# 2I006
# Projet : Le Robot Trieur
# Première partie : Algorithme "au plus proche"

Ariana Carnielli et Lisa Kacel

## 1 Introduction

Ce projet s'intéresse à la recherche automatique des solutions pour le jeu du "robot trieur". Dans ce jeu, un robot se déplace dans un grille à $m$ lignes et $n$ colonnes. Chaque case possède une couleur de fond et peut comporter une pièce colorée, le nombre de cases et pièces d'une même couleur étant identique. Le robot peut porter au plus une pièce, il peut se déplacer d'une case à toutes les cases voisines, et peut échanger la pièce qu'il porte contre la pièce dans la case où il est, prendre la pièce de la case où il est s'il ne porte pas déjà une pièce, ou laisser la pièce qu'il porte dans la case où il est si celle-ci n'a pas déjà une pièce. L'objectif du jeu est de ranger chaque pièce dans une case de même couleur. Une case comportant une pièce de même couleur que son fond est appelée *noire*.

L'implémentation du jeu a été fournie pour ce projet. L'objectif de cette première partie du projet est de donner quelques implémentations différentes d'un algorithme pour résoudre le problème, appelé algorithme "au plus proche", et comparer leurs complexités et vitesses d'exécution.

L'algorithme "au plus proche" consiste à, tant qu'il reste une case non-noire, vérifier si le robot porte une pièce ou pas. S'il porte une pièce, le robot se déplace à la case la plus proche avec la même couleur de fond que la pièce et la dépose. Sinon, il cherche la plus proche case non-noire avec une pièce et se déplace pour la prendre. En cas d'égalité de distance, la priorité est donnée à la case en haut, à gauche. Les implémentations diffèrent par la méthode utilisée pour trouver la case la plus proche où se déplacer.

Il est intéressant à noter que cet algorithme ne donne pas forcément la solution avec le nombre minimal de déplacements. En effet, considérons l'exemple de la grille de la Figure 1. On note les déplacements du robot vers le haut, le bas, la gauche et la droite respectivement par `U`, `D`, `L` et `R`, et l'échange de pièce du robot par `S`. Si le robot se trouve initialement à la case en haut à gauche, une application de l'algorithme "au plus proche" donne la solution suivante, à 10 déplacements :

`S R S L S R R S R S D S L S L S U R S`

Néanmoins, une autre solution possible, à 7 déplacements, est :

`S R R S R S D S L S L S U S`

On remarque par cet exemple que chercher à chaque fois la case la plus proche avec la même couleur peut ne pas donner la solution optimale.

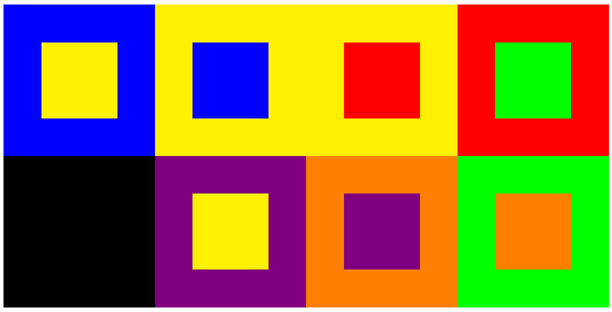

FIGURE 1 – Exemple de grille où l'algorithme "au plus proche" ne donne pas le nombre minimal de déplacements.

## 2 Version naïve

La première méthode utilisée pour chercher la case la plus proche est celle dite *naïve*, qui consiste à parcourir, à chaque fois, toute la grille du jeu à partir de la case $(0,0)$ afin de trouver la case la plus proche à la case où le robot se trouve. On a pour cela implémenté plusieurs fonctions utiles dans une bibliothèque de fonctions : la fonction `PlusCourtChemin` de la Question 1.2 et des petites fonctions qui aident à rendre le code plus lisible. La plupart d'entre elles retourne des booléens utilisés pour décider où le robot va se déplacer. En particulier, on a implémenté des fonctions répondant à la Question 1.3.

