



UNIVERSITÉ  
DE LORRAINE



Master Informatique

# IA d'assistance pour jeu de gestion de crise

Rapport  
en vue de la validation de l'UE Initiation à la recherche

2024-2025

## Étudiants

Tristan LAGARDE  
Morade AIACHI  
Emanuel FERNANDES DOS SANTOS

## Encadrants

Olivier BUFFET  
Vincent THOMAS  
Stéphane GORIA



## DÉCHARGE DE RESPONSABILITÉS

L'Université de Lorraine n'entend donner ni approbation ni improbation aux opinions émises dans ce rapport, ces opinions devant être considérées comme propres à leur auteur.

## Remerciements

Nous tenons à remercier vivement nos encadrants, Monsieur Olivier BUFFET, enseignant chercheur au LORIA, Monsieur Vincent THOMAS, enseignant chercheur au LORIA et Monsieur Stéphane GORIA, enseignant chercheur au CREM, pour leur accueil, leur temps consacré et le partage de leur expertise au quotidien. Grâce aussi à leur confiance nous avons pu nous accomplir totalement dans nos missions. Ils furent d'une aide précieuse dans les moments les plus délicats.

# Table des matières

<b>1</b>	<b>Introduction</b>	
<b>2</b>	<b>Processus de Décision Markovien</b>	<b>1</b>
2.1	Définitions . . . . .	1
2.2	Itération-valeur . . . . .	2
2.3	Real Time Dynamic Programming . . . . .	3
<b>3</b>	<b>Méthode</b>	<b>4</b>
3.1	Jeu . . . . .	4
3.2	IA de conseil . . . . .	6
3.3	IA d'assistance . . . . .	10
<b>4</b>	<b>Expérimentation</b>	<b>11</b>
4.1	Évaluation du taux de succès de l'IA . . . . .	11
4.2	Performance de l'algorithme Itération-Valeur . . . . .	11
<b>5</b>	<b>Conclusion</b>	<b>13</b>
<b>6</b>	<b>Annexes</b>	<b>1</b>
<b>7</b>	<b>Bibliographie</b>	<b>19</b>
<b>8</b>	<b>Glossaire</b>	
<b>9</b>	<b>Quatrième de couverture</b>	

# 1 Introduction

Les **serious games** sont des jeux de table dits sérieux, c'est-à-dire qu'ils mettent en scène des situations sérieuses, parfois réalistes, où les décisions sont par moments difficiles et nécessitent souvent de prendre en compte de nombreux **paramètres** et de prévoir plusieurs **tours** à l'avance des événements intervenant dans le futur. On peut compter parmi ce genre des jeux comme *Civilization*, *Age of Empires* [2] ou *Hostage Rescue Squad*. Dans ce dernier, où l'on incarne une équipe d'élite de sauvetage, l'aléatoire est prédominant et il peut être compliqué de trouver un chemin menant à l'otage sans introduire une part de risque non négligeable et qui nécessite donc d'envisager plusieurs routes.

Ainsi, pour comprendre comment aider les joueurs de wargames, nous devons introduire plusieurs notions. Un **agent** est un programme capable de prendre des décisions dans son environnement, et connaît tous les aspects de celui-ci. L'**état** de cet environnement est la collection de tous ses éléments variables, comme la position du joueur, des ennemis, de l'otage, etc. L'environnement est un **environnement stochastique**, c'est-à-dire qu'il est régi par l'aléatoire : par exemple, un même personnage se déplaçant d'une case à une autre en partant du même état de départ peut ne pas forcément arriver sur le même état d'arrivée. L'état d'arrivée est déterminé grâce à une fonction de **transition**, qui indique dans quel état l'environnement arrive en partant d'un état de départ et d'une action à effectuer. Une **action** est le coup d'un personnage. Ici, chaque personnage effectue tous ses coups avant de passer au prochain. Enfin, on évalue les actions selon une fonction de **récompense**, qui va indiquer la plus-value de l'action en fonction de l'état de départ et de l'état d'arrivée.

Le but ici est de créer un agent basé sur la **planification probabiliste**, capable de pouvoir **conseiller** le joueur en lui proposant des **actions pertinentes** pour la situation du jeu dans laquelle il se trouve. Il doit être capable de raisonner sur les **conséquences à long terme**, mais procéder ainsi peut rendre les actions et **explications complexes** ; il faut ainsi trouver un moyen de présenter les informations aux joueurs de façon pertinente pour qu'ils puissent l'utiliser au mieux. Aussi, ce genre d'algorithmes peut être très coûteux en temps surtout sur des jeux de grande complexité : il faut optimiser ces calculs afin que le temps d'exécution reste raisonnable, voire utiliser des **heuristiques** qui seront plus rapides au détriment de la qualité des conseils.

Pour cela, nous allons détailler la notion de **Processus de Décision Markovien** et plus précisément deux algorithmes faisant partie de cette famille de planification probabiliste : l'**Itération-Valeur** et le **Real Time Dynamic Programming**. Ensuite, nous décrirons le serious game inspirant le jeu développé, *Hostage Rescue Squad*, ainsi que l'architecture du projet, les différentes étapes du développement de celui-ci et les difficultés rencontrées pendant, puis l'implémentation de l'IA de conseil et l'IA d'assistance. Enfin, nous évaluerons le taux de réussite de l'IA de conseil et la performance de l'implémentation d'ITÉRATION-VALEUR.

## 2 Processus de Décision Markovien

### 2.1 Définitions

Nous allons définir les notions nécessaires à la compréhension des algorithmes par la suite.

Un **état** est la collection de tous les éléments variables de l'environnement.

Une **transition**  $T(s_t, a_t, s_{t+1})$  est un triplet avec  $s_t$  l'état de départ, auquel on applique l'action  $a_t$  pour arriver à l'état  $s_{t+1}$ . Si l'environnement de l'agent est stochastique, alors la probabilité d'arriver à l'état  $s'$  si l'action  $a$  a été faite sur l'état  $s$  se note  $P(s'|s, a)$ . La transition est dite **markovienne** si la probabilité d'arriver en  $s'$  à partir de  $s$  ne dépend que de  $s$  et non de l'historique des états antérieurs.

La **récompense** de l'action  $a$  sur l'état  $s$  pour arriver en  $s'$  se note  $R(s', a, s)$  et est bornée par  $\pm R_{max}$ . Les récompenses d'une séquence d'états  $h = [s_0, a_0, s_1, a_1, \dots, s_n]$  sont dites **additives** si la récompense de  $h$  est

$$U_h([s_0, a_0, s_1, a_1, \dots, s_n]) = R(s_0, a_0, s_1) + R(s_1, a_1, s_2) + \dots + R(s_{n-1}, a_{n-1}, s_n) \quad (1)$$

et sont dites **additives dévaluées** si l'utilité d'une séquence d'états  $h$  vaut

$$U_h([s_0, a_0, s_1, a_1, \dots, s_n]) = R(s_0, a_0, s_1) + \gamma R(s_1, a_1, s_2) + \gamma^2 R(s_2, a_2, s_3) + \dots + \gamma^{n-1} R(s_{n-1}, a_{n-1}, s_n) \quad (2)$$

avec  $\gamma \in [0, 1]$  le **facteur de dévaluation**. Ce facteur quantifie dans quelle mesure un agent préfère une récompense actuelle aux futures : si  $\gamma$  proche de 0, alors il accordera peu d'importance aux récompenses futures, mais si proche de 1, alors il préférera les récompenses à long terme.

Une **politique**  $\pi$  est une solution qui spécifie quelle action l'agent doit effectuer pour tous les états qu'il pourrait atteindre. On note  $\pi(s)$  l'action recommandée par la politique  $\pi$  pour l'état  $s$ . On mesure la qualité d'une politique par l'**utilité espérée** des historiques d'environnement générés qu'elle génère, notée  $U_h([s_0, a_0, s_1, a_1, \dots, s_n])$ . Une, ou plusieurs, politique  $\pi^*$  est **optimale** si elle génère l'utilité la plus élevée.

