Sorbonne Université

Master Androide 2ème Année

Projet MADI

Algorithmes pour la planification dans le risque

Auteurs :

Clémence BOURGUE
Ariana CARNIELLI

1 Introduction

Dans ce projet on s'intéresse au problème de planification de trajectoire d'un robot se déplaçant dans une grille rectangulaire contenant des obstacles. Le robot démarre son mouvement dans une case donnée, fixée dans ce projet comme le coin en haut à gauche de la grille, et veut atteindre une case but, fixée comme le coin en bas à droite. Pour cela, il dispose de quatre actions, correspondant à des déplacements dans les quatre directions, mais le résultat de ces actions est aléatoire : le robot a une probabilité $p > 0.5$ d'arriver dans la case cible de l'action mais une probabilité $1 - p$ d'arriver dans une case voisine de la case cible ($\frac{1-p}{2}$ pour chaque case voisine), lorsque ces cases voisines existent. Notre objectif dans ce projet est d'étudier ce problème de planification de trajectoire en le modélisant par un processus décisionnel Markovien (PDM) et sous deux situations différentes : la première lorsque chaque case a un coût et on cherche à minimiser le coût total du chemin (présentée dans les Sections 2 et 3), et la deuxième lorsque l'on cherche à minimiser plusieurs critères et chaque case a un coût pour un de ces critères, chaque critère étant identifié à une couleur (présentée dans la Section 4).

1.1 Implémentation

L'implémentation de ce projet a été faite en Python dans un fichier principal `projet_madi.py` et un fichier Jupyter notebook `tests.ipynb` contenant les tests réalisés. Une documentation du fichier `projet_madi.py` est aussi fournie.

Le fichier `projet_madi.py` contient deux classes principales, `Grille` et `Visualisation`, ainsi que plusieurs fonctions qui implémentent les techniques utilisées dans ce projet. La plupart de ces fonctions sera décrite dans la suite de ce rapport, dans les sections correspondantes à leurs implémentations. On décrit dans cette section uniquement les classes `Grille` et `Visualisation` et la fonction `simulation`, qui implémentent tout ce qui est demandé dans la partie 1 du projet.

Avant de détailler ces classes, on présente quelques choix d'implémentation fixés dans ce projet. Tout d'abord, concernant les actions à prendre, les actions d'aller vers le haut, la droite, le bas, ou la gauche sont représentés dans notre code par les entiers 0, 1, 2, et 3, respectivement. Ainsi, une stratégie pure est représentée par un tableau d'entiers en dimension 2 contenant en chaque case un des entiers $0, \ldots, 3$. Pour les stratégies mixtes, on utilise des tableaux de flottants de dimension 3 : sa case `[i, j, a]` donne la probabilité de prendre l'action a dans la case (i, j).

1.1.1 La classe `Grille`

La classe `Grille` sert à représenter un grille, qui est stockée comme un tableau principal `numpy` de dimension 2 contenant des entiers et initialisée de façon aléatoire. L'entier -1 représente un mur, alors que les autres entiers représentent la couleur de la case : dans les Sections 2 et 3, cette couleur encode le coût de chaque case, selon un tableau de coûts `tab_cost` qui est aussi passé en argument au constructeur de `Grille` et dont la taille est utilisée pour inférer le nombre de couleurs. La distribution de couleurs est faite de façon aléatoire uniforme par défaut mais la loi peut être modifiée à travers le paramètre `proba_coul` passé en argument au constructeur de `Grille`.

Dans la Section 4, on considérera des problèmes d'optimisation multi-objectifs et alors les couleurs des cases représenteront non pas le coût en soi, mais le critère correspondant à la case. La valeur du coût est stockée dans un autre tableau `numpy`, de même dimension que le tableau principal, et initialisée aléatoirement avec les chiffres de 1 à 9 selon une loi uniforme par défaut (qui peut être modifiée en une autre loi à l'aide de l'argument `proba_nb` de la classe `Grille`). La classe `Grille` stocke aussi la probabilité p qu'une action mène bien à la case cible voulue et non pas à une case voisine.

Des méthodes auxiliaires ont été implémentés dans la classe `Grille` pour simplifier la manipulation des grilles, et notamment le calcul des probabilités de transition du processus décisionnel Markovien. Les détails d'implémentation de ces méthodes sont fournis dans la documentation annexe.

1.1.2 La classe `Visualisation`

La classe `Visualisation` sert à créer des fenêtres graphiques permettant d'afficher des grilles, à y naviguer avec le robot à l'aide des flèches du clavier, à visualiser des politiques pures, et à simuler des politiques pures ou mixtes à l'aide de la touche espace, qui exécute un pas de la stratégie. Cette classe a été fortement inspirée des programmes fournis avec l'énoncé et crée la fenêtre graphique à l'aide du module `tkinter` de Python.

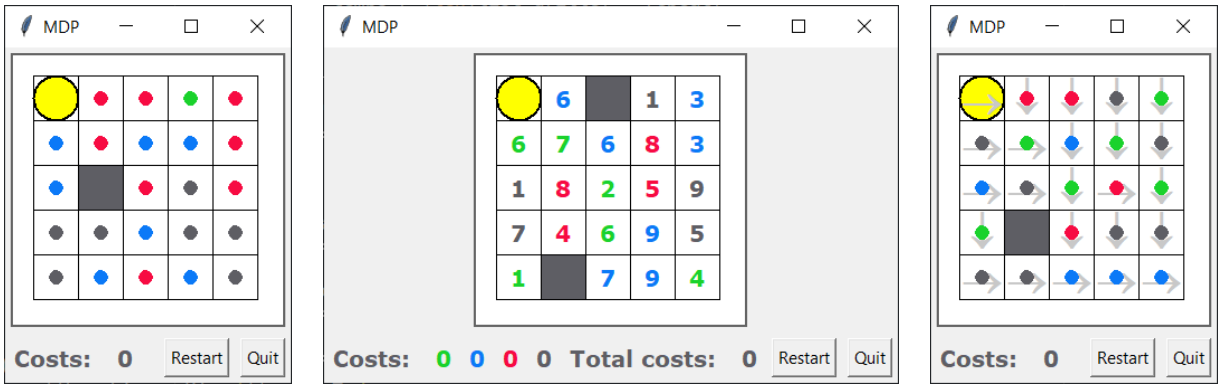
Le constructeur de la classe `Visualisation` prend en argument la grille et les couleurs utilisées pour représenter les couleurs des cases (sous forme de liste de chaîne de caractères, chaque chaîne représentant une couleur sous notation hexadécimale).

La fonction principale de la classe `Visualisation` est la fonction `view`, dont l'appel lance l'ouverture de la fenêtre graphique. Son paramètre principal est le `mode`, qui peut être égal à `''couleur''`, pour représenter chaque case par un couleur (cas mono-objectif), ou `''chiffre''`, pour mettre un chiffre sur chaque case et utiliser la couleur de la case pour représenter ce chiffre (cas multi-objectif). Les stratégies peuvent aussi être passées à cette fonction par son argument `strategy`, qui peut contenir soit un tableau en dimension 2 pour les stratégies pures ou 3 pour les stratégies mixtes. Lorsqu'une stratégie pure est passée en argument, elle est affichée sous forme de flèches grises dans chaque case et l'utilisateur peut la simuler à l'aide de la touche espace. Cette visualisation de la stratégie n'est pas faite dans le cas d'une stratégie mixte pour ne pas surcharger la grille avec les informations des lois de probabilité de chaque case, mais la simulation à l'aide de la touche espace est bien disponible dans ce cas aussi.

La plupart des fonctions de la classe `Visualisation` sont internes et servent à gérer les différentes étapes de la visualisation d'une grille. La gestion de mouvement est faite par la fonction `_move`, qui, outre la réalisation du mouvement, est aussi responsable pour calculer et mettre à jour l'affichage des coûts en temps réel. Pour les coûts on a fait le choix, cohérent avec la suite de ce projet, de considérer que, dans un déplacement, le coût payé est celui de la case d'où le robot *sort*. Ainsi, un déplacement d'une case de coût 1 à une case de coût 4 aura pour effet une augmentation du coût total de 1, et le 4 sera pris en compte uniquement au prochain pas. On a fait ce choix pour être cohérent avec la politique de récompenses des processus décisionnels Markoviens utilisés, dans lesquels on considère que la récompense à chaque pas de temps est une fonction uniquement de l'état d'origine et de l'action prise (et, dans notre cas, la récompense ne dépend pas de l'action).

Il y a une légère différence entre la visualisation et les méthodes de résolution implémentées dans les différentes sections de ce rapport. Les méthodes de résolution considèrent un modèle par un processus décisionnel Markovien qui continue à l'infini, ce qui veut dire que, si le robot est sur une case voisine à un mur et décide de prendre une action qui le conduira vers ce mur, son coût est augmenté du coût de la case où il est. La visualisation, cependant, ne montre pas cette augmentation, et considère au contraire qu'une action qui mène vers un mur ne se passe pas. Ce choix a été pris afin d'éviter des coûts qui augmentent à l'infini dans la visualisation.

La Figure 1 montre, sur des grilles 5×5 générées de façon aléatoire, de différentes visualisations possibles : le mode `''couleur''` dans (a), le mode `''chiffre''` dans (b), et le mode `''couleur''` avec visualisation d'une stratégie pure dans (c). On rappelle que, bien que non représentée sur la Figure 1, la visualisation de stratégies pures est aussi disponible dans le mode `''chiffre''`.



