École Centrale Lyon

UE APPRO
Stratégies de résolution de Problèmes
Rapport

BE1 - Problème du parcours du cavalier

Élèves :
Matías Duhalde
matias.duhalde@ecl22.ec-lyon.fr

Enseignant :
Alexandre Saidi

2 octobre 2022

Table des matières

1 Introduction

Ayant un échiquier de taille n (e.g. $n \times n$), la pièce de cavalier, et sa case de départ, le but du problème consiste à trouver un parcours du cavalier qui traverse toutes les cases de l'échiquier une seule fois. Le problème est particulièrement intéressant, à cause du patron de déplacement de la pièce : elle se déplace en L, c'est-à-dire de deux cases dans une direction et ensuite une case perpendiculairement. Effectivement, il est impossible de trouver une solution quand la taille de l'échiquier est < 5 n'importe quelle soit sa case de départ (sauf le cas où $n = 1$ dont la solution est triviale), et pour les cas avec $n \geq 5$, il y a des cases de départ pour le cavalier dont le problème n'a pas de solution.

Ce *BE* et ce rapport a pour but d'utiliser des **algorithmes à essais successifs (*AES*)** pour résoudre ce problème.

2 Modélisation

Le problème du parcours du cavalier peut être modelé comme un graphe non dirigé, chaque **cas** de l'échiquier étant un **sommet**, et ses **arêtes** étant définies par quels autres **cases** peuvent être atteints dans un déplacement de cavalier. Par conséquent, ce que nous essayons de trouver correspond à un **chemin *Hamiltonien*** (un chemin qui ne visite chaque nœud du graphe qu'une seule fois). Puisque les arêtes ne sont pas pondérées, il n'y a pas de "meilleure solution", donc chacune est aussi bonne que l'autre. Donc, on peut traiter chaque étape du parcours comme un nouvel état, et de revenir à un état précédent une fois que on constate que on ne peut pas aller plus loin dans ce parcours actuel. Cela peut être réalisé par un **parcours en profondeur (*DFS*)**, en faisant un **retour arrière (*backtrack*)** chaque fois qu'on ne trouve pas une possibilité à suivre. Tout cela correspond à un *AES*.

3 Algorithme

Pour résoudre le problème, on propose trois algorithmes différents, tous basés sur **AES**. Au niveau du code, ils ont été implementés en **Python 3**.

3.1 AES naïve

3.1.1 Description

La première version de l'algorithme utilise une approche naïve. L'idée est très similaire au pseudo-code base des AES, en faisant les modifications nécessaires pour travailler avec l'échiquier, mais sans ajouter aucune heuristique. Ci-dessous on peut trouver le pseudo-code de l'algorithme utilisé. Dans le pseudo-code de l'algorithme 1, *Prometteur* vérifie si la case tentative est valide comme successeur, c'est-à-dire, la case n'a pas été déjà visitée.

Function AES_parcours_cavalier_un_succes_suffit
Données : G : un graphe sous forme de matrice,
N : Taille échiquier,
$Case\_actuelle$ (X, Y) : Les cordonnées qu'on visite dans cet appel,
Num_etape : entier numéro de la prochaine case
Résultat : Succès ou Échec (un booléen)
début
    si $Num\_etape > N^2$ **alors**
        retourner *Succès*
    sinon
        pour tous $Case\_suivante$ ***successeur de*** $Case\_actuelle$ *dans* G **faire**
            si $Prometteur(G, N, Case\_suivante)$ **alors**
                Inscrire Num_etape dans $Case\_suivante$
                si $AES\_parcours\_cavalier\_un\_succes\_suffit(G, N, Case\_suivante, Num\_etape + 1) = 1$ **alors**
                    retourner *Succès*
                fin
                Effacer Num_etape inscrit dans $Case\_suivante$
            fin
        fin
    fin
    retourner *Échec*
fin

Algorithme 1 : AES problème cavaliers naïve

3.1.2 Implémentation

Le code référencé dans cette section se trouve dans le fichier `cavaliers_v1.py`. Des outils et fonctions auxiliaires (utilisées aussi dans les autres implémentations) se trouvent dans le fichier `utils.py`.