Un **processus de décision de Markov** est la spécification d'un problème de décision séquentiel pour un environnement stochastique entièrement observable avec un modèle de transition markovien et des récompenses additives. Il est composé de :

- un ensemble d'états dont  $s_0$  l'état initial
- un ensemble d'actions  $Actions(s)$  pour chaque état  $s$
- un modèle de transition  $P(s'|s, a)$
- une fonction de récompense  $R(s)$  pour chaque état  $s$

Soit  $U(s)$  le maximum d'utilité espérée à l'étape  $s$ . On définit  $U(s)$  par l'**équation de Bellman** :

$$U(s) = \max_{a \in A(s)} \sum_{s'} P(s'|s, a) [R(s, a, s') + \gamma U(s')], \quad (3)$$

C'est-à-dire que l'utilité d'un état est la récompense immédiate associée à la transition suivante plus l'utilité dévaluée espérée de l'état suivant, à supposer que l'agent choisisse l'action optimale.

Une **fonction de valeur d'action** ou **Q-fonction** est une fonction  $Q(s, a)$  de l'utilité espérée d'effectuer  $a$  sur l'état  $s$ , définie par :

$$Q(s, a) = \sum_{s'} P(s'|s, a) [R(s, a, s') + \gamma U(s')] \quad (4)$$

On a donc  $U(s) = \max_a Q(s, a)$  et  $\pi_*(s) = \arg \max_a Q(s, a)$ .

## 2.2 Itération-valeur

ITÉRATION-VALEUR est un algorithme qui s'apparente à une **énumération** de toutes les Q-fonctions. Cet algorithme va, pour chaque état, choisir l'action qui apporte la meilleure utilité selon l'équation de Bellman. Une fois cette étape réalisée, on note  $\delta$  la différence absolue d'utilité la plus grande et on relance une itération tant que ce  $\delta$  est au-dessus d'un seuil.

Plus formellement, soit  $S$  l'ensemble des états,  $A(s)$  les actions réalisables à partir de l'état  $s$ ,  $\gamma$  la dévaluation et  $\varepsilon$  le seuil d'arrêt.

ITÉRATION-VALEUR est définie par la fonction suivante [7]

---

### Algorithm 1 Itération-Valeur

---

```

1: function ITÉRATION-VALEUR( $pdm, \varepsilon, \gamma$ )
2:   repeat
3:      $U \leftarrow U'$ 
4:      $\delta \leftarrow 0$ 
5:     for all  $s \in S$  do
6:        $U'[s] \leftarrow \max_{a \in A(s)} Q\text{-Valeur}(pdm, s, a, U)$ 
7:       if  $|U'[s] - U[s]| > \delta$  then
8:          $\delta \leftarrow |U'[s] - U[s]|$ 
9:       end if
10:    end for
11:  until  $\delta \leq \varepsilon(1 - \gamma)/\gamma$  return  $U$ 
```

---

Ici  $\delta$  représente la différence maximale d'utilité à chaque itération.  $\gamma$  dirige l'importance des **récompenses à long terme**. En effet, à mesure que  $\gamma$  diminue, les utilités des états voisins sont de plus en plus **dévaluées** et donc l'agent aura un comportement proche d'un **algorithme glouton**. À l'inverse, avoir un  $\gamma$  proche de 1 augmentera l'importance des utilités des états voisins, et donc l'agent aura tendance à effectuer des actions pour obtenir une récompense à long terme. Cependant, augmenter  $\gamma$  augmente de façon **exponentielle** le nombre d'itérations nécessaires pour l'arrêt de l'algorithme. Un juste milieu doit donc être trouvé entre l'efficacité de la politique et le temps de calcul.

Cet algorithme va nous produire une **politique optimale**, mais possède tout de même certaines limitations. D'abord, elle requiert de **stocker en mémoire** tous les états, nombre qui croît de façon exponentielle avec le nombre de paramètres composant un état. Par exemple, supposons qu'un état soit composé de 9 entiers pouvant chacun prendre 20 valeurs, qu'un entier soit représenté en mémoire par 4 octets et qu'il n'y ait pas de surcoût lié à la gestion des états en mémoire. Stocker tous les états nécessite alors

$$20^9 \times 4 \text{ octets} = \frac{2,048.10^{11}}{1024^3} \text{ To} \approx 1907,34 \text{ To} \quad (5)$$

À titre de comparaison, *El Capitan*, le supercalculateur le plus puissant du monde [6], comporte 5.4375 pétaoctets [5] de mémoire vive. Ce simple exemple remplirait donc  $\approx 34,25\%$  de sa mémoire vive. Ensuite, le nombre d'itérations peut théoriquement croître de façon exponentielle à mesure que  $\gamma$  se rapproche de 1 [7], et pour chaque itération l'algorithme doit passer en revue toutes les actions pour les états pour calculer les Q-valeurs. En reprenant l'exemple ci-dessus et en supposant qu'il y ait 4 actions par état, et que chaque calcul de la Q-valeur ne constitue qu'une opération en réel à point flottant, il y a au total

$$20^9 \times 4 \text{ opérations} = 2,048.10^{11} \text{ opérations} \quad (6)$$

Le processeur le plus puissant disponible dans le commerce [4] est l'*AMD EPYC 9655P*, et il dispose d'une puissance de calcul en moyenne à  $7,1026.10^{11}$  opérations en point flottant par seconde [3]. Il lui faudrait donc dans le meilleur cas au moins  $\approx 288$  ms pour faire une seule itération. Si l'algorithme converge en 1000 itérations, alors ce processeur ne finira au mieux qu'en 4 minutes et 48 secondes.

Cet algorithme n'est donc pas praticable sur des moyens et gros états contenant un nombre de facteurs élevé ou pouvant prendre beaucoup de valeurs distinctes.



## 2.3 Real Time Dynamic Programming

REAL TIME DYNAMIC PROGRAMMING (ou RTDP) est une **heuristique** qui essaie de répondre à la problématique de l'Itération-Valeur. Cet algorithme part de l'état initial, choisit l'action à faire selon un **tirage**, met à jour l'utilité de l'état de départ et se place dans l'état  $s'$  selon  $T(s, a, s')$ . L'algorithme continue cette itération tant qu'il n'a pas atteint son objectif, auquel cas il reprend à l'état initial. Cette boucle continue tant qu'il y a du **temps restant**, c'est-à-dire soit un temps imparti pour réaliser l'algorithme, soit un nombre d'itérations limite.

Plus formellement, soit  $s$  l'état initial de l'agent, RTDP est définie par la fonction suivante [1]

---

**Algorithm 2** RTDP

---

```
1: procedure RTDP( $s$ )
2:   repeat
3:     EssaiRTDP( $s$ )
4:   until condition d'arrêt
5: end procedure
6:
7: procedure ESSAIRTPD( $s$ )
8:   while non BUT( $s$ ) do
9:      $a \leftarrow \text{ActionGloutonne}(s)$ 
10:     $J(s) \leftarrow Q(s, a)$ 
11:     $s \leftarrow \text{ChoisirEtatSuivant}(s, a)$ 
12:   end while
13: end procedure=0
```

---

Cet algorithme a plusieurs avantages. Premièrement, il peut être effectué **en ligne**, c'est-à-dire qu'il n'a pas besoin d'être pré-calculé à l'avance pour donner une politique. On peut tout de même améliorer la qualité de sa politique en calculant au départ une politique avec un temps imparti plus long, puis au fur et à mesure que l'agent se retrouve dans des états inconnus, on effectue RTDP avec un temps imparti plus raisonnable. Deuxièmement, il n'a pas besoin de stocker tous les états en mémoire. En effet, puisque RTDP se base sur des tirages, il aura tendance à ne voir que les états les plus souvent rencontrés et délaissera les plus rares. Cet avantage est très important notamment quand beaucoup de facteurs sont compris dans un état, auquel cas le nombre d'états total croît de façon exponentielle. Troisièmement, son temps d'exécution n'est pas lié au nombre total d'états. Puisqu'il se contente de découvrir les états au fur et à mesure des essais, son temps d'exécution n'est gouverné que par le temps imparti et le temps de calcul de la transition. Enfin, RTDP laisse la possibilité de choisir entre le temps d'exécution et la qualité de la politique par le biais du temps restant. Ce paramètre impose un temps imparti et laisse l'algorithme trouver la meilleure politique dans le temps imparti.