(a) (b) (c)

FIGURE 1 – Exemples de visualisation dans (a) le mode ‘‘`couleur`’’, (b) le mode ‘‘`chiffre`’’, et (c) le mode ‘‘`couleur`’’ avec une stratégie pure

1.1.3 La fonction `simulation`

La fonction `simulation` sert à simuler de façon automatisée l'exécution d'une stratégie sur une grille, sans visualisation. Les deux principaux arguments de cette fonction sont la grille et la stratégie à tester. La fonction prend aussi en compte le paramètre γ donnant l'amortissement du futur dans le coût du robot : son coût à l'instant t est multiplié par γ^t. Un bonus passé en argument est donné au robot lorsqu'il arrive dans sa case but, ce bonus est donné une seule fois lorsque le robot arrive sur cette case et la simulation s'arrête ensuite. Le paramètre `mode` permet de choisir comment le coût sera calculé : en mode ‘‘`couleur`’’, chaque case a un coût scalaire dépendant de sa couleur, alors que, en mode ‘‘`chiffre`’’, on considère un coût total vectoriel selon les différentes couleurs. Afin d'éviter une boucle infinie dans la fonction lorsque la stratégie passée en argument n'amène pas à la case but, un argument permet de fixer le nombre maximal d'itérations. Finalement, la fonction permet aussi de placer le robot dans n'importe quelle case de la grille au départ à l'aide de l'argument `init_robot`.

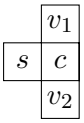
2 Recherche d'une trajectoire de moindre risque par itération de la valeur

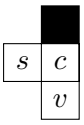
2.1 Modélisation

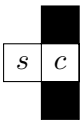
Pour modéliser ce problème comme un processus décisionnel Markovien il nous faut un ensemble d'états, un ensemble d'actions possibles, une fonction de transition et une fonction de récompense.

L'ensemble d'états S est choisi comme l'ensemble des cases (i, j) de la grille qui ne sont pas des obstacles. L'ensemble d'actions A contient quatre éléments représentant les déplacements vers le haut, à droite, vers le bas et à gauche et dénotés respectivement par $\uparrow, \rightarrow, \downarrow$ et $\leftarrow$. Pour la fonction de transition, on utilise les probabilités données à l'énoncé ; par exemple, pour l'action $\rightarrow$ à partir de l'état s, et en notant par c la cible de l'action (la case juste à droite de s) et v_1 et v_2 les voisins de c en haut et en bas, on a les possibilités suivantes en fonction de quelles cases parmi c, v_1 ou v_2

contiennent des obstacles :

$$T(s,\rightarrow,s') = \begin{cases} p & \text{si } s' = c \\ \dfrac{1-p}{2} & \text{si } s' = v_1 \text{ ou } s' = v_2 \\ 0 & \text{sinon} \end{cases}$$ dans la configuration 

$$T(s,\rightarrow,s') = \begin{cases} \dfrac{1+p}{2} & \text{si } s' = c \\ \dfrac{1-p}{2} & \text{si } s' = v \\ 0 & \text{sinon} \end{cases}$$ dans la configuration 

$$T(s,\rightarrow,s') = \begin{cases} 1 & \text{si } s' = c \\ 0 & \text{sinon} \end{cases}$$ dans la configuration 

$$T(s,\rightarrow,s') = \begin{cases} 1 & \text{si } s' = s \\ 0 & \text{sinon} \end{cases}$$ dans la configuration 

Les autres configurations et autres actions se déduisent de celles ci-dessus par symétrie.

Finalement, la fonction de récompense ne dépend pas de l'action et se calcule d'après la fonction coût c donné à l'énoncé de la façon suivante :

$$R(s,a) = \begin{cases} M & \text{si } s \text{ est la case but} \\ -c(\text{couleur}(s)) & \text{sinon} \end{cases}$$

où M est une valeur pour inciter l'arrivée à la case but et couleur(s) désigne la couleur de la case s.

L'équation de Bellman donnant la valeur optimale $V^*(s)$ pour s une case quelconque différente de la case but est donc :

$$V^*(s) = -c(\text{couleur}(s)) + \gamma \max_{a\in\{\uparrow,\rightarrow,\downarrow,\leftarrow\}} \left(\sum_{s'\in S} T(s,a,s')V^*(s') \right)$$

Quand on est dans la case but b, en bas et à droite, l'équation reste la même sauf pour la récompense qui est maintenant M :

$$V^*(b) = M + \gamma \max_{a\in\{\uparrow,\rightarrow,\downarrow,\leftarrow\}} \left(\sum_{s'\in S} T(b,a,s')V^*(s') \right)$$

Or, dans ce cas l'action optimale est de prendre une action impossible (à droite ou vers le bas) pour pouvoir rester dans cette case. On peut alors réécrire $V^*(b)$ comme $V^*(b) = M + \gamma V^*(b)$ d'où $V^*(b) = \frac{M}{1-\gamma}$. Ainsi, continuer le PDM à l'infini une fois qu'on arrive à la case but est équivalent à arrêter le PDM à la case but en donnant comme récompense $M' = \frac{M}{1-\gamma}$.

2.2 Implémentation de l'algorithme d'itération de la valeur

L'algorithme d'itération de la valeur a été implémenté dans la fonction `pol_valeur`, qui prend en argument une grille, la valeur de γ, la valeur de M, et la tolérance ε utilisée comme critère d'arrêt. Elle prend aussi un mode en argument : lorsqu'il est égal à `''couleur''`, l'algorithme calcule l'itération de la valeur pour le problème traité dans cette section, alors que le mode `''somme_chiffre''`

correspond au problème résolu dans la Section 4.1 (multi-critères avec optimisation de la somme des critères). La fonction retourne la politique optimale et le nombre d'itérations faites pour la trouver.

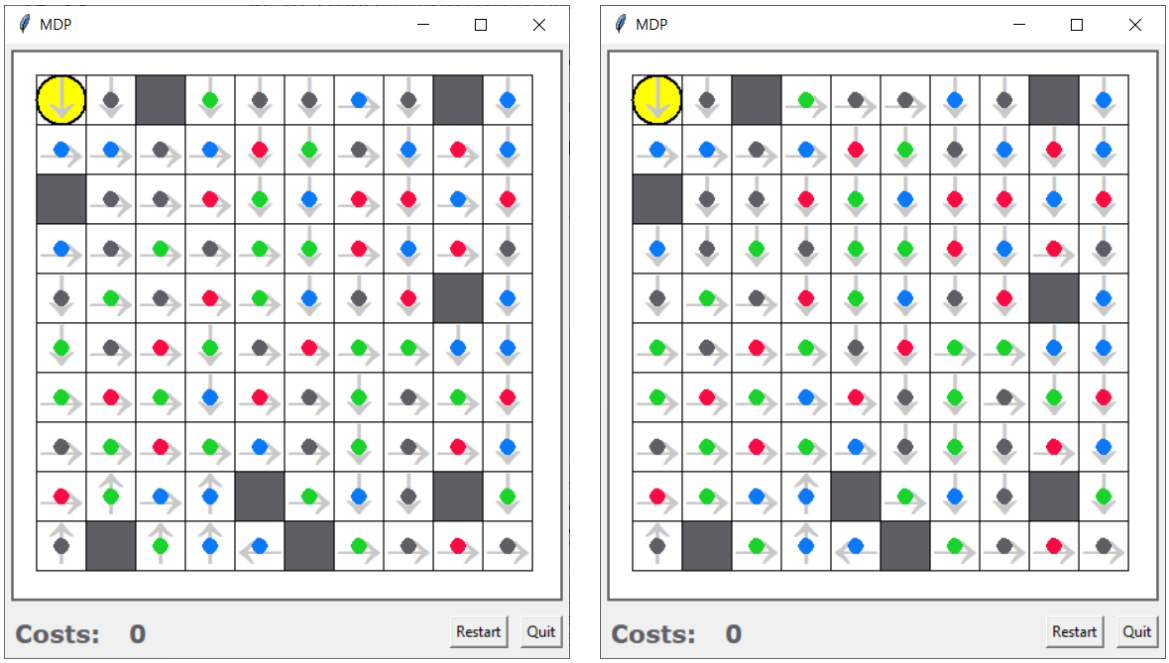
Dans l'implémentation, on fixe la valeur $V^*(b) = \frac{M}{1-\gamma}$ et on calcule $V^*(s)$ pour tout $s \neq b$ en suivant l'algorithme vu en cours. La fonction V^* est représentée par un tableau de dimension 2 de même taille que la grille. Ainsi, ce tableau contient aussi des cases correspondant aux murs, qui ne sont pas des états possibles. Ces cases ne sont pas prises en compte dans le calcul car la probabilité d'y aller est toujours 0. Leur valeur est fixée à 0 par défaut. La politique optimale retournée par la fonction est aussi un tableau de même taille que la grille, avec une valeur par défaut de 0 (aller vers le haut) dans les cases correspondant aux murs.

2.3 Résultats

Cette section présente tous les résultats concernant la résolution par itération de la valeur. On commence par les Sections 2.3.1, 2.3.2, 2.3.3 et 2.3.4, qui répondent essentiellement à la partie 2b de l'énoncé. La Section 2.3.5 présente les réponses à la partie 2c de l'énoncé, alors que la partie 2d est détaillée dans la Section 2.3.6. Tous les tests présentés ici sont implémentés dans le fichier Jupyter notebook annexe, `tests.ipynb`.

2.3.1 Exemple de calcul de stratégie sur une grille

Des cellules implémentées dans le fichier Jupyter notebook permettent de tester la résolution par itération de la valeur en variant la taille de la grille et les valeurs de p, γ et M. À titre d'exemple, on présente, dans la Figure 2, la résolution obtenue sur une instance de taille 10×10 avec $p = 1$ dans (a) et $p = 0.6$ dans (b), et $\gamma = 0{,}9$ et $M = 1000$.



(a) (b)

FIGURE 2 – Exemple de calcul de stratégie par itération de la valeur avec (a) $p = 1$ et (b) $p = 0.6$

On observe tout d'abord que les résolutions sont cohérentes avec ce qui était attendu : on a bien des flèches qui font en sorte que le robot se dirige vers la case en bas à droite. En plus, le

robot contourne bien les obstacles (comme on peut voir sur la stratégie optimale de la case $(9, 4)$; on considère ici que les coordonnées de la case en haut à gauche sont $(0, 0)$ et on donne d'abord le numéro de ligne et ensuite celui de colonne).

