

---

**RAPPORT : METHODES ET OUTILS DE L'IA ET LA RO**  
*projet : multirobot wars*

---

BECIRSPAHIC LUCAS

# 1 Introduction

Dans le cadre de ce projet on cherche à créer une intelligence artificielle qui capture le plus de case possibles, pour ce faire on cherche à définir plusieurs comportements.

## 2 Algorithmes génétiques

Pour définir les paramatres exactes associées aux senseur(c'est à dire les poids de notre neurones) on utilise des algorithmes génétiques.

On définit une fonction *fitness*, qui permet de voir à quel points l'objectif de notre robot est respecté pour des paramètres données. Ils faut donc la maximiser.

La première aproche, la plus simple consiste à tirer aléatoirement les parametres et garder le champion : selui avec la meilleur fitness. Néanmoins cette methode prend beaucoup de temps à converger et de ce fait donne souvent un optimum local

Une approche plus intelligente est de faire une mutation gaussienne sur notre champion afin de faire varier les paramtres tout en gardant les propriétés souhaitables de notre champion. On introduit un  $\sigma$  pour faire varier l'amplitude de la mutation. Néanmoins avec un  $\sigma$  fixe, on s'approche de la valeur optimal mais trop brutalement, car on manque de finess. C'est pourquoi on fait bouger le  $\sigma$  de la manière suivante :

Si la fitness trouvée est meilleur que la précédante, on augmente  $\sigma$  pour qu'il soit plus grand.

Sinon on considère que la variation va trop loin et on réduit  $\sigma$

– **Eviteur d'obstacle**  $fitness = vt * (1 - vr) * MinSenseurValue$

– **Colle mur**  $fitness = vt * (1 - vr) * (1 - MinSenseurValue)$

– **Traqueur**  $fitness = vt * (1 - vr) * MinSenseurEnemis$

Dans cette section, nous allons aborder brièvement des stratégies simples, qui peuvent servir de base pour nos autres stratégies ou comme adversaire.

– **stratégie naïve** : Cette stratégie va ramasser les fioles les plus proches du joueur. Ces résultats sont mauvais à cause de l'heuristique simpliste qui considère que maximiser le nombre de fioles ramassées suffit pour gagner.

– **stratégie meilleur valeur proche** : Légère amélioration de l'algorithme naïf qui tient compte des valeurs associés aux fioles et de leurs distances. On prend la fiole qui maximise *valeur - distance*

– **stratégie regroupement** : On améliore notre stratégie en se basant sur l'heuristique suivante : il vaut mieux prendre une fiole qui est proche des autres qu'au milieu de nulle part. Pour ce faire, on introduit  $d$  : la distance d'une fiole  $f_1$  vis à vis des autres fioles :

$$d = \sum_{f_2 \in \text{fioles}} distance(f_1, f_2)$$

On prend la fiole qui maximise  $\beta * valeur - distance - \alpha * d$ . Avec  $\alpha$  un paramètre compris entre 0 et 1 pour réduire l'importance de  $d$  et  $\beta$  un paramètre compris entre 1 et 2 pour augmenter l'importance de la valeur associée aux fioles. J'ai introduit ces paramètres car  $d$  avait une valeur trop importante. A cause de cela, notre intelligence artificielle n'allait pas chercher des fioles intéressantes. Une autre solution aurait été de normaliser les valeurs de mes parametres afin qu'ils aient un poids équivalent.

### 3 La stratégie contre

Dans cette stratégie on suppose que l'on connaît la stratégie de l'adversaire. Le concept de l'algorithme est le suivant : on a un le chemin des fioles par lequel notre personne passe initialement et le chemin de l'adversaire. On cherche à améliorer notre score en allant prendre les fioles dans un ordre différent, de manière astucieuse pour améliorer notre score a partir de la connaissance du chemin adverse.

– *Quelques notations* : Soit un chemin, la liste ordonnée des fioles par lequel un personnage va passer, on le représente comme une liste de tuples (fiole, distance).

L'attribut distance représente la distance cumulée pour aller à cette fiole depuis mon état initial en passant par les états antérieurs de mon chemin.