Il existe aussi une version améliorée de RTDP, nommée LABELED REAL TIME DYNAMIC PROGRAMMING (ou LRTDP), où à chaque état rencontré dont la différence d'utilité ne dépasse pas un seuil  $\epsilon$ , on associe un label qui indique qu'au-delà de cet état la solution optimale a été trouvée, et donc qu'il n'est pas nécessaire de continuer au-delà de cet état. Cette amélioration permet d'arrêter prématurément le parcours pour éviter de faire des calculs inutiles et améliorer davantage la politique.

## 3 Méthode

### 3.1 Jeu

Le jeu se base sur le serious game *Hostage Rescue Squad*. Dans ce jeu solo, on incarne une équipe de sauvetage dans une mission d'extraction d'otage. Chaque mission se déroule sur une carte comportant des murs, fenêtres et portes, des obstacles dans lesquels les personnages peuvent se cacher, un ou plusieurs otages à sauver, des terroristes et les opérateurs contrôlés par le joueur. À chaque tour du joueur, il choisit quel opérateur contrôler puis lance trois dés. Les valeurs obtenues par le jet déterminent quelles actions sont possibles : l'opérateur peut se déplacer d'une case pour chaque dé ayant une valeur entre 1 et 4, tirer sur un terroriste en ligne de mire pour chaque dé ayant une valeur égale à 5 ou 6, lancer une grenade, lever son bouclier ou faire une élimination silencieuse pour chaque paire de valeurs égales, etc. Une fois tous les dés utilisés ou le tour passé, le joueur contrôle le prochain opérateur de son choix et quand tous les opérateurs ont fini, le tour du joueur se termine et celui des terroristes commence. Comme pour le tour du joueur, on lance autant de dés que le niveau de menace et en fonction des valeurs obtenues, les terroristes peuvent se déplacer selon une routine, tirer ou appeler des renforts. Ce niveau de menace, compris entre 2 et 7, représente l'agressivité des terroristes et peut augmenter ou diminuer au fil de la mission : un opérateur qui tire, lance une grenade, bloque un tir avec le bouclier augmente la menace, un appel de renfort quand 5 ennemis sont présents l'augmente également de 1, un opérateur qui se calme le diminue de 1. L'élimination silencieuse, contrairement au tir, ne fait pas augmenter la menace, mais nécessite d'être à côté d'un terroriste. Tuer tous les terroristes fait automatiquement baisser la menace à 2 mais les terroristes gardent tout de même une chance d'appeler des renforts. La mission est réussie dès que tous les opérateurs se situent en bas du plateau avec tous les otages, mais elle échoue si l'un des opérateurs ou des otages se fait abattre.

Le jeu est réalisé en **Java 17** avec la librairie graphique **JavaFX 21** et la gestion du versionnage est faite avec **Git**. Ce choix s'est fait par familiarité des membres du groupe, car nous avons tous utilisé ces technologies par le passé pendant notre cursus. Tout au long du développement, nous avons essayé de favoriser l'extensibilité du jeu lors de la conception. Dès le départ, nous avons généré les personnages, les cases et les coups afin de faciliter l'ajout de fonctionnalités à l'avenir, chose confirmée par la suite du développement quand nous avons implémenté, par exemple, l'élimination silencieuse ou les obstacles.

Dans un premier temps, afin d'implémenter plus facilement les algorithmes de planification, nous avons réduit la quantité de fonctionnalités : au plus deux ennemis, un seul opérateur, pas de murs ni obstacles et seule la possibilité de tirer, se déplacer et passer son tour. Aussi, nous avons décidé de modifier la façon dont l'aléatoire a d'influencer la partie ; ici nous ne lançons plus de dés mais le joueur peut décider de faire n'importe quelle action. Cependant, celles-ci ont désormais un coût en points d'actions (ou PA) et une probabilité de succès. Ainsi, un opérateur peut ne pas pouvoir faire un coup par manque de ressources ou le rater par manque de chance. Plus précisément, le joueur disposait de deux PA, se déplacer coûtait 1 PA et réussissait 95% du temps et le tir coûtait 1 PA et réussissait 85% du temps. De même, les ennemis n'ont plus de dés, mais effectuent tous les mêmes deux coups aléatoires avec 70% de chances de se déplacer vers la prochaine case de leur routine et 30% d'essayer un tir. Aussi, le niveau de menace restait constant au niveau 2.

Dans cette première version rudimentaire, représentée dans l'image 1, presque aucune information n'était accessible au joueur. Sauf en observant le code source, il ne pouvait savoir les probabilités de succès ou les coûts de ses coups, le fonctionnement des coups ou leurs conditions, ou même son nombre de PA restants. Après cette version, nous nous sommes concentrés sur l'implémentation d'ITÉRATION-VALEUR, des obstacles, et sur la correction de tous les bugs. Nous détaillons l'implémentation de l'algorithme dans la section 3.2. Quand nous avons obtenu une version fonctionnelle de IV, nous avons décidé en plus de faire augmenter la menace quand le joueur effectue un tir.

Cette version étant principalement consacrée à des fonctionnalités non visuelles, nous avons décidé par la suite d'améliorer l'interface pour rendre les probabilités de succès et les coûts des coups plus accessibles, afficher le nombre de PA restants du joueur ainsi que de laisser la possibilité de recommencer la partie. Cette version est représentée dans l'image 2.

Après cette version, une autre série de corrections de bugs est initiée et nous ajoutons l'affichage et le bouton d'exécution du coup recommandé par l'IA, l'élimination silencieuse, se calmer, les murs, l'affichage du niveau de menace, et l'IA d'assistance. Les détails d'implémentation de cette IA sont dans la section 3.3. Cette version est illustrée dans l'image 3.

Finalement, le jeu s'est divisé en deux versions. Dans la première, illustrée dans l'image 5, nous avons amélioré l'interface graphique afin de l'harmoniser et le rendre plus proche du jeu original, et pour faire apparaître encore plus d'informations au joueur en affichant les avertissements de l'IA d'assistance et

en affichant le fonctionnement de chaque coup en survolant le bouton du coup. Nous avons également caché par défaut les conseils de l'IA et ajouté un bouton pour alterner la visibilité des conseils.

Dans la seconde, illustrée dans l'image 6, nous avons ajouté la réapparition des ennemis aux points de réapparition et, pour des raisons expliquées dans la section 3.2, nous avons changé l'algorithme de l'IA de conseil de ITÉRATION-VALEUR à RTDP.

Ces deux versions devaient, à terme, être fusionnées, mais par manque de temps cela ne s'est pas fait. Au final, nous obtenons, pour la déclinaison *a*, le diagramme de classes illustré dans l'image 7. Certains paquets sont prévus pour recevoir d'autres fonctionnalités, comme par exemple le paquet **separations** qui devait aussi contenir les fenêtres et les portes. D'autres fonctionnalités devaient être implémentées, comme la présence de plusieurs routines pour les terroristes, la possibilité pour eux de pouvoir changer de routine, la présence de plusieurs opérateurs, le lancer de grenades ou les boucliers. Cependant, le manque de temps a aussi empêché le développement de ceux-ci. Aussi, la structure est conçue pour laisser la possibilité d'appliquer les algorithmes à n'importe quel MDP. Par exemple, on pourrait développer une version du jeu où le but est de désamorcer des bombes, tuer des vagues de terroristes ou juste s'extraire d'une mission périlleuse, et les algorithmes resteraient valides tant que les adaptateurs sont implémentés.

