Explication des Résultats du Programme de Simulation Robot Nettoyeur

Antoine BARTCZAK, Thomas LESIEUX et Lilian BRIAUT

October 22, 2024

1 Introduction

Dans ce projet, nous avons dû concevoir une simulation d'un robot aspirateur et trouver un algorithme à la fois efficace et fonctionnel. Nous avons opté pour un **algorithme de balayage**, du même style qu'un algorithme de largeur d'abord une priorité directionnel est inscrite dans le code et pour éviter que le robot retourne sur les cases déjà nettoyer il considère toutes les cases nettoyer comme des obstacles.

Une alternative est de laisser le robot se déplacer de manière aléatoire dans toutes les directions. D'après Tobias EDWARD et Jacob SÖRME, l'algorithme le plus efficaces pour les robots aspirateurs serait **l'algorithme de déplacement aléatoire** plutôt qu'un algorithme de pathfinding structuré. Bien que le balayage permette au robot de ne passer qu'une seule fois sur chaque zone, un déplacement aléatoire a l'avantage de permettre plusieurs passages sur les zones les plus accessibles. Cela est particulièrement pertinent, car ces zones sont souvent les plus utilisées par les habitants et nécessitent donc plus de nettoyage.

L'autre choix que nous avons fait est de **quadriller la zone** pour limiter les directions possibles du robot et faciliter ainsi la programmation des mouvements. Chaque case du quadrillage équivaut à la taille du robot (environ 30 cm x 30 cm), ce qui permet de décomposer chaque pièce en zones précises. En cas de déplacement aléatoire, le fait de limiter les directions permet de réduire les mouvements dans des angles trop variés, ce qui améliore l'efficacité en concentrant les déplacements dans des zones spécifiques.

Ce compromis entre les différents algorithmes vise à garantir à la fois une couverture complète des surfaces et une gestion efficace des ressources, tout en répondant aux besoins de nettoyage de manière optimale.

Tous les codes sont déposé sur ce github : https://github.com/StarsL0l/Projet-Robot-Aspirateur/tree/main

2 Fonctions Principales

2.1 Fonction `initialize_grid()`

Cette fonction initialise la grille en plaçant des murs autour des bords et en ajoutant des obstacles aléatoires à l'intérieur de la zone de nettoyage. Voici les étapes principales :

- Création des murs : Chaque côté de la grille est entouré de murs pour éviter que le robot sorte de la zone.

- Placement des obstacles : Un nombre aléatoire d'obstacles est placé sur la grille.

Résultat attendu : Une grille avec des murs et des obstacles aléatoirement répartis.

2.2 Fonction `main()`

C'est la fonction principale qui contrôle le flux du programme. Elle :

- Initialise la grille.
- Place le robot à une position aléatoire où il y a de la poussière.
- Boucle sur les événements de jeu pour gérer les déplacements du robot.

Elle gère aussi la pile (`stack`) pour enregistrer les positions visitées et permettre au robot de revenir en arrière s'il est bloqué.

Résultat attendu : Le robot parcourt la grille et nettoie toute la poussière.

2.3 Fonction `drawGrid()`

Cette fonction dessine la grille à l'écran en utilisant Pygame. Pour chaque case de la grille, elle détermine si la case contient de la poussière, est nettoyée, ou contient un mur ou un obstacle, et dessine en conséquence.

- `Poussière` : Dessine un rectangle gris.
- `Nettoyé` : Dessine un rectangle blanc.
- `Mur` : Dessine un rectangle noir ou une texture.
- `Obstacle` : Dessine un rectangle noir ou une texture.

Résultat attendu : Une représentation visuelle de la simulation.

2.4 Fonction `moveRobot(i, j, orientation)`

Cette fonction gère le déplacement du robot sur la grille. Elle marque la case actuelle comme nettoyée et dessine le robot à sa position en fonction de l'orientation donnée.

- Si la texture est activée, elle charge et dessine une image du robot.
- Si la texture est désactivée, elle dessine un simple rectangle bleu.

Résultat attendu : Le robot nettoie la case et se déplace.

