Projet MOGPL

Mariana Duarte Ferreira (21501059), Caroline Zouloumian (21501928)

Décembre 2025

1 Introduction

Dans le cadre de ce projet, nous nous intéressons à l'optimisation du déplacement d'un robot utilisé dans le dêpot d'un grand magasin pour le transport d'objets. L'objectif principal est de minimiser le temps de transport du robot entre un point de départ et un point d'arrivée donnés.
Le robot se déplace uniquement en ligne droite sur une grille rectangulaire de taille $N \times M$. Il est contrôlé par deux types de commandes : avancer de n cases pour $n \in \{1, 2, 3\}$, ou de tourner à gauche ou à droite. Chaque commande nécessite une seconde pour s'exécuter.
Le but de notre projet est de développer un programme capable de déterminer le temps minimal nécessaire pour que le robot atteigne son point d'arrivée à partir d'un point de départ donné.

2 Modélisation

Pour une matrice d'entrée de taille $M \times N$, nous construisons le graphe orienté (V, A) défini comme suit. Comme le robot se déplace le long des lignes de la grille, nous considérons les coordonnées des sommets allant de 0 à M pour les lignes et de 0 à N pour les colonnes.

$$\text{Direction}(d) = \begin{cases} \text{Nord} & \text{si } d = 0, \\ \text{Est} & \text{si } d = 1, \\ \text{Sud} & \text{si } d = 2, \\ \text{Ouest} & \text{si } d = 3. \end{cases}$$

$$\Delta(d) = \begin{cases} (-1, 0) & \text{si } d = 0 \text{ (Nord)}, \\ (0, 1) & \text{si } d = 1 \text{ (Est)}, \\ (1, 0) & \text{si } d = 2 \text{ (Sud)}, \\ (0, -1) & \text{si } d = 3 \text{ (Ouest)}. \end{cases}$$

$$V = \{\, (i, j, d) \mid i \in \{0, \ldots, M\},\ j \in \{0, \ldots, N\},\ d \in \{0, 1, 2, 3\},\ (i, j) \notin \mathcal{O} \,\}.$$

$\mathcal{O} \subseteq \{0, \ldots, M\} \times \{0, \ldots, N\}$ est l'ensemble des positions interdites (obstacles).

$$A = A_{\text{rotation}} \cup A_{\text{déplacement}}$$

$$A_{\text{rotation}} = \{((i, j, d), (i, j, (d \pm 1) \bmod 4)) \mid (i, j, d) \in V\}$$

$$A_{\text{déplacement}} = \left\{ \big((i, j, d), (i + k\Delta_i(d),\ j + k\Delta_j(d),\ d)\big) \;\middle|\; \begin{array}{l} (i, j, d) \in V, \\ k \in \{1, 2, 3\}, \\ (i + s\Delta_i(d),\ j + s\Delta_j(d)) \notin \mathcal{O},\ \forall s \leq k \end{array} \right\}$$

Notons que la condition

$$(i + s\Delta_i(d), j + s\Delta_j(d)) \notin \mathcal{O}, \quad \forall s \leq k$$

implique les contraintes suivantes :

si avancer de 1 case est impossible, alors avancer de 2 et 3 cases est impossible aussi.
si avancer de 2 cases est impossible, alors avancer de 3 cases est impossible.

Par exemple, pour la matrice

$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$$

nous avons le graphe suivant :