### 3.2 IA de conseil

Le sujet de recherche étant «IA d'assistance pour jeu de gestion de crise», l'une de nos principales tâches était de développer une intelligence artificielle pouvant conseiller le joueur dans le choix de ses coups. Pour répondre à cette problématique, nous avons conçu une IA exploitant les résultats de l'algorithme Itération-Valeur défini plus tôt, que nous avons appliqué à l'ensemble des états du jeu. Avant de détailler le procédé, il est important de bien comprendre la structure d'un état et la définition d'un coup (action) dans notre jeu.

Définissons d'abord un état du jeu, nous utilisons les notations suivantes

- $w$  : Largeur du plateau de l'environnement
- $h$  : Hauteur du plateau de l'environnement
- $n$  : Nombre d'opérateurs
- $pam$  : Nombre de points d'actions maximum d'un opérateur
- $m$  : Le nombre d'ennemis
- $r_i$  : Indice de la case dans la routine où se situe le  $i$ -ème ennemi
- $p_i$  : Indice de la case du plateau où se situe le  $i$ -ème opérateur
- $pa_i$  : Nombre de points d'actions restants de l'opérateur  $i$
- $tr$  : Taille de la routine

Un état  $s$  est alors défini par

$$s = \left\{ \begin{array}{ll} \text{Position Opérateur 1} & = (p_1 \in \{0, \dots, (w \times h) - 1\}) \\ \vdots & \\ \text{Position Opérateur } n & = (p_n \in \{0, \dots, (w \times h) - 1\}) \\ \text{Points d'action restants Opérateur 1} & = (pa_1 \in \{0, \dots, pam\}) \\ \vdots & \\ \text{Points d'action restants Opérateur } n & = (pa_n \in \{0, \dots, pam\}) \\ \text{Position Ennemi 1} & = (r_1 \in \{0, \dots, tr - 1\} \text{ s'il est en vie, sinon } r_1 = -1) \\ \vdots & \\ \text{Position Ennemi } m & = (r_m \in \{0, \dots, tr - 1\} \text{ s'il est en vie, sinon } r_m = -1) \\ \text{Indicateur Otage récupéré } r & \in \{0, 1\} \\ \text{Niveau Menace } me & \in \{2, \dots, 7\} \end{array} \right.$$

Le nombre d'états possibles est donc :

$$(w.h)^n \times (pam + 1)^n \times (tr + 1)^m \times 2 \times 6 = 12(w.h)^n \times (pam + 1)^n \times (tr + 1)^m \quad (7)$$

Dans la version actuelle du jeu, nous avons  $w = 7$ ,  $h = 10$ ,  $n = 1$ ,  $pam = 2$ ,  $tr = 16$ ,  $m = 2$ , et on obtient un total de 728 280 états possibles. Or, on ne possède qu'un seul opérateur, les ennemis se déplacent selon une même routine prédéfinie et ils effectuent tous les mêmes coups, donc le nombre de combinaisons de positions ennemies est égal à la taille de la routine. Aussi, la menace n'augmente que s'il y a un ennemi tué par un tir, donc dans le pire des cas on peut avoir un niveau de menace  $\leq 3$ . Le nombre d'états accessibles est donc

$$\sum_{k=0}^m 70 \times 17^k \times 2 \times 3 \quad (8)$$

Ce qui nous donne

$$70 \times 17^2 \times 2 \times 3 + 70 \times 17^1 \times 2 \times 3 + 70 \times 17^0 \times 2 \times 3 = 128'940 \quad (9)$$

Cependant, il existe des états qui sont techniquement possibles mais qui ne sont pas valides. On peut donc procéder à des optimisations pour réduire le nombre d'états valides. Il faut tout de même prendre en considération le fait que ces restrictions sont valides pour cette version du jeu et qu'elles ne le seraient peut-être plus par la suite et devront donc être supprimées.

Premièrement, on ne peut pas avoir un ennemi et un opérateur sur la même case. Effectivement, si un opérateur et un ennemi se retrouvaient sur la même case, l'ennemi aurait éliminé l'opérateur et la partie serait finie. Par exemple, si la case 3 de la routine ennemie correspond à la case 16 du plateau, alors un état invalide peut être

$$s = (16, 2, [3, -1], 0, 3) \quad (10)$$

Deuxièmement, si tous les ennemis sont morts alors la menace vaut obligatoirement 2. Une des règles du jeu spécifiait que lorsque le dernier ennemi en vie est éliminé, le niveau de menace revient au minimum, il s'agit donc ici d'une application de la règle plus qu'une optimisation mais qui élimine tout de même des états générés par le programme. Un état invalide serait donc par exemple

$$s = (12, 2, [-1, -1], 0, 3) \quad (11)$$

Troisièmement, si aucun ennemi n'est mort alors la menace ne peut pas avoir augmenté. Dans notre version du jeu, le seul moyen d'augmenter la menace est d'effectuer et réussir un tir sur un ennemi, ce qui impliquerait que l'ennemi soit mort, donc tous les états où tous les ennemis sont vivants mais avec une menace différente de 2 ne sont donc pas valides, par exemple

$$s = (12, 2, [1, 5], 0, 5) \quad (12)$$

Dernièrement, le niveau de menace ne peut pas dépasser le niveau de menace minimum + le nombre d'ennemis morts. Comme expliqué auparavant, le seul moyen d'augmenter la menace est de tirer, donc le pire cas serait le cas où l'opérateur élimine tous les ennemis en effectuant des tirs et donc, sans prendre en compte la remise à niveau de la menace lors de l'élimination de tous les ennemis, la menace vaudrait

$$\text{menace} = \text{menace minimum} + \text{nombre d'ennemis morts} \quad (13)$$

Cette valeur représente donc la borne supérieure du niveau de menace pour la version actuelle. Tout état ayant un niveau de menace supérieur serait invalide, par exemple

$$s = (12, 2, [1, -1], 0, 5) \quad (14)$$

Avec toutes ces optimisations, nous obtenons 23'349 états accessibles. Nous pouvons donc constater qu'après ce filtrage nous avons divisé le nombre d'états par environ 5,52.

Définissons ensuite une action. Une action est définie comme un coup, ciblant une direction et étant effectué par un opérateur. Soit  $c$  un coup, il est défini par

$$c = \left\{ \begin{array}{l} \text{Déplacement vers l'une des 4 directions possibles,} \\ \text{ou} \\ \text{Tir vers l'une des 4 directions possibles,} \\ \text{ou} \\ \text{Élimination silencieuse vers l'une des 4 directions possibles,} \\ \text{ou} \\ \text{Se calmer,} \\ \text{ou} \\ \text{Ne rien faire.} \end{array} \right. \quad (15)$$

Chaque opérateur dispose d'un nombre limité de points d'action, et peut ici effectuer des coups tant qu'il lui en reste. De plus, un opérateur doit compléter tous ses coups avant qu'un autre opérateur ne puisse agir, ce qui signifie qu'il est impossible de mélanger les coups entre les opérateurs. Si un des coups est "Ne rien faire", il est forcément le dernier coup du tour du joueur. Le nombre de coups possible est donc

$$4 + 4 + 4 + 1 + 1 = 14 \quad (16)$$

Les coups ont chacun un coût qui peut être différent, dans notre jeu le coût du déplacement, du tir et de l'élimination est de 1 PA, celui de se calmer est de 2 PA et celui de ne rien faire est de  $pa_i$  PA, i.e. le coup consomme tous les PA restants de l'opérateur.