2.5 Fonction `getNextPosition(i, j, orientation)`

Cette fonction calcule la prochaine position à laquelle le robot doit se déplacer. Le robot vérifie dans l'ordre les directions suivantes pour trouver de la poussière :

1. Gauche
2. Bas
3. Haut
4. Droite

Si aucune poussière n'est trouvée, le robot reste sur place et garde son orientation actuelle.

Résultat attendu : Le robot trouve une case contenant de la poussière ou reste sur place.

2.6 Fonction `move(stack, grid, robot_x, robot_y, orientation, cleaning_finished)`

Cette fonction déplace le robot en fonction de sa position actuelle, de l'état de la grille et de son orientation. Elle gère également les retours en arrière lorsque le robot se retrouve dans une impasse. Le déplacement se fait de la manière suivante :

1. La fonction `getNextPosition(robot_x, robot_y, orientation, grid)` est appelée pour déterminer la prochaine position et orientation du robot.
2. Si le robot ne peut pas avancer (impasse détectée), deux cas se présentent :
    - Si la pile `stack` n'est pas vide, le robot revient en arrière à la dernière position sauvegardée, inverse son orientation, puis continue l'exploration.
    - Si la pile est vide, cela signifie que toutes les positions ont été explorées et le nettoyage est terminé. Le drapeau `cleaning_finished` est alors activé.
3. Si le robot peut avancer, sa position et son orientation actuelles sont sauvegardées dans la pile `stack` (sauf si elles sont déjà présentes), puis il se déplace vers la nouvelle position.

Résultat attendu : Le robot avance vers une nouvelle position ou, en cas d'impasse, il revient en arrière. Si toutes les cases ont été explorées, le nettoyage est considéré comme terminé.

3 Pseudocode

3.1 Algorithme de balayage

La première solution envisagée a été de considérer les cases déjà nettoyées comme des murs et de ne plus passer dessus. Pour ce faire, nous avons créé plusieurs fonctions. La première permet de trouver la prochaine case à nettoyer en appliquant une priorité de direction, ce qui produit un effet de balayage. La deuxième fonction s'active lorsque le

robot se retrouve coincé entre des obstacles ou des cases déjà nettoyées. Dans ce cas, le robot revient sur ses pas jusqu'à retrouver la zone de départ ou découvrir une nouvelle case à nettoyer.

Pour le flowchart de cette algorithme cf figure 1 en annexe.

3.1.1 Pseudocode de `getNextPosition(i, j, orientation)`

```
[language=Python]
Function getNextPosition(i, j, orientation)
    If grid[i - 1][j] == Poussière Then
        Return i - 1, j, 90
    End If

    If grid[i][j + 1] == Poussière Then
        Return i, j + 1, 180
    End If

    If grid[i][j - 1] == Poussière Then
        Return i, j - 1, 0
    End If

    If grid[i + 1][j] == Poussière Then
        Return i + 1, j, 270
    End If

    Return i, j, orientation
End Function
```

3.1.2 Pseudocode de `move()`

```
Function move(stack, grid, robot_x, robot_y, orientation, cleaning_finished)
    # Obtenir la prochaine position du robot
    (new_x, new_y, new_orientation) = getNextPosition(robot_x, robot_y, orientation, grid)

    # Vérifier si le robot ne peut pas avancer (impasse détectée)
    If (new_x, new_y) == (robot_x, robot_y) Then
        If stack is not empty Then  # Si le robot peut revenir en arrière
            (robot_x, robot_y, orientation) = pop(stack)  # Revenir à la dernière position
            orientation = inverse_orientation(orientation)  # Inverser l'orientation du robot
        Else
            cleaning_finished = True  # Aucun autre mouvement possible, nettoyage terminé
        End If
    Else
        # Mettre à jour l'orientation et vérifier si la position n'a pas été explorée
        orientation = new_orientation
        If stack is empty OR (robot_x, robot_y) not in stack Then
            # Ajouter la position actuelle dans la pile
            push(stack, (robot_x, robot_y, orientation))
        End If
        # Déplacer le robot à la nouvelle position
        robot_x = new_x
        robot_y = new_y
    End If

    # Retourner les mises à jour
    Return stack, grid, robot_x, robot_y, orientation, cleaning_finished
End Function
```