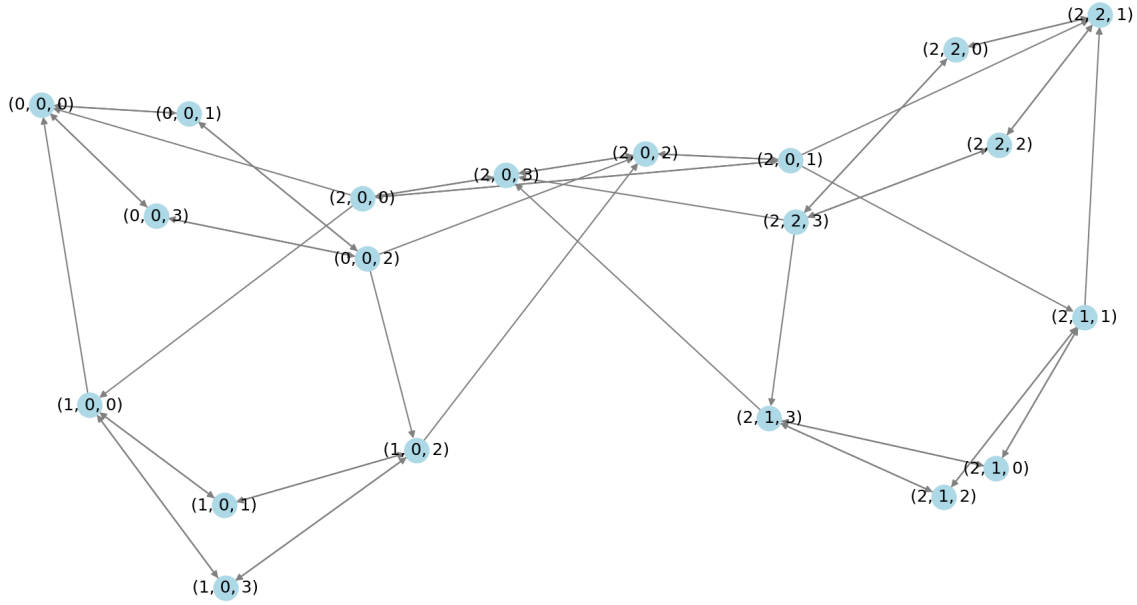

Figure 1: Exemple de graphe

3 Algorithme

Afin de trouver le chemin de coût minimal, nous avons implémenté deux algorithmes de plus court chemin connus: l'algorithme Breadth-First-Search et l'algorithme A*.

3.1 BFS

Algorithme. L'algorithme BFS explore le graphe par niveau, de manière uniforme, visant l'ensemble des états accessibles en un certain nombre de mouvements. Nous utilisons donc une liste First-In-First-Out pour garantir cet ordre d'exploration, et un ensemble contenant les états déjà visités pour éviter les répétitions. L'algorithme s'arrête lorsque la position (i, j) du nœud courant correspond à l'état d'arrivée. Comme notre graphe est construit de manière à ce que chaque arête représente une action et que toutes les actions ont le même temps d'exécution, le coût d'un chemin correspond directement à son temps d'exécution. Le chemin trouvé est donc automatiquement le plus court en termes de temps.

Raisonnement. Nous avons choisi d'implémenter BFS car toutes les actions ont un coût uniforme, et cet algorithme explore en premier les chemins les plus courts. Nous avons préféré BFS à Depth-First-Search, car si DFS est plus rapide pour trouver un chemin possible, il ne l'est pas pour trouver le meilleur chemin. BFS évite des explorations profondes non pertinentes.

Complexité. Dans le pire des cas (dans un graphe où il n'y aurait aucun obstacle), le nombre total d'états serait:

$$|V| = 4nm,$$

car chaque case ouverte de la grille peut être associée aux quatre directions possibles du robot (nord, sud, est, ouest). Donc le graphe contient un nombre total d'états donné par:

$$|V| \leq 4nm,$$

Dans le pire des cas, chaque sommet a 5 voisins (avancer de 1, 2 ou 3, tourner à gauche, tourner à droite). Donc:

$$|A| = 5 \times |V|.$$

La complexité temporelle de BFS est donc :

$$O(|V| + |A|) = O(nm),$$

puisque $|A|$ est linéaire en $|V|$. BFS va donc mieux fonctionner sur de petites grilles, mais deviendra coûteux lorsque la taille de la grille augmentera.

3.2 A*

Algorithme. L'algorithme A* combine, pour chaque état, le coût réellement parcouru depuis le départ (g) et l'estimation du coût restant pour atteindre la cible (h). Il calcule alors la valeur $f = g + h$.

