Rapport IA-Jeux Chancerel / Kerautret

Structure du plateau

Le plateau de jeu est un hexagone composé de 61 cases. Dans l'état initial, 55 cases sont des icebergs à collecter, 6 cases sont des bateaux. Un joueur possède 3 bateaux et l'adversaire en possède 3. Les bateaux sont placés aux extrémités de l'hexagone. Notre implémentation propose une ArrayList comprenant 9 ArrayList dans lesquelles nous ajoutons des Cases.

La première ArrayList représente le plateau dans son entièreté. Les ArrayList à l'intérieur représente les 9 lignes du plateau. Dans chacune de ces lignes, nous ajoutons entre 5 et 9 Cases. Une Case est un objet java que nous avons créé. Il possède plusieurs champs : la valeur de la case qui est une String, une position qui est un Point 2D, un score et un scoreVoisin qui sont des entiers et une valeur booléenne indiquant si la case a été visitée durant un parcours.

La valeur de chaque case peut être soir « o » pour représenter un iceberg, soit « . » pour représenter une case vide (l'iceberg a été collecté), soit « B », soit « R » représentant un bateau noir ou rouge.

Aussi, nous avons 2 ArrayList comprenant des Point 2D pour garder en mémoire les positions des bateaux rouges et des bateaux noirs. Ainsi, nous connaissons les positions de nos 3 bateaux ainsi que des bateaux ennemis.

Cette organisation est simple à implémenter mais nous la regrettons car la mise à jour des positions de chaque bateau est dédoublée. En effet, le plateau entier contient aussi ces informations. Aussi, nous n'avons pas utilisé de Map, les parcours du plateau sont donc obligatoirement des double boucles for, plus couteuse en temps.

Nous avons aussi été obligé de convertir un Point 2D en String de manière en renvoyer les moves de la manière spécifiée par la consigne (Exemple : C2-D2). En effet, notre point correspondant à C2 est en réalité un Point (x=2, y=1). La valeur doit être soustraite par 1 pour coller à la réponse serveur.

Cependant, sa facilité d'implémentation et de visualisation nous a aidé à avancer plus vite dans l'écriture de nos fonctions de recherche.

Algorithmie

Nous devions dans un premier temps trouver les coups possibles dans un état de plateau donné. Grâce à notre liste de bateaux nous pouvions avoir directement les positions de nos bateaux. Nous avons ensuite implémenté une fonction permettant de trouver les coups à 1 de distance d'un bateau. Nous avons d'abord géré les positions exceptionnelles comme les bords ou les coins. Ensuite, nous avions 1 ou 2 voisins latérales. Enfin, nous avons entre 1 et 4 voisins qui se situent au-dessus ou en dessous du bateau. Pour les trouver rapidement nous avons conçu la règle algorithmique suivante : si la ligne au-dessus existe et qu'elle est plus grande que la ligne du bateau, alors nous avons 2 voisins possibles en (x – 1, y) et en (x-1, Y+1), si elle est plus petite nous avons 2 voisins possibles en (x – 1, y – 1) et (x – 1, y). Lorsqu'on regarde les ligne en dessous, on remplace les x-1 par x+1.

C'est dans cette fonction que nous vérifions si un voisin est un bateau. S'il l'est, la position du bateau ennemi n'est pas ajoutée à la liste des voisins à 1 de distance.

Une fois que nous avions tous les voisins de tous nos bateaux, il fallait restreindre cette liste de coups possible aux coups qui nous rapproche d'un iceberg. Nous avons alors implémenté un algorithme BFS permettant de trouver les positions de tous les icebergs sur le plateau en partant de chacun de nos bateaux. Nous appliquons un second BFS en partant cette fois-ci des icebergs pour remonter vers les cases voisines de nos bateaux et en mettant à jour un score sur chaque case correspondant à la distance bateau-iceberg. La ou les case(s) voisine(s) de chaque bateau ayant le score le plus bas sont ajoutés dans la liste de coups possibles. C'est cette liste que nous renvoyons à l'appel de possibleMoves.

BestMove

La fonction bestMove effectue plusieurs étapes. Tout d'abord, on récupère la liste de possibleMoves de notre joueur. Ensuite, on joue chaque move sur une copie du board sur laquelle on appelle ensuite minMax. Minmax vient jouer un coup, d'abord de l'adversaire, puis du joueur ; on teste alors toutes les combinaisons jusqu'à une certaine profondeur. On en ressort une évaluation de chaque état final et on prend le plus favorable pour le joueur face à l'adversaire.

Pour cela, on utilise une fonction eval qui va renvoyer le score du jouer actuel auquel on vient soustraire le score de l'adversaire, afin d'obtenir un score de cette suite de moves. Ainsi, on peut en ressortir un premier déplacement qui sera approximativement le meilleur calculé.

Conclusion

Ce projet était intéressant du fait du défi qu'il présentait. Nous avons rencontré des difficultés à implémenter la fonction possibleMoves car le board étant hexagonal et non pas carré, les voisins ne sont pas juste les 4 cases voisines du tableau dans lequel nous l'avons représenté.

De plus, la fonction bestmove a présenté des problèmes : la copie du board semblait ne faire qu'une référence au premier board, ce qui le modifiait, rendant impossible de calculer les autres moves.