On a implémenté deux fonctions `RechercheCaseNaif_c` et `RechercheCaseNaif_nn` qui retournent les coordonnées $(k,l)$, respectivement, de la case la plus proche d'une case $(i,j)$ donnée et dont la couleur de fond est une couleur $c$ donnée, et de la case la plus proche de $(i,j)$ non-noire et comportant une pièce. On a utilisé ces deux fonctions et les fonctions données pour implémenter une résolution par l'algorithme "au plus proche". On a aussi crée un programme principal `Solveur` qui prend en argument les dimensions de la grille, le nombre de couleurs, la graine pour le générateur aléatoire et le numéro du solveur à utiliser (1 pour le solveur naïf, d'autres numéros ayant été ajoutés pour les solveurs implémentés dans les sections suivantes) et qui enregistre la solution trouvée dans un fichier.

Pour toutes les simulations de temps d'exécution de ce rapport, nous avons désactivé l'enregistrement de la solution (après avoir vérifié que la solution enregistrée était bonne) afin de mesurer uniquement le temps de calcul et ne pas avoir des structures de solution occupant trop de mémoire pour les dimensions importantes de grille. Le nombre de couleurs a été choisi comme étant égal à 10 pour toutes les simulations, et le temps d'exécution donné dans les Tables 1 à 4 correspond à une moyenne sur 4 simulations pour chaque dimension de la grille.

Expérimentalement, en utilisant la fonction `clock` de la bibliothèque `time.h`, on a trouvé les temps d'exécution de la Table 1. On remarque que le temps moyen d'exécution dépasse les $30s$ pour des grilles de taille d'environ $270 \times 270$.

| Dimensions de la grille | Temps moyen d'exécution |
|---|---|
| $250 \times 250$ | $22{,}62s$ |
| $260 \times 260$ | $26{,}89s$ |
| $270 \times 270$ | $30{,}37s$ |

Table 1 – Temps d'exécution de la version naïve.

# 3 Version circulaire

La méthode circulaire de recherche de la case $(k, l)$ la plus proche d'une case $(i, j)$ donnée consiste à démarrer la recherche depuis $(i, j)$ et chercher de façon circulaire, en augmentant la distance de 1 à chaque fois jusqu'à ce qu'une case avec la propriété voulue soit trouvée. Par rapport à la version naïve, l'avantage est que l'on peut arrêter la recherche dès qu'une première case est trouvée, sans avoir besoin de parcourir toute la grille.

On a implémenté les deux fonctions `RechercheCaseCirculaire_c` et `RechercheCaseCirculaire_nn`, analogues à celles implémentées pour la version naïve, et un solveur les utilisant. Les résultats expérimentaux de temps moyens d'exécution sur 4 simulations sont données dans la Table 2, qui les compare avec les résultats de la Table 1. On remarque que le temps d'exécution moyen de la version circulaire est plus petit, car la recherche s'arrête dès qu'une case a été trouvée. On peut donc résoudre des grilles de dimension jusqu'à environ $330 \times 330$ en moins de $30s$.

| Dimensions de la grille | Temps moyen d'exécution, version naïve | Temps moyen d'exécution, version circulaire |
|---|---|---|
| $250 \times 250$ | $22{,}62s$ | $10{,}02s$ |
| $260 \times 260$ | $26{,}89s$ | $10{,}64s$ |
| $270 \times 270$ | $30{,}37s$ | $12{,}73s$ |
| $320 \times 320$ | | $26{,}88s$ |
| $330 \times 330$ | | $30{,}94s$ |
| $340 \times 340$ | | $34{,}98s$ |

Table 2 – Temps d'exécution des versions naïve et circulaire.

On a aussi fait des tests en augmentant le nombre de couleurs pour une taille fixé de la grille. On observe que, comme attendu, le temps d'exécution de la méthode naïve reste constant, car il parcourt toujours toutes les cases de la grille pour une recherche. Néanmoins, le temps d'exécution de la méthode circulaire augmente, car, avec plus de couleurs, il lui faut en moyenne chercher plus loin pour trouver une case.

Pour la complexité de la méthode naïve dans le cas d'une grille $n \times n$, la recherche parcourt toujours toutes les $n^2$ cases de la grille, et est ainsi effectuée en $\Theta(n^2)$. Pour résoudre la grille, il faut effectuer cette recherche une fois pour chaque pièce de la grille, soit $n^2$ pièces, donc la complexité est en $\Theta(n^4)$.

