IA02

Rapport de Projet

Jeu de Gyges

Devaux Tony
Donin Stéphan

Sommaire

Affichage du plateau de jeu

Pour cette section, nous définirons un plateau de jeu type contenant en première ligne du joueur Nord, comme du joueur Sud, de gauche à droite, un pion à un cercle, puis une pion à deux cercles, puis un pion à trois cercles, puis un pion à trois cercles, puis un pion à deux cercles, et finalement un pion à un cercle.

Nous avons choisi de représenter ce plateau de jeu type de la façon suivante :

```
          ---------
          |   N   |
-------------------------
| 1 | 2 | 3 | 2 | 2 | 1 |
-------------------------
|   |   |   |   |   |   |
-------------------------
|   |   |   |   |   |   |
-------------------------
|   |   |   |   |   |   |
-------------------------
|   |   |   |   |   |   |
-------------------------
| 1 | 2 | 3 | 3 | 2 | 1 |
-------------------------
          |   S   |
          ---------
```

N représente le joueur Nord comme la case Nord, objectif du joueur Sud, et inversement avec S.
1 représente un pion à un cercle.
2 représente un pion à deux cercles.
3 représente un pion à trois cercles.

Stratégie d'affichage.

L'affichage va avoir lieu en plusieurs temps. On observe ainsi l'algorithme suivant :

- Affichage de la structure fixe du plateau de jeu
- Pour chacune des lignes, on fera appel au prédicat `afficherLigne` avec le bon numéro de ligne à afficher

Le prédicat `afficherLigne` fonctionne de la manière suivante :

- Il reçoit en paramètre le numéro de la ligne qu'il doit afficher
- Ensuite, il fait appel au prédicat `afficherCase` en lui fournissant en paramètre le numéro de la case à afficher. Pour cela, on additionnera le numéro de ligne, multiplié par 10, à 1, 2, 3, 4, 5 ou 6, selon la case à afficher.

Le prédicat `afficherCase` teste 4 cas distincts :

- Le numéro de case reçu en paramètre appartient à la première sous-liste du descripteur de jeu : ainsi, l'algorithme affichera donc « 1 », qui représente un pion à un cercle.
- Le numéro de case reçu en paramètre appartient à la deuxièmere sous-liste du descripteur de jeu : ainsi, l'algorithme affichera donc « 2 », qui représente un pion à deux cercles.
- Le numéro de case reçu en paramètre appartient à la troisième sous-liste du descripteur de jeu : ainsi, l'algorithme affichera donc « 3 », qui représente un pion à trois cercles.
- Le numéro de case n'appartient à aucune des trois sous-listes : la case est donc libre et l'algorithme affichera un blanc.

Initialisation du plateau de départ.

Dans un premier temps, on affichera les conventions de saisie à l'utilisateur ; on fera appel au prédicat `afficherConsignes` pour cela.
Ensuite vient la fameuse étape de la saisie...

Nous allons alors demander à chacun des deux joueurs la place initiale de chacun de ses pions (premier pion à un cercle, deuxième pion à un cercle, etc...). À chaque entrée de l'utilisateur, un prédicat de contrôle sera appelé afin de vérifier deux choses :

- Le joueur Nord ne peut placer ses pions que sur la première ligne du plateau, et le joueur Sud ne peut placer ses pions que sur la dernière ligne du plateau
- Un joueur, n'importe lequel, ne peut placer un pion là où il en a déjà placé un.

Pour cela, nous allons lister les 12 cas de placement possibles dans le prédicat, et utilisons une liste dans laquelle nous ajoutons tous les choix précédents de place de l'utilisateur. Lors de l'appel à ce prédicat de contrôle, l'appartenance du choix de la place à cette liste sera donc contrôlée. C'est ainsi que fonctionne le prédicat `caseValide`.

Ensuite, une fois cette étape de contrôle passée, le prédicat d'initialisation du plateau va alors ajouter le choix de l'utilisateur à l'atome contenant le descripteur courant du plateau de jeu. Ainsi, on utilisera pour cela le prédicat `ajoutDansSousListe`, permettant d'ajouter un élément dans la première, deuxième ou troisième sous-liste d'une liste donnée en argument. C'est grâce à ce prédicat que nous allons construire le descripteur de plateau de jeu petit à petit.