On observe quelques différences entre le cas $p = 1$ et $p = 0.6$ et, en guise d'exemple, on regarde ce qui se passe sur la case $(0, 3)$. Lorsque $p = 1$, la stratégie optimale trouvée sur cette case a été $\downarrow$, ce qui la mène à la case $(1, 3)$, qui est bleue, et évite la case noire $(0, 4)$ où le robot irait avec l'action $\rightarrow$. Cependant, avec $p = 0.6$, en prenant l'action $\downarrow$, le robot a une chance de se retrouver à la case $(1, 2)$, qui est noire et ne lui rapproche pas du du but. Il préférera alors prendre l'action $\rightarrow$, car elle lui permettra à coup sûr de se rapprocher du but.

2.3.2 Le cas d'une grille impossible

Comme les murs dans les grilles sont tirés de façon aléatoire, il est tout à fait possible que la grille soit "impossible", dans le sens où il n'y a pas de chemin entre la case de départ et la case but, ou même que l'une parmi les cases de départ ou but soient des murs. Dans ce cas, l'algorithme d'itération de la valeur marche toujours et il est intéressant à analyser ce comportement. Pour cela, on a pris la même grille qu'avant et on l'a transformée pour insérer un mur à la case but. La politique optimale calculée par itération de la valeur est montrée dans la Figure 3 (calculée avec $p = 1$ et $M = 1000$).

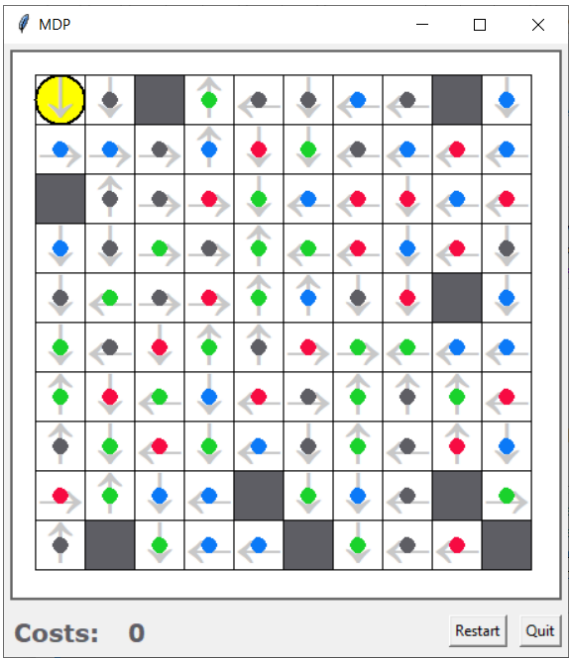


FIGURE 3 – Résolution par itération de la valeur d'une grille impossible

On remarque, dans la Figure 3, que le robot cherchera à minimiser sont coût à horizon infini en se dirigeant soit vers la case verte bloquée la plus proche (une case verte voisine d'un mur) et en prenant dans cette case l'action de se diriger vers le mur pour y rester, soit en se dirigeant vers une paire de cases vertes voisines entre lesquelles il oscillera. En absence d'une case but lui donnant une récompense, c'est cette stratégie de chercher les cases bloquantes moins coûteuses qui optimisera son critère à horizon infini.

Afin de tester automatiquement si une grille est possible, une méthode `est_possible` a été ajoutée à la classe `Grille`. Elle teste d'abord si les cases initiale et but sont possibles et, si c'est le

cas, alors elle fait un parcours en profondeur sur la grille à partir de la case initiale en cherchant la case but pour déterminer si elle est atteignable à partir de la case initiale. Comme les temps d'exécution et nombre d'itérations peuvent dépendre de façon importante du fait d'avoir des grilles possibles ou pas, dans la suite, tous les tests de temps ont été réalisés en n'utilisant que des grilles possibles.

2.3.3 Effet du bonus M

Le bonus M de la case but s'avère avoir un impact très important sur le comportement de la stratégie optimale trouvée. En effet, intuitivement, si M est trop petit, le robot n'aura pas assez d'incitation à aller vers la case but et donc, lorsqu'il démarre assez loin, il est possible qu'il reste bloqué, comme s'il ne voyait pas la case but, dans une situation similaire à celle décrite dans la Section 2.3.2. D'autre part, si M est trop grand, le robot aura une incitation trop forte à aller vers la case but en minimisant le nombre de pas pour y parvenir, même si pour cela il passe par des cases trop chères : il considérera qu'il vaut mieux passer par des cases chères mais arriver plus tôt à la case but que de prendre un chemin plus long. Il semble ainsi qu'il faut trouver toujours une valeur de M adaptée à l'objectif que l'on souhaite.

Afin de vérifier ces intuitions, on a construit une grille spéciale sans murs et avec une structure particulière et testé l'algorithme d'itération de la valeur pour divers valeurs de M, en fixant $\gamma = 0{,}8$ et $p = 0{,}6$. Les résultats sont donnés dans la Figure 4.

On observe dans la Figure 4 que, pour M petit, lorsque le robot est loin de la case but, il se dirige vers la case verte la plus proche et y reste, comme dans le cas de la Section 2.3.2 où la case but n'est pas atteignable : le robot n'est pas assez incité à l'atteindre. D'autre part, pour M très grand, le robot se dirige parfois vers des cases qui lui font avoir un coût plus important. Pour des valeurs modérées de M, par contre, comme $M = 2$, par exemple, on observe un comportement plus proche de celui attendu : le robot se dirige toujours vers la case but mais il prend parfois des chemins plus longs pour éviter de passer par des cases avec un coût élevé.

Les valeurs de M qui permettent d'avoir ces différents régimes dépendent de divers paramètres, comme la taille de la grille, les coûts des couleurs et les valeurs de p et γ. On a observé expérimentalement que la solution semble moins sensible à des variations en M lorsque p est proche de 1.

2.3.4 Temps d'exécution et nombre d'itérations de l'algorithme d'itération de la valeur

On a implémenté des fonctions `tester_temps` et `tester_iterations` permettant de tester le temps de calcul moyen (à l'aide de la fonction `process_time` du module `time` de Python) et le nombre d'itérations moyen de fonctions de résolution de ce processus décisionnel Markovien. Ces deux méthodes prennent en argument la fonction à tester, une liste de grilles sur lesquelles on teste, et les autres arguments à passer à la fonction testée. Pour le test de temps d'exécution, la fonction est appelée un certain nombre de fois sur chaque grille (contrôlé par l'argument `repeat`). Pour le test du nombre d'itérations, on suppose que la fonction retourne, en deuxième sortie, le nombre d'itérations qu'il lui a fallu pour converger.

Ces tests ont été appliqués à l'algorithme d'itération de la valeur sur des grilles de taille 10×10, 10×15 et 15×20, avec différentes valeurs de γ et p. Pour chaque taille de grille, 50 grilles ont été générées et, pour le test de temps, l'algorithme est exécuté 10 fois sur chacune de ces grilles. On a garanti, à l'aide de la méthode `est_possible`, qu'il y a un chemin entre la case initiale et la case but dans chaque grille testée. La quantité de 50 grilles a été choisie à la place des 15 grilles suggérées dans l'énoncé car cela permet de réduire la variance tout en gardant un temps de calcul total raisonnable. Les résultats sont donnés dans les Tables 1 et 2.

(a) (b) (c)

(d) (e) (f)

FIGURE 4 – Effet du bonus M d'atteindre la case but sur la politique optimale avec (a) $M = 0$, (b) $M = 1$, (c) $M = 2$, (d) $M = 10$, (e) $M = 50$ et (f) $M = 100$

On observe dans les Tables 1 et 2 que, comme attendu, le temps de calcul et le nombre d'itérations augmentent avec la taille du tableau et avec la valeur de γ. Pour $\gamma = 0{,}5$, le nombre d'itérations semble varier très peu avec la taille du tableau : en effet, comme dans ce cas γ^n converge très rapidement vers 0 lorsque n augmente, le robot regarde très peu dans le futur et il ne prend pas trop en compte le futur au-delà de quelques cases proches, ce qui facilite la convergence de la fonction valeur. Comme 10×10 est déjà assez grand pour $\gamma = 0{,}5$, l'augmenter davantage ne change pas beaucoup le nombre d'itérations. Cependant, dans chaque itération, le tableau contenant les valeurs de V^* est plus grand, et donc le temps de calcul de chaque itération augmente, augmentant ainsi la valeur totale. Lorsque γ augmente, le robot voit de plus en plus loin et la taille du tableau a un impact plus important sur le nombre moyen d'itérations. Concernant les variations de p, on remarque que le nombre d'itérations diminue en moyenne lorsque p augmente, ce phénomène étant plus prononcé lorsque γ est plus proche de 1.

	$\gamma = 0{,}5$		$\gamma = 0{,}7$		$\gamma = 0{,}9$	
	$p = 0{,}6$	$p = 1$	$p = 0{,}6$	$p = 1$	$p = 0{,}6$	$p = 1$
10×10	37,0	42,7	56,5	41,4	78,0	54,9
10×15	63,5	63,6	111,0	106,5	126,8	98,0
15×20	157,1	133,2	249,8	238,8	342,9	283,7

TABLE 1 – Temps de calcul moyen (en millisecondes) de l'algorithme d'itération de la valeur

	$\gamma = 0{,}5$		$\gamma = 0{,}7$		$\gamma = 0{,}9$	
	$p = 0{,}6$	$p = 1$	$p = 0{,}6$	$p = 1$	$p = 0{,}6$	$p = 1$
10×10	18,8	18,7	25,1	20,1	33,2	25,2
10×15	19,2	19,0	32,5	32,3	37,8	29,9
15×20	19,5	19,2	35,3	35,1	50,0	42,1

TABLE 2 – Nombre moyen d'itérations de l'algorithme d'itération de la valeur

2.3.5 Étude de l'impact du coût

Dans cette partie, on fixe $p = 0{,}8$, $\gamma = 0{,}9$, et on étudie ce qui se passe lorsque la fonction de coût $c(x)$ est changée à $c^q(x)$ pour quelques valeurs de q. On a d'abord observé que, en laissant un bonus constant à la case but, on arrive à des situations de blocage lorsque q augmente, dans lesquelles, comme décrit à la Section 2.3.3, le robot décide de rester bloqué plutôt que d'aller vers la case but. Pour éviter ce type de phénomène, on a décidé d'augmenter la récompense de la case but avec q et on a pris alors un bonus de M^q, avec M fixé à 4.