Avec ces paramètres et  $pam = 2$ , on dénombre

- 1 combinaison de fin de tour immédiat
- 12 choix de coût 1 PA puis,
  - soit un coup de coût 1 PA, donc 12 choix
- 1 combinaison de fin de tour
- 1 combinaison de se calmer

. Le nombre de combinaisons de coups possibles est donc au total

$$1 + 12(12 + 1) + 1 = 1 + 156 + 1 = 158 \quad (17)$$

Toutefois, un coup en lui-même n'est pas suffisant pour le déroulement du jeu, il faut également une direction pour la plupart des coups. On définit donc une direction par l'orientation vers laquelle un opérateur peut effectuer un coup. Les directions valides sont **HAUT**, **BAS**, **GAUCHE**, **DROITE** ou **AUCUNE**. La direction **AUCUNE** est utilisée par certains coups qui ne nécessitent pas de mouvement ou de visée directionnelle, comme se calmer ou ne rien faire.

Maintenant que les états et coups sont définis voyons comment on implémente un MDP. Pour résoudre les équations de Bellman, il est nécessaire de connaître 4 éléments : les états  $S$ , la distribution  $P(s'|s, a)$ , les actions  $A(s)$ , et la fonction de récompense  $R(s, a, s')$ . Ici, MDP est une interface qui est définie par quatre méthodes. La première, **getEtats**, génère une liste de tous les états valides du MDP et nous permet d'obtenir  $S$ . Son implémentation dans le jeu est détaillée dans l'algorithme 3.

---

**Algorithm 3** Génération des États

---

```

1:  $S \leftarrow \emptyset$ 
2: for  $HR \in \{0, 1\}$  do
3:   for  $PO \in \{0, 1, \dots, (w \times h) - 1\}^n$  do
4:     for  $PA \in \{0, 1, \dots, pam\}^n$  do
5:       for  $PE \in \{0, 1, 2, \dots, tr\}^m$  do
6:         for  $me \in \{2, 3, \dots, 7\}$  do
7:            $e \leftarrow \text{État}(HR, PO, PA, PE, me)$ 
8:           if valide( $e$ ) then
9:              $S \leftarrow S \cup \{e\}$ 
10:          end if
11:        end for
12:      end for
13:    end for
14:  end for
15: end for
16: return  $S$ 

```

---

La seconde, **getCoups**, génère la liste de tous les coups valides du MDP quand l'agent se situe sur l'état  $s$  et nous permet d'obtenir  $A(s)$ . Son implémentation dans le jeu est détaillée dans l'algorithme 4.

---

**Algorithm 4** Récupération des coups valides

---

```

1:  $C \leftarrow \emptyset$ 
2: for  $e \in S$  do
3:   for  $c \in \{\text{Déplacement}, \text{Tir}, \text{ÉliminationSilencieuse}\}$  do
4:     for  $d \in \{\text{HAUT}, \text{BAS}, \text{GAUCHE}, \text{DROITE}\}$  do
5:       if valide( $c, d, e$ ) then
6:          $C \leftarrow C \cup \{(e, (c, d))\}$ 
7:       end if
8:     end for
9:   end for
10:  if valide( $\text{FinDeTour}, \text{AUCUN}, e$ ) then
11:     $C \leftarrow C \cup \{(e, (\text{FinDeTour}, \text{AUCUN}))\}$ 
12:  end if
13:  if valide( $\text{SeCalmer}, \text{AUCUN}, e$ ) then
14:     $C \leftarrow C \cup \{(e, (\text{SeCalmer}, \text{AUCUN}))\}$ 
15:  end if
16: end for
17: return  $C$ 

```

---

La troisième, **transition**, génère une distribution de tous les états accessibles du MDP à partir de  $s$  en effectuant le coup  $c$  et nous permet d'obtenir  $P(s'|s, c)$ . Son implémentation dans le jeu est détaillée dans l'algorithme 5.

---

**Algorithm 5** Calcul de la distribution des états

---

```
1:  $T \leftarrow \emptyset$  ▷ Liste vide de Paires ((État de départ, Coup), (État d'arrivée, Proba))
2:  $s_1 \leftarrow \text{ÉtatAprèsCoup}(s, c, \text{Succès})$  ▷ Succès : Proba succès à 100%
3:  $s_2 \leftarrow \text{ÉtatAprèsCoup}(s, c, \text{Échec})$  ▷ Echec : Proba Echec à 100%
4:  $T \leftarrow T \cup \{(s, c) \rightarrow (s_1, \text{ProbaSuccès}(c))\}$ 
5:  $T \leftarrow T \cup \{(s, c) \rightarrow (s_2, 1 - \text{ProbaSuccès}(c))\}$ 
6: for  $t \in T$  do
7:    $s \leftarrow \text{etatDepart}(t)$ 
8:   if  $\text{estFinDeTour}(s)$  then ▷ Tour des ennemis
9:      $T \leftarrow T \setminus \{t\}$ 
10:    for  $C \in \{\text{Déplacement}, \text{Tir}\}^{me_s}$  do ▷  $me_s$  : Niveau de menace de s
11:      for  $e \in \text{ennemis}(s)$  do
12:         $s' \leftarrow s$ 
13:         $p \leftarrow 1$ 
14:        for  $c \in C$  do
15:           $p \leftarrow p \times \text{ProbaSuccès}(c)$ 
16:           $s' \leftarrow \text{ÉtatAprèsCoup}(s', c, \text{Succès})$ 
17:        end for
18:         $T \leftarrow T \cup \{((\text{État de départ}, \text{Coup}) \rightarrow (s', p))\}$ 
19:      end for
20:    end for
21:  end if
22: end for
23: return  $T$ 
```

---

La dernière, **recompense**, calcule la récompense d'une action  $a$  effectuée depuis un état  $s$  vers un état  $s'$ , et nous permet d'obtenir  $R(s, a, s')$ . Pour cela, on définit des récompenses spécifiques afin de modifier le comportement de l'agent appliquant la politique obtenue. Ces récompenses sont :

- *valeurReussite*, c'est-à-dire la récompense liée à la réussite de la mission
- *valeurObjectif*, c'est-à-dire la récompense liée à la récupération du ou des objectifs, ici l'otage
- *valeurEchec*, c'est-à-dire la pénalité liée à l'échec de la mission
- *valeurTuerEnnemi*, c'est-à-dire la récompense liée à l'élimination d'un terroriste, peut importe par quelle méthode
- *valeurDeltaMenace*, c'est-à-dire la récompense ou pénalité liée à la diminution ou augmentation du niveau de menace
- *valeurDeplacement*, c'est-à-dire la pénalité lié au déplacement de l'opérateur

Ainsi, on peut choisir de favoriser l'élimination des terroristes en augmentant *valeurTuerEnnemi*, ou lui faire finir la mission au plus vite possible en diminuant *valeurDeplacement*.

Nous pouvons maintenant regarder comment l'IA fonctionne. En amont des parties, l'agent génère la politique optimale selon ITÉRATION-VALEUR. Pour chaque état possible, l'agent garde la paire (coup, direction) associée à la meilleure Q-Valeur, ce qui génère une politique optimale. Dans le jeu, après chaque coup, l'agent observe l'état actuel et propose au joueur le coup optimal selon la politique conçue. Cette approche garantit que les conseils fournis tiennent compte à la fois de la récompense immédiate et des conséquences sur les tours futurs, offrant ainsi au joueur un appui s'il est perdu sans pour autant lui imposer ses choix.

Dans la déclinaison  $b$  de la version finale, l'IA se base sur RTDP car le nombre d'états était trop volumineux. En effet, puisque les terroristes peuvent réapparaître, l'écart entre deux terroristes n'est plus forcément constant, et on ne peut plus considérer que le déplacement des terroristes se fait comme si un seul terroriste n'existait. Un rapide calcul nous donne alors comme borne supérieure du nombre d'états

$$12(w.h)^n \times (pam + 1)^n \times (tr + 1)^m = 12(7 \times 10)^1 \times (2 + 1)^1 \times (16 + 1)^5 = 3'578'039'640 \quad (18)$$

Avec les ressources que nous avons, il n'est pas raisonnable d'appliquer IV sur un nombre d'état aussi important, d'une part par le manque de mémoire vive pour stocker tous les états, et d'autre part par le manque de puissance de calcul pour exécuter l'algorithme en un temps raisonnable.