Ci-dessous le code qui implémente l'algorithme décrit dans la dernière section. On profite de la programmation orientée objet de Python pour construire l'algorithme, étant la plupart des fonctions définies dans une classe qui conserve l'état de l'échiquier. La fonction (ou le méthode) `résoudre` du code définit le cas de départ. Aussi, pendant l'exécution du code, on enregistre des statistiques pour analyser le fonctionnement du programme, spécifiquement le nombre de tentatives et le nombre de *backtracks* qui l'algorithme fait.

```
def resoudre(self) -> bool:
    """Résoudre le problème du cavalier

    Returns:
        bool: True s'il est possible pour le cavalier de parcourir tout
    l'échiquier. Sinon, False.
    """
    return self.aes_parcour_cavalier_un_succes_suffit(self.
    case_de_depart, 2)

def aes_parcour_cavalier_un_succes_suffit(self, derniere_case_traitee:
    Case, prochain_num_etape: int) -> bool:
    """Partie récursive (AES) du problème du cavalier

```

```
    Args:
        derniere_case_traitee (Case): Case où le cavalier se trouve maintenant
        prochain_num_etape (int): étape actuelle du parcours

    Returns:
        bool: False si l'algorithme ne trouve pas une solution pour l'état actuel, sinon True
    """
    x_actuel, y_actuel = derniere_case_traitee
    if prochain_num_etape > self.nombre_de_cases:
        return True
    for x_offset, y_offset in Echiquier.TAB_DELTA_X_Y:
        nouvelle_case = (x_actuel + x_offset, y_actuel + y_offset)
        self.tentatives += 1
        if self.prometteur(nouvelle_case):
            self.fixer_case(nouvelle_case, prochain_num_etape)
            res = self.aes_parcour_cavalier_un_succes_suffit(
                nouvelle_case, prochain_num_etape + 1)
            if res:
                return True
            self.liberer_case(nouvelle_case)
            self.backtracks += 1
    return False

def prometteur(self, case: Case) -> bool:
    """Vérifier si case est valide pour le cavalier

    Args:
        case (Case): case tentative pour y aller

    Returns:
        bool: True si la case est valide, sinon False
    """
    _x, _y = case
    return 0 <= _x < self.dimension and 0 <= _y < self.dimension and not self.is_taken(case)
```

Listing 1 – Code d'algorithme AES naïve

Les fonctions `fixer_case` et `liberer_case` simplement changent la valeur de la case donnée, et `is_taken` vérifie si la case donnée est déjà occupée (sa valeur est différent à -1). Aussi, `Echiquier.TAB_DELTA_X_Y` contient les directions possibles que le cavalier peut suivre.

3.1.3 Analyse

Le code peut être exécuté en Python (version 3.7 ou supérieure requise), en utilisant la commande suivant : `python cavaliers_v1.py`. Tout de suite, le fragment de code dans `if __name__ == '__main__'` sera exécuté et demandera la dimension de l'échiquier et la case de départ à l'utilisateur. À la fin de l'exécution, on pourra voir le résultat, s'il y en a un, ou **Échec** dans le cas négatif, suivi des statistiques collectées. Un exemple est montré dans l'image 1. Le fonctionnement de toutes les autres versions du problème est la même.

La figure 2 montre les résultats de l'exécution du code avec $N = 5$, avec chaque cas

```
tracert6@DESKTOP-MPDV ~/Documents/INF/BE/cavaliers  master ± python cavaliers_v1.py
Taille de la matrice ? (>4; défaut 5) 5
Départ du cavalier ? (0<=x,y<=4; défaut 0 0) 1 1
Solution trouvée:
[[23. 12. 17.  2. 25.]
 [ 6.  1. 24. 11. 16.]
 [13. 22.  7. 18.  3.]
 [ 8.  5. 20. 15. 10.]
 [21. 14.  9.  4. 19.]]
Nombre de tentatives : 4169553
Nombre de backtracks : 521181
Temps de calcul : 2.20560359954834s
```

Figure 1 – Exemple d'exécution

de départ possible. La couleur représente si une solution a été trouvée, vert étant un succès et rouge un échec. Le premier nombre dans chaque cas est le temps d'exécution (moyenné après 10 essais), le second le nombre de **tentatives** et le troisième le nombre de ***backtracks***.

