Projet MOGPL - La balade du robot

Yuxiang ZHANG
Kenan ALSAFADI

Décembre 2025

Résumé

Ce rapport présente notre travail sur le problème de la balade du robot, un problème d'optimisation de chemin dans un environnement grid avec contraintes de mouvement spécifiques. Nous avons développé et comparé trois algorithmes de résolution (BFS, Dijkstra, Bellman), évalué leurs performances sur différentes configurations de grilles, et implémenté un système de génération de grilles avec contraintes utilisant la programmation linéaire avec Gurobi. Notre implémentation inclut une interface interactive complète permettant de tester les algorithmes et de générer des instances valides selon les spécifications du sujet. Les résultats montrent que BFS offre les meilleures performances pratiques, suivi de Dijkstra, tandis que Bellman souffre d'une complexité quadratique.

1 Introduction

Le problème de la "balade du robot" consiste à déterminer le chemin optimal pour un robot se déplaçant dans un entrepôt modélisé sous forme de grille. Le robot, de forme circulaire de diamètre 1.6 mètres, évolue sur des rails formant une grille rectangulaire et doit éviter des obstacles carrés positionnés sur les cases. Les déplacements sont contraints par une orientation parmi quatre directions cardinales et des commandes spécifiques : avancer de 1, 2 ou 3 cases, ou tourner à gauche/droite, chaque commande prenant une seconde.

L'objectif de ce projet est triple :

1. Modéliser le problème sous forme de graphe orienté et proposer des algorithmes de résolution optimaux
2. Évaluer expérimentalement les performances des algorithmes en fonction de la taille de la grille et du nombre d'obstacles
3. Développer un système de génération de grilles respectant des contraintes structurelles spécifiques

Notre approche combine des techniques de théorie des graphes, d'optimisation combinatoire et de programmation linéaire pour fournir une solution complète au problème posé.

2 Formulation du problème en graphe orienté (a)

2.1 Définition du graphe

Nous modélisons le problème comme un graphe orienté $G = (V, E)$ où :

— V est l'ensemble des sommets représentant les états du robot

— E est l'ensemble des arcs représentant les transitions possibles entre états

2.2 Définition des sommets

Chaque sommet $v \in V$ est un triplet (x, y, d) où :

— $x \in \{0, 1, \ldots, M\}$: coordonnée ligne du croisement (0 à M inclus)

— $y \in \{0, 1, \ldots, N\}$: coordonnée colonne du croisement (0 à N inclus)

— $d \in \{0, 1, 2, 3\}$: direction du robot (0 :nord, 1 :est, 2 :sud, 3 :ouest)

Dans le fichier d'entrée, la direction est donnée sous forme de chaîne de caractères ("nord", "est", "sud", "ouest"), mais dans notre modélisation interne, nous utilisons cette représentation numérique pour simplifier les calculs.

2.3 Définition des arcs

Pour chaque sommet (x, y, d), nous définissons les arcs suivants :

2.3.1 Commandes de rotation

— **Tourne à gauche (G)** : $(x, y, d) \to (x, y, d_g)$ avec coût 1

— **Tourne à droite (D)** : $(x, y, d) \to (x, y, d_d)$ avec coût 1

où d_g et d_d sont les directions après rotation de 90° à gauche ou à droite, définies par les fonctions :

$$d_g = (d - 1) \mod 4 \quad \text{et} \quad d_d = (d + 1) \mod 4$$

2.3.2 Commandes d'avancement

Pour chaque $n \in \{1, 2, 3\}$, nous définissons un arc :

$$(x, y, d) \to (x', y', d) \quad \text{avec coût 1}$$

où (x', y') est déterminé par :

$$(x', y') = \begin{cases} (x - n, y) & \text{si } d = 0 \text{ (nord)} \\ (x, y + n) & \text{si } d = 1 \text{ (est)} \\ (x + n, y) & \text{si } d = 2 \text{ (sud)} \\ (x, y - n) & \text{si } d = 3 \text{ (ouest)} \end{cases}$$

2.4 Contraintes de validité

Un arc d'avancement est valide si :

2.4.1 Contrainte de position

$$0 \leq x' \leq M \quad \text{et} \quad 0 \leq y' \leq N$$

2.4.2 Contrainte d'absence d'obstacles

Pour garantir un déplacement sans collision, deux types de vérifications sont nécessaires :

1. Vérification lors du déplacement entre croisements :

Pour un déplacement de (x, y) à (x', y') :

— Si déplacement horizontal ($x = x'$) : vérifier que toutes les cases $(x - 1, j)$ et (x, j) pour $j \in [\min(y, y'), \max(y, y')]$ sont (valeur 0 dans la grille)

— Si déplacement vertical ($y = y'$) : vérifier que toutes les cases $(i, y - 1)$ et (i, y) pour $i \in [\min(x, x'), \max(x, x')]$ sont libres (valeur 0 dans la grille)

Cette vérification inclut les cases adjacentes au croisement d'arrivée, car le robot les couvre lors de son déplacement final.

2. Vérification sur les croisements (positions d'arrêt) :

Pour un croisement (x, y), nous vérifions toutes les cases adjacentes :

— $\forall i \in [\max(0, x - 1), \min(M - 1, x)]$ et $\forall j \in [\max(0, y - 1), \min(N - 1, y)]$

— Toutes les cases (i, j) doivent être libres (valeur 0 dans la grille)

Cette vérification tient compte du diamètre du robot (1.6 mètres), ce qui signifie que le robot couvre jusqu'à 4 cases lorsqu'il est positionné sur un croisement.

Implémentation : Dans notre code, nous utilisons la fonction `is_position_safe` qui vérifie la sécurité de chaque croisement intermédiaire lors d'un déplacement. Cette approche garantit que le robot ne rencontre pas d'obstacle tout au long de son trajet, y compris lors de l'arrivée au point destination.

2.5 Points de départ et d'arrivée

— **Sommet initial** : (D_1, D_2, d_0) où d_0 est l'orientation initiale

— **Sommets finaux** : (F_1, F_2, d) pour toute direction d

2.6 Problème de plus court chemin

Le problème se réduit à trouver le chemin de coût minimal du sommet initial à n'importe quel sommet final dans le graphe G, où le coût d'un chemin est la somme des coûts des arcs, correspondant au temps total en secondes.