### 3.3 IA d'assistance

Une partie importante du sujet consiste également à créer un agent capable d'avertir le joueur en cas de danger. Au stade actuel du jeu, le danger ne vient que du fait de mourir en étant dans la ligne de mire de l'ennemi au moment de son tir ou en étant sur la case sur laquelle il se déplace. Pour cela, nous utilisons une exploration probabiliste des états, c'est-à-dire que nous explorons les états proches en choisissant la première action sur l'état initial puis successivement des coups et directions aléatoires et nous comptons le nombre d'états échecs que nous rencontrons. Si le taux d'états échec dépasse un certain seuil alors l'agent affiche le coup initial avec la direction associée qu'il estime dangereuse. Plus formellement, l'agent exécute l'algorithme suivant

---

**Algorithm 6** Probabilité d'échec en effectuant le coup  $a_0$

---

```
1: function PROBAECHEC( $s_0, a_0, nbIters, nbMax$ )
2:    $nbEchec \leftarrow 0$ 
3:   for  $i \leftarrow 1, nbIters$  do
4:      $etat \leftarrow T(s_0, a_0)$ 
5:      $nbCoups \leftarrow 1$ 
6:     while non estTerminal( $etat$ ) ou  $nbCoups \leq nbMax$  do
7:        $action \leftarrow \text{tirerAleatoirement}(\text{actionsValides}(etat))$ 
8:        $etat \leftarrow T(etat, action)$ 
9:        $nbCoups \leftarrow nbCoups + 1$ 
10:    end while
11:    if estEchec( $etat$ ) then
12:       $nbEchec \leftarrow nbEchec + 1$ 
13:    end if
14:  end for
15: end function
16: return  $nbEchec/nbIters$ 
```

---

Bien que simple, cet algorithme nous permet de régler le niveau de sensibilité de l'agent, notamment en diminuant la valeur de  $nbMax$  pour qu'il ne voie que le danger à court terme ou à l'inverse l'augmenter pour qu'il voie plus loin dans le futur, et nous permet de choisir entre vitesse d'exécution et confiance dans le résultat, notamment en diminuant  $nbIters$  pour aller plus vite ou l'augmenter pour être sûr du danger. Cet algorithme est aussi **parallélisable**, ce qui augmente la confiance dans le résultat sans augmenter significativement le temps de calcul. Dans le jeu, nous utilisons l'algorithme avec  $nbIters = 10'000$ ,  $nbMax = 10$  et un seuil à 60% afin d'avoir un avertissement des dangers assez proches et une confiance suffisante. Un exemple est illustré sur l'image 8



## 4 Expérimentation

### 4.1 Évaluation du taux de succès de l'IA

Pour évaluer la capacité de l'agent à choisir le coup optimal, on simule des parties où l'opérateur n'exécute que les coups recommandés par l'agent. Pour les expériences ci-dessous, les paramètres sont les suivants

- $\gamma = 0.9935$
- $\epsilon = 0.001$
- *valeurReussite* = 5'000'000
- *valeurObjectif* = 5'000'000
- *valeurEchec* = -10'000'000
- *valeurTuerEnnemi* = 300'000
- *valeurDeltaMenace* = 600'000
- *valeurDeplacement* = -400'000

Les valeurs des récompenses sont artificiellement augmentées pour que les utilités puissent se propager correctement pendant l'exécution d'ITÉRATION-VALEUR. On compte aussi le nombre de tours effectués avant de finir la mission, qu'elle soit gagnée ou perdue.

Pour la première expérience, on utilise les probabilités de succès suivantes :

- *probaSuccesDeplacement* = 95%
- *probaSuccesTir* = 85%
- *probaSuccesEliminationSilencieuse* = 80%
- *probaSuccesCalmer* = 30%

On effectue cette expérience sur 10'000'000 parties, et on observe que l'agent en gagne 9'366'892, qui résulte en un taux de victoire de 93,366% en finissant en moyenne en 12,28 tours.

Pour vérifier que le taux de succès de l'agent est affecté par les pourcentages de succès des coups, nous avons pour cette expérience retiré 10% de chance de succès des coups. On obtient donc les probabilités de succès

- *probaSuccesDeplacement* = 85%
- *probaSuccesTir* = 75%
- *probaSuccesEliminationSilencieuse* = 70%
- *probaSuccesCalmer* = 20%

On effectue cette expérience encore une fois sur 10'000'000 parties, et on observe que l'agent en gagne 9'049'244, ce qui fait un taux de succès de 90,49% en finissant en moyenne en 13,86 tours. Les performances de l'agent sont donc bien affectées négativement quand on diminue les chances de succès de ses coups.

On voit donc que la politique de l'agent n'est pas déterministe, l'agent a tout de même des chances de perdre même en choisissant les coups optimaux pour chaque état et son taux de victoire est affecté par les taux de succès des coups.

### 4.2 Performance de l'algorithme Itération-Valeur

En plus de mesurer l'efficacité de la politique générée par l'agent, il est également intéressant d'évaluer les performances techniques de l'algorithme, notamment en fonction de la taille de l'espace d'états. Pour cela, nous avons fait varier le nombre d'ennemis présents dans l'environnement, ce qui a pour effet d'accroître le nombre d'états et donc le temps requis pour arriver vers la politique optimale. Cette expérience n'a été réalisée qu'à titre d'observation, dans des versions non officielles du jeu où le nombre d'ennemis dépasse les 2 prévus dans la version jouable. Nous conservons cependant les mêmes paramètres que pour les tests précédents. Les résultats suivants ont été obtenus en utilisant un processeur Ryzen 7 6800H et en faisant une moyenne des temps sur 6 exécutions :

Nombre d'ennemis	Nombre d'états générés	Nombre d'itérations	Temps de convergence
2	23 349	101	13 secondes
3	73 765	112	59 secondes
4	196 757	127	3 minutes 25 secondes
5	265 029	141	7 minutes

TABLE 1 – Tableau des performances d'IV en fonction du nombre d'ennemis

On observe donc que le nombre d'états croît de manière non linéaire avec le nombre d'ennemis. En effet, lorsque l'on passe de 2 à 3 ennemis, le nombre d'états générés est multiplié par  $\approx 3.1$ , tandis que le temps de convergence est multiplié par  $\approx 3.7$ . Cela montre qu'ajouter un seul ennemi a un impact significatif. En ajoutant un quatrième ennemi, le nombre d'états subit une augmentation par  $\approx 2.7$  par rapport à la version précédente. Le temps de convergence augmente encore fortement avec un facteur de multiplication de  $\approx 3.5$ . Enfin, en poussant l'expérience jusqu'à 5 ennemis, le nombre d'états est multiplié de manière plus modérée par un facteur de  $\approx 1.3$  par rapport à la version avec 4 ennemis, en revanche le temps d'exécution, lui, double pratiquement. On remarque donc qu'à partir d'un certain seuil, la complexité, que ce soit en termes de nombre d'états ou de temps de calcul, n'augmente plus aussi sévèrement.

