

Rapport technique
Mini projet : Le jeu du carré, IAs joueuses

Ange Jennyfer NGUENO FOKAM - 170840

Julien PETRIGNET 217307

Titouan FREVILLE - 217821

Sara EL AICHI 226328

15 juin 2017

Table des matières

I	Le jeu, présentation et compréhension	1
1	Historique, principe de base, type	2
1.1	Type de jeu	2
1.2	Compréhension	2
II	Intelligence Artificielle, let's play	4
2	Définition des IA	5
2.1	IA - Base. Niveau 0	5
2.1.1	Principe	5
2.1.2	Heuristique	5
2.1.3	Algorithmes	6
2.1.4	Complexité	7
2.2	IA - Premières armes. Niveau 1	8
2.2.1	Principe	8
2.2.2	Heuristique	8
2.2.3	Algorithmes	8
2.2.4	Complexité	10
2.3	IA - La connaissance. Niveau 2	11
2.3.1	Principe	11
2.3.2	Heuristique	11
2.3.3	Algorithmes	12
2.3.4	Complexité	13
III	Modélisation	15
3	Environnement	17
3.1	Tableau	17
3.2	Graph	18

3.3	Solution choisie	18
4	Information de jeu	19
4.1	IA 1 - Structure	19
4.2	IA 2 - Stucture	19
IV	Choix du Langage	21
V	Simulation de fonctionnement	23
5	Initialisation	25
6	Tour de jeu	26
6.1	IA0	26
6.2	IA1	26
6.3	IA2	27

Table des figures

6.1	Forme de la grille 4x4 après instantiation	28
6.2	Noeud à ajouter dans l'arbre des possible et grille mise à jour pour l'appel récursif	29
6.3	Etat de la grille après les 3 coups joués.	30
6.4	Etat de la grille après que l'IA 0 aie joué.	30
6.5	Etat de la grille après que l'IA 1 aie joué.	31
6.6	Etat de la grille après que l'IA 2 aie joué.	31

Résumé

Le projet IAs joueuses pour le jeu du carré a pour objet de répondre au sujet de 4AIT de l'école Supinfo, promotion 2018. L'objectif du projet est de réaliser au moins deux intelligences artificielle capables de jouer au jeu du carré. Les intelligences proposées devront être capables 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 sélectionné afin de déterminer son type (décision, forçage, anticipation, calcul, réaction, ...). Nous pourrons alors nous interroger sur les différentes IAs proposées puis sur les langages utilisables afin de sélectionner le plus intéressant pour nous.

Ce rapport présente 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 décrit pour la première fois en 1889 par Edouard Lucas, mathématicien. Le principe du jeu est très simple, et nécessite peu de matériel. Le but est de créer des formes carrées. Pour cela, chaque joueur va à tour de rôle tracer un segment sur un quadrillage permettant de représenter un côté pour un futur carré. Le joueur 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 dames, dans le sens où, lorsque le jeu est maîtrisé, l'objectif devient de forcer le joueur adverse à fermer un certain nombre de carrés pour pouvoir en récupérer plus par la suite. C'est donc un jeu de forçage et de décision. Fonctionnant sur des formes géométriques, il va donc se concentrer sur la capacité à lire le plan représenté 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 là 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 points que lui.

Cette stratégie essentielle est toutefois complexe, et n'est pas ce que les joueurs seront capables 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 plusieurs IAs possibles, plus précisément, nous avons introduit deux IAs simples. 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 intéressant, et ce n'est pas pleinement une IA dans le sens où elle va simplement automatiser un processus de traitement de données. 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 liaisons 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 sommets sans côtés) de façon aléatoire sur la grille. Puis elle complétera les carrés ayant un côté. Une priorité absolue est donnée pour fermer les carrés, ce qui implique que l'IA ferme de façon systématique les carrés visibles jouables à son tour. Elle est incapable de calculer le revenu en points d'une de ses actions.

2.1.2 Heuristique

Cette IA a donc besoin de peu de connaissance, et peu 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 cette IA puisse fonctionner.

2.1.3 Algorithmes

Pour la définition de l'algorithme, nous abstrayons la lecture et les types de données par la fonction : lireGrille(*grille*) dont on suppose le typage connu, et qui nous renvoie les informations voulues.