Pour la méthode circulaire, le pire cas correspond à lorsque la case recherchée est le plus éloignée possible de la case courante, auquel cas il faut parcourir toutes les $n^2$

cases de la grille, mais elle peut être plus courte, lorsque par exemple la case recherchée est voisine de la case courante. On a donc une complexité de $O(n^2)$. Comme pour la méthode naïve, il faut effectuer la recherche une fois pour chacune des $n^2$ pièces, donnant donc une complexité en $O(n^4)$.

Les expérimentations permettent de retrouver cette évaluation de la complexité. Pour la méthode naïve, en utilisant la première et la troisième lignes du tableau, on obtient $\frac{31,36s}{22,94s} \approx 1,34$ et $\left(\frac{270}{250}\right)^4 \approx 1,36$, ce qui valide la complexité en $\Theta(n^4)$. De même, pour la méthode circulaire, en utilisant la première et la dernière lignes du tableau, on obtient $\frac{34,98s}{10,02s} \approx 3,49$ et $\left(\frac{340}{250}\right)^4 \approx 3,42$, ce qui valide également le calcul théorique de la complexité en $O(n^4)$.

# 4 Version par couleur

La méthode par couleur consiste à créer un tableau de dimension 1 et longueur égale à la quantité de couleurs et contenant dans chaque case une liste doublement chainée avec les indices des cases ayant une même couleur de fond. Pour effectuer la recherche de la case la plus proche d'une couleur donnée, il suffit de parcourir la liste chainée correspondante. On ne fait donc un parcours complet de la grille qu'une seule fois, pour la construction du tableau de listes. Néanmoins, on utilise encore la méthode circulaire pour trouver une case non-noire contenant une pièce.

On a commencé par implémenter une bibliothèque de fonctions pour la manipulation de listes chainées, avec des fonctions d'initialisation, insertion en queue en $\Theta(1)$, suppression en $\Theta(1)$, affichage, etc. Ensuite, on a implémenté la fonction `LDC rechercherPlusProcheCase` de recherche de case plus proche d'une case $(i, j)$ passée en argument. Cette fonction parcourt toute la liste chainée passée en argument pour trouver la case la plus proche.

En utilisant cette fonction et la fonction `RechercheCaseCirculaire_nn` de la partie précédente, on a implémenté un nouveau solveur pour résoudre le robot trieur. Les résultats expérimentaux des temps moyen d'exécution sont donnés dans la Table 3. On remarque que le temps d'exécution moyen est beaucoup plus petit que pour les deux autres versions, étant de l'ordre de $30s$ pour des grilles de taille d'environ $590 \times 590$.

Étudions la complexité de la méthode par couleur en fonction de $n$ et du nombre maximal de pièces d'une même couleur $\alpha$. Pour initialiser le tableau de listes chainées, il faut parcourir la grille une fois, la complexité de cette étape étant donc en $\Theta(n^2)$. Pour rechercher la case la plus proche de couleur $c$, il faut parcourir une fois la liste chainée de la couleur correspondante, et alors la complexité est $\Theta(\ell_c)$, où $\ell_c$ est la longueur de cette liste. Comme $\alpha$ est la taille de la plus grande liste, on obtient ainsi une complexité en $O(\alpha)$ pour la recherche. Comme il faut répéter la recherche pour les $n^2$ pièces, en prenant en compte aussi l'initialisation, la complexité totale est en $\Theta(n^2) + O(n^2\alpha) = O(n^2\alpha)$. La taille $\alpha$ de la plus grande liste est bornée par le nombre de pièces $n^2$ et ainsi, en fonction de $n$, on a une complexité en $O(n^4)$.

| Dimensions de la grille | Temps moyen, version naïve | Temps moyen, version circulaire | Temps moyen, version par couleur |
|---|---|---|---|
| $250 \times 250$ | $22{,}62s$ | $10{,}02s$ | $1{,}055s$ |
| $260 \times 260$ | $26{,}89s$ | $10{,}64s$ | $1{,}176s$ |
| $270 \times 270$ | $30{,}37s$ | $12{,}73s$ | $1{,}441s$ |
| $320 \times 320$ | | $26{,}88s$ | $2{,}984s$ |
| $330 \times 330$ | | $30{,}94s$ | $3{,}113s$ |
| $340 \times 340$ | | $34{,}98s$ | $3{,}313s$ |
| $580 \times 580$ | | | $28{,}71s$ |
| $590 \times 590$ | | | $30{,}48s$ |
| $600 \times 600$ | | | $33{,}04s$ |

Table 3 – Temps d'exécution des versions naïve, circulaire et par couleur.

