Rapport technique
Mini Projet : Le jeu du carré, IA joueuses

Ange Jennyfer NGUENO FOKAM - 170840
Julien PETRIGNET 217307
Titouan FREVILLE - 217821
Sara EL AICHI 226328

14 juin 2017

Table des matières

Résumé

Le projet IA joueuses pour le jeu du carré à pour but de répondre au sujet de 4AIT de l'école Supinfo, promotion 2018. L'objectif du projet est de réalisé aux moins deux intelligences artificielle capable de jouer au jeu du carré. Les intelligences proposées devront être capable de jouer entre elle ou contre un joueur extérieur.

A partir de cet énoncé, plusieurs solutions s'offrent à nous. Pour pouvoir répondre le mieux aux problèmes, il nous faut tout d'abord analyser le jeu sélectionner afin de déterminer son type (décision, forçage, anticipation, calcul, réaction, ...). Nous pourrons alors nous interroger sur les différentes IA proposable puis sur les langages utilisables afin de sélectionner le plus intéressant pour nous.
Ce rapport à pour but de présenter toute la démarche de réflexion que nous avons eu afin de prendre une décision nous permettant de résoudre le problème de la façon la plus intéressante.

Première partie

Le jeu, présentation et compréhension

Chapitre 1

Historique, principe de base, type

Le jeu du carré est un jeu décris en 1889 pour la première fois par Edouard Lucas, mathématicien. Le principe du jeu est très simple, et nécessite peu de matériel. Le but est de créer des forme carré. Pour cela, chaque joueur va a tour de rôle tracer un segment sur un quadrillage permettant de représenter un côté pour un future carré. Le jouer ayant fermé le plus de carré remporte la partie.

1.1 Type de jeu

Le jeu des petits carrés est un jeu de réflexion. Il fonctionne comme le jeu de dame, dans le sens ou, lorsque le jeu est maîtriser, l'objectif devient de forcer le jouer adverse a fermer un certain nombre de carré pour pouvoir en récupérer plus par la suite. C'est donc un jeu de forçage et de décision. Fonctionnant sur des formes géométrique, il va donc se concentré sur la capacité à lire le plan représenter par la grille et à venir correctement enfermer l'adversaire dans une situation de fermeture perdante. Nous allons donc chercher à permettre à nos IA une bonne compréhension de l'occupation de l'espace, et une bonne lecture du plan grillagé.

1.2 Compréhension

Détaillons maintenant la stratégie principale du jeu.
Le jeu se base donc sur la géométrie et plus particulièrement les formes rectangulaires, et les carrés. La première phase du jeu va avoir pour objectif de créer des zones de « non droit » tel que si un joueur place un trait dans cette zone, il donne une grande quantité de points à son adversaire. Ensuite,

chaque joueur va devoir essayer d'agrandir sa zone de non droit et en « posséder » une depuis la qu'elle il récupérera plus de points que son adversaire.
L'objectif est donc de créer un couloir (appeler plus généralement « serpent » en raison de sa forme) longiligne puis de forcer l'adversaire à jouer en bordure de ce couloir tel que dès qu'il ferme un carré dans cette zone, le joueur récupère plus de point que lui.
Cette stratégie essentiel est toutefois complexe, et n'est pas ce que les joueurs seront capable de produire en premier. Afin de proposer des IAs équilibrées, il nous faudra donc une IA incapable de prévoir/comprendre ce Fonctionnement.
Nous pouvons déjà entrevoir deux système pour concevoir nos IA : une IA simpliste complétant la grille et fermant les carrés dès que possible, et une IA plus complexe gérant la notion basique de serpent sans anticiper sur les coups à venir.

Deuxième partie

Intelligence Artificielle, let's play

Chapitre 2

Définition des IA

Dans la partie précédente, nous avons parlé de plusieur IA possible, plus précisément, nous avons introduit deux IA simple. Nous allons ici continuer ce travail afin de redéfinir les IAs et les compléter.