## 5 Conclusion

## 6 Annexes

### Table des figures

1	Première version du jeu. L'opérateur est en bleu, les terroristes en rouge, l'otage en vert et les cases valides pour le coup sélectionné sont contournés en jaune. . . . .	1
2	Deuxième version du jeu. Les obstacles sont en gris. Les nombres inscrits dans les cases correspondent à leurs coordonnées $(x, y)$ , et sont à des fins purement de débogage. . . . .	2
3	Troisième version du jeu. Les coups sont, de gauche à droite, passer son tour, tirer et se déplacer . . . . .	3
4	Troisième version du jeu. Les coups sont, de gauche à droite, passer son tour, tirer, se déplacer, faire une élimination silencieuse et se calmer. Les murs sont les lignes noires. Les nombres dans les cases correspondent à l'identifiant de la case et sont à des fins purement de débogage. . . . .	4
5	Déclinaison <i>a</i> de la version finale du jeu. L'opérateur est le carré vert avec "Andrew" inscrit dessus, les terroristes sont les carrés gris aux contours noirs, l'otage est le carré rouge aux contours noirs et les murs sont les lignes brunes. Les conseils de l'IA sont désactivés par défaut. . . . .	4
6	Déclinaison <i>b</i> de la version finale du jeu. Les points de réapparition des terroristes sont en orange. . . . .	5
7	Diagramme de classes simplifié de la version finale du jeu . . . . .	5
8	Exemple d'affichage de l'IA d'assistance. Ici, se déplacer vers la gauche résulte en une fin de mission dans 86,6% des cas en moins de 10 coups, et se déplacer vers la droite dans 77,9% des cas. . . . .	6

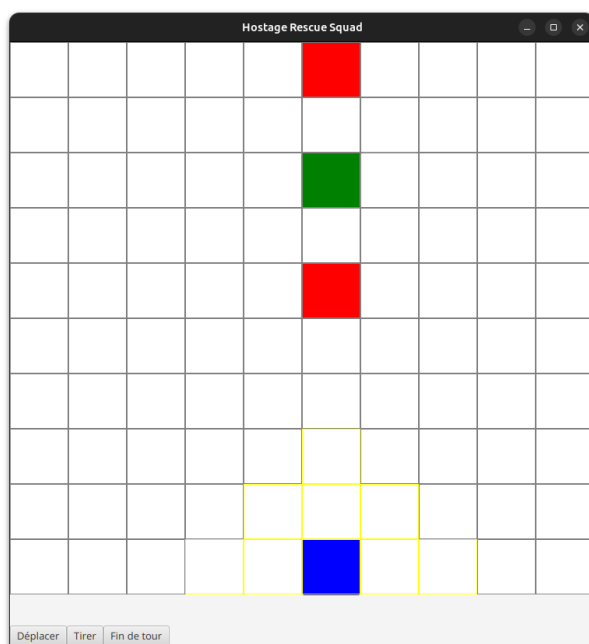


FIGURE 1 – Première version du jeu. L'opérateur est en bleu, les terroristes en rouge, l'otage en vert et les cases valides pour le coup sélectionné sont contournés en jaune.

Hostage Rescue Squad						
Jeu Aide						
0,0	1,0	2,0	3,0	4,0	5,0	6,0
0,1	1,1	2,1	3,1	4,1	5,1	6,1
0,2	1,2	2,2	3,2	4,2	5,2	6,2
0,3	1,3	2,3	3,3	4,3	5,3	6,3
0,4	1,4	2,4	3,4	4,4	5,4	6,4
0,5	1,5	2,5	3,5	4,5	5,5	6,5
0,6	1,6	2,6	3,6	4,6	5,6	6,6
0,7	1,7	2,7	3,7	4,7	5,7	6,7
0,8	1,8	2,8	3,8	4,8	5,8	6,8
0,9	1,9	2,9	3,9	4,9	5,9	6,9
Déplacer Tirer Fin de tour						

FIGURE 2 – Deuxième version du jeu. Les obstacles sont en gris. Les nombres inscrits dans les cases correspondent à leurs coordonnées  $(x, y)$ , et sont à des fins purement de débogage.

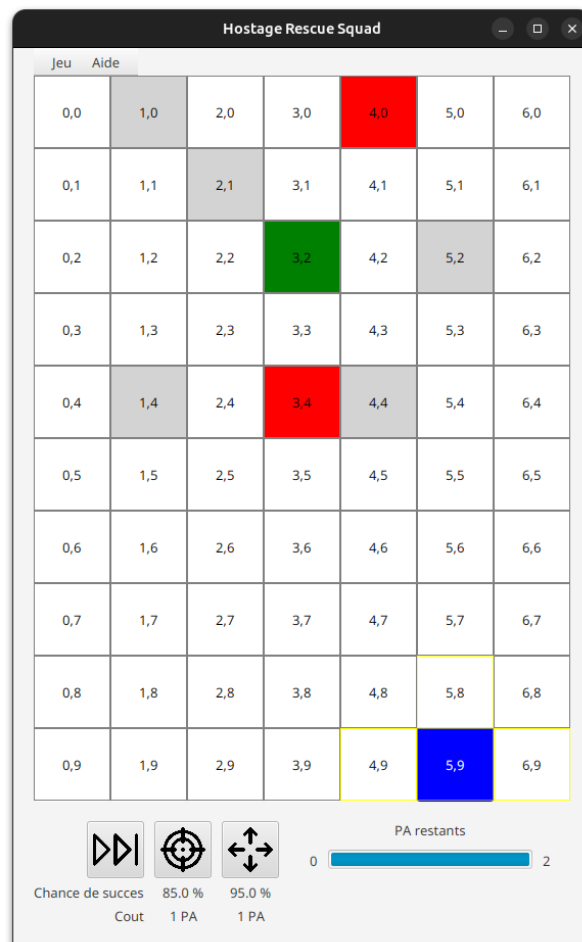


FIGURE 3 – Troisième version du jeu. Les coups sont, de gauche à droite, passer son tour, tirer et se déplacer



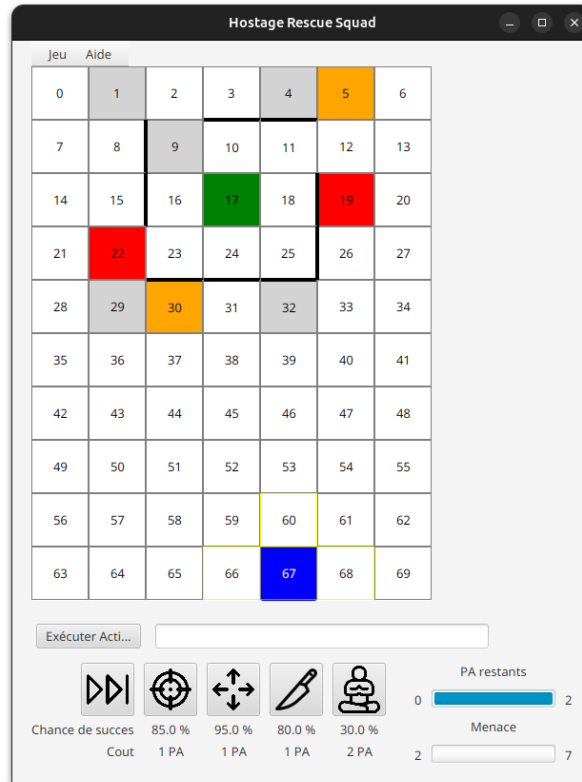


FIGURE 6 – Déclinaison *b* de la version finale du jeu. Les points de réapparition des terroristes sont en orange.

