Sorbonne Université
Faculté des Sciences et Ingénierie

Projet MOGPL

La balade du robot

Réalisé par :

Salah Djamel
Azizi Naoufel

Encadrant :

Evripidis Bampis

Année Universitaire : 2025–2026

Département Informatique

Contents

1 Introduction générale

Le projet que nous présentons s'inscrit dans le cadre du cours de *Modélisation et Optimisation pour la Gestion de la Production et de la Logistique* (MOGPL). Il a pour objectif d'étudier la navigation optimale d'un robot se déplaçant sur une grille dont certaines cases peuvent contenir des obstacles. Le robot évolue selon un ensemble de commandes prédéfinies — rotations et déplacements — chacune ayant un coût unitaire. L'enjeu consiste à déterminer une séquence minimale de commandes permettant au robot d'atteindre une position cible à partir d'un état initial.

Ce problème, en apparence simple, combine plusieurs difficultés : la présence d'obstacles, la prise en compte de l'orientation du robot, et les contraintes géométriques imposées par la structure du déplacement. Pour le traiter de manière rigoureuse, nous proposons d'abord une modélisation sous forme de graphe orienté, permettant d'interpréter la navigation comme un problème de plus court chemin. Cette abstraction nous conduit naturellement à l'utilisation d'un parcours en largeur (BFS), parfaitement adapté à la structure uniforme des coûts.

Afin d'évaluer en pratique l'efficacité de l'algorithme proposé, nous mettons ensuite en place une série d'expériences portant d'une part sur la taille de la grille, et d'autre part sur le nombre d'obstacles. Ces analyses permettent de valider la cohérence entre les résultats théoriques et les performances observées.

Enfin, la dernière partie du projet concerne la génération contrôlée d'obstacles sur la grille via la résolution d'un programme linéaire en nombres entiers, résolu à l'aide du solveur Gurobi. Cette modélisation offre un moyen systématique et flexible de produire des environnements de test réalistes, tout en respectant un ensemble de contraintes structurelles.

L'ensemble du rapport vise ainsi à articuler modélisation, algorithmique, optimisation et expérimentation, dans une approche complète typique des problématiques rencontrées en recherche opérationnelle.

2 Formulation du problème comme un plus court chemin dans un graphe orienté

Pour modéliser le déplacement du robot, nous adoptons une représentation par un graphe orienté $G = (V, E)$ dans lequel chaque sommet encode un état complet du robot et chaque arc représente une commande réalisable. Cette modélisation permet de transformer le problème de navigation en un problème combinatoire classique : la recherche d'un plus court chemin.

2.1 Ensemble des sommets V

Un sommet du graphe correspond à un état (i, j, o), où :

- (i, j) désigne une case libre de la grille (i.e. une case de valeur 0 dans l'instance) ;
- $o \in \{\text{nord}, \text{sud}, \text{est}, \text{ouest}\}$ représente l'orientation du robot.

Nous devons distinguer explicitement l'orientation car elle conditionne la possibilité d'effectuer une avance ; deux états situés sur la même case mais avec deux orientations différentes ne donnent pas accès aux mêmes actions.

On obtient ainsi l'ensemble des sommets :

$$V = \{(i, j, o) \mid (i, j) \text{ est libre}, \ o \in \{\text{nord}, \text{sud}, \text{est}, \text{ouest}\}\}.$$

Chaque case libre génère exactement quatre sommets, ce qui donne au plus $4MN$ états. Cette augmentation du nombre de sommets est un coût nécessaire pour représenter précisément les contraintes directionnelles du robot.

2.2 Ensemble des arcs E.

Un arc correspond à l'exécution d'une commande élémentaire durant exactement une seconde. Toutes les commandes ayant la même durée, le graphe obtenu est un graphe non pondéré au sens du plus court chemin.

Deux catégories d'actions sont possibles : tourner et avancer.

2.2.1 Commandes de rotation.

Depuis un état (i, j, o), le robot peut :

- tourner à gauche : $o' = \text{gauche}(o)$,
- tourner à droite : $o' = \text{droite}(o)$.

Ces actions ne modifient pas la position, seulement l'orientation. Elles sont modélisées par les arcs :

$$(i, j, o) \to (i, j, o') \quad \text{de coût 1.}$$

Ces transitions jouent un rôle important car elles permettent au robot de se réorienter avant d'avancer dans une direction donnée.