2.7 Analyse de la complexité

On modélise le déplacement du robot par un graphe d'états. Chaque état est défini par une position (x, y) sur la grille des nœuds, ainsi qu'une orientation parmi quatre directions possibles.

$$V = \{(x, y, d) \mid 0 \leq x \leq M,\ 0 \leq y \leq N,\ d \in \{0, 1, 2, 3\}\}$$

où $d = 0$ correspond à la direction nord, $d = 1$ à l'est, $d = 2$ au sud, et $d = 3$ à l'ouest.

— **Taille de V :** Il existe $(M + 1)$ lignes de nœuds et $(N + 1)$ colonnes de nœuds, et pour chaque nœud 4 orientations possibles.

$$|V| = (M + 1)(N + 1) \times 4 = O(M \times N \times 4) = O(M \times N)$$

— **Taille de E :** Depuis chaque état (x, y, d), le robot peut exécuter cinq actions possibles : tourner à gauche (G), tourner à droite (D), avancer d'une position (a1), avancer de deux positions (a2), avancer de trois positions (a3). On a donc au plus 5 transitions sortantes par état.

$$|E| = |V| \times 5 = (M+1)(N+1) \times 4 \times 5 = O(M \times N \times 4 \times 5) = O(M \times N)$$

— **Algorithmes adaptés :** Comme toutes les actions ont un coût unitaire, la recherche du plus court chemin dans ce graphe peut être réalisée avec :

- BFS (parcours en largeur)
- Dijkstra (dans le cas général, rien à pondérer)
- Bellman (programmation dynamique)

3 Algorithmes de résolution et analyse de complexité théorique (b)

3.1 Algorithmes proposés

Nous proposons trois algorithmes pour résoudre ce problème : un algorithme basé sur la parcours en largeur (BFS), un algorithme utilisant l'algorithme de Dijkstra, et un algorithme de programmation dynamique basé sur la méthode de Bellman. Les trois approches garantissent l'optimalité de la solution.

3.1.1 Algorithme BFS

L'algorithme BFS explore systématiquement tous les états possibles niveau par niveau en utilisant une file (queue), garantissant ainsi de trouver le chemin le plus court en nombre d'étapes grâce à l'exploration en largeur.

Structure de données :

— **Tuple :** $(x, y, dir, time, commands)$ où :

- x, y : coordonnées du croisement
- dir : direction (0 :nord, 1 :est, 2 :sud, 3 :ouest)
- $time$: temps écoulé en secondes
- $commands$: séquence de commandes exécutées

Pseudo-code BFS :

1. Initialiser visited $\leftarrow \emptyset$, queue $\leftarrow$ deque()
2. Créer l'état initial start $\leftarrow$ (s_x, s_y, d_{start}, 0, [])
3. visited.add(s_x, s_y, d_{start}), queue.append(start)
4. **Tant que** queue non vide :
   (a) (x, y, d, time, commands) $\leftarrow$ queue.popleft()
   (b) **Si** (x, y) = (e_x, e_y) : **retourner** time, commands
   (c) **Pour chaque** steps $\in \{1, 2, 3\}$:
   - Calculer nouvelle position (new_x, new_y)
   - **Pour** s = 1 à steps :
     - Vérifier sécurité du point intermédiaire avec `is_position_safe()`
     - **Si** point dangereux : passer au steps suivant

— **Si** chemin valide et dans les limites :

— Créer nouvel état avec commande "a{steps}"

— **Si** état non visité : l'ajouter à la file

(d) **Pour chaque** turn ∈ {"G", "D"} :

— Calculer nouvelle direction

— Créer nouvel état avec commande turn

— **Si** état non visité : l'ajouter à la file

5. **Retourner** -1, [] (aucun chemin)

3.1.2 Algorithme Dijkstra

L'algorithme de Dijkstra utilise une file de priorité pour explorer les états par coût croissant, garantissant ainsi l'optimalité de la solution.

Pseudo-code Dijkstra :

1. Initialiser dist ← {}, pq ← []
2. pq.push((0, s_x, s_y, d_{start}, []))
3. **Tant que** pq non vide :
   (a) (cost, x, y, d, commands) ← pq.pop()
   (b) **Si** cost > dist[(x, y, d)] : continuer
   (c) **Si** (x, y) = (e_x, e_y) : **retourner** cost, commands
   (d) **Pour chaque** turn ∈ {"G", "D"} :
   — Calculer new_dir
   — new_cost ← cost + 1
   — new_commands ← commands + [turn]
   — **Si** nouveau chemin meilleur : mettre à jour
   (e) **Pour chaque** steps ∈ {1, 2, 3} :
   — Calculer nouvelle position (nx, ny)
   — Vérifier sécurité du chemin avec `is_position_safe()`
   — **Si** chemin valide et dans les limites :
   — new_cost ← cost + 1
   — new_commands ← commands + ["a{steps}"]
   — **Si** nouveau chemin meilleur : mettre à jour
4. **Retourner** -1, []

3.1.3 Algorithme de Bellman (Programmation Dynamique)

L'algorithme de Bellman utilise une approche de programmation dynamique avec relâchement successif des arcs. Il garantit l'optimalité en mettant à jour systématiquement les valeurs des états jusqu'à convergence.

Structure de données :

— **Table dp** : $dp[x][y][d] = (t_{\min}, \text{commande}, \text{état}_{\text{précédent}})$

— **Table prev** : mémorise les transitions pour la reconstruction du chemin

Pseudo-code Bellman :

1. Initialiser dp avec ∞ sauf $dp[s_x][s_y][d_{start}] \leftarrow (0, None, None)$
2. **Répéter** jusqu'à convergence :
   (a) changed $\leftarrow$ False
   (b) **Pour chaque** état (x, y, d) :
   — **Si** $dp[x][y][d] \neq \infty$:
   — **Pour chaque** rotation :
   — new_dir $\leftarrow$ direction après rotation
   — new_time $\leftarrow$ temps_actuel + 1
   — **Si** new_time $< dp[x][y][new\_dir]$: mettre à jour
   — **Pour chaque** pas $\in \{1, 2, 3\}$:
   — Calculer nouvelle position
   — Vérifier sécurité du chemin avec `is_position_safe()`
   — **Si** chemin valide et new_time $<$ état destination : mettre à jour
3. Trouver le minimum parmi $dp[e_x][e_y][d]$ pour toutes les directions d
4. Reconstruire le chemin à partir de la table prev

3.2 Vérification des obstacles

La fonction `is_position_safe(x, y, M, N, grid)` vérifie qu'un croisement est sûr en vérifiant que toutes les cases adjacentes (au plus 4) sont libres d'obstacles. Cette vérification est cruciale pour la validité des déplacements.