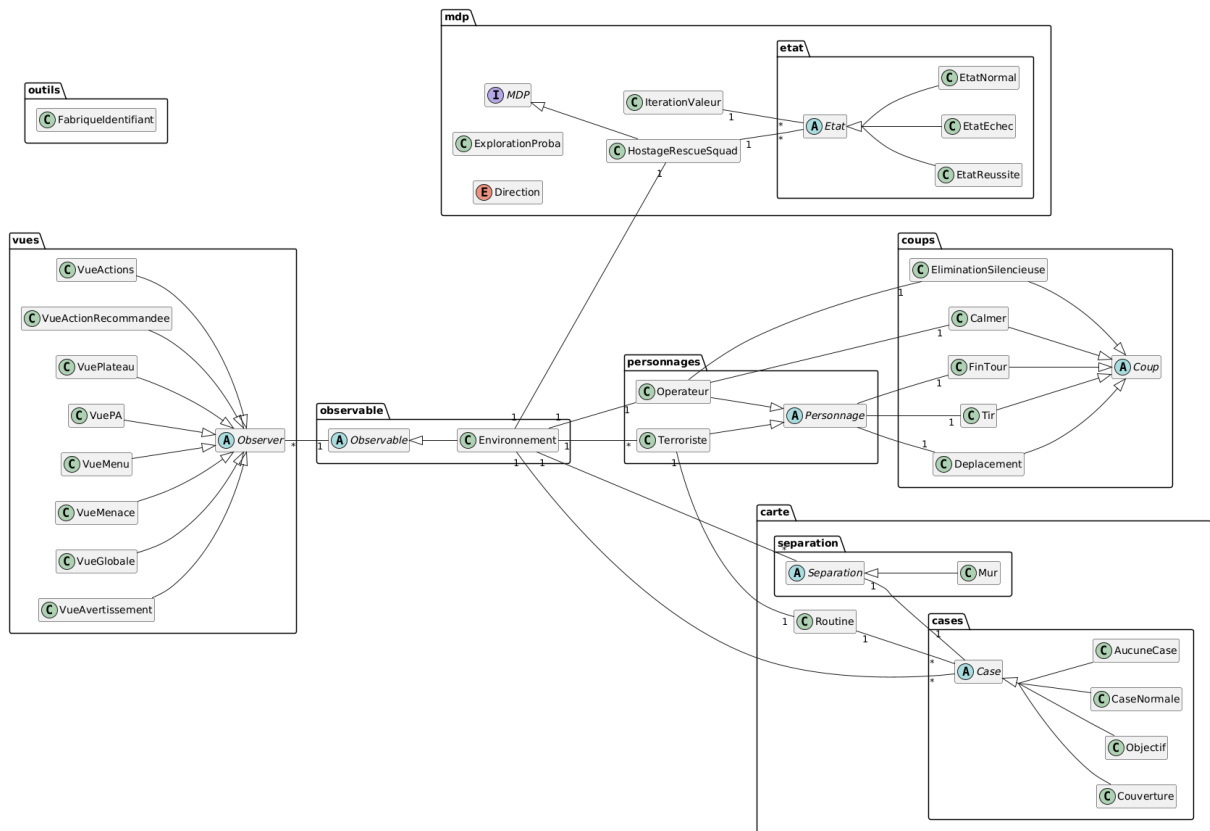


FIGURE 7 – Diagramme de classes simplifié de la version finale du jeu



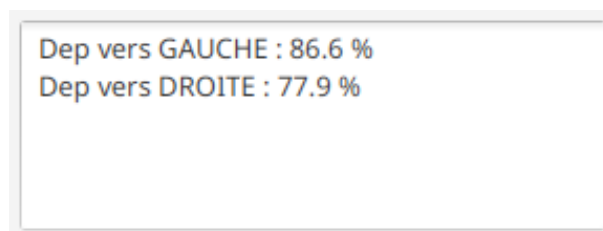


FIGURE 8 – Exemple d’affichage de l’IA d’assistance. Ici, se déplacer vers la gauche résulte en une fin de mission dans 86,6% des cas en moins de 10 coups, et se déplacer vers la droite dans 77,9% des cas.

## 7 Bibliographie

### Références

- [1] Olivier Buffet and Douglas Aberdeen. Planification robuste avec (1) rtdp. In *CAP*, pages 127–142, 2005.
- [2] [https://fr.wikipedia.org/wiki/Jeu\\_de\\_guerre](https://fr.wikipedia.org/wiki/Jeu_de_guerre) (consulté le 26/04/2025).
- [3] <https://www.cpubenchmark.net/cpu.php?cpu=AMD+EPYC+9655Pid=6354> (consulté le 27/04/2025).
- [4] [https://www.cpubenchmark.net/high\\_end\\_cpus.html](https://www.cpubenchmark.net/high_end_cpus.html) (consulté le 27/04/2025).
- [5] <https://www.llnl.gov/article/52061/lawrence-livermore-national-laboratorys-el-capitan-verified-worlds-fastest-supercomputer> (consulté le 27/04/2025).
- [6] <https://www.top500.org/system/180307/> (consulté le 27/04/2025).
- [7] Stuart Russell, Peter Norvig, Fabrice Popineau, Laurent Miclet, and Claire Cadet. *Intelligence artificielle : une approche moderne (4 édition)*. Pearson France, November 2021.

## 8 Glossaire

LORIA : Laboratoire Lorrain de Recherche en Informatique et ses Applications  
CREM : Centre de REcherche en Médiation  
MDP : Markov Decision Process, ou processus de décision Markovien en français  
IV : Iteration-Valeur  
RTDP : Real Time Dynamic Programming  
PA : Point d'Action  
Bug : erreur ou problème informatique  
Débugage : correction de bugs IA : Intelligence Artificielle

## Annexes



### Déclaration sur l'honneur contre le plagiat

Je soussigné(e),

Nom, Prénom, Fernandes dos Santos Emanuel

Régulièrement inscrit à l'Université de Lorraine

N° de carte d'étudiant : 32105861

Année universitaire : 2024-2025

Niveau d'études : L ou M

Parcours : Informatique

N° UE : 8JU31N01

Certifie qu'il s'agit d'un travail original et que toutes les sources utilisées ont été indiquées dans leur totalité. Je certifie, de surcroît, que je n'ai ni recopié ni utilisé des idées ou des formulations tirées d'un ouvrage, article ou mémoire, en version imprimée ou électronique, sans mentionner précisément leur origine et que les citations intégrales sont signalées entre guillemets.

Conformément à la loi, le non-respect de ces dispositions me rend passible de poursuites devant la commission disciplinaire et les tribunaux de la République Française.

Fait à Vandoeuvre-lès-Nancy, le 1/05/2025

Signature :

## Déclaration sur l'honneur contre le plagiat

Je soussigné(e),

Nom, Prénom,

Lagarde Tristan

Régulièrement inscrit à l'Université de Lorraine

N° de carte d'étudiant : 32107914

Année universitaire : 2024-2025

Niveau d'études : L ou M

Parcours : Informatique

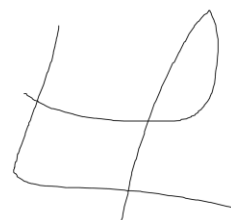
N° UE : 8JU31N01

Certifie qu'il s'agit d'un travail original et que toutes les sources utilisées ont été indiquées dans leur totalité. Je certifie, de surcroît, que je n'ai ni recopié ni utilisé des idées ou des formulations tirées d'un ouvrage, article ou mémoire, en version imprimée ou électronique, sans mentionner précisément leur origine et que les citations intégrales sont signalées entre guillemets.

Conformément à la loi, le non-respect de ces dispositions me rend passible de poursuites devant la commission disciplinaire et les tribunaux de la République Française.

Fait à TOUL , le 01/05/2025

Signature :





## Déclaration sur l'honneur contre le plagiat

Je soussigné(e),

Aiachi Morade

Régulièrement inscrit à l'Université de Lorraine

N° de carte d'étudiant : 30906125

Année universitaire : 2024-2025

Niveau d'études : M

Parcours : Informatique

N° UE : 8JU31N01

Certifie qu'il s'agit d'un travail original et que toutes les sources utilisées ont été indiquées dans leur totalité. Je certifie, de surcroît, que je n'ai ni recopié ni utilisé des idées ou des formulations tirées d'un ouvrage, article ou mémoire, en version imprimée ou électronique, sans mentionner précisément leur origine et que les citations intégrales sont signalées entre guillemets.

Conformément à la loi, le non-respect de ces dispositions me rend passible de poursuites devant la commission disciplinaire et les tribunaux de la République Française.

Fait à Liverdun , le 01/05/2025

Signature :



## 9 Quatrième de couverture