

Projet MADI – Processus de décision Markovien et apprentissage par renforcement.

Defiolles Julien et Rey Simon

Le but de ce projet est de développer une application permettant de résoudre des labyrinthes non déterministes. Il s'appuie pour cela sur la résolution des processus de décision Markovien ainsi que l'apprentissage par renforcement.

Le projet est développé en Python et utilise le logiciel sous licence Gurobi pour la résolution des programmes linéaires, pour l'affichage graphique la librairie pygame est utilisée.

Notations. On considère dans la suite un donjon carré donné. On note n la taille du donjon définit le nombre de cases dans une ligne (ou une colonne) du donjon, le donjon a donc $n \times n$ cases. La case départ est la case à la position $(n-1, n-1)$ et le trésor est en position $(0, 0)$. On note \mathcal{C} l'ensemble des positions pour lesquelles il n'y a pas de mur. Pour une position $p \in \llbracket 0, n-1 \rrbracket \times \llbracket 0, n-1 \rrbracket$, on note $adj(p)$ les positions adjacentes qui ne sont pas des murs.

1 Gestion des donjons

Pour représenter les donjons nous avons développé une classe `Dungeon` qui stocke un donjon comme une matrice de case, chaque case étant définit par son type (Clé, Piège, Trésor, Portail...).

Cette classe nous permet de créer un donjon à partir d'un fichier définissant un donjon selon le standard suivant : la première ligne donne la dimension du donjon les coordonnées étant séparées par un espace, les autres lignes définissent les cases du donjon, ligne par ligne sans séparateur. Les caractères utilisés pour chaque case sont donnés en Table 1.

Ainsi le donjon donné dans l'énoncé s'écrit :

```
8 8
te  rwck
wrww_c_r
_ e _r
pw _p_r
swwew ww
_r r_cp
wwew w
e o
```

La classe permet aussi d'écrire un donjon dans un fichier, ce qui permet de les sauvegarder.

Les donjons peuvent aussi être générés aléatoirement, dans ce cas les cases obligatoires (Départ, Trésor, Clé et Épée) sont positionnées dès le départ, les autres cases sont remplies en suivant les probabilités données dans la table précédente.

L'amélioration que nous avons décidé de mettre en œuvre est de prendre en compte des points de vie pour le joueur. Ainsi à la création d'un joueur un paramètre définit ses points de vie initiaux. Nous avons donc changé les effets de certaines cases, les ennemies et les pièges retirent donc des points de vie plutôt que de tuer le joueur (voir la fonction de transition du PDM en Table 2 pour plus de précisions).

2 Déroulement d'une partie

La classe `State` définit les états d'une partie (position, possessions et vie du joueur). C'est aussi dans cette classe qu'est implémentée la fonction d'évaluation des états pour le PDM.

Nous avons aussi créé une classe `Player` qui permet de représenter le joueur. On y retrouve sa position, les objets qu'il possède ainsi que ses points de vie. Cette classe peut sembler ne pas apporter d'informations supplémentaires que la classe `State`, cependant cela permet d'avoir une organisation du projet plus cohérente du point de vue sémantique et permet de bien séparer les méthodes spécifiques aux deux.


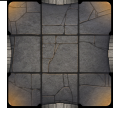
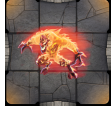

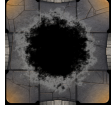



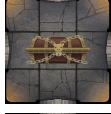

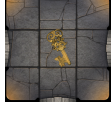

Case	Caractère associé	Probabilité de création	Image utilisée	Case	Caractère associé	Probabilité de création	Image utilisée
Mur	w	0.2		Vide	␣	0.4	
Ennemie	e	0.1		Piège	r	0.1	
Fissure	c	0.05		Portail	p	0.05	
Plateforme	–	0.1		Départ	o		
Trésor	t			Épée	s		
Clé	k			Héro			

TABLE 1 – Informations sur les cases des donjons

La classe `Graphics` gère l’affichage de la partie grâce à la librairie `pygame`. Elle permet d’afficher un donjon, les déplacements du joueur, des messages ainsi que la stratégie calculée par les PDM.