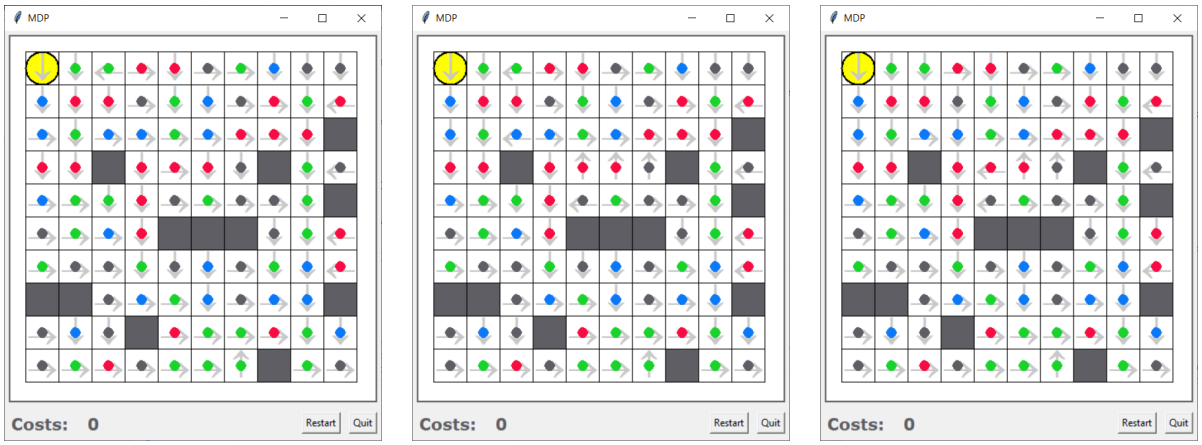
On remarque d'abord que l'augmentation de q donne une augmentation beaucoup plus importante des cases qui ont déjà un coût élevé : les cases noires augmentent plus que les rouges, qui augmentent plus que les bleues, qui augmentent plus que les verts, qui, elles, restent constantes. Ainsi, on s'attend à, en général, observer des comportements qui auront tendance à éviter de plus en plus les cases noires.

En variant q de 1 à 10 en pas de 1, on observe que, à chaque fois, on a des petits changements, en général pas plus que deux ou trois cases entres deux valeurs de q consécutives. Vu la complexité du critère à être optimisé, qui prend en compte d'une façon assez délicate le bonus M, le taux γ et la probabilité p, il est difficile à interpréter tous les changements de toutes les étapes et quelques changements peuvent paraître a priori peu intuitifs, mais il ressort de l'analyse de tous ces changements qui, globalement, on a de plus en plus tendance à éviter les cases noires lorsque c'est possible, ensuite à éviter les rouges, et ainsi de suite. Afin d'illustrer, la Figure 5 représente les cas $q = 1$, $q = 5$ et $q = 9$.

2.3.6 Modélisation et résolution avec un autre critère de coût

On considère maintenant le problème de la partie 2d de l'énoncé, dans lequel, comme indiqué à l'énoncé, "une trajectoire est meilleure qu'une autre si elle traverse moins de cases noires ou, en cas d'égalité, si elle traverse moins de cases rouges ou, en cas d'égalité sur les noires et les rouges, si elle traverse moins de case bleues, ou bien encore, en cas d'égalité sur les bleues, si elle traverse moins de cases vertes". On modélise ce problème comme le même PDM que l'on étudie jusqu'à présent, mais avec un choix de coût selon les couleurs afin de satisfaire ces contraintes.

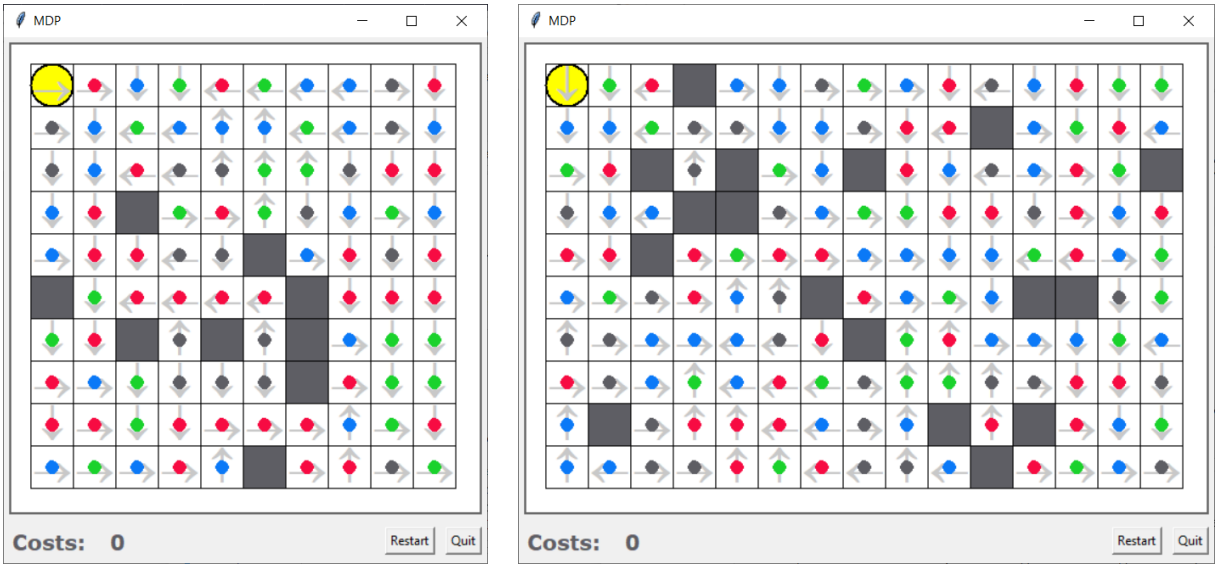
Plus précisément, prenons C une constante suffisamment grande de telle sorte qu'il y ait dans la



(a) (b) (c)

FIGURE 5 – Effet de la puissance q du coût des cases sur la politique optimale avec (a) $q = 1$, (b) $q = 5$ et (c) $q = 10$

grille toujours strictement moins que C cases d'une même couleur (C peut être, par exemple, égal à la quantité de cases de la grille, puisque l'on ne prend pas en compte dans notre modèle la couleur de la case but). On attribue au vert le coût de 1, au bleu le coût de C, au rouge le coût de C^2, et au noir le coût de C^3. Alors toute trajectoire doit éviter de passer par une case noire si possible : le coût qu'elle a en passant par une case noire, C^3, est plus grand que le coût qu'elle obtiendrait en passant par toutes les cases rouges de la grille, qui serait au maximum $(C-1)C^2$, avec un raisonnement analogue pour les autres couleurs.



(a) (b)

FIGURE 6 – Grilles de taille (a) 10×10 et (b) 10×15 résolues avec le tableau de coûts $[1, C, C^2, C^3]$

La Figure 6 illustre cette technique sur deux grilles, l'une de taille 10×10 et l'autre de taille 10×15. On a trouvé que, dans les deux cas, le choix de $M = C^2$ donne un bon compromis entre l'incitation à aller vers la case but et la bonne prise en compte des coûts de chaque case.

Dans la Figure 6(a), on observe que, pour aller de la case initiale à la case but en évitant les cases noires, il faut absolument passer par la case de coordonnées $(8, 5)$ (on rappelle la convention que la case initiale a pour coordonnées $(0, 0)$ et que l'on note d'abord l'indice de ligne et ensuite celui de colonne), et toutes les flèches dirigent vers un chemin empruntant cette case. En plus, on observe que, pour éviter d'avoir la chance de tomber sur une case noire, la stratégie optimale préfère faire des détours plutôt que de prendre le risque de tomber sur une case noire. Ainsi, par exemple, en $(0, 1)$, plutôt que juste descendre (qui donnerait une probabilité de $\frac{1-p}{2} = 0{,}1$ d'arriver à une case noire), la stratégie optimale va à droite vers $(0, 2)$, ensuite vers le bas vers $(1, 2)$ sans le risque de tomber sur une case noire, et uniquement après vers la gauche et vers le bas. Le même type de détour apparaît dans la case $(8, 2)$: plutôt qu'aller directement vers la droite pour franchir la case $(8, 5)$, ce qui comporte le risque de tomber sur l'une des cases noires $(7, 3)$ à $(7, 5)$, la stratégie va plutôt vers le bas en $(8, 2)$ pour s'éloigner des cases noires avant d'aller à droite, et ne remonte que lorsque c'est nécessaire. La stratégie de monter en $(8, 7)$ sert, quant à elle, à éviter la case noire $(9, 8)$.

Une analyse similaire peut être faire dans la Figure 6(b). On observe qu'il est impossible d'aller de la case initiale à la case but sans passer par au moins une case noire, mais la stratégie optimale essaie de minimiser le nombre de cases noires prises, en essayant de passer par la case noire $(5, 2)$ au début en évitant les autres cases noires voisines. Ensuite, l'objectif est de passer par la case $(4, 5)$ pour éviter les cases noires $(7, 7)$, $(8, 7)$ et $(9, 7)$. Tout le long du chemin, la stratégie optimale évite ainsi, dès que possible, les cases noires.

3 Recherche d'une trajectoire de moindre risque par programmation linéaire

3.1 Programme linéaire en stratégies mixtes

Comme vu en cours, la fonction V^* d'un PDM peut être trouvée comme solution du programme linéaire suivant :

$$(\mathcal{P}) \begin{cases} \min \displaystyle\sum_{s \in S} \mu_s v_s \\ \text{s.c. } v_s - \gamma \displaystyle\sum_{s' \in S} T(s, a, s') v_{s'} \geq R(s, a), \qquad \forall s \in S, \forall a \in A \end{cases}$$

où les variables de décision v_s représentent les valeurs de $V^*(s)$ et μ_s, $s \in S$, sont des constantes strictement positives quelconques satisfaisant $\sum_{s \in S} \mu_s = 1$.