2.1 IA - Base. Niveau 0

La première IA que nous allons décrire servira de base à toutes les IAs suivantes. C'est l'IA la plus simple que l'on puisse faire pour obtenir un résultat interressant, et ce n'est pas pleinement une IA dans le sens ou elle va simplement automatiser un processus de traitement de donnée.
L'unique objectif de cette IA est d'être capable de jouer au jeu.

2.1.1 Principe

L'IA basique a donc pour objectif simple de pouvoir placer correctement les liasons sur la grille, et être capable de prendre des points. Pour ce faire, elle va se contenter de placer les liaisons hors carré (des liaisons entre deux somets sont côté) de façon aléatoire sur la grille. Puis elle complétera les carrés ayant un côté. Une priorité absolue est donné pour fermer les carrés, ce qui imp̀lique que l'IA ferme de façon systématique les carrés visible jouable à son tour. Elle est incapable de calculer le revenu (en point d'une de ses actions).

2.1.2 Heurisitque

Cette IA a donc besoin de peu de connaissance, et peux de processus. Ces connaissances seront par la suite présente dans toutes les IAs.

— Capable de relier deux sommets correctement
— Capable de lire la grille
— Capable de trouver les carrés fermable

Ces trois éléments sont suffisant pour que cet IA puisse fonctionner.

2.1.3 Algorithmes

Pour la définition de l'algorithme, nous abstrayon la lecture et les types de données par la fonction : lireGrille(grille) dont on suppose le typage connue, et qui nous renvoie les informations voulues.
L'algoritmhe que nous allons utiliser pour la première IA est simpliste. Il utilise la fonction de lecture lireGrille pour récupéré soit un carré soit une liaison vide. Nous allons définir de façon abstraite la fonction lireGrille capable de renvoyer ces information puis l'algorithme de l'IA.

```
function lireGrille(grille)
    case ← currentCase
    for all case in reachableCase do
        if case is closable then
            return case
        else
            if grilleEnd then
                return randomCase
            end if
        end if
    end for
end function

function IA0
    while true do
        caseToPlay ←lireGrille(grille)
        if caseToPlay is closable then
            closeCase()
            countPoints()
        else
            drawCaseSegment()
        end if
        waitOverPlayer()
    end while
end function
```

Les fonctions abstraite drawCase, closeCase, countPoints et waitOverPlayer sont considiré comme de compléxité neutre et instantanée par rapport aux autres fonctions. Leur utilisation est évidente ainsi que ce qu'elle font.

2.1.4 Complexité

Spaciale

Assez peu de compléxité spaciale pour cette algorithme. Une variable est utilisé pour la grille, une pour stocker la case en cours d'analyse, une pour le score et une pour attendre l'autre jouer. Soit 4 variable simultanées. La plus part de ces variables contiennes une valeur courte pouvant être stocker faiclement et occupant très peu d'espace. La grille sera un peu plus longue, et son poid sera dépendant de la taille de celle-ci ainsi que du choix de représentation. On peut donc dire que la complexité spaciale CS est équivalente à la compléxité spaciale de la grille CSG. L'occupation de la mémoire est donc très raisonable et, elle augmentera de façon linéaire en fonction du nombre d'élément de la grille.

Temporelle

La compléxitée temporelle de cette algorithme est assez facile a calculer : $CT_{ia0}(n)$ = O($CT_{lireGrille}(n)$*$4n$) = O(n*$4n$)=O($4n^2$) = O(n^2), où n est le nombre de case de la grille.
Cette algorithme est donc très peu optimisé, est long a s'éxuté. Son temps d'éxécution augmlente en fonction du carré du nombre de case. Il devient donc rapidement au dela d'un temps d'attente supportable. Cependant, ce temps d'éxécution est globale sur toute l'éxution de l'IA, c'est a dire, globalement, si nous faisons s'affronter les deux IAs, cette augmentation de temps est valalble. Dans une utilisation en joueur contre IA, nous pouvons réduire la compléxité de l'éxécution a la complexité de jouer un tour, le joueur n'ayant qu'a attentendre le coup suivant de l'IA a chaque étape. Nous avons alors une compléxité linéraire en fonction du nombre d'élément de la grille, l'IA ne lisant qu'une seule fois la grille complète à chaque tour de jeu.