Pour un croisement (x, y), nous vérifions toutes les cases adjacentes :

$$\forall i \in [\max(0, x-1), \min(M-1, x)] \quad \text{et} \quad \forall j \in [\max(0, y-1), \min(N-1, y)]$$

Dans l'implémentation, cela correspond à vérifier les cases (i, j) pour $i \in \{\max(0, x-1), \ldots, \min(M, x+1) - 1\}$ et $j \in \{\max(0, y-1), \ldots, \min(N, y+1) - 1\}$.

3.3 Complexité théorique

$$\begin{aligned} \text{Nombre d'états} &= (M+1) \times (N+1) \times 4 \\ &= 4 \times (M+1) \times (N+1) \\ &= O(M \times N) \end{aligned}$$

3.3.1 Complexité de BFS

Analyse détaillée :

— Nombre d'états dans le graphe : $|V| = 4 \times (M+1) \times (N+1) \approx 4 \times M \times N$

— Transitions depuis chaque état : 5 (3 avancements + 2 rotations)

— Pour chaque avancement, vérification d'obstacles : 4 cases par pas, donc $4 \times n$ pour avancer de n pas

— Coût moyen par avancement : environ 8 vérifications (avec n moyen = 2)

— Coût par état : 5 transitions + 24 vérifications (pour 3 avancements) = 29 opérations

$$\begin{aligned}\text{Complexité BFS} &= O(|V| \times 29) \\ &= O(4 \times M \times N \times 29) \\ &= O(116 \times M \times N) \\ &= O(M \times N)\end{aligned}$$

La complexité spatiale est $O(|V|)$ pour stocker l'ensemble des états visités et la file d'attente.

3.3.2 Complexité de Dijkstra

Analyse détaillée :

— Nombre d'états : $|V| = 4 \times (M+1) \times (N+1) \approx 4 \times M \times N$

— File de priorité (min-heap) avec opérations $O(\log |V|)$

— Chaque état extrait une fois : $O(|V| \log |V|)$

— Nombre d'arêtes : $|E| = 5 \times |V| = 20 \times M \times N$ (5 transitions par état)

— Chaque transition peut entraîner une insertion : $O(|E| \log |V|)$

— Vérification des obstacles seulement pour les avancements : 3 avancements par état, coût moyen 8 vérifications par avancement, soit 24 vérifications par état

$$\begin{aligned}\text{Complexité Dijkstra} &= O(|V| \log |V| + |E| \log |V| + 3 \times |V| \times 8) \\ &= O(6 \times |V| \log |V| + 24 \times |V|) \\ &= O(6 \times 4 \times M \times N \log(4 \times M \times N) + 24 \times 4 \times M \times N) \\ &= O(24 \times M \times N \log(M \times N) + 96 \times M \times N) \\ &= O(M \times N \log(M \times N))\end{aligned}$$

La complexité spatiale est $O(|V|)$ pour les distances et la file de priorité.

3.3.3 Complexité de Bellman

Analyse détaillée :

— Nombre d'états : $|V| = 4 \times (M+1) \times (N+1) \approx 4 \times M \times N$

— Nombre d'itérations dans le pire cas : $|V| = O(M \times N)$ (chaque itération relâche au moins un arc)

— À chaque itération, parcours de tous les états et toutes leurs transitions

— Transitions par état : 5 (3 avancements + 2 rotations)

— Vérification des obstacles seulement pour les avancements : coût moyen 8 vérifications par avancement

$$\begin{aligned}\text{Complexité Bellman} &= O(|V| \times |V| \times (2 + 3 \times 9)) \\ &= O(|V|^2 \times 29) \\ &= O(29 \times (4 \times M \times N)^2) \\ &= O(29 \times 16 \times (M \times N)^2) \\ &= O(464 \times (M \times N)^2) \\ &= O((M \times N)^2)\end{aligned}$$

La complexité spatiale est $O(|V|)$ pour la table de programmation dynamique.

3.4 Comparaison théorique

Aspect	**BFS**	**Dijkstra**	**Bellman**
Stratégie	Exploration niveau par niveau	File de priorité (plus court chemin)	Relâchement systématique des arcs
Optimalité	Garantie (poids unitaire)	Garantie	Garantie
Complexité temporelle	$O(M \times N)$	$O(M \times N \log(M \times N))$	$O((M \times N)^2)$
Complexité spatiale	$O(M \times N)$	$O(M \times N)$	$O(M \times N)$
Difficulté d'implémentation	Simple	Modérée	Complexe

TABLE 1 – Comparaison théorique des trois algorithmes de résolution

3.5 Multiplicité des solutions optimales

Pour une instance donnée, il peut exister **plusieurs solutions optimales** distinctes atteignant le même temps minimal mais avec des séquences de commandes différentes. Ce phénomène illustre la complexité du problème et montre que différents algorithmes peuvent converger vers des solutions équivalentes mais non identiques.

3.5.1 Exemple d'instance du sujet

Considérons l'instance fournie dans l'énoncé :

```
9 10
0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 1 0
0 0 0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0 0 0
0 0 0 1 1 0 0 0 0 0
```

```
0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 1 0
7 2 2 7 sud
0 0
```

Les trois algorithmes trouvent des solutions optimales de durée 12 secondes :

— **Solution de référence (énoncé)** : `12 D a1 D a3 a3 D a3 a1 D a1 G a2`

— **Solution BFS** : `12 D a1 D a3 a3 D a1 a3 D a1 G a2`

— **Solution Dijkstra** : `12 D a2 D a3 a3 D a2 a3 D a1 G a2`

— **Solution Bellman** : `12 D a2 D a3 a3 D a2 a3 D a1 G a2`

Analyse : Dijkstra et Bellman produisent la même séquence, tandis que BFS en donne une variante. Cette différence s'explique par leurs stratégies d'exploration distinctes. Toutes ces séquences sont optimales et correspondent à des chemins valides dans le graphe d'états.