2.2.2 Commandes d'avance.

Le robot peut également effectuer une commande `Avance(n)` avec $n \in \{1, 2, 3\}$. Une telle commande est possible si et seulement si les n cases successives dans la direction correspondant à l'orientation o sont libres d'obstacles.

Si ces conditions sont satisfaites, il atteint une nouvelle position (i', j'), et l'on ajoute au graphe l'arc :

$$(i, j, o) \to (i', j', o) \quad \text{de coût 1.}$$

Cette modélisation permet de conserver un graphe homogène dans lequel chaque commande correspond exactement à un arc, quel que soit le nombre de cases parcourues. Cela simplifie grandement la structure du problème.

2.3 Sommet initial et sommets d'arrivée.

Le sommet initial est donné par :

$$s = (D1, D2, o_0),$$

où $(D1, D2)$ et o_0 sont fournis dans le fichier d'entrée.

La destination est atteinte dès que le robot arrive sur la case $(F1, F2)$, quelle que soit son orientation finale. Ainsi, tout état :

$$t = (F1, F2, o), \quad o \in \{\text{nord}, \text{sud}, \text{est}, \text{ouest}\},$$

est considéré comme une solution valide. Cela évite d'exiger des rotations supplémentaires inutiles qui n'influencent pas la position finale.

2.4 Objectif du problème.

L'objectif consiste à déterminer une séquence de commandes de durée minimale permettant d'aller de s à un sommet d'arrivée $(F1, F2, o)$. Formellement :

$$\text{trouver le plus court chemin entre } s \text{ et } (F1, F2, o), \quad \forall o.$$

Comme chaque action dure une seconde, le temps total correspond exactement au nombre d'arcs dans le chemin.

2.5 Méthode de résolution.

Comme tous les arcs possèdent un coût identique, le graphe est non pondéré. Dans ce cas, la recherche du plus court chemin s'effectue de manière optimale grâce au **parcours en largeur (BFS)**. Cet algorithme explore le graphe couche par couche : il examine tous les états à distance 1, puis à distance 2, etc. Le premier état d'arrivée rencontré correspond donc nécessairement à un chemin minimal.

L'algorithme de Dijkstra pourrait également être utilisé, mais dans le contexte présent, où tous les coûts sont égaux, il se réduit exactement à un BFS, tout en introduisant une complexité supplémentaire liée à l'utilisation d'une file de priorité. Ainsi, BFS est à la fois plus simple, plus efficace et parfaitement adapté au problème.

Cette formulation en graphe orienté constitue une base solide pour le développement d'un algorithme fiable et optimal, et elle s'intègre très naturellement dans la suite du projet, où un parcours en largeur sur l'espace des états permettra de déterminer la séquence minimale de commandes à exécuter.

3 Algorithme de résolution et complexité

La modélisation effectuée dans la question (a) nous conduit naturellement à considérer un espace d'états de taille $O(MN)$ où chaque état encode à la fois la position du robot et son orientation. Le problème consiste alors à déterminer une séquence minimale de commandes pour atteindre la destination. Du fait que toutes les actions ont un coût identique égal à 1, l'algorithme le plus adapté est le parcours en largeur (BFS), qui permet d'explorer cet espace de manière exhaustive mais ordonnée.

Avant de passer à la justification théorique, nous présentons ci-dessous le pseudocode de l'algorithme.

3.1 Pseudo-code

Input: Grille $grid[M][N]$, position de départ (D_1, D_2), orientation initiale o_0, position d'arrivée (F_1, F_2)
Output: Temps minimal et liste des commandes, ou `-1` si aucun chemin
Créer un tableau `visited[M][N][4]` initialisé à **faux**
Créer des tableaux `prev_i`, `prev_j`, `prev_o`, `prev_cmd`
Créer une file FIFO Q
`visited[`D_1`][`D_2`][`o_0`]` $\leftarrow$ **vrai**
Enfiler (D_1, D_2, o_0) avec distance 0
while Q *n'est pas vide* **do**
  $(i, j, o), d \leftarrow$ Défiler Q
  if $(i, j) = (F_1, F_2)$ **then**
    retourner la trajectoire obtenue en remontant les tableaux `prev`
  end
  `// 1) Rotations gauche et droite`
  foreach $r \in \{G, D\}$ **do**
    $o' \leftarrow$ orientation résultante
    if *`visited[i][j][o']` est* ***faux*** **then**
      `visited[i][j][o']` $\leftarrow$ **vrai**
      Stocker (i, j, o) dans `prev_i,j,o'` et r dans `prev_cmd`
      Enfiler (i, j, o') avec distance $d + 1$
    end
  end
  `// 2) Avances de 1, 2 ou 3 cases`
  for $n \in \{1, 2, 3\}$ **do**
    $(i', j') \leftarrow$ case atteinte après une avance de n cases depuis (i, j) dans l'orientation o
    if *les n cases intermédiaires sont libres et (i', j') est dans la grille* **then**
      if *`visited[i'][j'][o]` est* ***faux*** **then**
        `visited[i'][j'][o]` $\leftarrow$ **vrai**
        Stocker (i, j, o) dans `prev_i,j,o` et an dans `prev_cmd`
        Enfiler (i', j', o) avec distance $d + 1$
      end
    end
  end
end
return `-1// [`
r]Aucun chemin trouvé