Nous stockons les états à explorer dans une file de priorité triée selon f. A chaque itération, on extrait l'état dont f est minimal. Si cet état correspond à la position d'arrivée, la recherche s'arrête. Sinon, on met à jour les valeurs de g et f de chacun ses voisins. Nous utilisons le dictionnaire `came_from`, qui enregistre pour chaque nœud son prédécesseur, ce qui permet de reconstruire le chemin emprunté une fois l'état de fin atteint.

Raisonnement. Contrairement à l'algorithme BFS, A* concentre la recherche vers la cible en utilisant une heuristique cohérente, ce qui réduit considérablement l'exploration.

Heuristique. Notre heuristique est une adaptation de la distance de Manhattan au contexte de notre problème. En effet, le robot peut se déplacer par blocs d'une, deux ou trois cases, chacune de ces possibilités ayant un coût de 1. C'est pourquoi la distance de Manhattan usuelle

$$d_{\text{Manhattan}}(d, a) = |x_d - x_a| + |y_d - y_a|$$

n'est pas une heuristique correcte pour notre problème. Pour résoudre ce problème, notre adaptation divise chaque déplacement horizontal ou vertical par 3 en arrondissant au supérieur.

$$\hat{x} = \left\lceil \frac{|x_d - x_a|}{3} \right\rceil, \qquad \hat{y} = \left\lceil \frac{|y_d - y_a|}{3} \right\rceil, \qquad h = \hat{x} + \hat{y}.$$

Notre heuristique reflète le nombre minimal de déplacements nécessaires pour rejoindre la cible, et non plus la distance nécessaire. Elle ne surestime pas le coût réel minimal, et est donc admissible.

Complexité. Comme présenté dans le paragraphe précédent, le graphe contient :

$$|V| \leq 4nm \quad \text{états}, \qquad |A| = 5 \times |V| = O(mn) \text{ arêtes.}$$

Chaque insertion dans la file de priorité coûte $O(\log V) = O(\log mn)$. Cela donne une complexité dans le pire des cas telle que:

$$O(|V| + |A| \log |V|) \;=\; O\big((mn + mn)\log(mn)\big) \;=\; O\big(mn \log(mn)\big).$$

En pratique, l'heuristique réduit cette complexité, et l'algorithme visite moins d'états que BFS, ce qui en fait une méthode efficace pour ce problème.

4 Tests

4.1 Résultats en fonction de la taille de la matrice

Pour chaque $n \in \{10, 20, 30, 40, 50\}$, on génère 10 instances de taille $N = M = n$ contenant n obstacles placés aléatoirement. Voici les résultats obtenus :

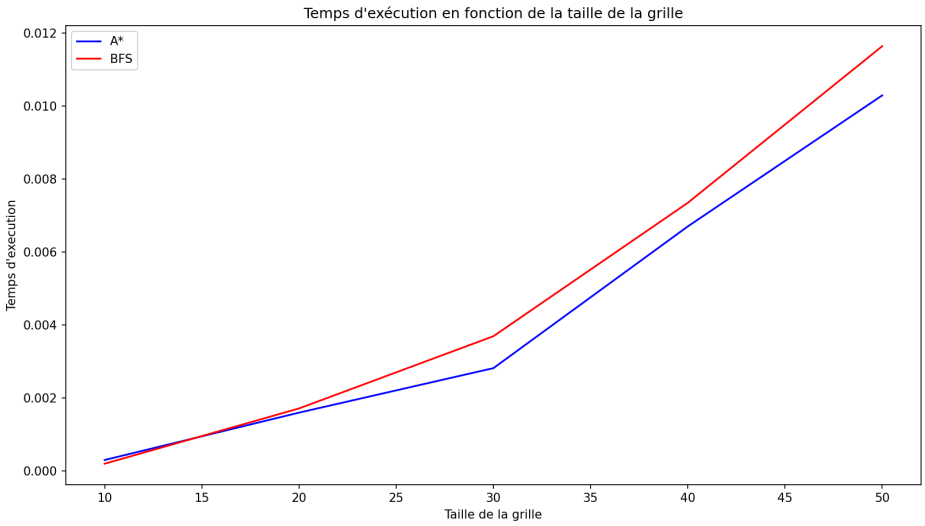

Figure 2: Temps d'exécution en fonction de la taille de la grille

On observe que l'algorithme A* présente systématiquement de meilleures performances que BFS. Cela est cohérent avec notre analyse de complexité : la courbe correspondant à BFS a une allure proche de $O(n^2)$, tandis que celle d'A* ressemble davantage à $O(n \log n)$.

