0 Conclusion

## 4.1 Conclusion générale

Pour conclure, la résolution du problème nous a permis à partir d'un problème donné, d'analyser un problème pour comprendre l'enjeu et de se documenter nous-mêmes sur les attentes du problème pour pouvoir le résoudre en proposant plusieurs algorithmes.

## 4.2 Améliorations possibles

**L'algorithme 1 et l'algorithme 2** ont déjà subi les améliorations que nous imaginions. En effet nous avions imaginés pour améliorer le temps d'exécution de trier les chemins par celui qui a le moins de voisins.

## 4.3 Solution alternative

Pour la solution alternative nous allons proposer un algorithme utilisant les graphes.

Contrairement aux **algorithmes 1 et 2**, notre algorithme alternatif va chercher tous les chemins possibles, les stocker dans une liste puis prendre le chemin qui parcours au moins une fois tous les sommets et l'afficher dans l'échiquier.

Voici le code permettant de trouver tous les chemins que nous allons utiliser pour illustrer les étapes de notre solution alternative :

```
def trouve_tous_chemins(self, sommet_dep, sommet_arr, chem=None):
    if chem is None:
        chem = []
    chem.append(sommet_dep)
    if sommet_dep == sommet_arr:
        return [chem]
    chems = []
    for sommet in self.aretes(sommet_dep):
        if sommet not in chem:
            for chem in self.trouve_tous_chemins(sommet, sommet_arr, chem):
                chems.append(chem)
    return chems
```

*Capture d'écran d'un code permettant de trouver tous les chemins. Il a été réalisé en R.07 Graphes.*

Voici une explication de la solution alternative :

1. Nous récupérons tous les chemins d'une liste. Cette liste va nous permettre de trouver le chemin "idéal" (celui qui parcours tous les sommets une fois et uniquement une.
2. Nous parcourons la liste des chemins. Dès que nous trouvons un chemin parcourant une fois tous les sommets (en vérifiant que le chemin que nous regardons ne contient pas de doublons) alors nous le stockons dans une variable.
3. Une fois le chemin idéal trouvé nous l'affichons dans l'échiquier pour montrer le chemin que le cavalier emprunte.
4. Si aucun chemin n'est trouvé alors le programme renvoie "aucune solution trouvée"

**Le point faible** de cet algorithme c'est son temps d'exécution (de par la recherche de tous les chemins et aussi la sélection du chemin "idéal"). Plus le nombre de données est élevé plus l'algorithme est lent.