3.2 Approche aléatoire

La deuxième solution envisagée était l'utilisation d'une direction aléatoire, ou plutôt pseudo-aléatoire. En théorie, sur une échelle de temps infinie, le robot serait contraint de passer **partout**. Bien sûr, dans la pratique, le temps disponible ne sera pas infini, mais on peut facilement imaginer un scénario où le robot dispose de 2 heures pendant l'absence des utilisateurs, ce qui serait suffisant pour nettoyer la majorité d'une pièce de taille modérée. De plus, comme mentionné dans l'introduction, cet algorithme serait particulièrement efficace pour les robots aspirateurs, car il leur permet de passer plusieurs fois sur les zones fréquemment utilisées, garantissant ainsi un nettoyage optimal.

D'autre part, l'algorithme de balayage une grande précision dans les mouvements du robot. Il est essentiel que le robot se déplace en ligne droite pour garantir une couverture précise et une optimisation efficace du nettoyage. Sans cela, le processus pourrait devenir inefficace. En revanche, l'algorithme de déplacement aléatoire n'a pas besoin d'une telle précision, ce qui simplifie les exigences matérielles. De plus, la complexité du code est également réduite : l'algorithme de balayage nécessiterait l'ajout de contrôles comme un PID ainsi que des capteurs supplémentaires pour corriger les trajectoires, tandis que ces ajustements ne sont pas nécessaires dans le cas du déplacement aléatoire.

Pour le flowchart de cette algorithme cf figure 2 en annexe.

```
While True Do
    Fill SCREEN with color (150, 150, 150)

    // Choix aléatoire d'une direction : 0=haut, 1=droite, 2=bas, 3=gauche
    direction ← Random integer between 0 and 3

    // Initialisation du déplacement
    dx ← 0
    dy ← 0

    // Calcul du déplacement selon la direction
    If direction equals 0 Then    // Haut
        dy ← -1
        orientation ← 0
    Else If direction equals 1 Then  // Droite
        dx ← 1
        orientation ← 270
    Else If direction equals 2 Then  // Bas
        dy ← 1
        orientation ← 180
    Else                             // Gauche
        dx ← -1
        orientation ← 90
    End If

    new_x ← robot_x + dx
    new_y ← robot_y + dy
End While
```

4 Simulation matérielle

Dans une implémentation réelle, nous opterions pour une carte Arduino Uno équipée d'un capteur à ultrasons, idéal pour la détection à courte distance. Le robot serait omnidirectionnel, avec une structure circulaire et équipé de deux roues de chaque côté, lui permettant de pivoter sur lui-même, comme la plupart des versions commercialisées.

Concernant la capacité de stockage, nous nous sommes interrogés sur la possibilité pour le robot de stocker à la fois la carte de la pièce et ses déplacements. Bien que l'Arduino soit programmé en C++, un langage très différent du Python, notamment dans la gestion des tableaux dynamiques, cette différence ne devrait pas poser de problème dans la réalisation du système ni dans l'algorithme suivi.

Nous avons estimé qu'en quadrillant la pièce avec des cases de 30x30 cm, un espace de 30 m^2 pourrait être représenté dans une matrice de taille gérable. Cette matrice, qui ne contiendrait que des 0 et des 1 pour indiquer les zones nettoyées ou non, serait stockable avec une large marge de sécurité en termes de capacité mémoire. De plus, un tableau permettant de suivre les derniers déplacements du robot serait également facile à gérer en termes de mémoire et de performance.

5 Temps d'exécution

Nous avons testé nos deux algorithmes afin de comparer le temps d'exécution minimum. Comme nous le pensions, l'algorithme de balayage est le plus efficace pour découvrir entièrement une pièce.