Le déroulement d’une partie est le suivant :

- Le mouvement du joueur est récupéré (dans la politique optimale du PDM ou par les entrées de l’utilisateur)
- La méthode `move` de la classe `Player` est appelée avec ce mouvement. Cette dernière réalise les actions suivantes :
 - Elle applique le mouvement du joueur en le déplaçant sur la case suivante
 - Elle applique toutes les réactions de la nouvelle case (attaque d’ennemies, téléportation par le portail...)
 - Si de nouvelles réactions doivent être appliquée à cause des réactions précédentes (téléportation d’un portail qui déclenche une attaque d’ennemie), elles sont appliquées jusqu’à épuisement des réactions.
- Si à l’issue du déplacement la partie se termine (victoire ou défaite) un message est affiché et le programme se termine. Sinon la même boucle continue.

3 Modélisation par les processus décisionnels Markoviens

Un Processus Décisionnel Markovien (PMD) est défini par un tuple $\langle S, A, T, R \rangle$ où S est un ensemble d’état, A un ensemble fini d’actions, $T : S \times A \rightarrow L(S)$ une fonction de transition et $R : S \times A \rightarrow \mathbb{R}$ une fonction de récompense immédiate. Pour modéliser le jeu comme un PMD il nous faut donc définir l’ensemble de ces éléments. Nous avons choisi la représentation suivante :

- Un état est un tuple $\langle t, s, k, pos, life \rangle$ où t, k et s sont des variables binaires représentant le fait que le joueur possède respectivement le trésor, l’épée ou la clé, $pos \in \llbracket 0, n - 1 \rrbracket \times \llbracket 0, n - 1 \rrbracket$ représente la position du joueur et $life \in \mathbb{N}$ son nombre de points de vie.

Type de case	Nouvel état	Probabilité
Départ	$\langle t, s, k, pos, life \rangle$	1
Vide	$\langle t, s, k, pos, life \rangle$	1
Clé	$\langle t, s, True, pos, life \rangle$	1
Épée	$\langle t, True, k, pos, life \rangle$	1
Trésor et $k = True$	$\langle True, s, k, pos, life \rangle$	1
Trésor et $k = False$	$\langle t, s, k, pos, life \rangle$	1
Fissure	$\langle t, s, k, pos, 0 \rangle$	1
Ennemie et $s = True$	$\langle t, s, k, pos, life \rangle$	1
Ennemie et $s = False$	$\langle t, s, k, pos, life - 1 \rangle$	0.3
	$\langle t, s, k, pos, life \rangle$	0.7
Piège	$\langle t, s, k, pos, life \rangle$	0.6
	$\langle t, s, k, pos, life - 1 \rangle$	0.1
	$\langle t, s, k, (n - 1, n - 1), life \rangle$	0.3
Plateforme	$\forall pos' \in adj(pos), \langle t, s, k, pos', life \rangle$	$1/ adj(pos) $
Portail	$\forall pos' \in \mathcal{C} \setminus \{pos\}, \langle t, s, k, pos', life \rangle$	$1/ \mathcal{C} \setminus \{pos\} $

TABLE 2 – Fonction de transition du PDM

L'ensemble S est alors défini comme l'ensemble des états possibles, soit l'ensemble des couples $\langle t, s, k, pos, life \rangle$ possible tels que la case à la position pos ne soit pas un mur.