# 5 Version par AVL

Dans la méthode par AVL, on utilise des arbres binaires de recherche équilibrées du type AVL pour faciliter la recherche de la case la plus proche avec une couleur donnée. Plus précisément, on crée un tableau `M` d'arbres AVL de dimension `m` × `nb_coul`, où `m` est le nombre de lignes de la grille et `nb_coul` le nombre de couleurs utilisées dans la grille. L'arbre `M[i][c]` contient les indices de colonne `j` de toutes les cases de la grille dans la ligne `i` et de couleur de fond `c`.

On a commencé par l'implémentation d'une bibliothèque pour la manipulation d'arbres AVL, avec des fonctions d'initialisation, insertion en $O(\log(\beta))$ (où $\beta$ est la quantité d'éléments de l'arbre), suppression en $O(\log(\beta))$, rotations droite et gauche en $O(1)$, équilibrage d'un nœud en $O(1)$, affichage, etc. Les fonctions de cette bibliothèque ont été testées en insérant, recherchant et supprimant plusieurs éléments d'un arbre.

On a également implémenté une fonction de recherche qui, étant donné un arbre AVL à entrées positives ou nulles et un entier $c$, recherche et retourne l'entier le plus proche de $c$ dans l'arbre (ou $-1$ si l'arbre est vide). Dans le cas où deux entiers de l'arbre sont à même distance de $c$, on retourne le plus petit. Cette fonction utilise une recherche dichotomique : si la valeur $c$ est plus petite que la racine de l'arbre, alors l'élément de l'arbre le plus proche de $c$ est soit sa racine, soit un élément de son sous-arbre gauche, avec la situation symétrique lorsque $c$ est plus grand que la racine de l'arbre. Une implémentation récursive de cette fonction de recherche permet ainsi d'avoir une complexité en $O(h)$, où $h$ est la hauteur de l'arbre, et donc en $O(\log(\beta))$ puisqu'un arbre AVL est équilibré.

Pour utiliser cette structure dans l'implémentation de l'algorithme "au plus proche", on a écrit une fonction de création du tableau d'arbres AVL `M` à partir d'une grille. Cette fonction parcourt toute la grille en insérant, pour les cases non-noires, son indice de colonne dans l'arbre correspondant à sa couleur de fond. On a aussi décidé de rajouter une dernière colonne à `M` contenant, pour chaque ligne, un arbre avec les indices de toutes les cases non-noires avec une pièce de la ligne, afin de pouvoir implémenter efficacement la recherche de cases non-noires.

On a en plus implémenté des fonctions de recherche de case par couleur et de

| Dimensions de la grille | Version naïve | Version circulaire | Version par couleur | Version par AVL |
|---|---|---|---|---|
| $250 \times 250$ | $22{,}62s$ | $10{,}02s$ | $1{,}055s$ | $0{,}3672s$ |
| $260 \times 260$ | $26{,}89s$ | $10{,}64s$ | $1{,}176s$ | $0{,}4063s$ |
| $270 \times 270$ | $30{,}37s$ | $12{,}73s$ | $1{,}441s$ | $0{,}4961s$ |
| $320 \times 320$ | | $26{,}88s$ | $2{,}984s$ | $0{,}7617s$ |
| $330 \times 330$ | | $30{,}94s$ | $3{,}113s$ | $0{,}8203s$ |
| $340 \times 340$ | | $34{,}98s$ | $3{,}313s$ | $0{,}9023s$ |
| $580 \times 580$ | | | $28{,}71s$ | $4{,}629s$ |
| $590 \times 590$ | | | $30{,}48s$ | $4{,}965s$ |
| $600 \times 600$ | | | $33{,}04s$ | $5{,}281s$ |
| $1000 \times 1000$ | | | | $26{,}95s$ |
| $1050 \times 1050$ | | | | $32{,}41s$ |
| $1100 \times 1100$ | | | | $38{,}50s$ |

Table 4 – Temps d'exécution des versions naïve, circulaire, par couleur et par AVL.

recherche de cases non-noires avec une pièce afin d'implémenter l'algorithme "au plus proche" par AVL. Pour les recherches de la case la plus proche, on parcourt les lignes du tableau `M` et, pour chaque ligne, on recherche la colonne la plus proche dans l'arbre AVL de la couleur correspondante, gardant à la fin la distance minimale par rapport à toutes les lignes. Le test du temps moyen d'exécution est donné dans la Table 4.