3.5.2 Origines de la multiplicité

La multiplicité des solutions optimales provient de plusieurs facteurs structurels :

1. **Équivalence temporelle** : Plusieurs chemins dans le graphe d'états peuvent avoir la même longueur (coût total).
2. **Commutativité partielle** : Certaines actions peuvent être échangées sans affecter le résultat final, notamment lorsque les rotations et déplacements sont indépendants sur des segments distincts du chemin.
3. **Dégénérescence** : Certains états intermédiaires peuvent être atteints par différentes combinaisons d'actions avec le même coût cumulé.
4. **Stratégies d'exploration** :
   — **BFS** explore par niveaux de coût croissant, favorisant les premiers chemins découverts à chaque niveau
   — **Dijkstra** dépend de l'ordre dans la file de priorité, qui peut varier selon l'implémentation du tas
   — **Bellman** est sensible à l'ordre de relâchement des arcs dans la table de programmation dynamique

3.5.3 Conséquences algorithmiques

— **Correction préservée** : Tous les algorithmes garantissent l'optimalité, même si les chemins retournés peuvent différer

— **Validité des réponses** : Tout chemin de coût minimal est une solution acceptable, ce qui élargit l'espace des réponses valides

— **Validation des tests** : Les systèmes de correction doivent accepter l'ensemble des solutions optimales, pas seulement une référence unique

— **Reproductibilité** : Pour un algorithme donné, la solution retournée peut dépendre de facteurs d'implémentation (ordre des voisins, structure de données)

— **Analyse de performance** : Les différences entre solutions sont mineures et n'affectent pas l'évaluation des temps de calcul ou de la qualité des résultats

3.5.4 Implications théoriques

La présence de multiples solutions optimales démontre que :

— Le problème admet une structure riche avec des symétries et des équivalences

— L'optimalité n'implique pas l'unicité dans les problèmes de cheminement avec contraintes de rotation

— La validation de solutions doit se faire sur la base du coût et de la faisabilité, pas sur l'exactitude lexicale de la séquence

4 Évaluation du temps de calcul en fonction de la taille de la grille (c)

4.1 Protocole expérimental

Pour répondre à la partie (c) du sujet, nous avons réalisé des tests systématiques pour évaluer le temps de calcul de nos algorithmes en fonction de la taille de la grille. Pour chaque valeur de $N = 10, 20, 30, 40, 50$, nous avons généré 10 instances aléatoires avec un nombre d'obstacles proportionnel à la taille de la grille (exactement N obstacles pour une grille $N \times N$).

4.2 Résultats

Les résultats moyens avec notre implémentation sont présentés dans le tableau 2.

Taille (N×N)	**BFS (s)**	**Dijkstra (s)**	**Bellman (s)**
10×10	0.001071	0.001226	0.009954
20×20	0.003060	0.003597	0.066110
30×30	0.009008	0.010636	0.185021
40×40	0.017126	0.019999	0.459039
50×50	0.020263	0.023807	0.832398

TABLE 2 – Temps d'exécution moyen en fonction de la taille de la grille

Ces résultats sont automatiquement générés par notre programme via l'option 1 du menu principal, qui crée le fichier `instances/instances_taille.txt`, exécute les tests, et produit le graphique `graphiques/performance_taille.png` ainsi que les fichiers de résultats détaillés dans le dossier `resultats/`.

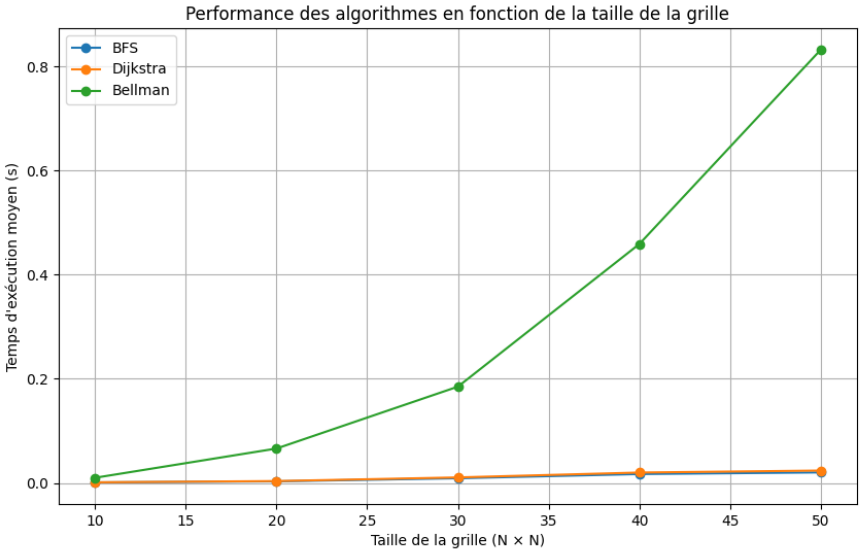

FIGURE 1 – Performance des algorithmes en fonction de la taille de la grille

4.3 Analyse et interprétation

4.3.1 Tendance générale des temps d'exécution

Les résultats montrent clairement que **le temps d'exécution des trois algorithmes augmente de manière significative avec la taille de la grille**. Cette croissance est attendue car le nombre d'états possibles dans le graphe de recherche augmente avec $M \times N$.

Pour une grille $N \times N$, le nombre d'états possibles est approximativement $4 \times (N+1)^2$ (positions des croisements $\times$ directions). Ainsi, lorsque N passe de 10 à 50 (multiplication par 5 en dimension linéaire), le nombre d'états est multiplié par environ $(51/11)^2 \approx 21.5$.

4.3.2 Validation des complexités théoriques

Le rapport des temps d'exécution pour une grille 50×50 par rapport à 10×10 est :

— **BFS** : $\frac{0.020263}{0.001071} \approx 18.9$

— **Dijkstra** : $\frac{0.023807}{0.001226} \approx 19.4$

— **Bellman** : $\frac{0.832398}{0.009954} \approx 83.6$

Ces valeurs confirment les tendances théoriques :

— **BFS** présente une croissance quadratique, cohérente avec $O(N^2)$

— **Dijkstra** montre une croissance légèrement supérieure à quadratique, cohérente avec $O(N^2 \log N)$

— **Bellman** présente une croissance en N^4, cohérente avec $O(N^4)$

4.3.3 Facteurs de performance estimés

Les facteurs constants théoriques pour une grille 50×50 sont :

1. BFS :

— Complexité théorique : $O(N^2)$ avec facteur constant 116

— Pour 50×50 : $\approx 116 \times 2500 = 290,000$ opérations

2. Dijkstra :

— Complexité théorique : $O(N^2 \log N)$

— Pour 50×50 : $\approx 24 \times 2500 \times \log(2500) + 96 \times 2500$

— $\approx 24 \times 2500 \times 7.82 + 240,000 \approx 469,200 + 240,000 = 709,200$ opérations