Algorithm 1: BFS pour le déplacement optimal du robot

3.1.1 Principe général de l'algorithme

Le parcours en largeur (BFS) est particulièrement adapté parce qu'il explore les états par niveaux . Cela signifie que :

- d'abord, il examine tous les états accessibles en une seule commande ;
- puis ceux accessibles en deux commandes ;
- puis ceux accessibles en trois commandes ; etc.

Cette organisation garantit qu'au moment où l'algorithme atteint pour la première fois une position (F_1, F_2), la séquence de commandes utilisée est nécessairement de longueur minimale. C'est précisément la propriété qui rend BFS optimal pour les graphes non pondérés.

L'algorithme repose sur une file FIFO Q contenant les états à explorer. À chaque itération, un état est retiré de la file, puis les nouvelles transitions valides sont générées :

- deux transitions de rotation ;
- jusqu'à trois transitions d'avance.

Chaque nouvel état non visité est inséré en fin de file pour exploration future.

3.1.2 Faisabilité des mouvements

Il ne suffit pas de considérer les arcs du graphe théorique : il faut également vérifier que le mouvement correspondant est réalisable sur la grille fournie en entrée. Deux tests se révèlent essentiels :

- **vertex_ok** : garantit que le sommet (i, j) représente une position sûre pour le robot. Cette vérification tient compte de la taille du robot et impose que les quatre cases adjacentes au point d'articulation soient libres.
- **edge_ok** : s'assure que le rail reliant deux sommets consécutifs n'est bloqué ni par un obstacle adjacent, ni par un dépassement de la grille.

Ces contrôles rendent la navigation réaliste : ils empêchent le robot de frôler un mur ou de traverser certaines zones impraticables. Ils introduisent également une dimension géométrique dans le BFS, contrairement à un simple graphe de cases adjacentes.

3.1.3 Reconstruction de la solution

Pour reconstituer la séquence optimale de commandes, BFS mémorise la provenance de chaque état visité grâce à des tableaux `parent`. Lorsque l'état d'arrivée est atteint, on remonte ces pointeurs jusqu'à l'état initial, ce qui donne la séquence des commandes dans l'ordre inverse.

Le renversement de la liste reconstitue alors la séquence correcte. Cette étape est très peu coûteuse en temps : la complexité est linéaire en la longueur du chemin, qui est généralement beaucoup plus petite que la taille du graphe.

3.1.4 Analyse de complexité

L'espace des états comporte au plus :

$$4MN \quad \text{états.}$$

Le facteur 4 provient des orientations possibles.

Pour chaque état, le robot peut générer un nombre constant de successeurs (au plus 5 : deux rotations et trois avances). Le BFS traite chaque état au plus une fois. Sa complexité est donc linéaire en le nombre d'états :

$$O(|V| + |E|) = O(MN).$$

Cette complexité garantit que l'algorithme est performant même pour des grilles de dimensions non triviales (par exemple 50×50), ce qui correspond exactement aux tailles utilisées dans les questions expérimentales.

La mémoire utilisée par les tableaux `visited` et `parent` est également proportionnelle à la taille de l'espace des états, soit :

$$O(MN).$$

En pratique, BFS s'avère extrêmement efficace : même pour des instances complexes, les temps de calcul restent de l'ordre de quelques millisecondes pour des tailles de grilles modestes, comme le montrent les résultats expérimentaux.

3.1.5 Conclusion de la section

La structure simple et régulière de l'espace d'états, ainsi que l'uniformité des coûts, font du parcours en largeur un choix naturel et optimal pour résoudre ce problème de navigation. De plus, l'intégration des tests géométriques `vertex_ok` et `edge_ok` permet d'appliquer BFS dans un environnement réaliste, sans perdre sa garantie d'optimalité. Le résultat est un algorithme à la fois rigoureux, robuste et très efficace en pratique.