Afin de retrouver les stratégies optimales aussi, et pas seulement les valeurs de $V^*(s)$, on passe au problème dual $(\mathcal{D})$ de $(\mathcal{P})$. Ce problème a autant de variables de décision que le problème $(\mathcal{P})$ a de contraintes, c'est-à-dire une pour chaque paire $(s, a) \in S \times A$, que l'on dénote par x_{sa}. L'écriture de ce problème dual donne alors, comme vu en cours :

$$(\mathcal{D}) \begin{cases} \max \displaystyle\sum_{s \in S} \sum_{a \in A} R(s, a) x_{sa} \\ \text{s.c. } \displaystyle\sum_{a \in A} x_{sa} - \gamma \sum_{s' \in S} \sum_{a \in A} T(s', a, s) x_{s'a} = \mu_s \qquad \forall s \in S \\ \qquad x_{sa} \geq 0 \qquad \forall s \in S, \forall a \in A \end{cases}$$

La stratégie mixte se déduit des valeurs x_{sa} solutions de $(\mathcal{D})$: pour $s \in S$ fixé, la probabilité de prendre l'action $a \in A$ dans la stratégie mixte optimale est proportionnelle à x_{sa}, et on peut donc

calculer la stratégie mixte optimale δ par normalisation de x :

$$\delta_{sa} = \frac{x_{sa}}{\sum_{a' \in A} x_{sa'}}.$$

Dans le cas présent (avec un seul objectif à optimiser), un résultat classique vu en cours garantit qu'il y a toujours une stratégie pure optimale, et on s'attend ainsi à ce que, pour chaque s, δ_{sa} soit égal à 1 pour une valeur de a et 0 pour les autres actions.

3.2 Programme linéaire en stratégies pures

Le programme linéaire $(\mathcal{D})$ donne a priori des stratégies mixtes. La difficulté pour imposer une stratégie pure est que les variables de décision x_{sa} ne donnent pas directement la stratégie mixte optimale mais uniquement des valeurs proportionnelles, et ainsi il ne suffit pas d'imposer que x_{sa} soient des variables binaires. La technique vue en cours pour résoudre ce problème est d'introduire des variables de décision binaires auxiliaires d_{sa} dont l'interprétation voulue est que $d_{sa} = 1$ si et seulement si a est l'action optimale à prendre dans l'état s. On veut ainsi que, si $x_{sa} > 0$, alors $d_{sa} = 1$ et, de façon équivalente, si $d_{sa} = 0$, alors $x_{sa} = 0$. En utilisant le fait que $x_{sa} \leq \frac{1}{1-\gamma}$ (ce qui est une conséquence du choix $\sum_{s \in S} \mu_s = 1$), ce comportement peut être obtenu avec la contrainte $(1-\gamma)x_{sa} \leq d_{sa}$. On obtient alors la reformulation suivante de $(\mathcal{D})$:

$$(\mathcal{D}_p) \begin{cases} \max \sum_{s \in S} \sum_{a \in A} R(s,a) x_{sa} & \\ \text{s.c.} \ \sum_{a \in A} x_{sa} - \gamma \sum_{s' \in S} \sum_{a \in A} T(s',a,s) x_{s'a} = \mu_s & \forall s \in S \\ \quad \sum_{a \in A} d_{sa} \leq 1 & \forall s \in S \\ \quad (1-\gamma) x_{sa} \leq d_{sa} & \forall s \in S, \forall a \in A \\ \quad x_{sa} \geq 0, d_{sa} \in \{0,1\} & \forall s \in S, \forall a \in A \end{cases}$$

Le problème $(\mathcal{D}_p)$ contient aussi une contrainte imposant le fait qu'il ne peut pas y avoir plus d'une action optimale par état. Il s'agit d'un problème linéaire à variables mixtes : les x_{sa} sont des variables de décision réelles, alors que les d_{sa} sont binaires. La stratégie pure optimale π se déduit des variables d_{sa} de façon immédiate : $\pi(s) = a$ si et seulement si $d_{sa} = 1$.

On remarque que cette modélisation n'est pas la seule pour pourvoir obtenir les stratégies optimales pures : on pourrait aussi, alternativement, considérer le problème primal $(\mathcal{P})$, le résoudre, et ensuite calculer la politique optimale à partir des valeurs de $V^*(s)$ trouvées en cherchant quelle action maximise chaque $V^*(s)$. Cette méthode marche bien dans le cas mono-objectif, mais la résolution par dualité a l'avantage de se généraliser au cas multi-objectif.

3.3 Implémentation

La résolution des problèmes linéaires $(\mathcal{D})$ et $(\mathcal{D}_p)$ a été faite en utilisant Gurobi en Python et implémentée dans les fonctions `pol_pl_mixte` et `pol_pl_pure`, respectivement. Ces fonctions prennent en argument la grille, la valeur de γ et la valeur de M. Comme la fonction `pol_valeur`, elles prennent aussi un mode en argument : le mode **''`couleur`''** correspond au problème traité dans cette section et **''`somme_chiffre`''** au problème résolu dans la Section 4.1. La fonction `pol_pl_pure` retourne une stratégie pure sous la forme d'un tableau de dimension 2, avec les mêmes conventions que la fonction

`pol_valeur` (valeur de 0 dans les cases correspondant à des murs), alors que `pol_pl_mixte` retourne une stratégie mixte sous la forme d'un tableau de dimension 3, avec la convention que, dans les cases correspondant à des murs, la probabilité retournée est la loi uniforme sur les quatre actions. Ces fonctions retournent aussi la valeur de la fonction objectif à l'optimum.

Pour l'implémentation, les paramètres μ_s ont été choisis tous égaux, c'est-à-dire $\mu_s = \frac{1}{|S|}$ (et on a aussi utilisé le fait que $\frac{1}{|S|} = \frac{4}{|S \times A|}$ puisqu'on a toujours exactement 4 actions par état, et $|S \times A|$ est le nombre de variables de décision de $(\mathcal{D})$, facilement retrouvable dans le code).

Pour calculer facilement les valeurs de $T(s', a, s)$ pour un état s donné, une fonction auxiliaire `proba_trans_arr` a été implémentée dans la classe `Grille`, qui parcourt les cases voisines s' de s et retourne un dictionnaire indexé par la paire (s', a) et contenant la probabilité que l'action a depuis s' conduise à s. Avec ce dictionnaire et les méthodes de la classe `tupledict` de Gurobi, l'implémentation des contraintes de $(\mathcal{D})$ et $(\mathcal{D}_p)$ devient assez immédiate.

3.4 Résultats

Comme dans la Section 2.3, tous les résultats présentés dans cette section ont été implémentés dans le fichier Jupyter notebook annexe, `tests.ipynb`.

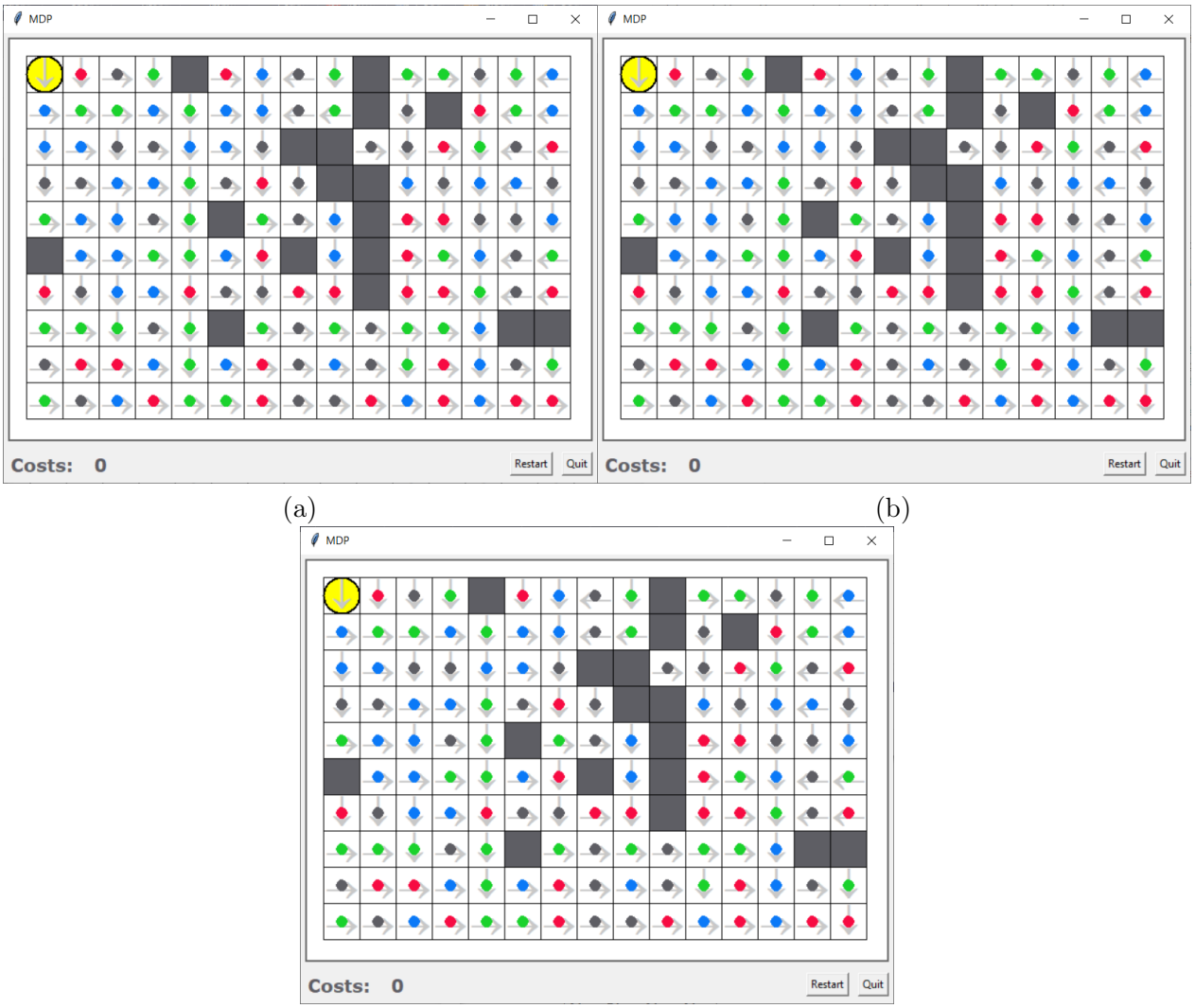
3.4.1 Exemple de calcul de stratégie sur une grille

Afin de tester les implémentations des fonctions `pol_pl_mixte` et `pol_pl_pure`, des cellules dans le fichier Jupyter notebook permettent de créer des grilles aléatoires et calculer les stratégies pure et mixte optimales par programmation linéaire, ainsi que la stratégie pure optimale par itération de la valeur, afin de la comparer aux autres.

