

Rapport IA

Choix de l'IA	1
Listing prolog	1
Heuristique	3
Tests réalisés	5

1. Choix de l'IA

Le but du projet était de réaliser une intelligence artificielle pour jouer au jeu Quantik. Nous avons réalisé deux IA, une qui respecte les règles du jeu mais qui réalise des coups aléatoires. Et, une seconde qui réalise un coup en fonction d'un poids qui est calculé sur toutes les cases.

Pour les deux IA, c'est Java qui conserve l'état de la grille (cela comprend les pions joués par notre IA et par le joueur adverse) et les pions disponibles pour jouer. Lorsque Java appelle une fonction prolog, il transmettra en paramètre ces informations.

Premièrement l'IA aléatoire, est une IA qui respecte les règles du jeu mais qui va placer son coup aléatoirement. L'IA va déterminer qu'elles sont les cases qui sont vides sur la grille de jeu. Parmi ces cases, l'IA va en tirer une aléatoirement. Ensuite, nous regardons si la case et le pion tiré respectent les règles du jeu. C'est-à-dire que le pion choisi ne peut déjà être dans la ligne, la colonne ou le carré de la case tirée. Si, la case et le pion respectent ces règles, le coup joué est retourné. Sinon, l'IA réalise des tirages jusqu'à obtenir un coup qui respecte les règles du jeu. Mais, nous utilisons jamais cette IA, nous privilégions l'heuristique.

Secondement l'IA qui calcule les poids, est une IA qui va calculer le poids des cases en fonction des pions déjà sur la grille. Nous expliquons cette stratégie en détail dans la partie 3 heuristique.

2. Listing prolog

Nous allons présenter les fonctions prolog réalisés.

- random(L,U,R) : Retourne un nombre aléatoire R compris entre L et U.
 - jouerCoupRandom(Ligne, Colonne, Pion) : Joue un coup aléatoire sans se soucier si le pion est disponible, si la case est vide et si cela respecte les règles du jeu.
 - jouerCoupRandomSurCaseVide(Grid,Ligne,Colonne,Pion) : Joue un coup aléatoire sur une case vide sans savoir si le pion est disponible ou si cela respecte les règles du jeu.
 - jouerCoupRandomSurCaseVideAvecPionRestant(Grid,Ligne,Colonne,Pion,PionRestant) : joue un coup aléatoire sur une case vide parmi les pions restant du joueur, mais ne respecte pas forcément les règles du jeu.
 - retournePionDansCase(Grid, Ligne, Colonne, Val) : retourne le pion présent aux coordonnées données.
 - verifLigne(Grid, Ligne, Pion) : vérifie si la ligne possède le pion donné en paramètre.
 - verifColonne([Ligne|Grid], Colonne, Pion) : vérifie si la colonne possède le pion donné en paramètre.
 - mod2Inf(Val, Inf) : retourne le plus grand multiplicateur de 2 inférieur ou égale à Val.
 - verifCarre(Grid, Pion, Px, Py) : vérifie si le carré possède le pion donné en paramètre.
 - verifCarreN(Grid, Pion, X, Y) : détermine quel carré on doit vérifier pour savoir si celui-ci possède le pion ou non.
 - verifCarreAll(Grid, Pion) : vérifie tous les carrés.
 - jouerCoup(Grid,Ligne,Colonne,Pion,ListePion,NvListePion) : Joue un coup aléatoire qui respecte les règles du jeu, les pions disponibles et si les cases sont vides.
-
- poidsLigne([L0|Ligne], Poids) : compte le nombre de pions présents sur une ligne
 - poidsColonne([Ligne|Grid], ColonneNb, Poids) : compte le nombre de pions présents sur une colonne
 - poidsCarree(Grid, ColonneNb, LigneNb, Poids) : compte le nombre de pions présents sur un carré
 - indexCarree(L,C,Ind) : attribue un chiffre au 4 carrés allant de 0 à 3
 - poidsMapping(Grid, PoidsMap) : calcul pour chaque ligne, colonne, carré, le nombre de pions présents
 - poidsMappingApply([Ligne|PoidsMap], Poids, CurrL, CurrC): applique les résultat de poidsMapping aux différentes cases du jeu, de manière à obtenir le poids de chacune d'elles -> voir partie détaillant le fonctionnement de l'heuristique
 - prioriteListe([Ligne|PoidsMap], PrioListes, NvPrioListes, CurrL, CurrC) : retourne une liste de coordonnées de cases à jouer ordonnées par priorité descendante.
 - flatten_level2 && flatten_level1 : fonction permettant d'aplatir les listes de coordonnées, la fonction flatten de base n'étant pas utilisable car aplatie totalement, hors on souhaite obtenir une liste de paires de coordonnées.
 - verifAll(Grid,L,C,P) : vérifie si le coup est valide, si la case est libre et que le pion n'est pas déjà présent dans la ligne, la colonne ou le carré.
 - pionRandom(PionRestant,P) : tire un pion aléatoire parmi les pions restants.
 - jouerPosition(Grid,[T|Res], PionRestant, L,C,P) : calcule le prochain coup à jouer en fonction des coups précédant et des pions restants et en fonction de la liste en second paramètre qui donne en premier élément la meilleur position à jouer
 - testTousLesPions(Grid,[P1|PionRestant],L1,C1,L,C,P) : teste parmi les pions restants, si un valide à la position donnée, la fonction retourne le premier pion valide.