En comparant avec la méthode par couleur, on remarque que la méthode par AVL est plus rapide. La dimension de la grille qui donne un temps moyen d'exécution de $30s$ est entre $1000 \times 1000$ et $1050 \times 1050$.

La complexité de la création du tableau d'AVL est en $O(n^2 \log(\beta))$, où $\beta$ est le nombre maximal d'éléments dans un arbre AVL du tableau. Il faut parcourir les $n^2$ éléments de la grille et les insérer dans un AVL, l'insertion étant faite, comme vu précédemment, en $O(\log(\beta))$. La complexité de la recherche de la case la plus proche par AVL est en $O(n \log(\beta))$. En effet, on fait une boucle sur les $n$ lignes du tableau `M` et, pour chaque ligne, on fait une recherche uniquement sur l'arbre de la couleur correspondante, qui se fait en $O(\log(\beta))$. Comme cette recherche est répétée pour les $n^2$ pièces de la grille, on a une complexité totale en $O(n^2 \log(\beta)) + O(n^3 \log(\beta)) = O(n^3 \log(\beta))$. Le nombre d'éléments $\beta$ du plus grand AVL est borné par le nombre de pièces $n$ d'une ligne et ainsi, en fonction de $n$, on a une complexité en $O(n^3 \log(n))$.

La Figure 2 donne les graphes des temps d'exécution de l'algorithme "au plus proche" pour les quatre méthodes précédentes. Pour chaque dimension $n \times n$ de la grille, les simulations ont été répétées 10 fois, le temps donné dans le graphique étant le temps moyen d'exécution. Comme montré par les calculs théoriques de complexité des questions précédentes, la méthode naïve est en $\Theta(n^4)$, les méthodes circulaire et par couleur sont en $O(n^4)$, et la méthode par AVL est plus rapide, en $O(n^3 \log(n))$. Parmi les méthodes qui ont la même complexité en $n^4$, les améliorations des méthodes circulaire et par couleur permettent de réduire le temps d'exécution par rapport à la méthode naïve.

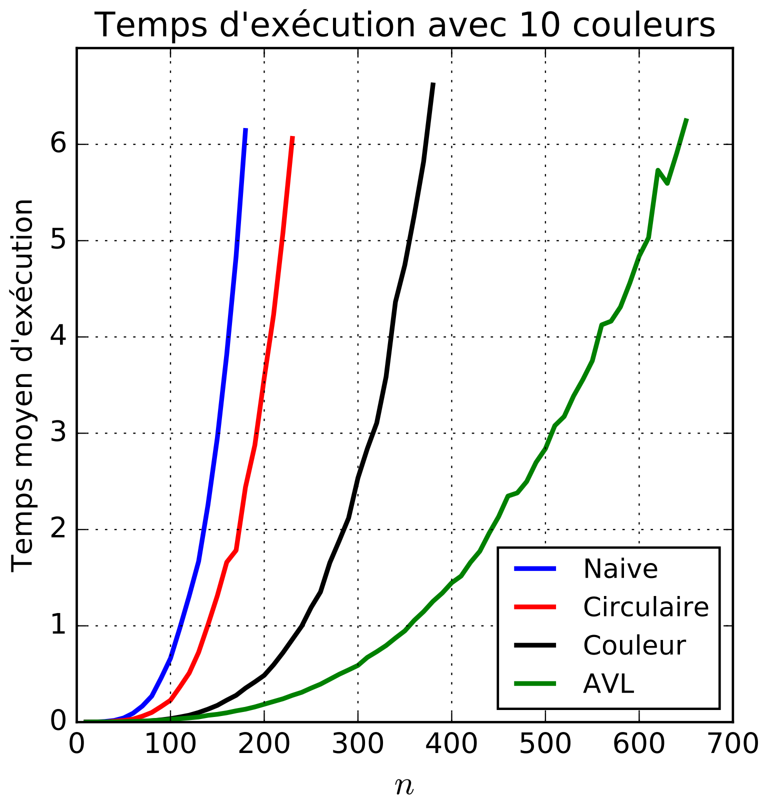


Figure 2 – Temps d'exécution des quatre méthodes différentes.