Les instances utilisées pour ces tests se trouvent dans un fichier nommé `instaces_grid` et les résultats obtenus pour l'algorithme A* et BFS se trouvent respectivement dans les fichiers `results_astar_instance` et `results_bfs_instance`.

4.2 Résultats en fonction de la quantité d'obstacles

Pour chaque $o \in \{10, 20, 30, 40, 50\}$, on génère 10 instances de taille 20×20 contenant o obstacles placés aléatoirement. Voici les résultats obtenus :

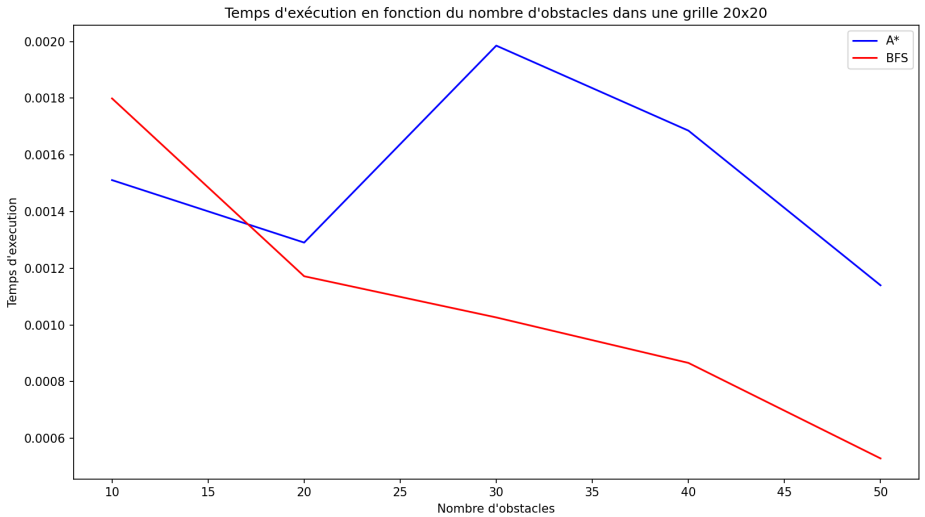

Figure 3: Temps d'exécution en fonction du nombre d'obstacles dans une grille 20×20

On observe que le temps d'exécution diminue lorsque le nombre d'obstacles augmente. En effet, plus il y a d'obstacles, moins il y a de nœuds accessibles, ce qui réduit l'espace de recherche pour les algorithmes. De plus, dans certains cas, A* peut être plus lent que BFS, car le calcul de l'heuristique ajoute un surcoût, surtout dans des environnements peu encombrés où BFS trouve rapidement la solution.

Les instances utilisées pour ces tests se trouvent dans un fichier nommé `instaces_obs` et les résultats obtenus pour l'algorithme A* et BFS se trouvent respectivement dans les fichiers `results_astar_obs` et `results_bfs_obs`.

4.3 Résultats supplémentaires

Nous avons jugé pertinent de comparer le temps d'exécution de A* selon que la solution existe ou non. Voici les résultats obtenus :