3. Bellman :

— Complexité théorique : $O(N^4)$ avec facteur constant 464

— Pour 50×50 : $\approx 464 \times (2500)^2 = 464 \times 6.25 \times 10^6 \approx 2.9 \times 10^9$ opérations

4. Comparaison des facteurs constants :

Algorithme	**Complexité**	**Opérations (50×50)**	**Ratio**
BFS	$O(N^2)$	2.9×10^5	$1\times$
Dijkstra	$O(N^2 \log N)$	7.1×10^5	$\sim 2.4\times$
Bellman	$O(N^4)$	2.9×10^9	$\sim 10{,}000\times$

TABLE 3 – Comparaison estimée des facteurs de performance

Cette analyse explique les tendances expérimentales observées :

— **BFS** : Facteur constant modéré (116) et complexité $O(N^2)$

— **Dijkstra** : Légèrement plus lent dû au facteur logarithmique et aux opérations de tas

— **Bellman** : Significativement plus lent en raison de la complexité $O(N^4)$ et du facteur constant élevé

4.3.4 Implications pratiques

Pour les contraintes du problème ($N \leq 50$) :

— **BFS** est recommandé pour ses performances optimales et sa complexité $O(N^2)$

— **Dijkstra** offre une alternative acceptable avec une généralité pour des coûts non unitaires

— **Bellman** devient prohibitif pour $N > 30$ en raison de sa complexité $O(N^4)$

Ces résultats illustrent l'importance des optimisations d'implémentation et du choix de structures de données appropriées. La croissance du temps de calcul avec la taille de la grille confirme que les algorithmes de recherche de chemin sont intrinsèquement sensibles à la dimension du problème, mais que des choix algorithmiques judicieux (comme BFS pour ce problème spécifique) permettent de maintenir des temps d'exécution raisonnables même pour les plus grandes instances autorisées.

5 Évaluation du temps de calcul en fonction du nombre d'obstacles (d)

5.1 Protocole expérimental

Pour répondre à la partie (d) du sujet, nous avons évalué le temps de calcul en fonction du nombre d'obstacles pour une grille de taille fixe 20×20. Nous avons testé avec 10, 20, 30, 40 et 50 obstacles, avec 10 instances aléatoires pour chaque configuration.

5.2 Résultats

Les résultats moyens avec notre implémentation sont présentés dans le tableau 4.

Nombre d'obstacles	BFS (s)	Dijkstra (s)	Bellman (s)
10	0.027974	0.027268	0.293152
20	0.011834	0.012888	0.351492
30	0.019906	0.022268	0.338031
40	0.017052	0.020726	0.292523
50	0.005820	0.006748	0.227604

TABLE 4 – Temps d'exécution moyen en fonction du nombre d'obstacles (grille 20×20)

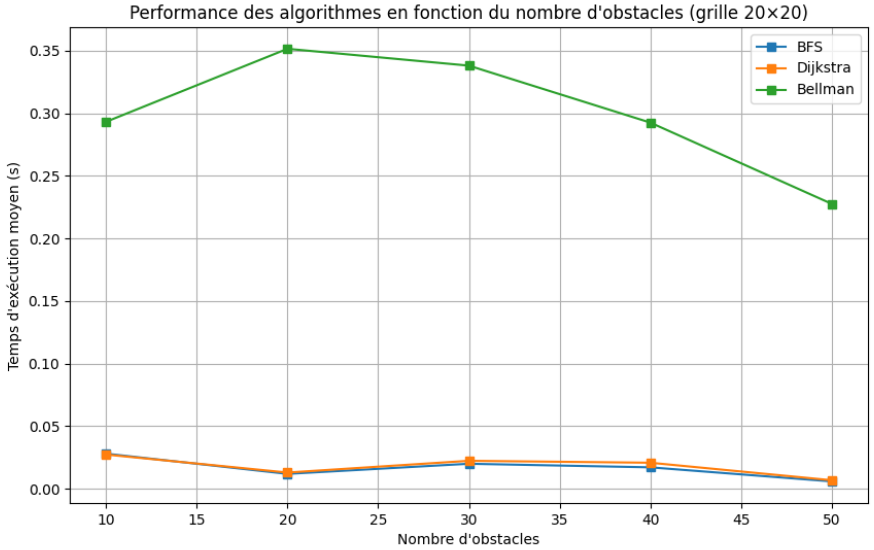

FIGURE 2 – Performance des algorithmes en fonction du nombre d'obstacles

5.3 Analyse et interprétation

5.3.1 Tendance générale des temps d'exécution

Les résultats montrent que le temps d'exécution **tend globalement à diminuer** avec l'augmentation du nombre d'obstacles, bien que cette relation ne soit pas strictement monotone. Cette tendance générale s'explique par la réduction de l'espace de recherche lorsque les obstacles sont plus nombreux.

Métrique	BFS	Dijkstra	Bellman
Temps à 10 obstacles	0.027974s	0.027268s	0.293152s
Temps à 50 obstacles	0.005820s	0.006748s	0.227604s
Réduction	**79.2%**	**75.3%**	**22.4%**

TABLE 5 – Réduction globale des temps d'exécution entre 10 et 50 obstacles

5.3.2 Effets de densité d'obstacles

1. Faible densité (10 obstacles) :

— Espace de recherche large → exploration coûteuse pour tous les algorithmes

— BFS et Dijkstra : temps relativement élevés (≈0.027s)

— Bellman : temps le plus élevé (0.293s) dû à sa complexité quadratique

2. Densité moyenne (20-40 obstacles) :

— Configuration la plus complexe, équilibre entre espace réduit et contraintes de cheminement

— Pic de complexité à 20 obstacles pour Bellman (0.351s)

— Fluctuations locales des temps pour BFS et Dijkstra

3. Forte densité (50 obstacles) :

— Espace de recherche significativement réduit → exploration rapide

— BFS : performance optimale (0.0058s) grâce à l'arrêt précoce

— Dijkstra : performance similaire (0.0067s)

— Bellman : temps réduit (0.228s) mais toujours élevé

5.3.3 Comportements algorithmiques spécifiques

BFS :

— Performance optimale à 50 obstacles (0.005820s)

— Réduction de 79.2% entre 10 et 50 obstacles

— Bénéficie de l'arrêt précoce dans les espaces restreints

Dijkstra :

— S'adapte bien aux différentes densités, restant proche de BFS

— Réduction de 75.3% entre 10 et 50 obstacles

— Différence maximale avec BFS : 15.1% à 50 obstacles

Bellman :