Soit  $s1$ , le chemin de mon joueurs en se basant sur la stratégie qui prend la meilleur fiole possible dans une distance raisonnable(stratégie meilleur valeur proche).

Soit  $s2$ , le chemin de mon adversaire obtenu en appliquant sa stratégie.

Soit la fonction  $distance(chemin, fiole)$  la distance qu'il me faut pour atteindre la fiole en passant par le chemin

On définit un dictionnaire de gain nommé  $dicoGain$  pour chaque fioles de la manière suivante :

– Si  $distance(s1, fiole) > distance(s2, fiole)$  alors  $dicoGain[fiole] = valeur\ de\ la\ fiole$

– Sinon  $dicoGain[fiole] = -1 * valeur\ de\ la\ fiole$

Une fois ces choses définis, on arrive au coeur de l'algorithme, on cherche à maximiser :

$$gain = \sum_{f \in fioles} dicoGain[f]$$

Pour ce faire, on dispose d'une opération de permutation qui change l'ordre de ramassage des fioles de  $s1$  en fonction de  $s2$ . On définit cette opération de la manière suivante :

**Algorithme :** Permutation(chemin du joueur, chemin de l'adversaire, fiole, case)

**Entrées :**  $s1$  : chemin,  $f$  : fiole,  $case$  : case

$old \leftarrow$  chemin du point de départ jusqu'à la fiole qui précède la case

$old \leftarrow$  on ajoute (case, distance(fiole precedente, case)) a la suite de old

$new \leftarrow$  on recalcule le chemin et les distances pour les fioles restantes

$chemin \leftarrow$  on concatene  $old$  et  $new$

**retourner**  $chemin$

Puis, on souhaite trouver la meilleure permutation parmi celles possibles de la manière suivante :

**Algorithme :** MeilleurPermutation(chemin du joueur, chemin de l'adversaire, fioles)

**Entrées :**  $s1$  : chemin,  $s2$  : chemin de l'adversaire,  $fioles$  : liste de fioles

**pour**  $f \in fioles$  **faire**

**pour**  $case \in s1$  **faire**

$new = Permutation(s1, s2, f, case)$

        On calcule le dicoGain pour le nouveau chemin

$new\ gain \leftarrow$  la nouvelle valeur du gain

**si**  $newgain > gain$  **alors**

$chemin \leftarrow new$

$gain \leftarrow new\ gain$

**retourner** ( $chemin, gain$ )

On réitère cette opération jusqu'à convergence (obtenir le même chemin 2 fois de suite). Une fois cette étape résolue on a le chemin final de notre algorithme et il nous reste plus qu'à le suivre pour savoir quel coup jouer.

– *Quelques remarques :* Etant donné que l'on ne peut pas connaître avec certitude la stratégie de l'adversaire, il faut avoir une fonction de prédiction qui étant donné la mémoire des coups précédents nous dit quel est la stratégie de l'adversaire.

Cette fonction est appelée à chaque étape de jeu et détermine la stratégie de l'adversaire. Si on ne peut trouver sa stratégie on applique la stratégie meilleur valeur proche.

La fonction de prédiction se base sur les stratégies de bases que j'ai définies. Donc si c'est une stratégie que je n'ai pas implémentée, mon algorithme ne pourra pas la contrer.

**Algorithme :** Prediction(Etats precedants, strategie)

**Entrées :**  $prec$  : l'ensemble des  $n$  coups précédents,  $strategiePrec$  : La stratégie précédente

**si**  $strategiePrec(prec[n-1]) = prec[n]$  **alors**

**retourner** ( $Vrai, strategiePrec$ )

**pour**  $strat \in strategies$  **faire**

**si**  $strategiePrec[n-1] = la\ position\ de\ l'adversaire\ à\ l'état\ précédent$  **alors**

**retourner** ( $Vrai, strat$ )

**retourner** ( $Faux, \emptyset$ )

– *Conclusion :* Il est difficile de d'évaluer et de comparer nos stratégies à cause de la nature aléatoire du jeu. Néanmoins, on pourrait faire des statistiques pour voir en moyenne comment ce débrouille les stratégies. Je n'ai pas effectué cette partie par manque de temps.