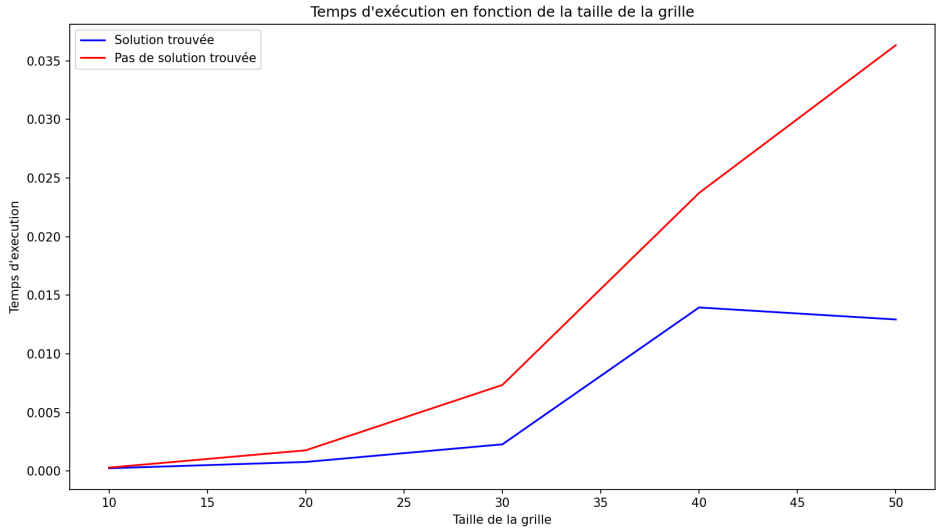

Figure 4: Temps d'exécution de A* en fonction de la taille de la grille

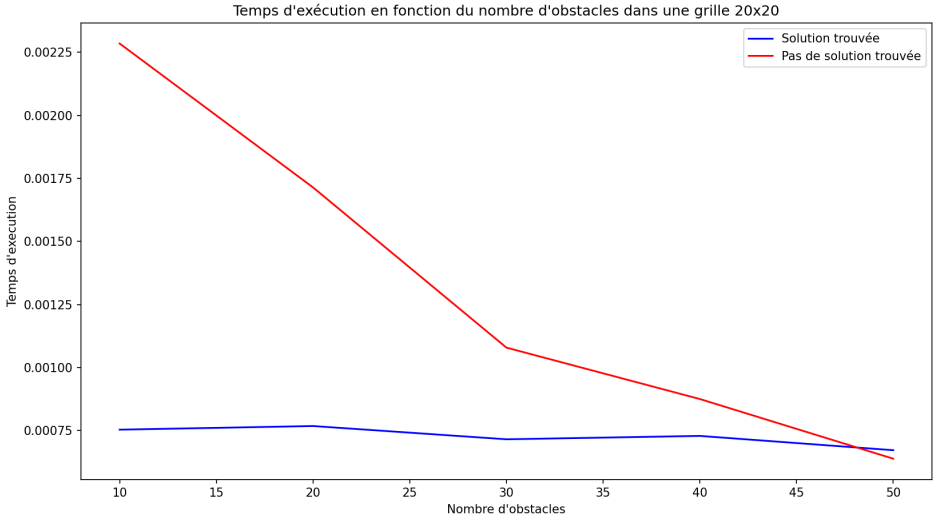

Figure 5: Temps d'exécution de A* en fonction du nombre d'obstacles dans une grille 20×20

Les résultats obtenus sont intéressants : le temps d'exécution de A* est toujours plus long lorsqu'aucun chemin n'existe que lorsqu'un chemin est trouvé. Cela s'explique par le fait que, dans le premier cas, A* doit explorer tous les nœuds du graphe.

5 Programme Linéaire

5.1 Problème d'optimisation

Les variables de notre programme linéaire sont

$$x_{i,j} \in \{0,1\}, \quad \forall i \in [\![0, M-1]\!], \quad \forall j \in [\![0, N-1]\!]$$

valant 1 si un obstacle est présent dans la case (i, j) et 0 sinon.

De plus, nous définissons des variables

$$w_{i,j} \in [\![0, 1000]\!], \quad \forall i \in [\![0, M-1]\!], \quad \forall j \in [\![0, N-1]\!]$$

qui représentent le poids entier entre 0 et 1000 de chaque case.