Une fois la demande des choix des deux joueurs terminés avec succès, le prédicat affichera alors le plateau via le prédicat `afficherPlateau` précédemment développé.
C'est ainsi que fonctionne le prédicat `plateauDepart`.

Est-ce qu'un coup est possible ?

Dans cette partie, nous devons développer 2 prédicats : l'un déterminera si un coup est possible, et l'autre déterminera le chemin emprunté par un coup donné afin d'aboutir.

On représentera un coup sous la forme d'une liste comme suit :
[JoueurQuiJoue, PositionDeDepart, PositionDArrivee]

Pour développer le premier prédicat, nous avons eu besoin de différents prédicats qui sont expliqués ci-dessous.

coordonneeValide(Coordonnee).

Ce prédicat teste si la valeur passée en argument est valide sur la configuration du jeu. Ceci revient alors à tester si le numéro de ligne et le numéro de colonne tous les deux associés à la coordonnée donnée n'excèdent pas tous les deux la valeur 6 et sont bien positifs.

existePion(Plateau, Pion).

Ce dernier vérifie l'existence de *Pion* dans le plateau *Plateau*. Pour ceci, il va tester l'appartenance de *Pion* aux trois sous-listes composant *Plateau* et s'arrêtera dès la première occurrence de *Pion* trouvée.

pionsSurLigne(Plateau, NumeroLigne, ListeDePions)

Ce prédicat retourne une liste *ListeDePions* contenant tous les pions sur la ligne *NumeroLigne* du plateau *Plateau*. Ainsi, si aucun pion n'est présent sur cette dernière, *ListeDePion* sera alors une liste vide.

pionsJouables(Plateau, Joueur, ListeDePions)

Ce prédicat retourne la liste *ListeDePions* telle que celle-ci comporte tous les pions que le joueur *Joueur* puisse jouer dans l'état courant du plateau *Plateau.*

nombreCercles(Plateau, Pion, Nombre)

Ce prédicat calcule le nombre de cercles *Nombre* que contient le pion *Pion*.

distanceCases(Case1, Case2, Distance)

Ce prédicat calcule la distance entre les deux cases *Case1* et *Case2*.

neRencontrePasDePion(Plateau, Case1, Case2)

Ce prédicat teste si, par n'importe quel chemin emprunté de *Case1* à *Case2*, le déplacement n'engendrera pas de rencontre avec un autre pion.
Ce prédicat s'effacera donc s'il existe au moins un chemin par lequel le déplacement n'engendre pas de rencontre avec un autre pion.

peutRebondir(Plateau, Origine, Destination)

Ce prédicat teste si un pion à la case *Origine* peut rebondir sur un pion placé sur la case *Destination.* Il teste notamment si *Origine* peut atteindre *Destination* et si ce déplacement n'engendre aucune rencontre avec un autre pion.

Plusieurs choses sont à étudier afin de vérifier si un coup est possible :

- Il faut tout d'abord vérifier que les deux coordonnées, celle d'origine et celle de destination, existent toutes les deux dans le plateau courant.
- Ensuite, il faudra vérifier que le pion joué (soit la coordonnée d'origine) est bien un des pions que le joueur est autorisé à jouer.
- Puis, il faudra vérifier la concordance avec le type de pion : c'est-à-dire qu'il faudra calculer la distance entre la coordonnée de départ et la coordonnée d'arrivée, et vérifier que cette distance soit bien permise par le type de pion joué.

À ce stade, si le déplacement d'un pion d'un point A à un point B n'engendre pas la rencontre avec un autre pion, alors un déplacement simple (par simple, nous entendons sans rebondissement ni déplacement d'autres pions) sera permis.

Cependant, si un pion A rencontre un autre pion B lors de son déplacement et si, et seulement si, ce pion B est à la même coordonnée que la destination du pion A, alors le pion A pourra soit rebondir, soit « effacer » le pion B.