4 Évaluation expérimentale du temps de calcul

Afin de mieux comprendre le comportement de notre algorithme dans des situations concrètes, nous avons mené une première campagne d'expérimentations visant à mesurer son temps d'exécution en fonction de la taille de la grille. Cette étude permet d'observer empiriquement comment l'augmentation de la dimension de l'espace des états influence la performance du BFS, ce qui complète l'analyse théorique effectuée dans la section précédente.

Pour cela, nous faisons varier la taille de la grille selon

$$N \in \{10, 20, 30, 40, 50\}.$$

Pour chaque valeur de N, dix instances sont générées aléatoirement. Chaque instance contient exactement $P = N$ obstacles, ce qui assure une densité proportionnelle à la taille de la grille. Les positions de départ et d'arrivée sont choisies aléatoirement mais strictement à l'intérieur de la grille afin d'éviter les sommets interdits situés sur la bordure. L'orientation initiale est sélectionnée uniformément parmi les quatre orientations possibles.

4.1 Script expérimental.

La procédure complète est automatisée grâce au script `experiences_Qc.py`. Celui-ci :

- génère les dix instances correspondant à chaque valeur de N ;
- écrit successivement ces instances dans un fichier d'entrée conforme au format imposé ;
- exécute notre algorithme BFS pour chaque instance ;
- mesure précisément le temps d'exécution grâce à un chronomètre haute résolution ;
- stocke pour chaque instance non seulement la solution mais aussi sa durée d'exécution.

Les temps sont ensuite agrégés afin de calculer pour chaque taille de grille :

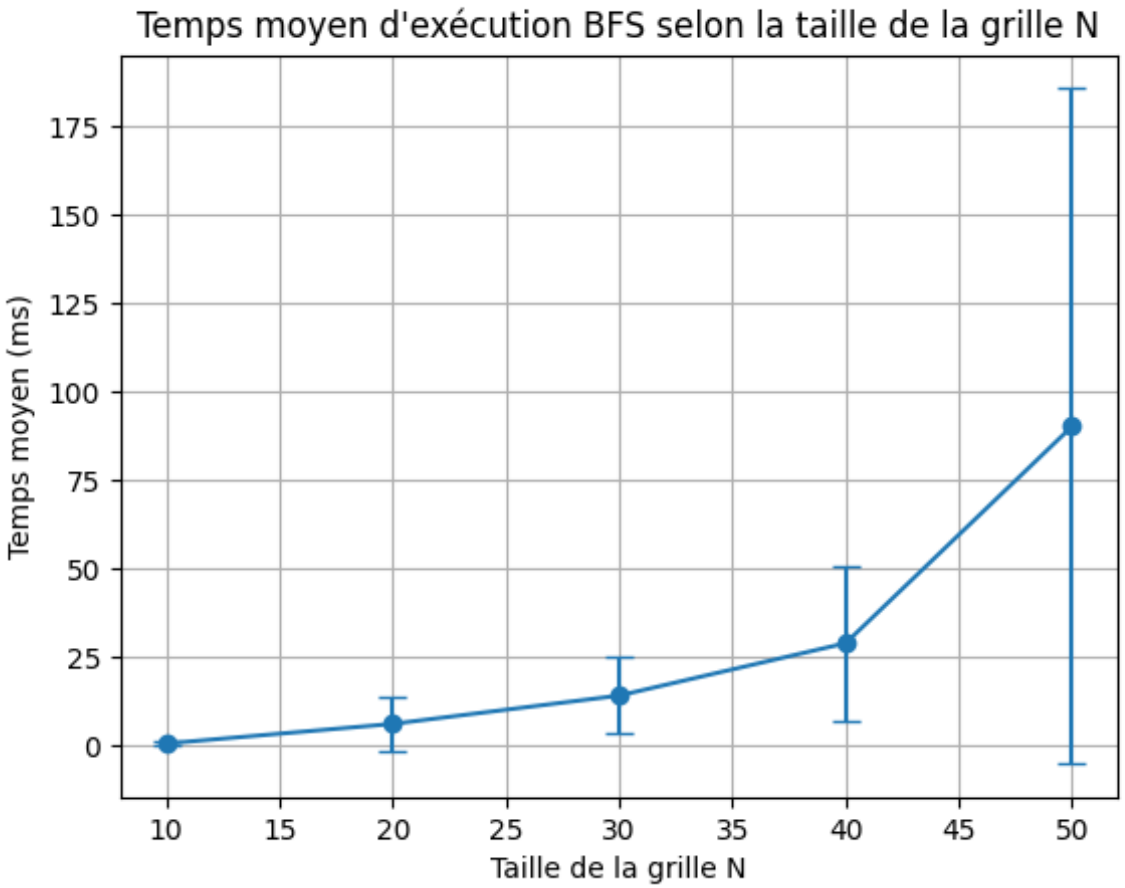
- le temps moyen d'exécution ;
- l'écart-type, qui renseigne sur la variabilité entre les instances.