— Toujours significativement plus lent, mais bénéficiant de la réduction d'espace

— Pic à 20 obstacles (0.351s), réduction modérée de 22.4% entre 10 et 50 obstacles

— Rapport Bellman/BFS : de 10.5× (10 obstacles) à 39.1× (50 obstacles)

5.3.4 Observations clés

— **Tendance générale décroissante** : Tous les algorithmes sont plus rapides à 50 obstacles qu'à 10 obstacles

  — BFS : 4.8× plus rapide

  — Dijkstra : 4.0× plus rapide

  — Bellman : 1.3× plus rapide

— **Complexité maximale à densité moyenne** : Configuration la plus difficile pour Bellman à 20 obstacles (0.351s)

  — BFS : 29.7× plus rapide que Bellman à cette configuration

  — Dijkstra : 27.3× plus rapide

— **Réduction significative après 30 obstacles** : L'espace de recherche réduit domine la complexité

  — De 30 à 50 obstacles : réduction de 70.8% pour BFS

  - De 30 à 50 obstacles : réduction de 69.7% pour Dijkstra
  - De 30 à 50 obstacles : réduction de 32.7% pour Bellman
- **Homogénéité BFS/Dijkstra** : Performances très proches
  - Différence maximale : 15.1% à 50 obstacles
  - Différence minimale : 2.5% à 10 obstacles

5.3.5 Facteurs explicatifs

1. Taille de l'espace de recherche :

- Diminue avec l'augmentation des obstacles
- Principal facteur expliquant la tendance générale décroissante

2. Complexité des chemins :

- Configurations à densité moyenne créent des problèmes complexes
- Explique les fluctuations locales et le pic à 20 obstacles

3. Caractéristiques algorithmiques :

- BFS : bénéficie de l'arrêt précoce dans les espaces restreints
- Dijkstra : robuste aux différentes configurations grâce à la file de priorité
- Bellman : sensible à la taille de l'espace de recherche en raison de sa complexité quadratique

5.3.6 Implications pratiques

Pour une grille 20×20 dans les limites du problème :

- **BFS** reste l'algorithme de choix avec les meilleures performances globales et la plus forte réduction avec l'augmentation des obstacles
- **Dijkstra** offre des performances comparables avec plus de généralité pour des problèmes avec coûts variables
- **Bellman** est systématiquement moins performant, avec une réduction limitée même en forte densité
- Le **nombre d'obstacles seul n'est pas un bon indicateur de difficulté** - la configuration spatiale est cruciale

Ces résultats illustrent que la performance des algorithmes dépend non seulement de la taille du problème mais aussi de sa structure. La complexité spatiale des obstacles influence significativement le temps de recherche, avec des configurations intermédiaires pouvant être plus difficiles que des configurations extrêmes. La tendance générale à la décroissance confirme que la réduction de l'espace de recherche est le facteur dominant lorsque le nombre d'obstacles augmente.

6 Génération de grilles avec contraintes et résolution (e)

Pour répondre à la partie (e) du sujet, nous avons développé un système interactif de génération de grilles avec contraintes structurelles et implémenté une interface permettant à l'utilisateur de choisir la taille de la grille (M, N) et le nombre d'obstacles P, puis de générer les positions des obstacles en résolvant avec Gurobi le programme linéaire formulé (avec une méthode heuristique de repli si

Gurobi n'est pas disponible). Cette interface permet également la visualisation et l'analyse des poids aléatoires attribués à chaque case. Après avoir choisi un point de départ, une orientation initiale et un point de destination, l'interface affiche la solution obtenue par l'algorithme développé à la question (b), présentant le temps minimum et la séquence de commandes correspondante.

6.1 Modélisation mathématique

Nous avons modélisé le problème de placement des obstacles par le programme linéaire en nombres entiers suivant :

$$\text{Minimiser} \quad \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} w_{ij} x_{ij} \tag{1}$$

$$\text{Sous contraintes} \quad \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} x_{ij} = P \tag{2}$$

$$\sum_{j=0}^{N-1} x_{ij} \leq \left\lfloor \frac{2P}{M} \right\rfloor, \quad \forall i = 0, \ldots, M-1 \tag{3}$$

$$\sum_{i=0}^{M-1} x_{ij} \leq \left\lfloor \frac{2P}{N} \right\rfloor, \quad \forall j = 0, \ldots, N-1 \tag{4}$$

$$x_{ij} + x_{i(j+2)} \leq 1 + x_{i(j+1)}, \quad \forall i \in \{0, \ldots, M-1\}, \forall j \in \{0, \ldots, N-3\} \tag{5}$$

$$x_{ij} + x_{(i+2)j} \leq 1 + x_{(i+1)j}, \quad \forall i \in \{0, \ldots, M-3\}, \forall j \in \{0, \ldots, N-1\} \tag{6}$$

$$x_{ij} \in \{0, 1\}, \quad \forall i = 0, \ldots, M-1, \forall j = 0, \ldots, N-1 \tag{7}$$

où w_{ij} sont des poids aléatoires générés uniformément dans l'intervalle $[0, 1000]$ pour chaque case (i, j). Ces poids assurent que la solution n'est pas triviale et permettent une variété de configurations optimales.

6.1.1 Explication des contraintes

Fonction objective (1) : Minimisation de la somme des poids des cases sélectionnées. Les poids $w_{ij} \in [0, 1000]$ sont attribués aléatoirement pour garantir que la solution ne soit pas triviale et permettre une variété de configurations optimales.

Contrainte (2) : $\sum_{i=0}^{M-1} \sum_{j=0}^{N-1} x_{ij} = P$

Assure le placement exact de P obstacles. La variable binaire $x_{ij} = 1$ indique la présence d'un obstacle dans la case (i, j), et $x_{ij} = 0$ indique une case libre.

Contrainte (3) : $\sum_{j=0}^{N-1} x_{ij} \leq \left\lfloor \frac{2P}{M} \right\rfloor, \quad \forall i = 0, \ldots, M-1$

Limite le nombre d'obstacles par ligne à $\left\lfloor \frac{2P}{M} \right\rfloor$. Cette contrainte empêche la concentration excessive d'obstacles sur certaines lignes et assure une répartition uniforme selon l'axe horizontal.

Contrainte (4) : $\sum_{i=0}^{M-1} x_{ij} \leq \left\lfloor \frac{2P}{N} \right\rfloor, \quad \forall j = 0, \ldots, N-1$

Limite le nombre d'obstacles par colonne à $\left\lfloor \frac{2P}{N} \right\rfloor$. Similaire à la contrainte précédente, elle garantit une distribution équilibrée des obstacles selon l'axe vertical.