- L'ensemble A des actions correspond aux mouvement du joueur, on a donc $A = \{\rightarrow, \downarrow, \leftarrow, \uparrow\}$.
Pour un état donné toutes les actions ne sont pas nécessairement disponibles, ainsi l'ensemble $A(st)$ des actions disponible pour un état $st = \langle t, s, k, pos, life \rangle$ dépend de la case à la position pos :
 - Si c'est une fissure (C), un portail (P), une plateforme (–) ou si le joueur est mort ($life = 0$), alors aucune action n'est possible, $T(st) = 0$
 - Sinon $T(st) \subseteq \{\rightarrow, \downarrow, \leftarrow, \uparrow\}$ de telle sorte que l'action ne mène pas contre un mur (on considère que le donjon est entouré de mur).
- La fonction de transition correspond à celle donnée dans l'énoncé. En partant d'un état $\langle t, s, k, pos_0, life \rangle$ et avec une action qui mène à une position pos , la fonction de transition est présentée en Table 2.
- La fonction d'évaluation des états est très simples tous les états donnent une récompense de -1 excepté l'état $\langle True, s, k, (n - 1, n - 1), life \rangle$ qui procure une récompense de 1000000 et les états $\langle s, k, t, pos, 0 \rangle$ qui procurent une récompense de -1000 .
Ainsi chaque pas est récompensé négativement pour pousser la résolution à choisir des chemins courts, mais c'est seulement en rapportant le trésor que le joueur peut marquer un score positif.

4 Résolution du PDM

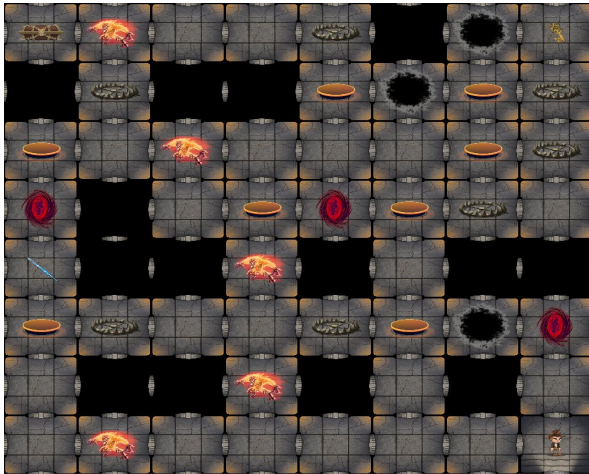
Pour résoudre le PDM décrit ci-dessus nous avons implémenté l'algorithme d'itération de la valeur ainsi que la résolution par la programmation linéaire.

Pour cela nous avons écrit une classe `BellmanEquation` dont une instance est créé pour chaque nœud du PDM. Cette classe est centrale dans la résolution du PDM, la méthode `next_value` retourne la nouvelle valeur d'un nœud étant donné la valeur courante et la méthode `get_constraints` retourne les contraintes du programme linéaire qui concerne le nœud.

L'algorithme d'itération de la valeur procède de manière habituelle. En commençant avec une valuation arbitraire, il itère la méthode `next_value` pour actualiser la valeur des nœuds. L'algorithme s'arrête dès lors que la précision est suffisante, c'est-à-dire dès lors que la différence entre la valuation courante et la valuation précédente est inférieure à 1.

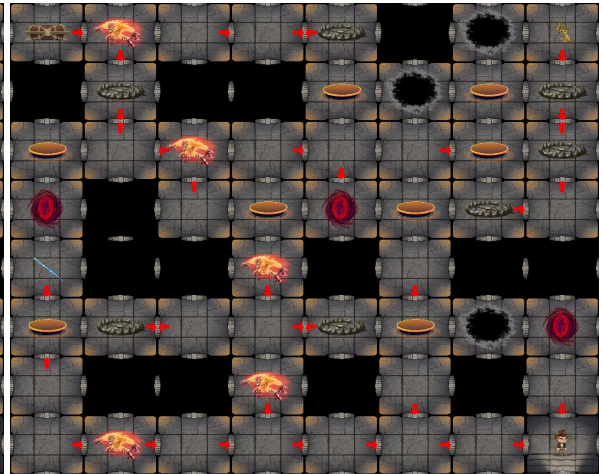
La résolution par la programmation linéaire est classique, la méthode `get_constraints` permet de générer toutes les contraintes du programme linéaire qui est ensuite résolu par Gurobi.