- jouerCoupHeuristique(Grid, PionRestant, L,C,P) : calcule le prochain coup à jouer en respectant les coups précédents, les pions restants et les règles du jeu.




3. Heuristique

A chaque fois qu'un coup est requis de la part du joueur, celui-ci interroge son moteur prolog en lui envoyant l'état actuel du plateau et les pions qu'ils n'a pas encore utilisé. Le coup qui est alors calculé respecte les règles du jeu, mais place également le joueur dans la position la plus favorable possible, ceci grâce à l'heuristique élaborée.

Cette position implique que dans la mesure du possible, on doit par exemple éviter de renvoyer une grille de jeu contenant une ligne, colonne ou un carré n'attendant qu'un pion pour être terminé, cas dans lequel on donnerait une victoire facile à l'adversaire. On doit donc de la même façon, tenter de jouer en priorité sur les cases qui peuvent nous rapporter la victoire.

La méthodologie appliquée par les fonctions de l'heuristique est la suivante :

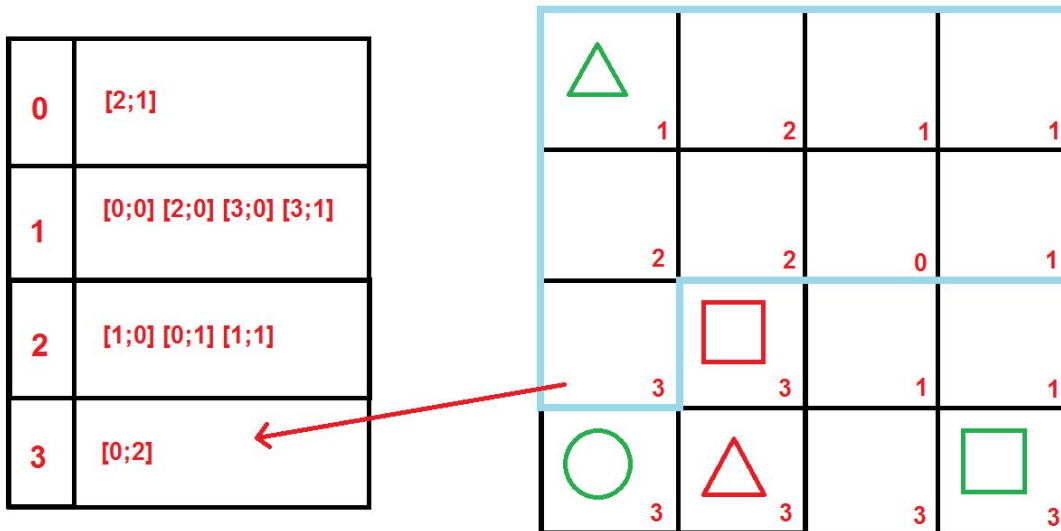
- Après la réception de la grille contenant les pions, une deuxième grille est créée, contenant elle des poids, chaque case de cette grille est assimilable à la case située à la même position dans la grille de pions. Le poids d'une case informe sur l'aboutissement du motif le plus complet auquel elle appartient, voici un schéma qui illustre ce principe :