L'algorithme que nous allons utiliser pour la première IA est simpliste. Il utilise la fonction de lecture lireGrille pour récupérer 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 abstraites `drawCase`, `closeCase`, `countPoints` et `waitOverPlayer` sont considérées comme de complexité neutre ou instantanée par rapport aux autres fonctions. Leur utilisation est évidente ainsi que ce qu'elle font.

2.1.4 Complexité

Spaciale

Assez peu de complexité spatiale pour cette algorithmme. Une variable est utilisée pour la grille, une pour stocker la case en cours d'analyse, une pour le score et une pour attendre l'autre joueur, soit 4 variables simultanées. La plus part de ces variables contiennent une valeur courte pouvant être stockée facilement et occupant très peu d'espace. La grille sera un peu plus longue, et son poids sera dépendant de la taille de celle-ci ainsi que du choix de représentation. On peut donc dire que la complexité spatiale CS est équivalente à la complexité spatiale de la grille CSG . L'occupation de la mémoire est donc très raisonnable et elle augmentera de façon linéaire en fonction du nombre d'élément de la grille.

Temporelle

La complexité temporelle de cette algorithmme est assez facile à 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.

Cet algorithmme est donc très peu optimisé et est long à s'exécuter. Son temps d'exécution augmente en fonction du carré du nombre de case. Il devient donc rapidement au delà d'un temps d'attente supportable. Cependant, ce temps d'exécution est globale sur toute l'exécution de l'IA, c'est à dire, globalement, si nous faisons s'affronter les deux IAs, cette augmentation de temps est valable. Dans une utilisation en joueur contre IA, nous pouvons réduire la complexité de l'exécution à la complexité de jouer un tour, le joueur n'ayant qu'à attendre le coup suivant de l'IA à chaque étape. Nous avons alors une complexité linéaire 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 noter 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 corridors. Par conséquent, elle va en priorité placer un segment au bout d'un autre segment en cherchant à relier deux sommets spécifiques. Le choix de liaison entre deux sommets se fait sur la quantité de points récupérables à la fermeture du serpent, cette valeur devant être maximisée. Lorsque l'IA a 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 serpent.

2.2.2 Heuristique

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 connaissances pour fonctionner correctement.

2.2.3 Algorithmes

```
function ADDMOVE(case, side, snakeList)  
  Match snakeList with  
  Empty  $\rightarrow [([(\textit{case}, \textit{side})], 1)]$   
  (snake, val) : : q  $\rightarrow$   
  if inSnake case snake then  
    if member case snake then  
       $((\textit{case}, \textit{side}) : : \textit{snake}, \textit{val}) : : \textit{q}$   
    else  
       $((\textit{case}, \textit{side}) : : \textit{snake}, \textit{val}+1) : : \textit{q}$ 
```

```

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

function MOVE( $grille, snakeList$ )
    Match  $snakeList$  with
    Empty  $\rightarrow$  let  $case \leftarrow$  randomCase() in addMove case randomInt(0,3)
     $snakeList$ 
    ( $snake, val$ ) : :  $\parallel \rightarrow$ 
    if isClosableSnake( $snake$ ) then
        closeSnake( $snake$ )
    end if
    ( $snake, val$ ) : : ( $snake2, val2$ ) : :  $q \rightarrow$ 
    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( $c2$ )
        else randomCase( $snake$ )
        end if
    end if
end function

```

```

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

```

Les fonctions non définies sont considérés comme abstraites et connues (car dépendantes du type de données) et soit en temps réel soit ayant une complexité maximale de n .

2.2.4 Complexité

Spaciale

Cette série d'algorithmes utilise plus de place que le précédent. Une variable supplémentaire doit être utilisée pour stocker la liste des serpents. La complexité de la liste de serpent dépendant de celle de la grille, nous pouvons dire que dans le pire des cas, elle sera de : $CSG * CSG$. En l'ajoutant à la complexité 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 où chaque élément de la grille se retrouve dans chaque serpent possible est quasiment impossible, sauf mal utilisation des fonctions de créations des serpents. On peut même dire que dans un cas idéal, où il n'existe qu'un serpent, la complexité spatiale sera linéaire en fonction du nombre d'élément de la grille. Par conséquent, en moyenne, la complexité spatiale de la fonction sera en moyenne logarithmique, aussi, on peut supposer que l'espace pris par la fonction augmentera énormément en fonction du nombre d'élément dans la grille pour des petites tailles puis aura tendance à se stabiliser pour des grilles de très grandes tailles.