L'ensemble du processus garantit des résultats reproductibles, obtenus dans des conditions contrôlées et homogènes.

4.2 Résultats obtenus.

Les résultats sont présentés dans le tableau suivant :

Taille N	**Temps moyen (ms)**	**Écart-type (ms)**
10	0.548	0.505
20	6.105	7.769
30	14.108	10.729
40	28.922	21.900
50	90.273	95.533



4.3 Analyse.

Les résultats montrent clairement une croissance globale du temps d'exécution avec la taille de la grille. Cette observation est en accord avec la complexité théorique en $O(N^2)$ du BFS appliqué à un espace d'états comportant $4MN$ sommets. Plus N augmente, plus le nombre d'états à explorer avant d'atteindre la destination devient important.

Cependant, un phénomène notable est la forte variabilité observée pour les valeurs les plus élevées de N, comme en témoignent les écarts-types particulièrement importants pour $N = 40$ et $N = 50$. Cette variabilité est directement liée à la structure aléatoire des obstacles :

- certaines configurations ouvrent un grand nombre de chemins faisables, ce qui permet à BFS d'atteindre rapidement la cible ;
- d'autres configurations créent des zones quasiment labyrinthiques, obligeant l'algorithme à explorer une grande partie de l'espace des états avant de trouver un chemin — voire à conclure qu'il n'en existe aucun.

Ainsi, deux grilles de même taille peuvent conduire à des temps d'exécution très différents. Ce comportement illustre parfaitement un trait caractéristique de BFS : sa sensibilité à la structure locale du graphe.

Malgré ces fluctuations, la tendance générale est bien celle attendue : le temps d'exécution augmente avec la taille de la grille, confirmant empiriquement la complexité théorique. Les résultats valident donc la cohérence et la pertinence de l'algorithme dans ce cadre d'application.

5 Influence du nombre d'obstacles sur le temps de calcul

Cette seconde série d'expérimentations vise à étudier le comportement du BFS pour une grille de taille fixe lorsque l'on fait varier la densité d'obstacles. Contrairement à la question précédente, la taille de la grille est ici constante, fixée à 20×20, tandis que le nombre d'obstacles varie selon

$$P \in \{10, 20, 30, 40, 50\}.$$

L'objectif est d'observer dans quelle mesure la difficulté du problème dépend non seulement de la taille de l'espace d'états, mais aussi de la structure géométrique du graphe, influencée par la présence d'obstacles.