		Carré : 0 Ligne : 1 Colonne : 2	
	1	2	1
1	1	2	0
1	1	 2	2
1	1	 2	2

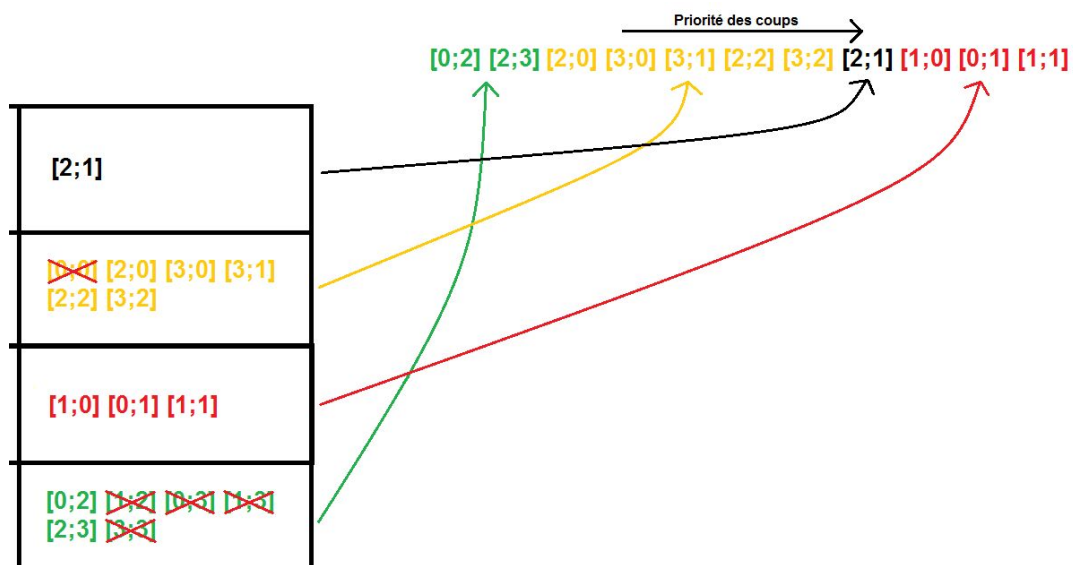
Comme on peut le voir, la case décrite appartient à un carré ne contenant aucun pion, son aboutissement est donc de 0, la ligne quant à elle à un aboutissement de 1 et la

colonne un aboutissement maximal de 2, c'est donc le poids de la colonne qui est attribué à la case.

- Une fois que cette étape est effectuée, il reste alors à déterminer à partir de ces données, une liste des cases à essayer par ordre de priorité, en effet il se peut qu'une case soit intéressante mais que les pions qu'il nous reste ou la disposition du plateau ne nous permette pas d'y jouer quoi que ce soit, on doit alors être capable de se rabattre sur un "deuxième meilleur choix". Un parcours de la grille permet de collecter les coordonnées des cases et de les trier par poids, comme sur l'exemple du schéma suivant :



- Les coordonnées des cases collectées sont ensuite mises bout à bout en suivant un ordre précis :
 - 1) D'abord les cases ayant un poids de 3
 - 2) Ensuite les poids de 1
 - 3) Les poids de 0
 - 4) Les poids de 2



On supprime au passage les coordonnées des cases sur lesquelles des pions ont déjà été placés. Le résultat de cet agencement est que l'IA, si elle ne peut jouer un coup gagnant, va tenter de jouer de manière sécurisée, sans donner de motifs à compléter à l'adversaire, ceci tout en cherchant à avancer des motifs comportant déjà 1 pion.