Les contraintes sont :

- $\sum_{i=0}^{M-1} \sum_{j=0}^{N-1} x_{i,j} = P$ (La grille doit contenir P obstacles)
- $\sum_{j=0}^{N-1} x_{i,j} \leq \frac{2P}{M}, \quad \forall i \in [\![0, M-1]\!]$ (Chaque ligne de la grille ne peut contenir plus de $2P/M$ obstacles)
- $\sum_{i=0}^{M-1} x_{i,j} \leq \frac{2P}{N}, \quad \forall j \in [\![0, N-1]\!]$ (Chaque colonne de la grille ne peut contenir plus de $2P/N$ obstacles)
- $x_{i,j-1} + x_{i,j+1} \leq 1 + x_{i,j}, \quad i \in [\![0, M-1]\!],\ j \in [\![1, N-2]\!]$ (Aucune ligne ne peut contenir la séquence 101)
- $x_{i-1,j} + x_{i+1,j} \leq 1 + x_{i,j}, \quad j \in [\![0, N-1]\!],\ i \in [\![1, M-2]\!]$ (Aucune colonne ne peut contenir la séquence 101)

Les contraintes interdisant les séquences 101 ont été les plus difficiles à établir. Pour les lignes (et de manière analogue pour les colonnes), nous avons procédé par disjonction de cas.

- Si $x_{i,j} = 0$:
  - Le cas $x_{i+1,j} = 0$ ou $x_{i-1,j} = 0$ doit être accepté. Ici, $x_{i+1,j} + x_{i-1,j} = x_{i+1,j}$.
  - Le cas $x_{i+1,j} = 1$ et $x_{i-1,j} = 0$ (et vice-versa) doit être accepté. Ici, $x_{i+1,j} + x_{i-1,j} = x_{i+1,j} + 1$.
  - Le cas $x_{i+1,j} = 1$ et $x_{i-1,j} = 1$ doit être interdit. Ici, $x_{i+1,j} + x_{i-1,j} = x_{i+1,j} + 2$.

  Pour que cela soit possible, on doit avoir $x_{i+1,j} + x_{i-1,j} \leq x_{i,j} + 1$.
- Si $x_{i,j} = 1$, alors toutes les valeurs de $x_{i+1,j}$ et de $x_{i-1,j}$ sont acceptées.
  - Si $x_{i+1,j} = 0$ et $x_{i-1,j} = 0$, alors $x_{i+1,j} + x_{i-1,j} = 0$, donc $x_{i+1,j} + x_{i-1,j} \leq x_{i,j} + 1$.
  - Si seulement une des cases voisines est occupée, $x_{i+1,j} + x_{i-1,j} = 1$, donc $x_{i+1,j} + x_{i-1,j} \leq x_{i,j} + 1$.
  - Enfin, si $x_{i+1,j} = 1$ et $x_{i-1,j} = 1$, alors $x_{i+1,j} + x_{i-1,j} = 2$, donc $x_{i+1,j} + x_{i-1,j} \leq x_{i,j} + 1$.

La fonction objectif est définie par

$$z = \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} x_{i,j}\, w_{i,j},$$

et notre but est de la minimiser.

5.2 Notre interface

L'utilisation de notre interface se déroule en deux étapes. Dans un premier temps, l'utilisateur saisit la taille de la grille (M, N) ainsi que le nombre d'obstacles qu'il souhaite placer. À partir de ces informations, la grille est générée en résolvant le programme linéaire présenté précédemment à l'aide du solveur Gurobi.

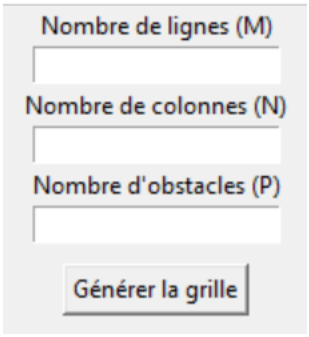

Figure 6: Saisie de la taille de la grille et de la quantité d'obstacles

Une fois la grille créée, la seconde étape de l'interface permet à l'utilisateur de choisir le point de départ, la direction initiale et le point d'arrivée.