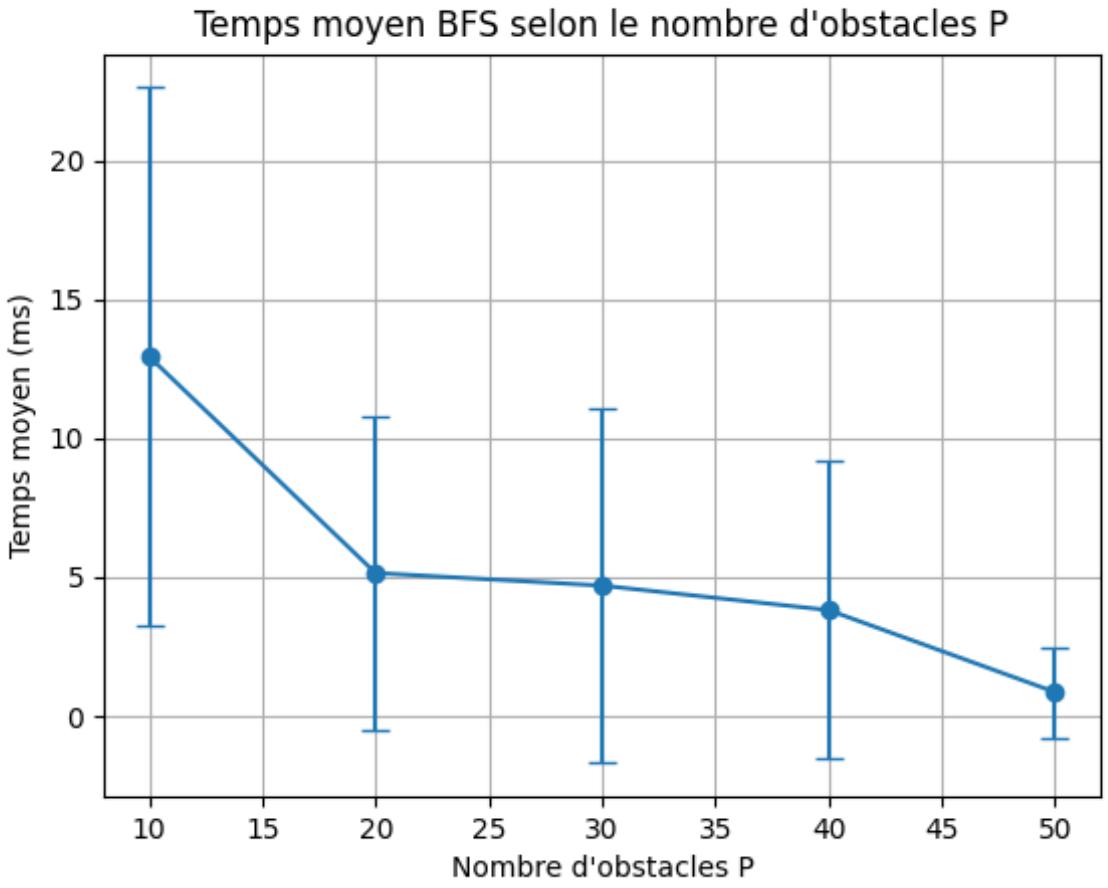
Comme précédemment, pour chaque valeur de P, dix instances aléatoires sont générées, en veillant à ce que :

- les positions de départ et d'arrivée soient strictement internes à la grille ;
- aucun obstacle ne recouvre ces positions ;
- l'orientation initiale du robot soit tirée au hasard.

Le script expérimental utilisé est le même que pour la question (c), garantissant une méthodologie homogène entre les deux séries de tests.

5.1 Résultats expérimentaux

Nombre d'obstacles P	**Temps moyen (ms)**	**Écart-type (ms)**
10	12.938	9.674
20	5.150	5.667
30	4.697	6.385
40	3.816	5.345
50	0.865	1.630



5.2 Analyse.

Contrairement au cas précédent, ici l'espace des états est fixe (une grille 20×20). L'évolution du temps d'exécution dépend donc uniquement de la manière dont les obstacles restreignent la liberté de mouvement du robot.

On observe une tendance nette :

le temps d'exécution diminue lorsque le nombre d'obstacles augmente.

Ce phénomène peut être expliqué par plusieurs facteurs :

- Lorsque P est faible, l'espace est relativement dégagé. BFS peut alors explorer une grande portion de la grille avant d'atteindre l'objectif, ce qui augmente fortement le temps d'exécution.
- À mesure que P augmente, le nombre de chemins possibles diminue, ce qui réduit mécaniquement la quantité d'états explorables.
- Certaines configurations fortement encombrées conduisent même à un échec rapide (aucun chemin), ce qui diminue encore le temps d'exploration.

Les écarts-types observés restent cependant élevés, reflétant la variabilité induite par la génération aléatoire des obstacles : une grille très encombrée mais comportant un couloir libre unique peut être résolue plus rapidement qu'une grille peu encombrée mais labyrinthique.

Ainsi, ces résultats mettent en évidence un comportement très intéressant : le BFS n'est pas seulement sensible à la taille de la grille, mais également à la structure des obstacles, qui peut réduire ou augmenter drastiquement la taille effective de l'espace accessible.

En résumé, l'augmentation du nombre d'obstacles tend à rendre l'exploration plus rapide, non pas parce que l'algorithme devient plus efficace, mais parce que la structure du graphe sous-jacent se simplifie du point de vue de l'exploration.

6 Modélisation d'un problème sous la forme d'un programme linéaire

Dans cette partie, l'objectif est de générer automatiquement une configuration d'obstacles satisfaisant plusieurs contraintes structurelles tout en minimisant un critère de coût imposé. Cette approche est particulièrement utile dans le contexte du projet, car elle permet de produire des grilles plausibles, contrôlées et variées, sans avoir à placer les obstacles manuellement. L'idée est de considérer que chaque case de la grille possède un poids reflétant son attractivité pour placer un obstacle : plus ce poids est faible, plus il est avantageux d'y placer un obstacle.