Possibilités d'améliorations:

Un inconvénient principal de cette implémentation de l'heuristique est que le choix du type de pion à poser est fait de manière totalement arbitraire, la stratégie se base uniquement sur une étude des positions. Ceci est directement lié au fait que l'heuristique n'est pas capable de déterminer des états du jeu possiblent après plusieurs échanges, ce qui lui permettrait de déterminer le meilleur pion à poser en fonction de la case dans l'immédiat.

Une amélioration qui s'est avérée complexe à mettre en oeuvre mais qui aurait pu voir le jour avec plus de temps était: partir de l'état actuel du plateau, jouer contre un joueur fictif utilisant également notre heuristique jusqu'à victoire de l'un d'entre nous, réitérer le processus en modifiant nos coups (position et type de pion) jusqu'à trouver un scénario gagnant pour nous, le premier coup de ce scénario gagnant est alors celui qu'il faut jouer dans l'instant contre le joueur réel. On pourrait même imaginer mémoriser l'arbre des chemins déjà calculés pour éventuellement le réutiliser si l'adversaire a effectué un coup similaire à ce que nous avons prévu.

Conclusion de l'heuristique:

En résumé nous sommes plutôt satisfait de l'heuristique que nous avons développée, même si elle ne saura peut être pas affronter des adversaires effectuant des analyses en profondeur des possibilités de jeu, elle pourra remporter face à des joueurs se comportant de manière arbitraire ou même ayant des heuristiques du même niveau.

4. Tests réalisés

Pour tester nos fonctions, nous avons réalisés plusieurs types de tests. Nous avons commencé par faire des tests unitaires dans le fichier prolog sur toutes les fonctions réalisées. Et, nous avons testé aussi ces fonctions via des tests unitaires en Java.

Les tests unitaires avec prolog, nous ont permis de vérifier le bon fonctionnement de nos fonctions dans des cas précis. Nous avons pu détecter un certain nombre de bug et assurer le bon déroulement de celle-ci. Ces tests ont aussi permis de donner des grilles de jeu spécifiques, cela a permis de tester lorsque l'IA ne peut plus jouer, ou quand elle va gagner.

Les tests unitaires avec Java étaient sensiblement les mêmes que ceux réalisés avec prolog. Ces tests avaient pour but de vérifier le bon fonctionnement entre Java et prolog de la bibliothèque JPL. On pouvait donner les mêmes paramètres aux fonctions et vérifier que nous recevions bien des valeurs cohérentes. C'est-à-dire des valeurs comprises dans la grille de jeu et qui respectent les règles du jeu. Ces fonctions étant déjà testées avec prolog nous savions qu'elles fonctionnaient. Mais cela a permis de vérifier la bonne transmission des données entre Java et Prolog

Pour terminer, nous avons aussi réalisé des tests en condition réelle. Nous avons réalisé des parties pour trouver et corriger des bugs. Pour tester dans ces conditions, nous avons deux clients et le serveur. Le premier client était celui qui jouait seul, donc notre intelligence artificielle. Le second client était un joueur qui attendait des saisies clavier. De ce fait, nous pouvions contrôler les coups et voir comment notre IA allait réagir face au coup que lui nous donnions. Nous avons donc pu tester, lorsque l'IA allait perdre et lorsqu'elle allait gagner. Nous avons aussi testé notre joueur contre d'autres joueurs pour voir comment celui réagit, cela a permis de détecter des bugs.