Rappelons que si un pion rebondit sur un second, alors il rebondira du même nombre de cases que le nombre de cercles composant ce deuxième.
Ce cas implique le fait de vérifier si un pion existe dans l'entourage du pion à déplacer, et si oui, il faudra alors vérifier si un déplacement à partir de la coordonnée de ce premier est possible vers la destination d'origine.
On observe alors le prédicat pionsAutour(Plateau, Pion, Liste) qui retourne la liste *Liste* composée de tous les pions à la portée du pion *Pion* sur le plateau *Plateau*. On se servira ainsi de ce prédicat afin de calculer si une case est atteignable par rebondissement.
Notons que pour éviter les débordements de mémoire et autres boucles infinies, nous sauvegarderons le parcours du pion afin de ne pas repasser sur les mêmes cases.

Dans le cas d'un effacement, le pion effacé sera alors placé librement sur le plateau, excepté derrière la première ligne de l'adversaire. Il s'agit alors simplement de vérifier que le pion de destination est dans la portée exacte (ni plus, ni moins) du pion initial.

Il faudra cependant tester une dernière chose avant de rendre cet algorithme performant : il faut tester si la destination ou le parcours du joueur ne comporte pas de case étant le départ de ce joueur. Pour être plus clair, le joueur NORD ne peut pas passer ni s'arrêter sur la case de départ côté NORD.

Jouer un coup

Avec les prédicats développés précédemment, le prédicat s'occupant de jouer un coup n'est alors plus d'une difficulté très élevée.

Tout d'abord, ce prédicat va vérifier que le coup passé en paramètre est un coup jouable dans l'état actuel de la carte grâce au prédicat coupPossible développé ci-dessus.
Une fois cette étape passée, une vérification sera faite et nous mènera à deux cas possible ; ainsi, on vérifiera que le pion de destination existe ou non sur le plateau :

- Si c'est le cas, alors on supprimera le pion qui est sur la case de destination, on supprimera le pion d'origine pour l'ajouter à la place du premier pion supprimé ; ensuite, on calculera la première case de libre dans le but de recevoir le pion qui a été effacé ; c'est le prédicat placeLibre qui s'occupera de ceci.
- Sinon, on effacera tout simplement le pion d'origine pour l'ajouter à la case de destination, tout le travail de vérification du coup ayant été fait au préalable dans le prédicat coupPossible.

Le prédicat placeLibre va examiner le plateau en vérifiant la disponibilité des cases. Ainsi, ce dernier va parcourir l'ensemble des cases du plateau dans un ordre bien défini qui est le suivant : il partira de la première ligne occupée par le joueur adverse et balaiera une à une les cases de chaque lignes et renverra la coordonnée de la première case libre qu'il trouvera.

Boucle interactive : partie entre deux joueurs humains

Pour la création de la boucle interactive permettant à deux joueurs humains de s'affronter dans un jeu de *gygues*, nous nous sommes beaucoup servis des prédicats asserta et retractall afin de gérer dynamiquement l'avancement des plateaux et le changement des joueurs au fil de la partie.

Ainsi, l'algorithme sera très simple : on demandera aux joueurs d'initialiser le plateau de jeu avec le prédicat plateauDepart, puis on fera jouer les joueurs un à un, chacun leur tour en faisant évoluer le plateau grâce aux prédicats asserta et retractall, jusqu'à ce que l'un des deux gagnent la partie.
Le prédicat repeat nous a été très utile également dans la génération de boucle avec condition de sortie.

Tous les prédicats développés jusqu'à maintenant seront directement ou indirectement utilisés dans ce prédicat-ci, tous les contrôles d'entrées et de validité seront donc implicitement effectués.

Coups Possibles

On distingue une fois de plus trois types de coups possibles dans un jeu de *Gygues* : les déplacements simples, les déplacements avec écrasement et les déplacements avec rebonds. Nous avons pris compte de la distinction entre ces trois types de déplacement dans la construction du prédicat destiné à déterminer les coups possibles selon une configuration donnée.

Déplacements simples

Pour ce type de déplacement, on récupère tout d'abord la liste de tous les pions avec lesquels le joueur peut jouer. Une fois cette liste obtenue, pour chacun des pions de cette liste, on vérifiera quelle case ces derniers peuvent atteindre, sans écraser ni sauter de pions. Une fois cette liste obtenue, on utilisera le prédicat transformationPionsCoups qui a pour but de transformer une liste de pions donnée en une liste de coups où chacun des pions composants la liste représentera la destination d'un coup. Un test sera effectué sur la liste finale afin d'en supprimer les doublons la composant.