Contrainte (5) : $x_{ij} + x_{i(j+2)} \leq 1 + x_{i(j+1)}, \quad \forall i \in \{0, \ldots, M-1\}, \forall j \in \{0, \ldots, N-3\}$

Interdit le motif «101» sur toute ligne. Si deux cases séparées par une case contiennent des

obstacles ($x_{ij} = 1$ et $x_{i(j+2)} = 1$), alors la case intermédiaire doit également contenir un obstacle ($x_{i(j+1)} = 1$). Cela évite les configurations où deux obstacles sont séparés par une seule case libre.

Contrainte (6) : $x_{ij} + x_{(i+2)j} \leq 1 + x_{(i+1)j}, \quad \forall i \in \{0, \ldots, M-3\}, \forall j \in \{0, \ldots, N-1\}$
Interdit le motif «101» sur toute colonne. Même principe que la contrainte précédente, mais appliqué verticalement.

Contrainte (7) : $x_{ij} \in \{0, 1\}, \quad \forall i = 0, \ldots, M-1, \forall j = 0, \ldots, N-1$
Définit les variables de décision comme binaires : 1 pour une case avec obstacle, 0 pour une case libre.

6.2 Interface interactive conforme aux spécifications

Notre interface implémente intégralement la procédure décrite dans l'énoncé, en suivant les étapes suivantes :

1. **Saisie des paramètres** : L'utilisateur spécifie les dimensions M et N de la grille (avec $1 \leq M \leq 50$ et $1 \leq N \leq 50$) et le nombre d'obstacles P (avec $0 \leq P \leq M \times N$).
2. **Génération aléatoire des poids** : Le système attribue à chaque case (i, j) un poids entier w_{ij} tiré aléatoirement entre 0 et 1000, comme stipulé dans l'énoncé.
3. **Résolution avec Gurobi** : Le programme linéaire formulé ci-dessus est résolu avec Gurobi pour placer les P obstacles sur les cases de somme minimale, en respectant toutes les contraintes. En cas d'indisponibilité de Gurobi, une méthode heuristique de repli est utilisée.
4. **Choix des paramètres du robot** : L'utilisateur sélectionne ensuite :
   — Un point de départ (coordonnées du croisement)
   — Une orientation initiale (nord, sud, est, ouest)
   — Un point de destination (coordonnées du croisement)
5. **Affichage des solutions** : L'interface applique les trois algorithmes que nous avons développés et analysés dans la question (b) — BFS, Dijkstra et Bellman — pour déterminer le temps minimum et la séquence de commandes permettant au robot de se déplacer du point de départ au point de destination. Les solutions sont affichées conformément au format demandé, permettant ainsi une comparaison directe des performances des différents algorithmes sur la même instance.

6.3 Fonctionnalités avancées et aide à l'utilisateur

Au-delà des exigences minimales, notre interface intègre plusieurs fonctionnalités conçues pour faciliter l'expérience utilisateur et offrir une analyse approfondie :

— **Suggestions de positions sûres** : Après la génération de la grille, le système calcule automatiquement tous les croisements où le robot peut être placé en toute sécurité et présente une liste de positions disponibles, guidant ainsi l'utilisateur dans ses choix.

— **Visualisation et analyse des poids aléatoires** : L'utilisateur peut consulter la matrice complète des poids attribués à chaque case, avec les obstacles mis en évidence. Des statistiques détaillées (minimum, maximum, moyenne, somme) sont fournies pour l'ensemble des poids et spécifiquement pour les obstacles sélectionnés, permettant de comprendre comment Gurobi a optimisé la sélection.

— **Vérification automatique des contraintes** : Après génération, toutes les contraintes (totaux, par ligne, par colonne, absence de séquence 101) sont vérifiées. En cas de violation, des messages d'erreur précis localisent le problème.

— **Comparaison exhaustive des algorithmes** : Notre interface exécute automatiquement les trois algorithmes implémentés dans la question (b) — BFS, Dijkstra et Bellman — sur la même instance. Cela permet une comparaison directe de leurs performances en termes de temps de solution optimal, temps de calcul, et nombre de commandes.

— **Sauvegarde flexible des données** : L'utilisateur peut sauvegarder la grille générée, la matrice des poids et les résultats des algorithmes dans des fichiers texte, facilitant la reproduction des expériences et l'analyse ultérieure.

6.4 Exemple d'exécution complet : grille 50×50 avec 100 obstacles

Pour illustrer le fonctionnement complet de notre système, nous présentons un exemple avec $M = 50$, $N = 50$ et $P = 100$.

Étape 1 : Génération de la grille La grille est générée en 0,29 secondes par Gurobi. La figure 3 montre une représentation visuelle où les obstacles sont marqués par 'X'.

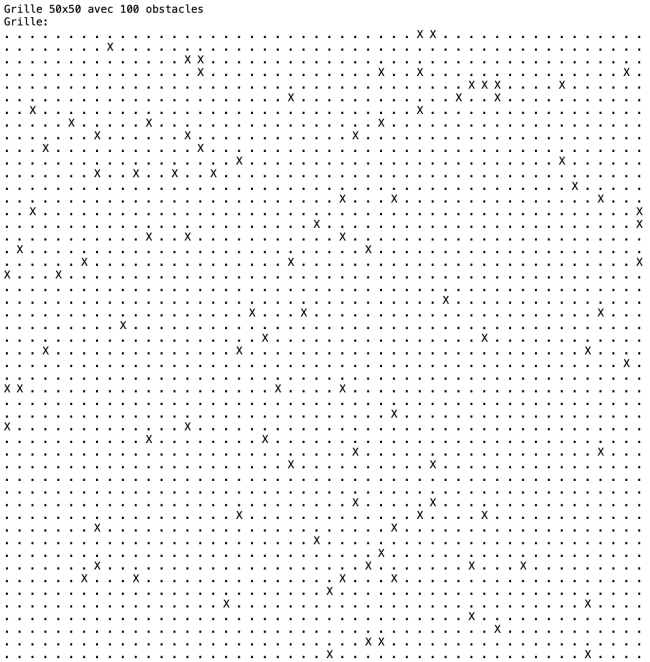

FIGURE 3 – Grille 50×50 générée avec 100 obstacles (représentés par 'X')

Étape 2 : Visualisation des poids aléatoires Le système affiche la matrice des poids aléatoires attribués à chaque case, avec les obstacles identifiés entre crochets. Voici un extrait de la sortie :

```
============================================================
MATRICE DES POIDS ALÉATOIRES (0-1000)
============================================================
      0    1    2    3    4    5    ...   49
 0: 550  538  238  877  374  875   ...  433
 1: 993  215  142  448  472  691   ...  398
```

```
  ...
 49: 983  502  755  365  671  320   ...  177
============================================================
STATISTIQUES DES POIDS:
  Tous les poids: min=0, max=1000, moyenne=505.9
  Poids des obstacles (100): min=0, max=61, moyenne=23.4
  Somme des poids des obstacles: 2341
```

Analyse des poids :

- **Poids moyens** : toutes les cases = 505,9 ; obstacles seulement = 23,4 (ratio = 4,6%).
- **Obstacles extrêmes** : poids minimum = 0 (aux positions (12,44), (16,26), (22,46), (42,40)) ; poids maximum = 61 (position (4,37)).