Cette IA est donc relativement rapide en terme d'attente pour le joueur entre chaque tour. Par contre, elle ne devrait présenter aucune difficulté après quelque partie. Il est également à noté que le temps sera un facteur beaucoup plus limitant que l'espace pour cette IA.

2.2 IA - Premières armes. Niveau 1

Développons maintenant une IA utilisant les structures de serpents introduite en première partie.

2.2.1 Principe

Cette seconde IA se base sur les mêmes connaissances que l'IA précédente. Cependant, nous allons modifier les règles de placement des liaisons et ajouter la capacité à compter les points. Ainsi, plutôt que de placer de façon aléatoire des côtés, cette nouvelle IA va chercher à créer des serpents et des coridors. Par conséquents, elle va en priorité placer un segment au bout d'un autre segment en cherchant a relier deux sommets spécifiques. Le choix de liaisons entre deux sommets se fait sur la quantité de points récupérables à la fermeture du serpent, cette valeur devant être maximisée. Lorsque l'IA à la possibilité de clore un serpent, elle va s'interroger sur la quantité de points qu'elle va gagner en le fermant par rapport à la quantité de point que l'adversaire peut récupérer. Si le rapport est positif, elle fermera systématiquement le serpend

2.2.2 Heurisitque

Notre seconde IA a donc besoin de nouvelle connaissance, bien qu'elle reste très simpliste.

— Capable de choisir le meilleur segment d'un serpent
— Capable de compter les points gagnables par chaque partie dans une configuration connue

Il lui faut donc deux nouvelles connaissance pour fonctionner correctement.

2.2.3 Algorithmes

function ADDMOVE(*case*, *side*, *snakeList*)
    Match *snakeList* with
    Empty → [([(*case*, *side*)],1)]
    (*snake*, *val*) : : *q* →
    if inSnake *case snake* **then**
        if member *case snake* **then**
            ((*case*, *side*) : :*snake*, *val*) : :*q*
        else
            ((*case*, *side*) : :*snake*, *val*+1) : :*q*

```
        end if
    else
        t : :addMove case side q
    end if
end function

function MOVE(grille,snakeList)
    Match snakeList with
    Empty → let case ←randomCase() in addMove case randomInt(0,3)
snakeList
    (snake, val) : : [] →
    if isClosableSnake(snake) then
        closeSnake(snake)
    end if
    (snake, val) : : (snake2, val2) : :q →
    if isClosableSnake(snake) then
        closeSnake(snake)
    else
        if closableWith(snake, snake2) then
            force(snake2)
        else
            move(grille,q)
        end if
    end if
end function

function FORCE(snake)
    if isClosableSnake(snake) then
        randomCase(snake)
    end if
    let (c1, c2) = snakeEnd(snake) in
    if closedSides(c1) < 3 then
        addSnakeSide(c1)
    else
        if closedSides(c2) < 3 then
            addSnakeSide(c2)
        elserandomCase(snake)
        end if
    end if
end function
```

function IA1($grille$,$snakeList$)
move($grille$,$snakeList$)
waitOverPlayer()
ia1($grille$, $snakeList$)
end function

Les fonctions non définis sont considéré comme abstraite et connue (car dépendante du type de données) et soit en temps réel soit sur une compléxité maximale de n.

2.2.4 Complexité

Spaciale

Cette série d'algorithme utilise plus de place que le précendent. Une variable supplémentaire doit être utiliser pour stocker la liste des serpents. La compléxité de la liste de serpent dépendant de celle de la grille, et on peut dire que dans le pire des cas, elle sera de : $CSG * CSG$. En l'ajoutant à la compléxité précédente, on a que $CSA1 = O(CSG + CSG^2) = CSG^2$. Aussi, dans le pire des cas, l'occupation de l'espace augmente exponentiellement vis à vis de la taille de la grille. Cependant, le cas ou chaque élément de la grille se retrouve dans chaque serpent possible est quasiment impossibe, sauf mal utilisation des fonctions de créations des serpents. On peut même dire que dans un cas idéale, ou il n'existe qu'un serpent, la compléxité spaciale sera linéaire en fonction du nombre d'élément de la grille. Par conséquent, en moyenne, la compléxité spaciale de la fonciton sera en moyenne logarithmique, aussi, on peut supposer que l'espace pris par la fonction augmentera énormement en fonction du nombre d'élément dans la grille pour des petites tailles puis aura tendance a se stabilisé pour des grilles de très grande taille.