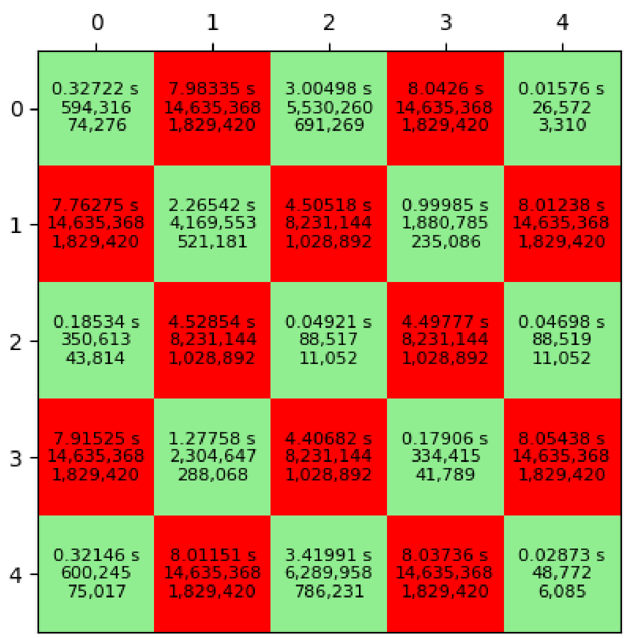

Figure 2 – Sommaire exécution de la première version avec $N = 5$

On constate que le temps d'exécution est proportionnel au nombre de tentatives et de retours en arrière. Dans les cas où il n'y a pas de solution, l'algorithme teste toutes les possibilités avant de retourner un échec, donc le temps d'exécution est généralement plutôt élevé par rapport aux cas positifs, où la solution est généralement trouvée assez tôt dans l'algorithme.

Pour $N = 6$, une solution partant de $(0, 0)$ est trouvée après 10.7 secondes, mais trouver un échec prend un temps déraisonnable. Cela ne fait qu'empirer avec un N plus élevé, donc ce rapport ne montre les résultats détaillés de cet algorithme que dans le cas où $N = 5$.

Pour obtenir un approximation de la complexité de l'algorithme, il est possible simplifier l'analyse si on assume que dans chaque case, le nombre de cases étant N^2, on aura toujours 8 autres cases disponibles (8 voisins). Donc, $8^{N^2} = 2^{3N^2} = O(2^{N^2})$. Cela correspond à une surestimation, mais il permet de voir que l'algorithme ne fonctionne pas très bien avec de grands N.

On propose des améliorations dans les sections suivantes, afin que on puisse trouver des solutions aux cas avec N plus élevé.

3.2 AES naïve avec bande

3.2.1 Description

La idée de cet algorithme est essentiellement la même au dernier, mais au niveau du code, une petite optimisation à été faite, décrite dans la section au-dessous.

3.2.2 Implémentation

Le code de Python se trouve dans le fichier `cavaliers_v2.py`. La différence se trouve au moment de construire la matrice qui représente l'échiquier : on ajoute un bande autour de la matrice qui représente l'échiquier.

```python
def creer_matrice_avec_bandes(dim: int) -> np.ndarray:
    """Créer une matrice avec bandes carrée de taille dim remplie de -1

    Args:
        dim (int): dimension de la matrice à créer (sans compter les bandes)

    Returns:
        np.ndarray: matrice résultant
    """
    m_base = np.zeros((dim + 4, dim + 4))
    for i in range(2, dim + 2):
        for j in range(2, dim + 2):
            m_base[i][j] = -1
    return m_base
```

Listing 2 – Création de l'échiquier avec bande

Malgré l'augmentation du nombre d'opérations au moment de construire la matrice, le résultat de ceci est que nous économisons quatre comparaisons en vérifient que la case tentative (fonction *prometteur*) soit valide.

```python
def prometteur(self, case: Case) -> bool:
    """Vérifier si case est valide pour le cavalier

    Args:
        case (Case): case tentative pour y aller

    Returns:
        bool: True si la case est valide, sinon False
    """
    return not self.is_taken(case)
```

Listing 3 – Validation de la case (cas avec bande)

3.2.3 Analyse

La logique générale de l'algorithme est la même, donc analyser à nouveau le résultat serait redondant. Cependant, nous nous intéressons par une éventuelle amélioration de la vitesse d'exécution du code.

La figure 3 montre les résultats de l'exécution du code de cette version avec $N = 5$, avec chaque cas de départ possible. La notation est la même à la figure 2.