Un exemple de résolution est donné en Figure 1 pour le donjon donné dans le sujet. Cinq politiques sont représentées : la politique initiale, celle lorsque le joueur possède la clé et celle lorsque le joueur possède le trésor et deux des politiques précédentes avec 4 points de vie. Ce n'est pas les seules politiques représentables, il faudrait aussi prendre en compte l'épée et tous les niveaux de points de vie. Il est



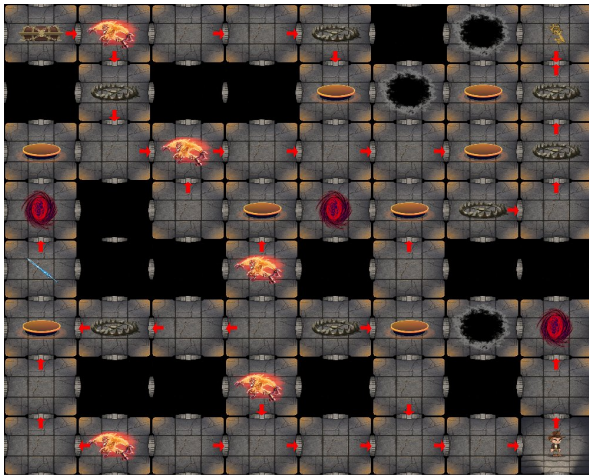
Welcome to Magic Maze, you are looking at the moves computed by the PDM resolution. | Player life = 1/1

(a) Donjon



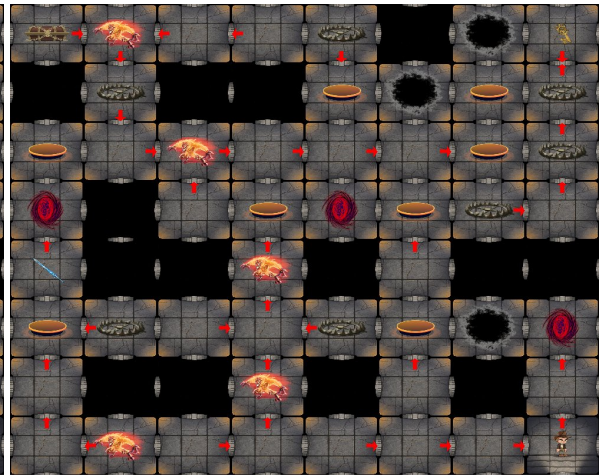
Welcome to Magic Maze, you are looking at the moves computed by the PDM resolution. | Player life = 1/1

(b) Politique optimale pour rentrer avec le trésor



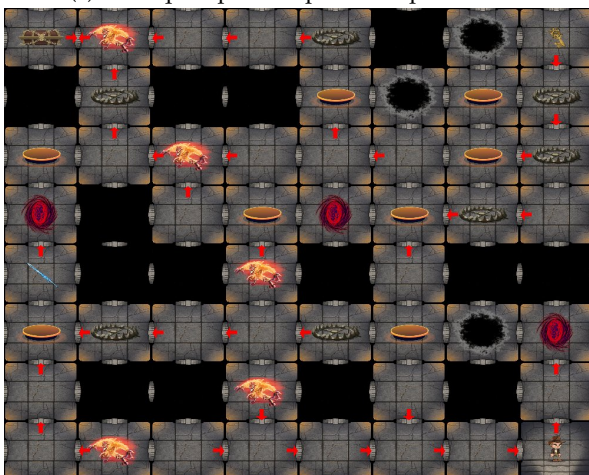
Welcome to Magic Maze, you are looking at the moves computed by the PDM resolution. | Player life = 1/1