Conditions de test :

- Pièce de 19x19 (sans les murs extérieurs)
- Pas d'obstacles
- Une seule et unique pièce

5.1 Algorithme de balayage

Cet algorithme étant fixe, le nombre de déplacements, et donc le temps d'exécution, dépend exclusivement de la taille de la pièce, soit du nombre de cases après subdivision, multiplié par 2 puisque le robot passe presque deux fois sur chaque case. Pour une pièce de 30 m^2, cela correspondrait à environ 666 déplacements.

5.2 Algorithme aléatoire

En revanche, l'algorithme aléatoire est presque impossible à prévoir (nous utilisons du pseudo-aléatoire car la graine n'est pas modifiée). Nous avons donc réalisé 10 tests sur une pièce de la même taille que celle utilisée pour l'algorithme de balayage. Nous avons tout de même apporté une légère modification au code pour ces tests, car en pratique l'objectif de l'algorithme aléatoire est de couvrir presque toute la surface. Afin d'éviter qu'il effectue 2000 mouvements pour ne trouver que quelques cases, nous avons décidé d'arrêter la simulation lorsque 95% de la pièce est nettoyée. Bien entendu, il s'agissait d'une situation particulière pour nos tests, car normalement, l'algorithme ne connaît pas son environnement.

Sur 10 essais, nous avons obtenu une moyenne de 3257 déplacements avec des extrêmes de 4861 et 1995. **Remarque :** Sans ce paramètre de 5%, les valeurs se situaient entre 6000 et 8000.

6 Conclusion

Dans cette simulation de robot aspirateur, nous avons exploré deux approches distinctes pour le déplacement du robot : l'algorithme de balayage et l'algorithme de déplacement aléatoire. Chacune de ces méthodes présente des avantages et des inconvénients qui dépendent des caractéristiques spécifiques de l'environnement et des besoins en nettoyage.

L'algorithme de balayage, qui consiste à considérer les cases nettoyées comme des obstacles, garantit une couverture systématique des surfaces. En ne repassant pas sur les zones déjà nettoyées, il permet au robot de nettoyer efficacement chaque partie de l'espace sans perte de temps. Il est bien sur envisageable de lui faire effectuer plusieurs cycle.

À l'inverse, l'algorithme de déplacement aléatoire, bien qu'il puisse sembler moins structuré, présente l'avantage de permettre au robot de passer plusieurs fois sur les zones les plus accessibles. Cela peut être bénéfique dans des espaces fréquemment utilisés, où le nettoyage régulier est essentiel. Cependant, cette approche peut également mener à des inefficacités, car le robot risque de ne pas couvrir certaines zones, notamment dans des configurations complexes ou à plusieurs pièces, rendant ainsi une couverture complète presque impossible ou extrêmement longue.

Bien sûr, les complexités matérielles ne sont pas les mêmes. L'algorithme de balayage nécessite une certaine précision et, par conséquent, davantage de capteurs. En revanche, l'algorithme aléatoire ne demande aucune précision particulière et nécessite donc moins d'équipement.

Nous avons également pu comparer leur efficacité. Le balayage s'avère nettement plus efficace, en demandant trois fois moins de temps. Peut-on alors affirmer que, sur une même échelle de temps, et en faisant fonctionner le robot en continu, quel que soit l'algorithme choisi, l'un serait meilleur que l'autre ? Là où le balayage assure une couverture uniforme, quelles que soient les zones plus ou moins sales, on pourrait supposer que l'algorithme aléatoire pourrait passer une seule fois sur certaines zones et 3 à 4 fois sur d'autres, favorisant ainsi le passage sur les zones les plus sales plutôt que sur les zones propres.

Finalement, le choix entre l'algorithme de balayage et l'algorithme aléatoire dépend des contraintes matérielles ainsi que des priorités en matière de nettoyage. Si l'objectif est d'assurer une couverture systématique et complète des surfaces les plus complexes, l'algorithme de balayage est préférable. En revanche, si le besoin est de concentrer le nettoyage sur les zones les plus utilisées ou sur une pièce en particulier, l'algorithme aléatoire peut s'avérer plus efficace. En somme, le compromis entre ces différentes approches est essentiel pour optimiser le fonctionnement d'un robot aspirateur, tout en répondant aux exigences spécifiques des utilisateurs.