La génération des obstacles devient alors un problème d'optimisation combinatoire, qui peut être formulé très naturellement sous forme d'un programme linéaire en nombres entiers, résoluble efficacement par un solveur d'optimisation moderne comme Gurobi.

6.1 Variables de décision.

Pour chaque case (i, j) de la grille, avec $i = 1, \ldots, M$ et $j = 1, \ldots, N$, on introduit une variable binaire :

$$x_{ij} = \begin{cases} 1 & \text{si un obstacle est placé sur la case } (i,j), \\ 0 & \text{sinon.} \end{cases}$$

Ces variables sont les éléments essentiels du modèle : elles déterminent entièrement la configuration finale de la grille. L'usage de variables binaires est indispensable car le placement des obstacles est un choix discret.

6.2 Données.

Chaque case (i, j) est associée à un poids $w_{ij} \in \{0, \ldots, 1000\}$, généré aléatoirement. Ces poids introduisent de la variabilité dans la génération des instances : deux grilles de même taille mais avec des poids différents conduiront généralement à des configurations d'obstacles distinctes.

On note P le nombre total d'obstacles à placer. Il constitue un paramètre d'entrée du modèle et permet de contrôler le degré de congestion de la grille.

6.3 Objectif.

Le but est de sélectionner P cases dans la grille sur lesquelles seront placés les obstacles, de manière à minimiser la somme des poids correspondants :

$$\min \sum_{i=1}^{M} \sum_{j=1}^{N} w_{ij}\, x_{ij}.$$

Cette fonction objectif privilégie les cases de faible poids. Ainsi, parmi toutes les configurations satisfaisant les contraintes, le solveur choisira celle qui minimise le coût global, ce qui introduit un comportement équilibré et non déterministe dans la construction des obstacles.

6.4 Contraintes.

Un certain nombre de contraintes viennent structurer le placement des obstacles afin d'éviter des configurations pathologiques (obstacles trop concentrés, colonnes bloquées, motifs rendant la navigation impossible, etc.).

- **Nombre total d'obstacles.**
  Le modèle doit poser exactement P obstacles :

  $$\sum_{i=1}^{M}\sum_{j=1}^{N} x_{ij} = P.$$

  Cette contrainte garantit que toutes les instances générées sont comparables entre elles lorsque l'on fixe P.

- **Bornes par ligne.**
  Pour éviter qu'une ligne soit surchargée, ce qui pourrait bloquer complètement une zone entière de la grille, on impose :

  $$\sum_{j=1}^{N} x_{ij} \leq \frac{2P}{M} \quad \forall i.$$

  Cette borne assure une répartition équilibrée des obstacles sur les lignes.

- **Bornes par colonne.**
  De façon symétrique :

  $$\sum_{i=1}^{M} x_{ij} \leq \frac{2P}{N} \quad \forall j.$$

  Cette condition empêche la formation de colonnes totalement obstruées, ce qui pourrait rendre impossible la navigation d'un côté à l'autre de la grille.

- **Interdiction du motif 101 sur les lignes.**
  Certaines configurations d'obstacles rendent la navigation extrêmement complexe, notamment lorsqu'une case libre est entourée par deux obstacles :

  $$x_{i,j} + x_{i,j+2} \leq 1 + x_{i,j+1} \quad \text{pour } i = 1, \ldots, M,\ j = 1, \ldots, N-2.$$

  Cette contrainte empêche exactement l'apparition du motif `1--0--1`. Elle agit comme une contrainte de cohérence locale .

- **Interdiction du motif 101 sur les colonnes.** De la même manière :

  $$x_{i,j} + x_{i+2,j} \leq 1 + x_{i+1,j} \quad \text{pour } i = 1, \ldots, M-2,\ j = 1, \ldots, N.$$

  Cela complète la protection verticale contre des motifs défavorables à la navigation.

6.5 Domaine des variables.

Les variables étant binaires, on impose :

$$x_{ij} \in \{0, 1\} \quad \forall i, j.$$

Cette contrainte rend le programme linéaire non seulement combinatoire mais également NP-difficile dans le cas général. Toutefois, les dimensions modestes du problème dans ce projet permettent à Gurobi de le résoudre quasi instantanément.