Tout d'abord, on remarque que, comme attendu, la stratégie mixte optimale calculée est en effet pure : toutes les lois de probabilité sur les actions sont des lois certaines, avec 100% de probabilité pour l'une des actions et 0 pour les autres. Les exceptions sont les cases correspondantes aux murs qui ne sont pas calculées par le programme linéaire et ont été fixées à des lois uniformes par défaut.

La Figure 7 montre le résultat de l'application des algorithmes par itération de la valeur et par programmation linéaire sur une grille de taille 10×15, avec $p = 1$, $\gamma = 0{,}9$ et $M = 100$. Pour le cas de la politique mixte, les flèches indiquent l'unique direction avec 100% de probabilité. On observe que les trois algorithmes donnent essentiellement le même résultat, avec des différences uniquement dans des cases dans lesquelles l'action optimale n'est pas unique. Par exemple, dans la case $(0, 2)$, la stratégie optimale donnée par programmation linéaire en stratégies mixtes est $\downarrow$, alors que, par programmation linéaire en stratégies pures et par itération de la valeur, la stratégie optimale est $\rightarrow$. Ces deux actions conduisent à exactement le même résultat : au bout de deux itérations, on sera sur la case $(1, 3)$, ayant traversé soit la case $(0, 3)$, soit la case $(1, 2)$, qui sont toutes les deux vertes (on rappelle que ces tests ont été réalisés avec $p = 1$). Cette situation se répète dans toutes les autres cases de la grille dans lesquelles les actions données par les algorithmes sont différentes. On a aussi une différence dans la case but, dans laquelle l'itération de la valeur donne $\downarrow$ comme action optimale mais les deux autres donnent $\rightarrow$, ce qui conduit exactement au même résultat puisque ces deux actions laissent le robot sur place.

Lorsque $p < 1$, ce phénomène d'avoir multiples actions possibles dans une case avec le même effet et le même coût final est beaucoup plus rare. Dans tous les tests réalisés avec $p < 1$, les politiques optimales obtenues étaient identiques, à l'exception de la case but, où l'on obtient toujours l'une d'entre $\downarrow$ ou $\rightarrow$ comme actions optimales.



(a) (b)

(c)

FIGURE 7 – Politiques optimales calculées (a) par itération de la valeur, (b) par programmation linéaire en stratégies pures, et (c) par programmation linéaire en stratégies mixtes

3.4.2 Temps d'exécution et valeurs de la fonction objectif

On a utilisé la fonction `tester_temps` décrite dans la Section 2.3.4 afin de tester le temps d'exécution des algorithmes par programmation linéaire. En plus, comme les fonctions `pol_pl_mixte` et `pol_pl_pure` retournent aussi en deuxième sortie la valeur à l'optimum de la fonction objectif des programmes linéaires, il est possible de réutiliser la fonction `tester_iterations`, implémentée originellement dans la Section 2.3.4 pour calculer le nombre moyen d'itérations par itération de la valeur, afin d'obtenir la valeur moyenne de la fonction objectif à l'optimum pour les deux algorithmes. La procédure expérimentale a été la même qu'à la Section 2.3.4, avec la différence que l'on a considéré uniquement des grilles de taille fixe 10×15. Les résultats de temps d'exécution et de valeurs de la fonction objectif sont données dans les Tables 3 et 4, respectivement.

Le temps d'exécution donné dans la Table 3 est plus important pour le calcul des stratégies pures que pour les stratégies mixtes. En effet, le calcul des stratégies mixtes se fait par résolution d'un programme linéaire à variables de décision continues, un problème polynomial, alors que le calcul

	$\gamma = 0{,}5$		$\gamma = 0{,}7$		$\gamma = 0{,}9$	
	$p = 0{,}6$	$p = 1$	$p = 0{,}6$	$p = 1$	$p = 0{,}6$	$p = 1$
Pure	151,6	104,9	141,9	105,3	136,0	103,7
Mixte	42,9	38,6	42,2	39,6	45,3	42,2

TABLE 3 – Temps de calcul moyen (en millisecondes) des algorithmes par programmation linéaire avec politiques pures et mixtes

	$\gamma = 0{,}5$		$\gamma = 0{,}7$		$\gamma = 0{,}9$	
	$p = 0{,}6$	$p = 1$	$p = 0{,}6$	$p = 1$	$p = 0{,}6$	$p = 1$
Pure	46,0	47,9	212,8	218,3	3291,0	3320,4
Mixte	46,0	47,9	212,8	218,3	3291,0	3320,4

TABLE 4 – Valeur de la fonction objectif à l'optimum des algorithmes par programmation linéaire avec politiques pures et mixtes

des stratégies pures fait intervenir un programme linéaire à variables mixtes, continues et discrètes, qui est un problème NP-difficile avec donc des temps de résolution beaucoup plus élevés. Comme le nombre de variables de décision et de contraintes est indépendant de p et de γ, ces paramètres ont un impact relativement peu important dans le temps de résolution.

Concernant p, le phénomène principal est que, lorsque $p = 1$, plusieurs coefficients des contraintes de $(\mathcal{D})$ et $(\mathcal{D}_p)$ deviennent 0, car, pour une case s donnée, le coefficient $T(s', a, s)$ est non-nul pour au plus 4 termes, et la matrice de contraintes est donc plus creuse que lorsque $p = 0{,}6$. L'impact de γ est beaucoup plus limité et subtil : une augmentation de γ semble faire réduire le temps de calcul du programme linéaire pour les stratégies pures lorsque $p = 0{,}6$, mais son impact est beaucoup plus limité dans les autres cas. En effet, la valeur de γ a un impact sur les positions des hyperplans définis par les contraintes de $(\mathcal{D})$ et $(\mathcal{D}_p)$, et donc dans la quantité de sommets réalisables du polyèdre des contraintes, mais d'une façon assez indirecte.

La comparaison avec la Table 1 des temps de calcul pour l'algorithme d'itération de la valeur montre que leurs caractéristiques par rapport aux dépendances en p et γ sont assez différentes. En effet, la convergence de l'itération de la valeur provient d'un principe de contraction avec un coefficient de contraction γ, et donc le temps de calcul dépend directement de γ et tend à l'infini lorsque γ tend vers 1. Cela n'est pas le cas pour l'approche par programmation linéaire. En plus, on remarque que les temps de calcul des stratégies mixtes par programmation linéaire sont moins élevés que ceux des stratégies pures par itération de la valeur, ce qui montre aussi l'intérêt de l'approche par programmation linéaire.

Comme les politiques optimales pures et mixtes coïncident, les valeurs à l'optimum des fonctions objectif des programmes linéaires $(\mathcal{D})$ et $(\mathcal{D}_p)$ coïncident, ce que l'on observe bien dans la Table 4. Comme $(\mathcal{D})$ est le dual de $(\mathcal{P})$, par dualité, ces valeurs à l'optimum sont aussi les valeurs optimales de $(\mathcal{P})$, et peuvent donc s'interpréter comme la moyenne de la fonction valeur $V^*(s)$ sur tous les états s (puisque l'on a choisi $\mu_s = \frac{1}{|S|}$; d'autres choix de μ_s auraient conduit à des moyennes pondérées). Cela donne ainsi l'espérance de la récompense du robot si on choisit sa position initiale de façon aléatoire dans la grille selon la loi μ_s.

Cette valeur est plus grande pour $p = 1$ que pour $p = 0{,}6$, ce qui était attendu : lorsque $p = 1$, il n'y a pas de perturbations aléatoires dans la trajectoire du robot et il peut donc éviter plus facilement les cases qui ont un coût grand et augmenter ainsi sa récompense. La valeur est aussi plus

grande lorsque γ augmente : comme le paramètre γ représente combien le futur est pris en compte dans la maximisation de l'agent, plus ce paramètre est grand, plus le futur sera pris en compte, et donc plus l'arrivée sur la case but avec la récompense M apparaîtra dans la fonction objectif.

4 Recherche d'une trajectoire équilibrée

Dans le cas présent, on a un processus décisionnel Markovien multi-objectifs : on a le même espace d'états S qu'avant, les mêmes actions A, les mêmes probabilités de transition T, mais la récompense R est désormais vectorielle, avec une entrée pour chaque type de ressource, et donc pour chaque couleur. Ainsi, avec 4 couleurs comme décrit à l'énoncé, $R(s,a) = (R_1(s,a), R_2(s,a), R_3(s,a), R_4(s,a))$ est un vecteur de taille 4 donnant les récompenses correspondant à chaque couleur dans l'état s et avec l'action a. D'après la construction donnée, chaque case contient un seul coût, correspondant à sa couleur, et ainsi $R(s,a)$ est un vecteur indépendant de a et dont trois composantes sont égales à 0, la composante correspondant à la couleur de la case s étant négative et égale à l'opposé du coût de s. Comme avant, on considère que la case but b a une récompense spécifique, que l'on prend ici comme $R(b,a) = (M,M,M,M)$ pour une valeur M assez grande.

4.1 Minimisation de la consommation totale de ressources

Étant données des suites $(s_t)_{t\geq 0}$ et $(a_t)_{t\geq 0}$ d'états et d'actions, issues de l'application d'une politique π pure ou mixte donnée à partir d'un état initial, la récompense obtenue en prenant ces suites d'états et d'actions est $\sum_{t=0}^{\infty} \gamma^t R(s_t,a_t)$, qui est maintenant un vecteur puisque chaque $R(s_t,a_t)$ est un vecteur. Sa i-ème composante, correspondant au i-ème critère, est $\sum_{t=0}^{\infty} \gamma^t R_i(s_t,a_t)$. Lorsque l'objectif est la minimisation de l'espérance de la consommation totale de ressources, on cherche alors à maximiser l'espérance de la récompense totale,

$$\mathbb{E}_\pi \left[\sum_{i=1}^{4} \sum_{t=0}^{\infty} \gamma^t R_i(s_t,a_t) \right], \tag{1}$$

où l'indice π dans l'espérance dénote le fait que les probabilités des séquences d'états et d'actions dépendent de la politique (pure ou mixte) π choisie.