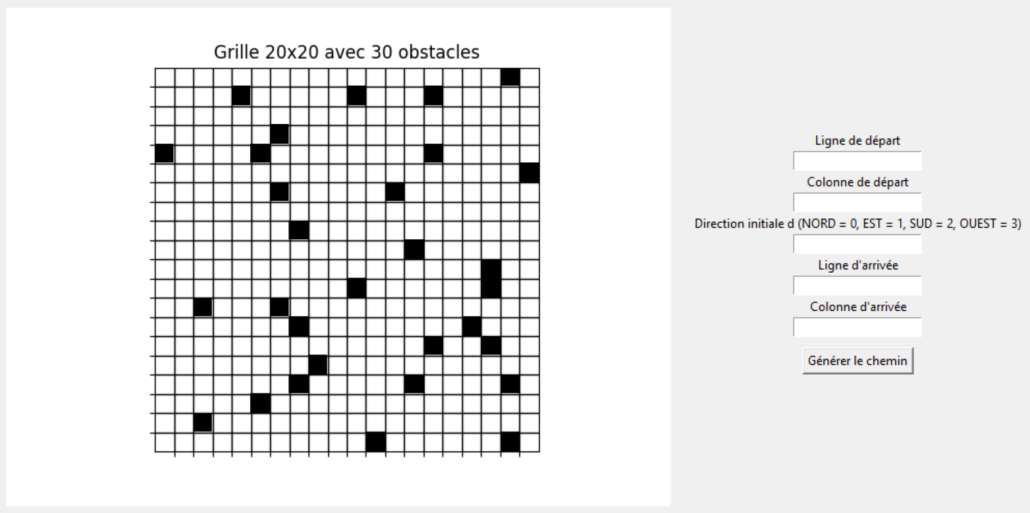

Figure 7: Grille générée

Si un chemin est trouvé, il est affiché sur la grille.

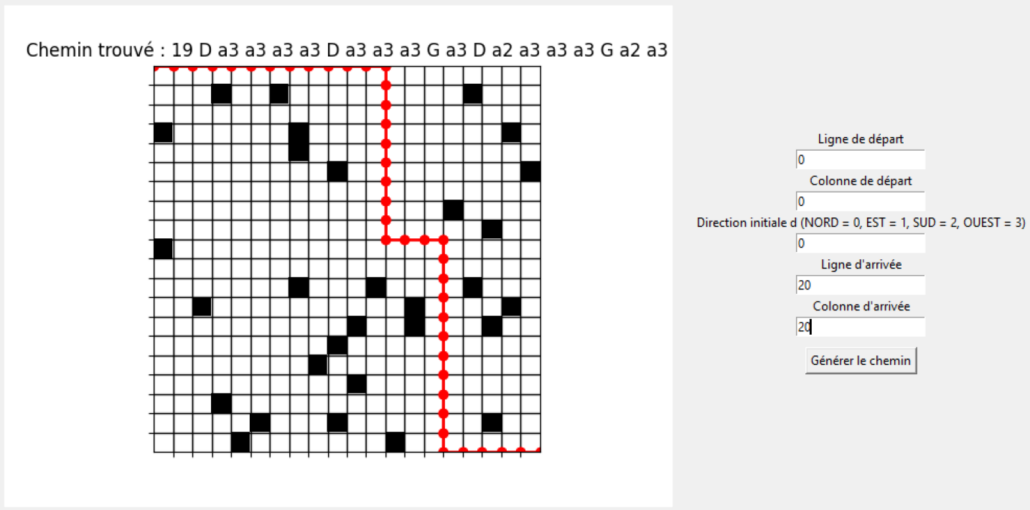

Figure 8: Chemin trouvé

Sinon, le message "aucun chemin trouvé" est présenté.