6.6 Discussion sur le modèle.

Ce programme linéaire présente un compromis intéressant entre :

- **variabilité**, grâce aux poids aléatoires,
- **contrôle structurel**, via les contraintes,
- **résolution efficace**, grâce à l'usage d'un solveur performant.

Il évite, en particulier, la génération d'obstacles concentrés dans une zone ou formant des motifs qui rendent les grilles irréalistes ou trop difficiles.

Il constitue donc un excellent générateur d'instances pour tester l'algorithme de navigation.

6.7 Interface de génération et de résolution

Nous avons conçu une interface Python permettant d'enchaîner automatiquement :

1. la génération d'une grille d'obstacles par résolution du PLNE ;
2. le choix des paramètres du robot (départ, orientation, arrivée) ;
3. l'exécution du BFS pour déterminer la trajectoire optimale.

L'utilisateur peut ainsi tester rapidement de nombreux scénarios différents.

6.8 Génération des obstacles avec Gurobi.

L'utilisateur saisit la taille (M, N) ainsi que le nombre d'obstacles P. Le programme :

- génère les poids w_{ij} ;
- construit le programme linéaire précédent ;
- demande à Gurobi de le résoudre ;
- extrait la grille optimisée sous forme d'une matrice binaire ;
- affiche cette grille pour visualisation.

Gurobi trouve en général la solution optimale en une fraction de seconde.

6.9 Sélection du point de départ et de l'arrivée.

Le robot se déplaçant sur les *sommets* de la grille (et non les cases), il est indispensable de choisir des coordonnées strictement intérieures :

$$1 \leq i \leq M-1, \qquad 1 \leq j \leq N-1.$$

Cette contrainte provient directement de `vertex_ok`, qui interdit toute position située sur la bordure, car le robot ne pourrait pas y être physiquement centré.

L'utilisateur renseigne également l'orientation initiale parmi :

$$\{\text{nord}, \text{est}, \text{sud}, \text{ouest}\}.$$

6.10 Lancement du BFS.

Une fois la configuration validée, l'interface appelle directement l'algorithme :

`bfs(grid, D1, D2, start_o, F1, F2)`.

Le résultat peut être :

- une séquence optimale de commandes, si un chemin existe ;
- `None` si la destination n'est pas atteignable.

La solution est ensuite affichée à l'utilisateur de manière lisible.

6.11 Résumé.

Cette interface constitue un outil complet qui :

- génère automatiquement des instances cohérentes à l'aide du modèle linéaire ;
- permet un choix flexible des paramètres du robot ;
- exécute immédiatement l'algorithme de navigation ;
- fournit une solution optimale ou détecte l'absence de chemin.

Elle relie ainsi harmonieusement les aspects théoriques (modélisation PLNE) et algorithmiques (BFS sur l'espace d'états).

7 Conclusion générale

Ce travail nous a permis d'aborder de manière intégrée plusieurs notions fondamentales de modélisation et d'optimisation. La première étape, consistant à reformuler le problème comme un plus court chemin dans un graphe orienté, a montré que l'interprétation correcte d'un problème réel peut souvent le ramener à une structure mathématique bien connue. Grâce à cette modélisation, l'utilisation du parcours en largeur (BFS) s'est imposée comme la méthode la plus naturelle et la plus efficace pour déterminer une trajectoire optimale, toutes les actions ayant un coût uniforme.

Les expérimentations réalisées ont permis de confirmer les tendances théoriques : la complexité en $O(MN)$ du BFS s'observe empiriquement, avec une croissance du temps de calcul lorsque la grille s'agrandit. À l'inverse, lorsque la taille de la grille est fixe, l'augmentation du nombre d'obstacles tend à réduire la zone réellement explorée, conduisant à des temps de calcul plus faibles en pratique. Ces résultats illustrent bien la sensibilité du BFS à la structure du graphe sous-jacent.

La dernière partie du projet, consacrée à la génération d'obstacles via un programme linéaire en nombres entiers, montre comment les outils d'optimisation peuvent enrichir la modélisation d'un problème et offrir un contrôle précis sur la structure des instances générées. Cette approche met en évidence la complémentarité entre optimisation combinatoire et algorithmique, deux dimensions essentielles de la recherche opérationnelle.

Au final, ce projet nous a permis de développer une solution complète allant de la modélisation théorique à l'implémentation algorithmique, en passant par l'analyse expérimentale et la formulation d'un problème d'optimisation. Il illustre de manière concrète l'importance d'une démarche structurée combinant rigueur mathématique, efficacité algorithmique et pertinence expérimentale.