(c) Politique optimale pour récupérer la clé



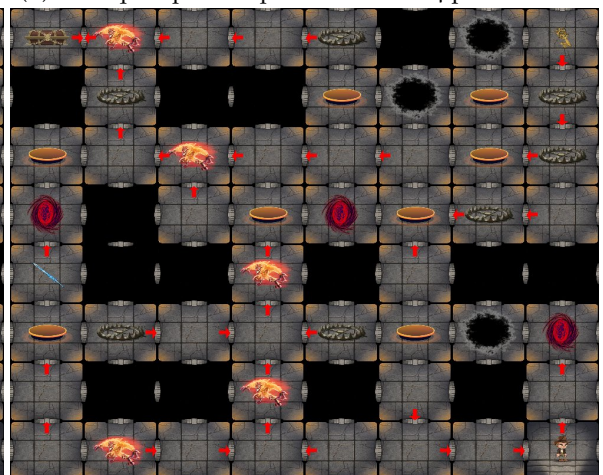
Welcome to Magic Maze, you are looking at the moves computed by the PDM resolution. | Player life = 4/4

(d) Politique optimale pour la clé avec 4 points de vie



Welcome to Magic Maze, you are looking at the moves computed by the PDM resolution. | Player life = 1/1

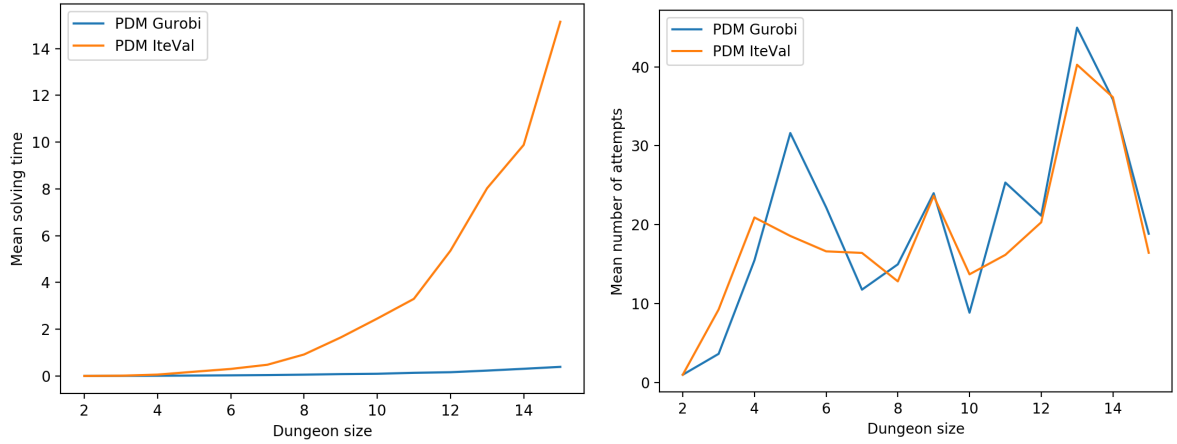
(e) Politique optimale pour aller chercher le trésor



Welcome to Magic Maze, you are looking at the moves computed by the PDM resolution. | Player life = 4/4

(f) Politique optimale pour le trésor avec 4 points de vie

FIGURE 1 – Politiques optimales pour le donjon de l'énoncé



(a) Temps moyen de résolution

(b) Nombre d'essai moyen avant la première victoire

FIGURE 2 – Comparaison des résolution par itération de la valeur et par programmation linéaire

intéressant d'observer les différences de politique lorsque le joueur a 4 points de vie. On observe alors que le joueur préfère combattre plusieurs ennemis (perte de points de vie) que de prendre le risque de tomber dans une fissure (mort assurée). Cela s'observe notamment sur les cases en haut à gauche entre les figures 1c et 1d.

D'autres exemples de résolution avec des donjons de taille différente peuvent être trouvés dans le dossier Images de l'archive.