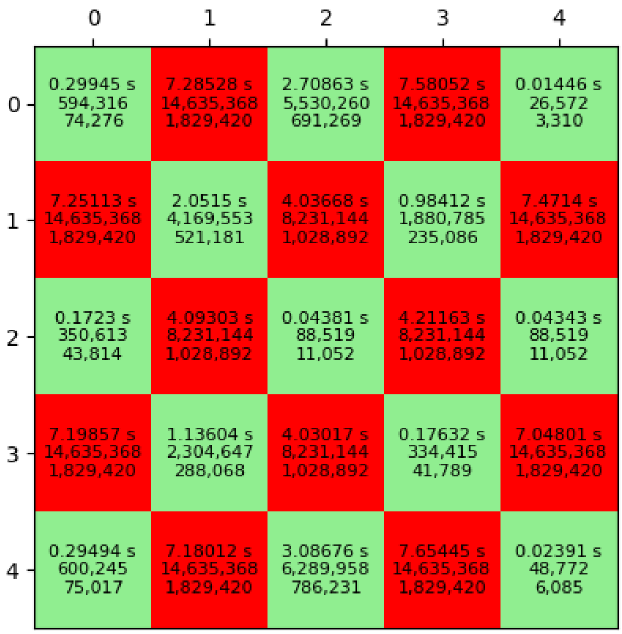

FIGURE 3 – Sommaire exécution de la deuxième version avec $N = 5$

Nous pouvons voir que la vitesse d'exécution du code est légèrement meilleure, grâce a l'amélioration décrit dans la dernière section. En moyenne, le temps d'exécution a diminué de 10%. Tous les autres statistiques (nombre de tentatives et retours arrières) restent les mêmes, car l'algorithme en essence est le même auquel de la première version.

3.3 AES avec heuristique

3.3.1 Description

Pour améliorer l'algorithme, on ajoute un **heuristique**. Dans chaque itération, pour définir un successeur, au lieu de choisir n'importe quelle case, on ira vers celui qui a le moins de **voisins**. Le nombre de voisins est définit par la quantité de cases **libres** où on peut aller d'une case donnée.

3.3.2 Implémentation

Le code de cette version se trouve dans le fichier `cavaliers_v3.py`. Pour savoir le nombre de voisins lors de l'exécution de l'algorithme, il est plus pratique de le maintenir pour chaque case que de le recalculer dans chaque itération. Pour le faire, on utilise une matrice dont le valeur de chaque case correspond au nombre de voisins de la case. La matrice est définie par le code suivant.

```
def obtenir_matrice_de_voisins(self) -> np.ndarray:
    """Obtenir la matrice de voisins de l'échiquier

    Returns:
        np.ndarray: Matrice de voisins calculée
    """
    voisins = np.zeros((self.dimension, self.dimension))
    for i in range(self.dimension):
        for j in range(self.dimension):
            for x_offset, y_offset in Echiquier.TAB_DELTA_X_Y:
                # echiquier a une bande donc il faut ajouter 2
                x_suivant = i + x_offset + 2
                y_suivant = j + y_offset + 2
                if self.prometteur((x_suivant, y_suivant)):
                    voisins[i, j] += 1
    return voisins
```

Listing 4 – Création de la matrice de voisins

Pendant l'exécution du code, chaque fois qu'on déplace le cavalier, il faut mise à jour le nombre de voisins. La même chose s'applique lors du retour en arrière.

```
def mise_a_jour_voisins(self, position_actuel: Case, inverse=False) ->
    None:
    """Mise à jour la matrice de voisins quand la position du cavalier
    change

    Args:
        position_actuel (Case): position du cavalier
        inverse (bool, optional): Si c'est False, la fonction réduira la
     quantité de voisins.
            Si c'est True, la fonction augmentera la quantité de voisins
    . Defaults to False.
    """
    _x, _y = position_actuel
    ajout = 1 if inverse else -1
    for x_offset, y_offset in Echiquier.TAB_DELTA_X_Y:
        x_suivant, y_suivant = _x + x_offset, _y + y_offset
        if self.prometteur((x_suivant, y_suivant)):
            # matrice de voisins n'a pas de bande donc il faut
    soustraire 2
            self.__voisins[x_suivant - 2][y_suivant - 2] += ajout
```

Listing 5 – Mise à jour de la matrice de voisins

Noter qu'on utilise un booléen pour définir si la fonction doit augmenter ou diminuer le nombre des voisins.