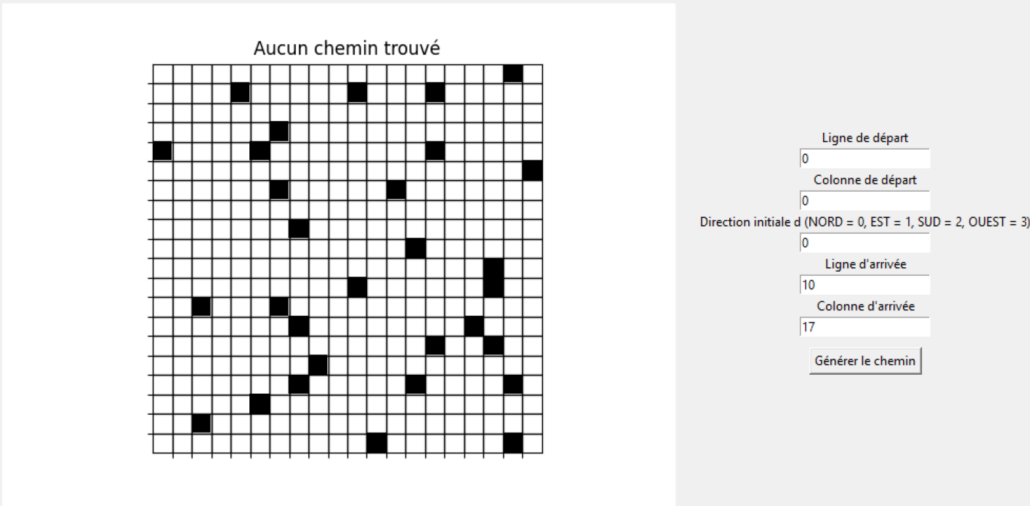

Figure 9: Aucun chemin trouvé

6 Conclusion

Afin de réaliser notre objectif, qui était de déterminer le temps minimal de déplacement d'un robot sur une grille en présence d'obstacles, nous avons utilisé une modélisation de la grille en un graphe orienté. Celui-ci prend en compte les contraintes de coûts d'actions (AVANCER et TOURNER) et les obstacles. Nous avons adapté deux algorithmes de plus court chemin à notre problème: Breadth-First-Search et A*. Nous avons ensuite, par le biais d'expérimentations, compris les efficacités différentes de nos deux solutions. La réalisation d'une interface pour visualiser le meilleur chemin dans des grilles générées selon les choix de l'utilisateur, et la programmation linéaire des obstacles de la grille de cette interface en utilisant Gurobi, complètent ce projet, dont toutes les parties ont été implémentées avec succès.

Les deux algorithmes implémentés, BFS et A*, rendent une solution correcte. Cependant, A* est plus performant grâce à une heuristique adaptée aux règles de déplacement du robot. Sur des grandes grilles, A* performe systématiquement plus rapidement que BFS, dont l'augmentation est quasi quadratique en fonction de la taille de la grille. Nous observons également que l'augmentation du nombre d'obstacles amène à une baisse de temps d'exécution, ce qui est dû à un espace réduit à explorer. Au contraire, une grille plus dégagée rend l'heuristique de A* moins utile mais toujours coûteuse. Cela fait que les temps d'exploration de A* et de BFS, qui trouve rapidement une solution, rivalisent dans ces cas particuliers. Au cours d'analyses supplémentaires, nous observons que A* devient plus lent dans le cas d'une solution inexistante, car l'algorithme doit alors explorer tout l'espace: un résultat cohérent.

Notre programme linéaire, utilisant le solveur Gurobi, modélise les contraintes de nombre d'obstacles, répartition, et d'absence de motif “101”, nous permettant de générer automatiquement, pour n'importe quelle taille de grille, les coordonnées des obstacles demandés.

Enfin, notre interface, implémentée à l'aide de Tkinter, rassemble ces différentes parties de notre implémentation. Sa première partie permet à l'utilisateur de choisir la taille de la grille et le nombre d'obstacles, et cette grille est générée à l'aide de notre programme linéaire. Ensuite, l'utilisateur peut choisir les positions de départ et d'arrivée pour trouver le meilleur chemin. Notre interface permet de visualiser ce chemin dans la grille, ou envoie un message d'échec s'il n'y a pas de solution. Ainsi, notre interface utilise notre algorithme ainsi que la programmation linéaire pour visualiser le meilleur chemin.

La mise en œuvre complète de ce projet nous a permis de combiner programmation linéaire avec interface graphique, et une compréhension approfondie des graphes et des algorithmes de plus court chemin dans différents scénarios.