En termes de résolution il est intéressant de mentionner le fait que le joueur utilise très fortement les portails. En effet ces derniers permettant avec une probabilité assez importante de gagner du temps (et du de recevoir moins de récompenses de -1), notamment lorsque le joueur est loin de la case visée.

5 Évaluation expérimental de la résolution du PDM

Pour évaluer expérimentalement les performances de nos algorithmes, nous avons mesurer à la fois les temps de résolution et le nombre d'essai nécessaire pour gagner le donjon.

La procédure de test est la suivante :

- Pour toute les tailles de donjon de 2 à 15, 10 instances sont générées
- Pour chaque instance, le temps de résolution et le nombre d'essai nécessaire pour gagner sont mesurer dix fois pour en prendre la moyenne
- Les graphiques sont ensuite tracés représentant les mesures selon la taille du donjon (moyenne sur les 10 instances d'une taille donnée et sur les 10 essais de chaque instance).

Les graphiques obtenus sont présentés en Figure 2 et 3.

La Figure 2a indique clairement que la résolution par la programmation linéaire en utilisant Gurobi est supérieure pour résoudre le PDM. En effet sur les donjons de taille importante le temps de résolution est bien inférieur en utilisant Gurobi allant jusqu'à un facteur 1/15 pour des donjons de taille 15. D'autres expérimentations ont permis de constater que Gurobi permet de résoudre rapidement des donjons de taille 30, soit 900 cases.

Nous avons aussi tracé le nombre d'essai moyen nécessaire pour obtenir une victoire, cela est présenté en Figure 2b. Cette figure ne permet cependant pas de dégager de tendance claire. Cela est sûrement dû au nombre de répétition (10) qui n'est pas suffisant pour avoir des moyennes pertinentes. De plus, nous faisons la moyenne sur des 10 instances de même taille mais très différente, la taille de l'instance n'est pas nécessairement corrélée à la probabilité de victoire.

Enfin, nous avons tracé le nombre moyen d'itération nécessaire à l'algorithme d'itération de la valeur pour atteindre sa solution optimal. Les résultats sont donnés en Figure 3. La progression est relativement constante pour un nombre d'itération qui reste faible. Il faut cependant noter que le nombre d'itération est qu'un indicateur du temps de résolution. En effet le temps pour une itération est bien plus important pour les grands donjons que pour les plus petits.

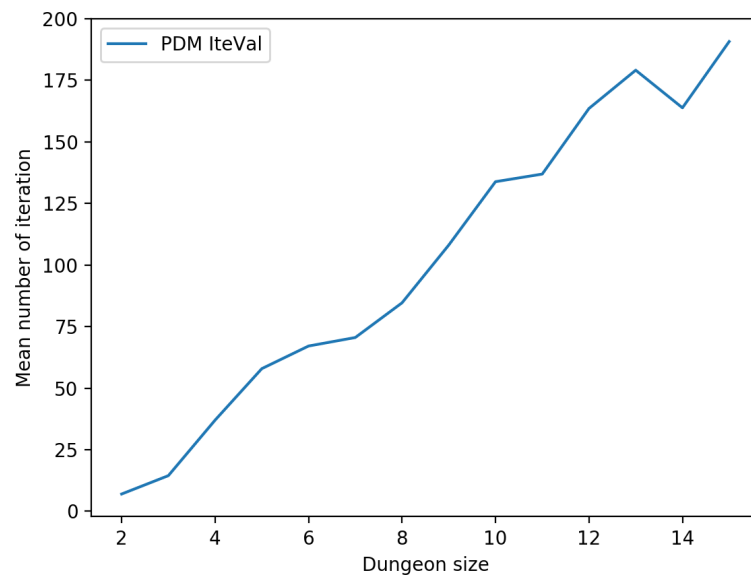


FIGURE 3 – Nombre de pas de l’algorithme d’itération de la valeur selon la taille du donjon

6 Apprentissage par renforcement