Déplacements avec écrasement

Ce prédicat est clairement inspiré du prédicat déjà développé ci-dessus. Il s'agira juste du même algorithme, à l'exception qu'on testera si le pion à ajouter à la liste de résultats est un pion existant sur le plateau (le cas des cases de destination libres étant déjà pris en compte dans le résultat du prédicat précédent).

Déplacements avec rebonds.

Le développement des prédicats pour cet objectif nous a posé quelques problèmes sérieux. Effectivement, l'algorithme fut abordé de la façon suivante au début :

Pour chaque pions i jouables par le joueur donné, nous avons trouvé tous les pions accessibles (au sens où la distance entre i et l'un de ces pions, et le nombre de cercles du pion i doivent être deux valeurs égales) ;
Pour chacun de ces pions j, nous avons alors cherché les cases accessibles ; deux possibilités s'offrent alors à nous :

- la case accessible est libre et on ajoute cette dernière au résultat
- la case disponible n'est pas libre et on relance la recherche à partir du pion de destination, ce qui créera alors l'effet de rebond sur le pion.

Cependant, cet algorithme engendrait des problèmes de complexité important, menant sans cesse à des dépassements de mémoire. Afin de pallier à ce problème, il fallut, comme l'énoncé le conseillait, poser une heuristique afin de limiter l'espace de recherche et ne pas s'exposer à des problèmes d'explosion combinatoire.

Une idée serait de ne se limiter qu'à un unique rebondissement. Ainsi, une fois la liste des pions accessibles par l'un des pions jouables par le joueur obtenue, il nous suffirait de relancer le prédicat de calcul sur les coups possibles à partir de chacun de ces pions (on ne prendrait alors en compte que

les déplacements simples et les déplacements avec écrasement). Ceci aura donc comme effet de ne limiter l'espace de recherche qu'à un seul niveau de rebond, à savoir le rebond sur les pions directement accessibles par les pions jouables du joueur ; on aura ainsi limité la complexité de ce prédicat afin de le rendre exploitable.

En effet, après avoir développé un prédicat basé sur cet algorithme, on obtient une complexité d'exécution raisonnable, renvoyant les résultats attendus dans un délai de temps très court.

Détermination d'un meilleur coup

Ce prédicat fonctionnera de pair avec le prédicat calculant tous les coups possibles pour une configuration de jeu donné. En effet, le prédicat de calcul de coups possibles va renvoyer une liste de coups, qui pourra ensuite être communiquée au prédicat responsable de la détermination d'un meilleur coup.

Ainsi, le prédicat commencera par regarder si la liste des coups disponibles contient un coup gagnant ou non. Si c'est le cas, elle retiendra donc ce coup.
Sinon, elle regardera tous les autres coups et retiendra le coup permettant le plus grand déplacement sur le plateau, que ce soit un déplacement diagonal, horizontal ou vertical.
En effet, une bonne idée aurait été de préférer les coups où le pions va le plus loin vers la case de son adversaire mais ceci peut facilement mener à une défaite : en effet, un des objectifs premiers de ce jeu, comme pour le jeu d'échecs, est de « coincer » l'adversaire de sorte à ce que nous soyons en mesure de jouer un pion le plus proche de sa case et ainsi atteindre cette dernière.
Rappelons que les pions ne sont pas attitrés à un joueur donné ; ils sont manipulables par les deux joueurs, tant qu'ils sont sur la première ligne de ce dernier.
Ainsi, en adoptant une stratégie où l'on donnerait la priorité les déplacements les plus loin vers l'objectif du joueur, on empêcherait cet effet d'encerclement de l'adversaire en lançant peu de pions seuls en avant en quelques sortes. C'est pour cette raison que tous les types de déplacements ont été pris en compte et que seule la distance parcourue engendrée par un coup a été prise en compte.

Afin d'éviter de toujours faire tourner les IA sur un même meilleur coup ne variant jamais, on stockera chacun des coups effectués dans une liste qui fera office de mémoire, afin de ne pas répéter ces coups et entrer dans une boucle sans fin.