Pour décider à quelle case on déplacera le cavalier, il faut trouver les meilleurs voisins, enfin, les cases qui ont le moins de voisins. Pour le faire en temps linéaire, premièrement on trouve la plus petite valeur, et après on trouve tous les cases voisines avec cette valeur.

```
def trouver_meilleurs_voisins_libres(self, position_actuel: Case) ->
    list[Case]:
    """Trouver la liste de meilleurs voisins libres d'une case
```

```python

    Args:
        position_actuel (Case): case d'où on retrouve les voisins

    Returns:
        list[Case]: liste de meilleurs voisins trouvée
    """
    k_min = self.trouver_le_premier_voisin_de_degre_minimal_libre(position_actuel)
    if k_min >= 0:
        return self.trouver_tous_les_meilleurs_voisins(position_actuel, k_min)
    # Il n'y a pas de voisins
    return []

def trouver_le_premier_voisin_de_degre_minimal_libre(self, position_actuel: Case) -> int:
    """Trouver le degré minimal libre (la valeur de la case avec le moins de voisins)

    Args:
        position_actuel (Case): case d'où on retrouve les voisins

    Returns:
        int: valeur de la case avec le moins de voisins
    """
    _x, _y = position_actuel
    # on commence par le maximum (9)
    k_min = 9
    for x_offset, y_offset in Echiquier.TAB_DELTA_X_Y:
        x_suivant, y_suivant = _x + x_offset, _y + y_offset
        if self.prometteur((x_suivant, y_suivant)):
            # matrice de voisins n'a pas de bande donc il faut soustraire 2
            if k_min > self.__voisins[x_suivant - 2][y_suivant - 2]:
                k_min = self.__voisins[x_suivant - 2][y_suivant - 2]
    return k_min

def trouver_tous_les_meilleurs_voisins(self, position_actuel: Case, k_min: int) -> list[Case]:
    """Trouver la liste de voisins de la position actuelle où le nombre de voisins est égal à
        k_min

    Args:
        position_actuel (Case): case d'où on retrouve les voisins
        k_min (int): valeur désirée

    Returns:
        list[Case]: liste de voisins trouvée
    """
    meilleurs_voisins = []
    _x, _y = position_actuel
    for x_offset, y_offset in Echiquier.TAB_DELTA_X_Y:
        x_suivant, y_suivant = _x + x_offset, _y + y_offset
        if self.prometteur((x_suivant, y_suivant)):
            # matrice de voisins n'a pas de bande donc il faut
```

```
    soustraire 2
                if k_min == self.__voisins[x_suivant - 2][y_suivant - 2]:
                    meilleurs_voisins.append((x_suivant, y_suivant))
     return meilleurs_voisins
```

Listing 6 – Trouver meilleurs voisins

Noter que toutes les opérations précédentes, à l'exception de la construction de la matrice de voisins, prennent un temps linéaire, puisqu'elles parcourent toutes la liste des mouvements possibles du cavalier (toujours de longueur 8), et effectuent des opérations à temps constant dans chaque boucle. La construction de la matrice de voisins prend temps quadratique.

Finalement. Au lieu d'aller dans toutes les cases possibles (comme dans l'algorithme naïve), on seulement ira dans celles qui sont les meilleures (les meilleurs voisins). La fonction de récursion change en conséquence.

```
def aes_parcour_cavalier_un_succes_suffit(self, derniere_case_traitee:
    Case,
                                            prochain_num_etape: int) ->
    bool:
     """Partie récursive (AES) du problème du cavalier

     Args:
         derniere_case_traitee (Case): Case où le cavalier se trouve
    maintenant
         prochain_num_etape (int): étape actuelle du parcours

     Returns:
         bool: False si l'algorithme ne trouve pas une solution pour l'é
    tat actuel, sinon True
     """
     if prochain_num_etape > self.nombre_de_cases:
         return True
     meilleurs_voisins = self.trouver_meilleurs_voisins_libres(
    derniere_case_traitee)
     for nouvelle_case in meilleurs_voisins:
         self.mise_a_jour_voisins(nouvelle_case)
         self.tentatives += 1
         if self.prometteur(nouvelle_case):
             self.fixer_case(nouvelle_case, prochain_num_etape)
             res = self.aes_parcour_cavalier_un_succes_suffit(
                 nouvelle_case, prochain_num_etape + 1)
             if res:
                 return True
             self.liberer_case(nouvelle_case)
             self.mise_a_jour_voisins(nouvelle_case, True)
             self.backtracks += 1
     return False
```