Temporelle

Les compléxités sont données pour les pires cas. Soit n le nombre de case de la grille.

$CT_{addMove} = \mathrm{O}(n * n) = \mathrm{O}(n^2)$
$CT_{force} = \mathrm{O}(CT_{isClosableSnake} + CT_{snakeEnd}) = \mathrm{O}(n + n) = \mathrm{O}(n)$
$CT_{move} = \mathrm{O}(CT_{isClosableSnake} + CT_{closeSnake} + CT_{closableWith} + CT_{force}) * n = \mathrm{O}(n + n + n + n) * n = \mathrm{O}(n^2)$

La compléxité temporelle au pire est crois donc exponentiellement en fonction du nombre d'élément dans la grille. Cependant, le cas ou chaque élément de la grille donne un serpent contenant tous les éléments de la grille ne devrait pas arrivé dans un contexte d'utilisation normale. On peut approximé la compléxité moyenne a $n * log(n)$ en prenant en compte le fait que la liste des serpents sera trié à chaque modification afin de conserver le serpent donnant le plus de point en premère position. Nous nous retrouvons alors dans la même situation que pour la compléxité spaciale.

Globalement, c'est algorithme sembe donc plus lourd que l'algorithme précédent pour un joueur contre joueur. Cependant, pour une grille très grande, il va tendre à se rapprocher de l'algorithme de niveau 0 en vitesse et en occupation de l'espace. Il est donc plus intéressant pour un jeu standart avec des personnes ayant une compréhension correcte du jeu, mais cherchant un adversaire relativement simple à battre. Il est d'autant plus intéressant que des joueurs cherchant un certain challenge devrait joueur sur des grilles relativement grande, et par conséquent, révéler l'efficacité de cette algorithme proposant un bon équilibre entre compléxité de jeu et temps de jeu.

2.3 IA - La connaissance. Niveau 2

Pour le joueur, l'IA 1 va être plus complexe a vaincre par sa capacité à choisir l'option avec meilleur gain. Cependant, elle reste extrèmement simple à vaincre car très sensible au forcing. Ce qui nous amène à une nouvelle IA.

2.3.1 Principe

L'IA que nous allons introduire ici est l'IA la plus complète que l'on puisse faire pour un jeu. Elle est simple à comprendre, mais extrèmement difficile à battre. Elle consiste à générer l'intégralité des évolutions possibles dans une grille de la grille vide a la grille pleine. Elle connait alors toutes les configurations possibles à chaque instant de la partie et vas toujours choisir le placement lui donnant le plus de condition de victoire à chaque instant.

2.3.2 Heurisitque

Notre troisième IA a donc besoin d'une seule chose en plus de l'IA 0 : la connaissance de toutes les configurations. Elle n'a plus besoin de savoir compter les points puisque elle sait qu'elle coup l'amène à la victoire. Elle n'a plus non plus besoin de savoir construire des serpents. Bref, il lui faut juste revenir aux connaissance de bases et créer l'arbre de connaissance correspondant à la grille demandé par le joueur.