Évaluation d'un plateau

Le but de ce prédicat est de déterminer vers quel joueur penche la victoire, étant donné un plateau donné. Le prédicat d'évaluation de plateau doit retourner une valeur numérique, cette dernière étant positive si la victoire tend vers le joueur Nord, et négative si la victoire tend vers le joueur Sud.

Pour cet objectif, on s'est servi d'un prédicat calculant le nombre d'éléments d'une liste, contenus dans des intervalles fermés donnés, et avec un coefficient donné (le nombre d'éléments trouvé sera multiplié par ce coefficient à la fin du prédicat).

Pour évaluer un plateau, nous avons procédé comme suit :

- Un plateau est décomposé en deux zones :
  - Les trois premières lignes qui appartiennent au joueur Nord dans le critère d'évaluation d'un jeu.
  - Les trois dernières lignes qui appartiennent au joueur Sud dans le critère d'évaluation d'un jeu.
- On comptera le nombre de pions présents dans la zone Nord et le nombre de pions dans la zone Sud. Appelons N le nombre de pions présents dans la zone Nord, et S le nombre de pions présents dans la zone Sud.
- On attribuera des coefficients d'importance différents selon l'avancement où se situent les pions ; ainsi, sur une échelle de 1 à 3, plus un pion sera prêt d'un départ, plus il aura un coefficient élevé.
- Une fois N et S trouvé, le résultat final sera la différence entre S et N, de sorte à ce que l'on coïncide bien avec les exigences (nombre négatif si la victoire tend plus vers le joueur Sud, et positif si elle tend plus vers le joueur Nord).

Répartition des tâches

La répartition des tâches dans ce projet fut plutôt simple : nous nous sommes distribué les prédicats ne représentant pas une difficulté majeure afin que nous puissions les faire individuellement.
Concernant les prédicats plus compliqués, nous avons préféré les construire ensemble, afin d'être certains d'emprunter les mêmes mécanismes de réflexion dans le cas d'une éventuelle reprise du code. Ainsi, nous nous sommes rencontrés le plus souvent possibles afin de poser les bases de la construction des prédicats complexes. Lorsqu'une rencontre n'était pas possible, l'un de nous deux continua le développement des prédicats dont le mécanisme de réflexion avait déjà été posé au préalable (par les deux binômes) et expliqua ensuite par mail les avancées effectuées dans le projet.
Ainsi, on peut dire que chacun d'entre nous à toucher à plus ou moins tous les prédicats de ce projet.

Difficultés rencontrées et améliorations possibles

Il n'y a pas eu de difficultés majeures dans la réalisation de ce projet. Les seules difficultés seraient à la rigueur la complexité du prédicat de calcul de coups possibles où il a fallu, comme conseillé dans l'énoncé, mettre en place une heuristique limitant le domaine de recherche de l'intelligence artificielle. Il a fallu également bien penser les stratégies d'évaluation de coups et de plateaux afin d'être le plus pertinent possible, mais avec une bonne analyse et un bon recul, ce ne fut pas d'une difficulté insurmontable.

Une amélioration possible est évidente : celle de l'heuristique limitant le domaine de recherche des coups possibles par l'intelligence artificielle. En effet, l'heuristique trouvée peut certaine être discutée et améliorée pour en donner une nouvelle plus efficace, qui sera certainement elle également destinée au même sort. On observera ainsi une potentielle amélioration infinie concernant cette heuristique, qui évoluera selon un esprit d'analyse de qualité, ainsi que dépendant du l'évolution du matériel informatique sur lequel cette intelligence artificielle sera embarquée.
On pourrait également améliorer la stratégie de détermination d'un meilleur coup à effectuer parmi une liste de coups donnée, ainsi que la stratégie d'évaluation d'un plateau de jeu, qui évoluera elle aussi selon l'esprit d'analyse des concepteurs.
Des aspects d'optimisation seront potentiellement envisageables, comme dans tout projet relevant d'une complexité élevée, comme c'est le cas dans les domaines de l'IA. Effectivement, chaque problème peut être abordé d'une façon différente, engendrant sans cesse des complexités différentes, meilleures comme moins bonnes.