Listing 7 – AES avec heuristique (voisins)

3.3.3 Analyse

Cette heuristique est irrévocable, c'est-à-dire, il n'aura pas de retours arrières si on suive le chemin des cases avec le moins de voisins, sauf dans le cas où il y a plusieurs meilleurs voisins. Si cela arrive, il existe la possibilité que l'algorithme retourne au futur (*backtrack*) et suive un autre meilleur voisin. Alors, si on assume qu'il n'y a qu'un meilleur voisin dans chaque récursion, on trouvera une solution sans faire un retour arrière. Donc, la complexité du *best-case* est $O(N^2)$. Cela peut être vérifié empiriquement. En regardant les figures 4, 5 et 6, on trouve que les cas où le nombre de retours arrières est 0, la quantité de tentatives est de $N^2 - 1$.

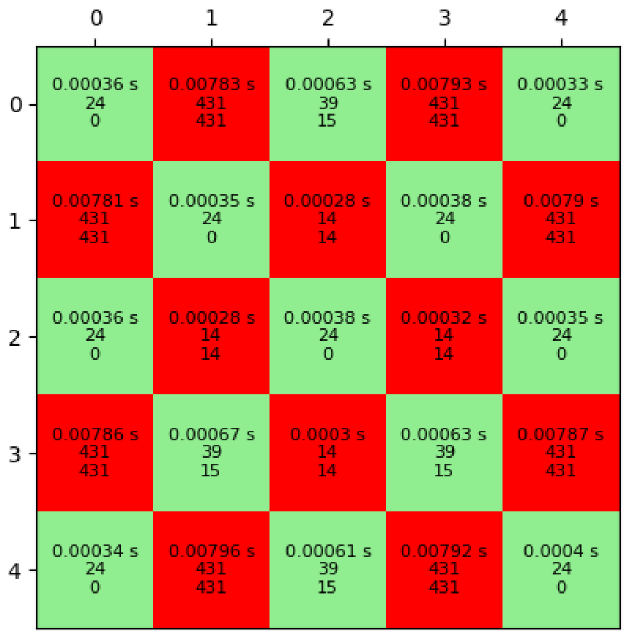

FIGURE 4 – Sommaire exécution de la troisième version avec $N = 5$

Il faut noter que dans les cas où il n'y a pas de solution, l'algorithme prend en général plusieurs tentatives pour le conclure, donc cet algorithme est n'est efficient que dans les cas positifs. Néanmoins, si on analyse les grilles de résultats (voir figures 4, 5 et 6) on peut noter la régularité suivante : quand $N > 4$ et N est pair, toutes les cases de départ ont une solution. Quand $N > 4$ et N est impair, seulement les cases blanches ont une solution. Une case blanche est laquelle où, si ses coordonnées sont (X, Y) (en partant de 0), $X + Y$ est pair.

Donc, on pourrait faire une quatrième version de l'algorithme qui vérifie la dimension et la case de départ avec cette règle avant de utiliser le **AES**.