Annexes

Figure 1: Algorithme de balayage

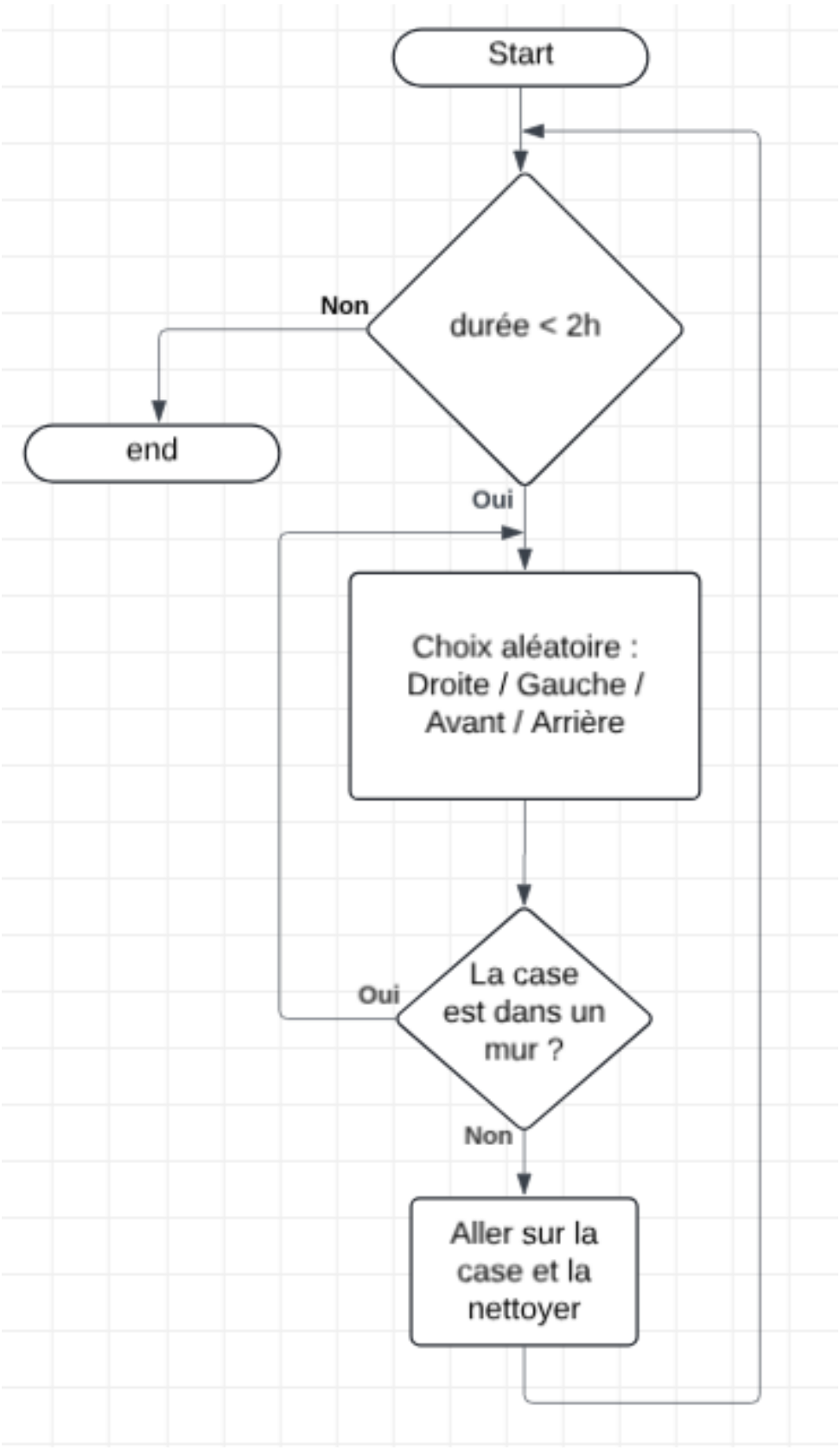

Figure 2: Algorithme aléatoire