2.3.3 Algorithmes

function $\text{INITTREEMOVE}(grille, moveTree)$
    Match $grille$ with
    $Empty \rightarrow EmptyTree$
    $case :: resteGrille \rightarrow$
    if isClosed($case$) **then** initTreeMove(q, $moveTree$)
    else
        for i from 0 to 3 **do**
            let $newGrid$ = play($case$,i,$grille$) in
            let $moveTreeFrom$ = initTreeMove($newGrid$, $EmptyTree$) in
            let $newMoveTree$ = addTree($moveTreeFrom$, $moveTree$) in
            initTreeMove($resteGrille$, $newMoveTree$)
        end for
    end if
end function

function $\text{SELECTPLAY}(treeMove)$
    Match $treeMove$ with
    $EmptyTree \rightarrow EndGame$
    $(Racine, lFils) \rightarrow$ playWinner($lFils$)
end function

function $\text{PLAYWINNER}(listTreeMove)$
    Match $listTreeMove$ with
    $Empty \rightarrow (EmptyTree, 0)$
    $t :: Empty \rightarrow$ (t, score(t))
    $t :: q \rightarrow$ let ($currentWinner$, $currentScore$) = playWinner(q) and $thisScore$ = score(t) in
    if $thisScore > currentScore$ **then**
        (t, $thisScore$)
    else
        ($currentWinner$, $currentScore$)
    end if
end function

function $\text{IA2}(moveTree)$
    let $newTree$ = playWinner($moveTree$) in
    let $playerTree$ = waitOverPlayer($newTree$) in
    ia2($playerTree$)
end function

Les fonctions *score* et *play* sont abstraite de compléxité simple (O(1)). La fonction *addTree* n'est pas présenter ici car dépendante de la modélisation des données, elle sera cependant assez classique (lire l'arbre jusqu'a arriver à la place de l'élément à ajouter et équilibré si nécessaire).

2.3.4 Complexité

Spaciale

La complexité spaciale de cette fonction est essentiellement liée à la taille de l'arbre des coups, et à sa création. Nous pouvons limiter l'impact en mémoire en utilisant de la récursivité terminale au lieu de la récursivité standart, permettant ainsi de ne plus puiser dans les ressources de la RAM. Cependant, le programme demandera tout de même de stocker la grille complète et l'arbre des possibles complet. On peut dire que la complexité spaciale globale est lié à la compléxité spaciale de la grille ainsi : $CSGlobale = CSGrille^(n!)$ ou n est le nombre de case.

Temporelle

Dans cette configuration, l'essentielle de la compléxité temporelle est lié à la génération de l'arbre des possibles. En effet, les fonctions $selectPlay$, $playWinner$ et $ia2$ auront au pire une compléxité de n.

La compléxité de la fonction de génération des coups est lié à la compléxité d'ajout d'un élément dans l'arbre. N'ayant pas présenter l'algorithme d'ajout (lié à la modélisation des données), nous notterons $CT_{ajoutArbre}$ sa valeur exacte, on peut cependant l'approximé à $n * log(n)$ étant donné la structure d'arbre et en anticiapant le fait que l'arbre ne sera pas équilibré via une fonction spécifique (étant donné son objectif), mais qu'il devrait être naturellement équilibré par le concept de jeu.

Ainsi, la compléxité de la fonction d'ajout sera :

$CT_{initTreeMove} = \mathrm{O}((4 * CT^n_{ajoutArbre})) = \mathrm{O}(CT^n_{ajoutArbre}) \quad \mathrm{O}((n * log(n))^n)$ $\mathrm{O}(n^n)$

Troisième partie

Modélisation

Maintenant que nos futures IA sont définis, voyons comment modéliser les données du problèmes.

Chapitre 3

Environement

L'environement de jeu consiste en un plateau présentant une grille. C'est sur cette grille que nos viendrons dessiner nos segment. Le plateau globale est de forme rectangulaire, idéalement carré. La grille est découper en carré, décomposer en ligne et somment. Plusieur solutions s'offrent alors à nous pour représenter l'environement. Nous pouvons définir un graphe contenant chaque sommet et les liasions entre eux, en utilisant une variable sur chaque noeud pour présenter la présence ou non d'une liaison, ou nous pouvons représenter chaque sommet de la grille dans une matrice, et utiliser une liste pour stocker les liaisons connue.

3.1 Tableau

La représentation via tableau est la représentation la plus informatique du problème, et la plus simple vis à vis de la représentation graphique du problème. Chaque point de la matrice représentant un sommet de la grille, il est facile de sélectionner des sommets puis de les liers dans une liste via un couple.
L'avantage de cette méthode est donc la facilité de gestion de la partie graphique du jeu. Cependant, il est alors compliquer de savoir qu'elle côté sont déjà tracer. Il est donc plus long de récupérer les informations quand aux corridors et serpents présent de le jeu, ainsi que de s'assurer que le coup est autorisé.