Si on définit la récompense scalaire $\widehat{R}(s,a) = \sum_{i=1}^{4} R_i(s,a)$, on observe alors que la récompense totale à maximiser, en échangeant l'ordre des sommes, est

$$\mathbb{E}_\pi \left[\sum_{i=1}^{4} \sum_{t=0}^{\infty} \gamma^t R_i(s_t,a_t) \right] = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t \sum_{i=1}^{4} R_i(s_t,a_t) \right] = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t \widehat{R}(s_t,a_t) \right].$$

Le dernier terme des égalités ci-dessus est exactement le terme qu'on cherche à maximiser dans un PDM à un seul objectif lorsque la récompense est donnée par $\widehat{R}$. Ainsi, on peut utiliser les techniques pour les PDMs à un seul objectif vues en cours, comme la fonction valeur V^* et les algorithmes utilisés dans les Sections 2 et 3, pour résoudre ce PDM.

Plus précisément, pour les politiques pures, on peut écrire les équations de Bellman comme dans le cas de la Section 2 :

$$V^*(s) = \max_{a\in A} \left[\widehat{R}(s,a) + \gamma \sum_{s'\in S} T(s,a,s') V^*(s') \right].$$

On peut alors déterminer une politique pure optimale par itération de la valeur. Il est aussi possible de déterminer des politiques pures optimales en résolvant un programme linéaire similaire à $(\mathcal{D}_p)$

de la Section 3.2, en remplaçant simplement $R(s,a)$ dans la fonction objectif par $\widehat{R}(s,a)$. De façon analogue, pour obtenir une politique mixte, on peut résoudre le même programme linéaire que dans la Section 3.1, mais avec les récompenses R remplacées par $\widehat{R}$:

$$(\widehat{\mathcal{D}}) \begin{cases} \max \sum_{s\in S}\sum_{a\in A} \widehat{R}(s,a)x_{sa} & \\ \text{s.c. } \sum_{a\in A} x_{sa} - \gamma \sum_{s'\in S}\sum_{a\in A} T(s',a,s)x_{s'a} = \mu_s & \forall s \in S \\ x_{sa} \geq 0 & \forall s \in S, \forall a \in A \end{cases}$$

4.2 Minimisation d'un critère équilibré

On s'intéresse maintenant à la minimisation d'un critère plus équilibré consistant à choisir un chemin qui minimise l'espérance du maximum des coûts selon les quatre critères différents pris en compte, ou, de façon équivalente, de choisir un chemin qui maximise l'espérance du minimum des récompenses selon les quatre critères, c'est-à-dire

$$\mathbb{E}_\pi \left[\min_{i\in\{1,2,3,4\}} \sum_{t=0}^{\infty} \gamma^t R_i(s_t,a_t) \right].$$

Il s'agit de la même expression que (1) mais avec la somme sur i remplacée par la minimisation sur i. Cependant, comme le minimum et la somme sur t ne peuvent pas être échangées, on ne peut pas procéder comme dans la Section 4.1 afin de l'écrire sous la forme $\mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t \widetilde{R}(s_t,a_t)\right]$ pour une récompense modifiée $\widetilde{R}$, et donc on ne peut pas appliquer directement les techniques usuelles des PDMs à un seul objectif. Le principe de la programmation dynamique de Bellman n'est pas satisfait dans ce cas, et donc on ne peut pas se servir d'une fonction valeur V^* ni des algorithmes qui l'utilisent, comme l'algorithme d'itération de la valeur.

Cependant, comme vu en cours, la formulation duale $(\mathcal{D})$ peut être utilisée dans le cas d'un PDM multi-critères. Pour le problème multi-critères en question, sa formulation comme un programme linéaire s'écrit :

$$(\mathcal{D}_{mc}) \begin{cases} \max \sum_{s\in S}\sum_{a\in A} R_i(s,a)x_{sa}, & \forall i \in \{1,2,3,4\} \\ \text{s.c. } \sum_{a\in A} x_{sa} - \gamma \sum_{s'\in S}\sum_{a\in A} T(s',a,s)x_{s'a} = \mu_s & \forall s \in S \\ x_{sa} \geq 0 & \forall s \in S, \forall a \in A \end{cases}$$

Plutôt que de résoudre ce problème multi-critères directement et trouver toutes les solutions Pareto-optimales (qui peuvent être en nombre exponentiel), on cherche dans cette question à le résoudre avec une fonction scalarisante particulière, le minimum sur tous les objectifs, ce qui correspond à considérer la fonction objectif

$$\max \min_{i\in\{1,2,3,4\}} \sum_{s\in S}\sum_{a\in A} R_i(s,a)x_{sa}.$$

Cette fonction n'est pas linéaire mais on peut utiliser la stratégie classique de transformer ce problème

en un programme linéaire en utilisant une variable de décision auxiliaire et 4 contraintes en plus :

$$(\mathcal{D}_{\min}) \begin{cases} \max z & \\ \text{s.c. } z \leq \sum_{s\in S}\sum_{a\in A} R_i(s,a)x_{sa}, & \forall i \in \{1,2,3,4\}, \\ \quad \sum_{a\in A} x_{sa} - \gamma \sum_{s'\in S}\sum_{a\in A} T(s',a,s)x_{s'a} = \mu_s & \forall s \in S \\ \quad x_{sa} \geq 0 & \forall s \in S, \forall a \in A \end{cases}$$

4.3 Implémentation

Comme vu dans la Section 4.1, la minimisation de la somme des coûts correspond à résoudre un PDM avec le coût modifié $\widehat{R}$. Ainsi, les mêmes fonctions `pol_valeur`, `pol_pl_mixte` et `pol_pl_pure` déjà décrites précédemment permettent de résoudre ce PDM : il suffit de leur donner en argument `mode` la chaîne de caractères **''somme_chiffre''**, qui choisira la bonne fonction de récompense $\widehat{R}$ à utiliser. Afin de simplifier les comparaisons à faire dans la suite, dans le mode **''somme_chiffre''**, la fonction `pol_pl_mixte` retourne en deuxième sortie non pas la valeur de la fonction objectif à l'optimum mais le vecteur de récompenses selon chacun des critères.

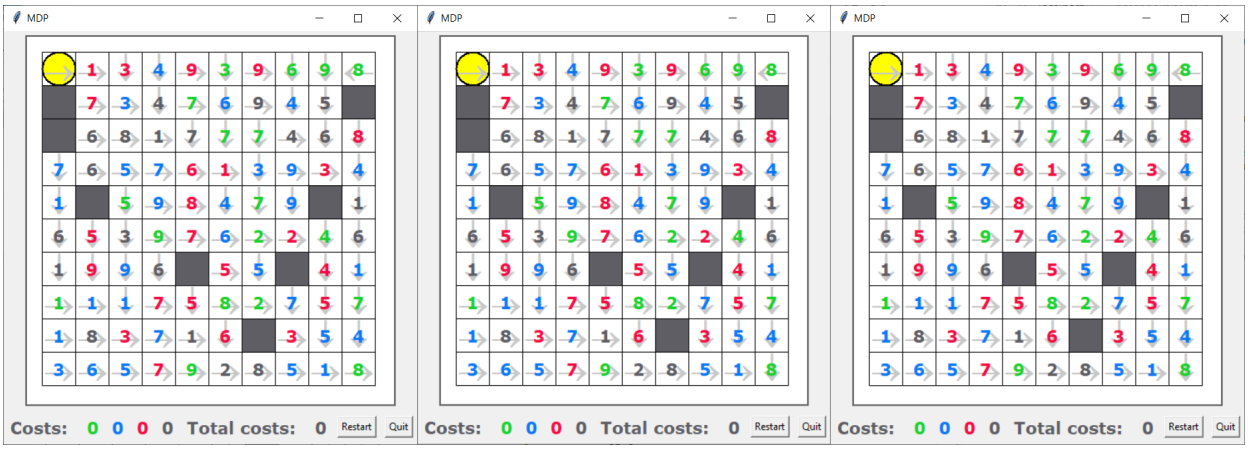
Concernant le critère équilibré de la Section 4.2, comme expliqué précédemment, ce critère ne correspond pas à la simple résolution d'un PDM mono-objectif, et donc une nouvelle fonction a été codée pour implémenter la résolution du problème $(\mathcal{D}_{\min})$. Cette fonction, `pol_pl_mixte_mo`, prend en argument la grille, la valeur de γ et la valeur de M, résout $(\mathcal{D}_{\min})$ à travers Gurobi, et retourne la stratégie mixte résultante sous la forme d'un tableau de dimension 3 (avec la même convention que la fonction `pol_pl_mixte` quant au traitement des murs). Comme `pol_pl_mixte`, la fonction `pol_pl_mixte_mo` retourne aussi le vecteur de récompenses selon chacun des critères.

4.4 Résultats

4.4.1 Exemple de calcul de stratégie pour la minimisation de la consommation totale de ressources

Afin de tester le mode **''somme_chiffre''** des fonctions `pol_valeur`, `pol_pl_mixte` et `pol_pl_pure`, on les a testés dans le fichier Jupyter notebook annexe sur une grille 10×10 avec $p = 1$, $\gamma = 0{,}9$ et $M = 100$. Les résultats sont présentés dans la Figure 8. Comme dans la Section 3.4.1, on observe que la stratégie mixte calculée est, comme attendue, une stratégie pure, avec dans chaque case une unique direction avec probabilité 1 et les autres directions avec une probabilité 0. Ainsi, la Figure 8(c) représente cette unique direction de probabilité 1.