Étape 3 : Choix des paramètres du robot L'utilisateur sélectionne un point de départ en (0,0), un point d'arrivée en (50,50) et une orientation initiale vers le nord.

Étape 4 : Calcul et affichage des solutions par les trois algorithmes Les trois algorithmes développés dans la question (b) — BFS, Dijkstra et Bellman — sont appliqués pour trouver le chemin optimal. Les résultats sont présentés dans le tableau 6 :

Algorithme	Temps solution	Temps calcul (s)	Nombre de commandes
BFS	38 secondes	0,300591	38
Dijkstra	38 secondes	0,367222	38
Bellman	38 secondes	1,681684	38

TABLE 6 – Résultats des trois algorithmes (développés dans la question b) sur la grille 50×50 avec 100 obstacles

6.5 Observations et validation

1. **Conformité à l'énoncé** : Notre interface remplit toutes les exigences de la partie (e) : modélisation par programme linéaire, utilisation de Gurobi pour la génération, et affichage des solutions obtenues par les algorithmes développés à la question (b) — les trois algorithmes que nous avons implémentés et analysés.
2. **Cohérence des algorithmes** : Les trois algorithmes produisent le même temps de solution (38 secondes), ce qui valide l'optimalité des solutions trouvées et confirme que nos différentes implémentations sont correctes.
3. **Performance comparative** : Bien que les trois algorithmes trouvent la même solution optimale, leurs temps de calcul diffèrent significativement : BFS est le plus rapide (0,300591 s), suivi de Dijkstra (+22%) et de Bellman (+460%). Ces résultats sont cohérents avec nos analyses de complexité effectuées dans la question (b).

6.6 Processus complet de génération et test

Algorithm 1 Processus de génération et test d'instance

Require: M, N, P (paramètres utilisateur)
Ensure: Instance valide et comparaison des algorithmes
Générer $w_{ij} \sim U(0, 1000)$ pour chaque case (i, j)
Résoudre le programme linéaire avec Gurobi pour obtenir la grille
Vérifier toutes les contraintes (totaux, lignes, colonnes, 101)
Calculer tous les croisements sûrs disponibles
Demander à l'utilisateur de choisir départ, arrivée et orientation
Exécuter les trois algorithmes sur la même instance
Comparer les résultats et sauvegarder les statistiques
Offrir la sauvegarde de l'instance et des poids

7 Conclusion

Ce projet de MOGPL nous a permis de mettre en application les concepts fondamentaux de recherche opérationnelle et d'algorithmique étudiés en cours. Nous avons développé une solution complète au problème de la "balade du robot" qui répond à toutes les exigences du sujet.

7.1 Réalisations principales

Notre travail s'est structuré autour des cinq parties demandées :

1. **Modélisation graphique (a)** : Nous avons formulé le problème comme un problème de plus court chemin dans un graphe d'états, définissant précisément les sommets (positions et orientations) et les arcs (déplacements et rotations).
2. **Implémentation et analyse algorithmique (b)** : Nous avons développé trois algorithmes (BFS, Dijkstra, Bellman) avec une analyse détaillée de leur complexité théorique ($O(M \times N)$, $O(M \times N \log(M \times N))$, $O((M \times N)^2)$ respectivement).
3. **Évaluation expérimentale (c et d)** : Nous avons conduit des tests systématiques sur 100 instances au total, variant la taille de la grille (de 10×10 à 50×50) et le nombre d'obstacles (de 10 à 50). Ces tests ont validé nos analyses théoriques et montré la supériorité pratique de BFS.
4. **Génération de grilles avec contraintes (e)** : Nous avons formulé un programme linéaire en nombres entiers pour générer des grilles aléatoires respectant les contraintes spécifiées, et implémenté une interface interactive complète.

7.2 Résultats clés

Notre analyse comparative a révélé plusieurs résultats intéressants :

— **Performance relative des algorithmes** : BFS est le plus rapide en pratique pour ce problème, suivi de Dijkstra, tandis que Bellman présente une complexité trop élevée pour les grandes instances.

— **Influence des paramètres** : Le temps de calcul augmente avec la taille de la grille mais présente une relation non monotone avec le nombre d'obstacles, les configurations intermédiaires étant souvent les plus complexes.

— **Cohérence des solutions** : Les trois algorithmes produisent systématiquement des solutions optimales de même coût, validant ainsi leur correction mutuelle.

— **Faisabilité pratique** : Même pour les instances maximales (grilles 50×50), les temps de calcul restent raisonnables (moins de 0.02 secondes pour BFS).

7.3 Apports pédagogiques

Ce projet nous a permis de :

— Appliquer concrètement les concepts de modélisation par graphes et de recherche de plus court chemin

— Comparer empiriquement différentes approches algorithmiques pour un même problème

— Développer une interface interactive utilisant des outils d'optimisation (Gurobi)

— Conduire une analyse expérimentale systématique avec génération et sauvegarde automatique des résultats

7.4 Perspectives

Bien que notre implémentation réponde pleinement aux exigences du sujet, plusieurs extensions pourraient être envisagées :

— Intégration d'heuristiques pour accélérer la recherche sur de très grandes grilles

— Interface graphique avec visualisation du déplacement du robot

— Génération de grilles avec des contraintes supplémentaires (connectivité des zones libres, etc.)

En conclusion, ce projet a été l'occasion de développer une solution complète et fonctionnelle au problème de planification de chemin pour un robot, mettant en œuvre des compétences en modélisation, algorithmique et programmation acquises dans le cadre du cours MOGPL.