3.2 Graphe

La représentation via graphe va permettre de compenser la faiblesse du tableau vis à vis des côté. Cependant, elle sera moins efficace d'un point de vue graphique. Le graphe a représenter doit stocker les informations sur la case. Vis a vis de la grille, une cas est un future carré, donc un point. Un point est fermé dès que ces 4 côtés sont fermé. Un sommet de notre graphe serait alors un triplet contenant le nom de la case (ex : C5), une liste booléen correspondant au côté de la case (vrai si fermé, faux sinon par exemple), et une liste contenant au plus quatre élément représentant les noms des cases adjacentes.
Nous avons alors une structures qui permet de facilement calculer les points de la carte, de visualiser les structures de corridors et serpents et de gérer les liaisons entre les cases. Bien que moi efficace sur le plan graphique, elle est beaucoup plus puissantes pour les algorithmes de résolution et plus rapides.

3.3 Solution choisie

Nous avons donc choisi la structure de graphe pour modéliser les données. L'objectif de notre travail étant principalement de créer des IAs capable de jouer au jeu, et non de permettre a deux joueurs de s'affronter. Nous avons naturellement choisi la solution la plus interressante du point de vue algorithmique.

Chapitre 4

Information de jeu

Une fois la représentation de l'environement choisi (en l'occurence, un graphe), il nous faut modéliser les coups et options de jeu. Encore une fois, il y a plusieur solutions. Cepenant, ici, les différentes solutions vont plus s'appliquer au choix d'implémentation d'IA qu'a un besoin. En effet, pour l'IA de niveau 0, il n'y a pas besoin de structure particulière pour symboliser les coups. Une simple lecture du graphe suffit. Les structures sont a définir pour l'IA de niveau 1 et 2.

4.1 IA 1 - Structure

La première IA a besoin d'une connaissance spécifique a chaque configuration du graphe : la quantité de points gagnable pour chaque serpent. Cette valeur peut s'obtenir en comptant le nombre de carré présent dans le serpent. Dans un soucis de vitesse, une liste contenant les différents serpents ainsi que le différentiel de points joueur/IA peut être intéressante a définir. Au quel cas, après chaque coups, cette liste est mise à jour. Dans cette liste, un serpent est un triplet contenant le nom de la case de départ, et celui de la case de fin, puis le différentiel de score (ex : le triplet (A0, A3, 4) représente la ligne allant de la première case à la troisième dans la rangé A et rapporte 4 points a l'IA quand elle le fermera).

4.2 IA 2 - Stucture

La structure a utilisé pour l'IA 2 est assez évidente, elle fait partie de sa définition. Il faut définir un arbre qui va stocker chaque confiuration possible du graphe en fonction du coup précédent. Nous allon alors avoir un arbre n-aire ayant pour racine la configuration initiale et ou chaque fils est la configuration suivante en fonction du coup jouer.

Quatrième partie

Choix du Langage

Maintenant que nous avons déterminé nos structure de donné et nos algorithmes, nous pouvons justifier un choix de langage de programmation.
Nos algorithmes et structures de donnés sont récursifs, par conséquent, s'orienter sur un langage impératif tel que C ou Java est peu adapté (même si les dernières version gère correctement la récursivité, ce ne sont pas des langages conçue dans ce but). De plus, nous avons des algorithme complexe à mettre en place. Aussi, un langage de programmation fonctionnelle semble plus approprié.
Parmi les différents langages de programmation fonctionnelle que nous connaission (LISP, élixir et OCaml), nous avons préféré choisir OCaml car un des membres du groupes était plus à l'aise avec OCaml que les autres, l'ayant pratiqué pendant 3 ans.
Nous avons donc choisi le langage sur le quel nous étions le plus expérimenté, et qui nous semble le plus adapté vis à vis de nos données.

Cinquième partie

Simulation de fonctionnement