On observe dans la Figure 8 que les stratégies optimales calculées par programmation linéaire en stratégies pures et mixtes coïncident. Elles sont aussi très similaires à celle calculée par itération de la valeur, les différences étant la case but (où, comme dans la Section 3.4.1, les deux stratégies $\downarrow$ ou $\rightarrow$ sont optimales) et la case $(8,7)$. Pour cette dernière, l'itération de la valeur donne $\rightarrow$ comme action optimale, alors que les deux approches par programmation linéaire donnent $\downarrow$. On observe que ces deux actions sont bien optimales, car les deux mènent à une case de coût 5 et, au bout de deux itérations, on arrive dans les deux cas à la même case $(9,8)$ (on rappelle que ces stratégies ont été calculées avec $p = 1$). Comme dans la Section 3.4.1, dans le cas $p < 1$, ce type de phénomène de multiples stratégies optimales est beaucoup plus rare et n'a pas été observé dans nos tests, à part dans la case but.



(a) (b) (c)

FIGURE 8 – Stratégies optimales pour la minimisation de la consommation totale de ressources calculées (a) par itération de la valeur, (b) par programmation linéaire en stratégies pures, et (c) par programmation linéaire en stratégies mixtes

4.4.2 Exemple de calcul de stratégie pour la minimisation d'un critère équilibré

On a également testé, dans le fichier Jupyter notebook annexe, la fonction `pol_pl_mixte_mo` sur la même grille utilisée dans la Section 4.4.1. Comme il ne s'agit pas d'un PDM à un critère scalaire, il n'y a pas de résultat théorique garantissant que la stratégie mixte optimale est une stratégie pure, et on observe bien que, en effet, dans certaines cases, la loi de probabilité sur les actions trouvée dans la stratégie optimale n'est pas déterministe. La Figure 9 représente la stratégie optimale trouvée en dessinant dans chaque case une flèche dans la direction de probabilité maximale.

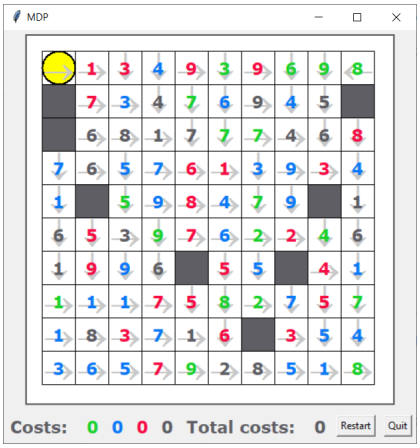


FIGURE 9 – Directions de probabilité maximale pour la stratégie mixte optimale pour la minimisation d'un critère équilibré calculées par programmation linéaire

Comme attendu, puisque le critère d'optimisation est différent, plusieurs cases sont différentes de celles de la Figure 8, mais il y a aussi plusieurs cases identiques puisque le robot a toujours pour but d'arriver à la même case but. On a observé, dans des tests sur quelques grilles, qu'il y a en général très peu de cases où la politique optimale n'est pas une loi de probabilité certaine. Dans la grille de ce test, uniquement trois cases contiennent des lois non-certaines, données dans la Table 5.

Case	$\uparrow$	$\rightarrow$	$\downarrow$	$\leftarrow$
$(2,4)$	0	0,1415	0,8585	0
$(5,8)$	0	0,0020	0,9980	0
$(7,2)$	0	0,6969	0,3031	0

TABLE 5 – Lois de probabilité de la stratégie optimale des cases de la grille ne contenant pas une loi de probabilité certaine

4.4.3 Temps de résolution de la minimisation d'un critère équilibré

À l'aide de la fonction `tester_temps`, on a testé le temps de résolution de l'algorithme de programmation linéaire pour la minimisation d'un critère équilibré implémenté dans la fonction `pol_pl_mixte_mo`, suivant la même procédure expérimentale qu'à la Section 2.3.4 mais avec les valeur $p = 0{,}7$ et $p = 1$ à la place de $p = 0{,}6$ et $p = 1$ en suivant la consigne donnée à l'énoncé. Les résultats de temps de calcul moyen sont donnés dans la Table 6

	$\gamma = 0{,}5$		$\gamma = 0{,}7$		$\gamma = 0{,}9$	
	$p = 0{,}7$	$p = 1$	$p = 0{,}7$	$p = 1$	$p = 0{,}7$	$p = 1$
10×10	25,0	27,2	29,6	25,9	28,9	25,5
10×15	39,2	43,1	46,4	40,5	47,7	39,3
15×20	93,0	79,3	99,8	80,8	103,3	88,1

TABLE 6 – Temps de calcul moyen (en millisecondes) de l'algorithme par programmation linéaire pour le calcul d'une stratégie optimale pour un critère équilibré

On observe dans la Table 6 que, en général, le temps de calcul diminue lorsque $p = 1$ et ce, pour la même raison qu'à la Section 3.4.2 : lorsque $p = 1$, plusieurs coefficients de la matrice de contraintes du programme linéaire à résoudre deviennent 0, et cette structure plus creuse accélère le temps de calcul. L'ordre de grandeur des temps de calcul pour la grille de taille 10×15 est essentiellement le même que celui donné dans la Table 3 pour le calcul des stratégies mixtes. Bien que les problèmes en question soient différents, les programmes linéaires $(\mathcal{D})$ et $(\mathcal{D}_{\min})$ résolus dans les deux cas sont très similaires : $(\mathcal{D}_{\min})$ contient les mêmes contraintes que $(\mathcal{D})$ et quatre contraintes en plus (une pour chaque couleur), ainsi que les mêmes variables de décision et une en plus, z. On remarque également dans la Table 6 que la taille de la grille a un impact important dans le temps de calcul, car le nombre de variables de décision et de contraintes augmente.

4.4.4 Performances moyennes de la minimisation équilibrée sur une instance test

Outre la stratégie optimale, la fonction `pol_pl_mixte_mo` retourne le vecteur avec les valeurs optimales du programme linéaire résolu selon chaque critère, qui correspondent à l'espérance des récompenses selon chaque critère avec la stratégie optimale et lorsque la position initiale du robot est choisie selon la loi μ (uniforme dans notre cas). Afin de vérifier cette correspondance, on compare dans cette section le vecteur retourné par `pol_pl_mixte_mo` avec une moyenne empirique issue de plusieurs simulations de la stratégie optimale calculée par cette même fonction, à l'aide de la fonction `simulation` décrite dans la Section 1.1.3. Cela a été fait dans la grille de la Figure 10 et les paramètres $p = 0{,}7$, $\gamma = 0{,}9$ et $M = 10$.

La fonction `pol_pl_mixte_mo` retourne comme vecteur de récompenses le vecteur (33,166, 33,166,

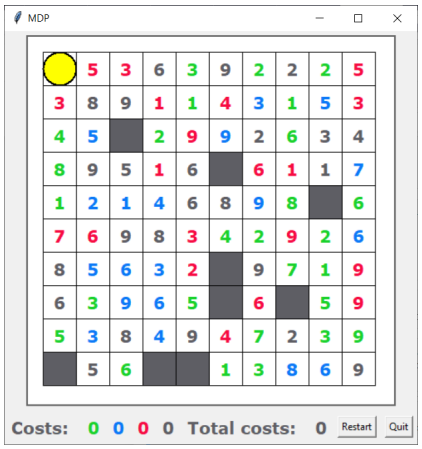


FIGURE 10 – Grille utilisée pour l'étude des performances moyennes de la minimisation équilibrée

34,987, 33,166) qui, comme attendu, cherche bien à équilibrer tous les critères. Afin d'utiliser la fonction `simulation`, il faut penser à une différence de comportement entre le PDM implémenté dans la fonction `pol_pl_mixte_mo` et la simulation de la fonction `simulation` : dans la première fonction, on considère que le PDM continue à l'infini, et donc, une fois arrivé à la case but, le robot reçoit la récompense M à chaque itération jusqu'à l'infini, alors que, pour la fonction `simulation`, le robot reçoit un bonus passé en argument lors de l'arrivée sur la case but et ensuite la simulation s'arrête. Afin de rendre ces deux choses comparables, le bonus reçu par le robot lorsqu'il arrive à la case but dans la simulation doit être comparable à gagner M à l'infini avec un facteur d'amortissement γ, ce qui veut dire que le bonus passé en argument doit être $\frac{M}{1-\gamma}$.

En répétant 10000 simulations (plutôt que les 15 demandées à l'énoncé, afin de réduire la variance des résultats), on obtient à la fin un vecteur de récompense moyen de (33,301, 33,166, 35,006, 33,125). On observe qu'il est très proche du vecteur retourné par la fonction `pol_pl_mixte_mo`, comme attendu.

4.4.5 Comparaison entre la minimisation d'un critère équilibré et de la consommation totale

On a tourné les fonctions `pol_pl_mixte` en mode `''somme_chiffre''` et `pol_pl_mixte_mo` sur trois grilles de tailles respectives 10×10, 10×15 et 15×20, et on a récupéré les valeurs des objectifs selon chaque critère. Les résultats obtenus sont donnés dans la Table 7.

Taille	Critère	Valeurs des objectifs
10×10	Consommation totale	(419,9, 422,3, 421,9, 419,7)
	Critère équilibré	(420,2, 420,8, 421,5, 420,2)
10×15	Consommation totale	(329,8, 332,9, 330,4, 332,7)
	Critère équilibré	(330,7, 331,5, 330,7, 331,2)
15×20	Consommation totale	(210,8, 212,0, 213,7, 210,0)
	Critère équilibré	(210,8, 211,6, 212,5, 210,8)

TABLE 7 – Comparaison entre les valeurs selon chaque objectif retournées par `pol_pl_mixte` en mode `''somme_chiffre''` et `pol_pl_mixte_mo` sur trois grilles de tailles différentes

La Table 7 a été calculée avec $p = 0{,}8$, $\gamma = 0{,}9$ et $M = 100$. On y observe le comportement attendu : l'optimisation de la consommation totale optimise la somme des récompenses, mais les récompenses selon chaque critère sont plus variées, alors que l'optimisation selon le critère équilibré, qui maximise le minimum des récompenses, conduit à des vecteurs plus équilibrés.