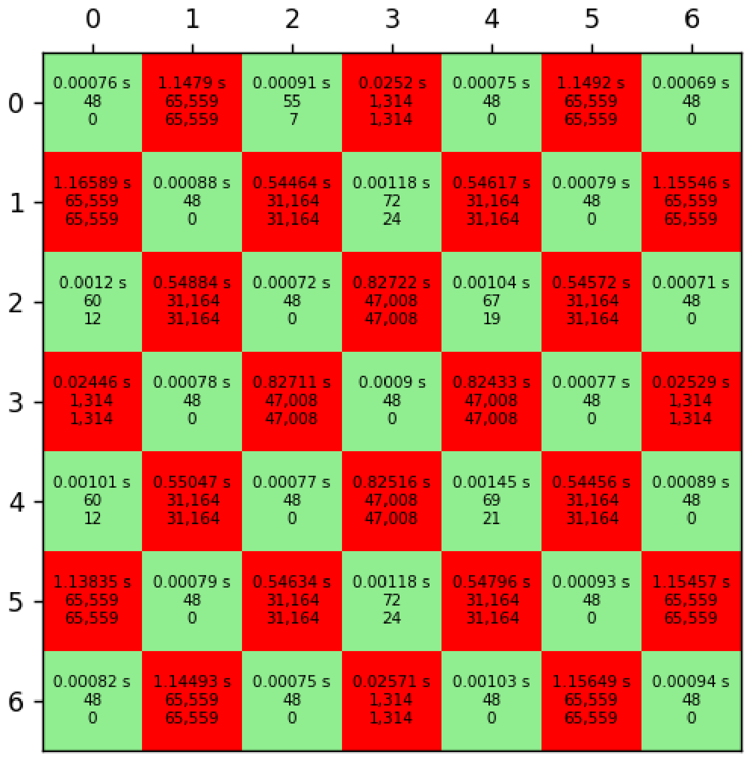

FIGURE 5 – Sommaire exécution de la troisième version avec $N = 7$

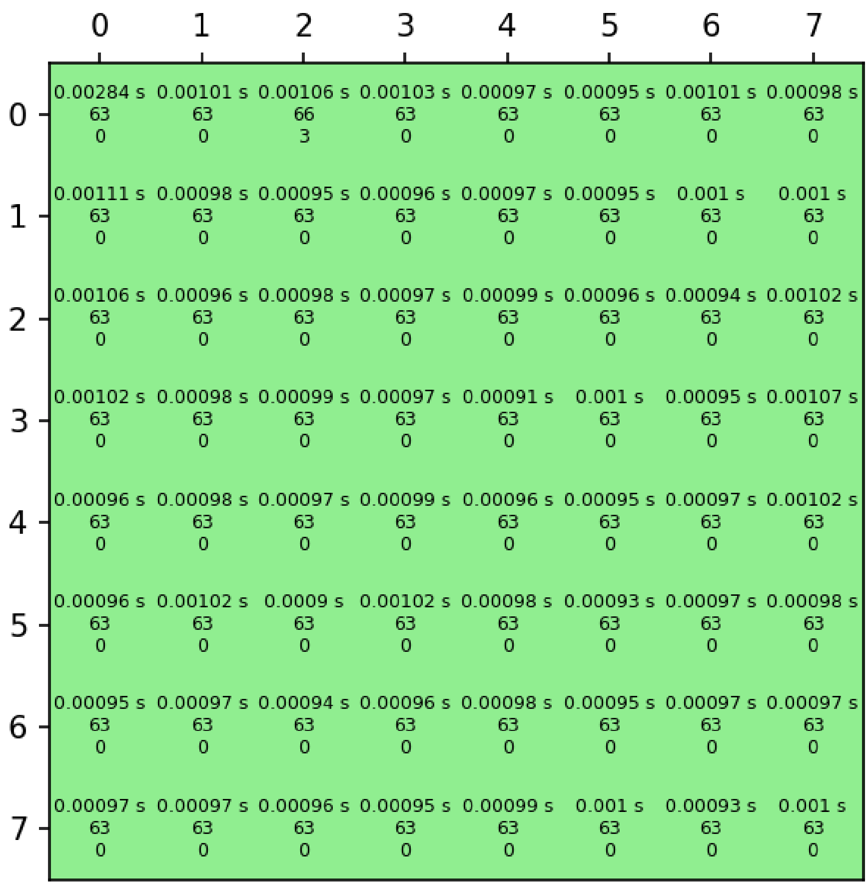

FIGURE 6 – Sommaire exécution de la troisième version avec $N = 8$

4 Conclusion

Dans ce rapport on a pu trouver un algorithme performant par les cas positifs, et on a proposé une version qui identifie tout de suite les cas sans solution.

Bien que la gestion d'opérations spécifiques à l'intérieur de l'algorithme soit importante et cela puisse apporter une petite amélioration (comme on a pu vérifier en ajoutant une bande à l'algorithme naïve), une vraie amélioration ne peut être obtenue qu'en essayant de repenser l'algorithme pour réduire sa complexité.

Les solutions naïves sont en général faciles de trouver (par rapport à une solution avec heuristique), mais aussi peu performantes, surtout dans les problèmes dont la taille est grande (un grand N, dans ce problème en particulier). Néanmoins, elles servent a mieux comprendre le problème et peuvent aider à trouver des heuristiques.