Temporelle

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

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

La complexité temporelle au pire croît donc exponentiellement en fonction du nombre d'élément dans la grille. Cependant, le cas où chaque élément de la grille donne un serpent contenant tous les éléments de la grille ne devrait pas arriver dans un contexte d'utilisation normale. Nous pouvons approximer la

complexité 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 première position. Nous nous retrouvons alors dans la même situation que pour la complexité spatiale.

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 complexité 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 le meilleur gain. Cependant, elle reste extrêmement 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 extrêmement difficile à battre. Elle consiste à générer l'intégralité des évolutions possibles du jeu, de la grille vide a la grille pleine. Elle connaît 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 Heuristitque

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 demandée par le joueur.

2.3.3 Algorithmes

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

```
function 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 IA2(moveTree)
  let newTree = playWinner(moveTree) in
  let playerTree = waitOverPlayer(newTree) in
  ia2(playerTree)
end function
```

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

2.3.4 Complexité

Spaciale

La complexité spatiale 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 non-terminal, 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. Nous pouvons dire que la complexité spatiale globale est liée à la complexité spatiale de la grille ainsi : $CSGlobale = CSGrille * (n!)$ ou n est le nombre de case. Nous pouvons en conclure que, si nous n'appliquons pas de récursivité terminale, nous allons très rapidement dépasser la mémoire d'un ordinateur standard, car pour chaque appel récursif, nous allons stocker un arbre de taille $CSGlobale$. Le nombre d'appel récursif augmentant avec la taille de la grille de façon linéaire, nous pouvons dire que l'algorithme occupe une large place dans la mémoire en récursif standard, et il est certain qu'une récursivité terminale permettrait de compenser cette faiblesse. Cependant, au vue des performances actuelle des ordinateur, nous pouvons l'implémenter en récursif non terminale en limitant la taille de la grille, et en se basant sur des tailles « standards » de l'ordre de $8 * 8$ par exemple.

Temporelle

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

La complexité de la fonction de génération des coups est liée à la complexité d'ajout d'un élément dans l'arbre. N'ayant pas présenter l'algorithme d'ajout (lié à la modélisation des données), nous noterons $CT_{ajoutArbre}$ sa valeur exacte, on peut cependant l'approximée à $n * \log(n)$ étant donné la structure d'arbre et en anticipant 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 complexité de la fonction d'ajout sera :

$$CT_{initTreeMove} = O((4 * CT_{ajoutArbre} * n)) = O(CT_{ajoutArbre}^n) \quad O((n * \log(n)) * n) \quad O(n^2 * \log(n)) > O(n^2)$$

Pour cet algorithme, la génération de l'arbre des possibles, et donc l'attente avant le lancement du jeu, va augmenter de façon exponentielle, c'est à dire que plus la grille va être grande, plus le joueur devra attendre avant de pouvoir commencer à jouer. Cependant, le choix d'un coup par l'ordinateur augmente de façon linéaire, car il lui suffit de trouver le bon élément dans la liste des possibilités suivant l'état actuelle de jeu. Cette liste ne pouvant dépasser le nombre d'élément présent dans la grille, et le nombre d'élément possible étant réduis à chaque étape (puisqu'il y a de moins en moins de case jouable), nous pouvons affirmer que l'IA devrait être de plus en plus rapide à jouer.

Pour cette dernière IA, nous avons affaire à une intelligence très difficile à battre. Cependant, le processus d'initialisation du jeu et de l'IA va être extrêmement lourd et long. Elle reste cependant intéressante pour des joueurs cherchant un challenge énorme, bien qu'il ne faille attendre un certain temps avant le début de la partie. Il est possible de gagner du temps en générant au préalable des modèles de grille et leurs arbres de coups. Au quel cas, si l'utilisateur choisit une grille connue, le processus de création n'est plus nécessaire, et le jeu sera en apparence aussi rapide que pour l'IA 0.

Troisième partie

Modélisation

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

Chapitre 3

Environnement

L'environnement de jeu consiste en un plateau présentant une grille. C'est sur cette grille que nous viendrons dessiner nos segments. Le plateau global est de forme rectangulaire, idéalement carré. La grille est découpée en carré, décomposée en ligne et en colonne. Plusieurs solutions s'offrent alors à nous pour représenter l'environnement. Nous pouvons définir un graph contenant chaque sommet et les liaisons entre eux, en utilisant une variable sur chaque nœud 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 connues.

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 lier 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 compliqué de savoir quels côtés sont déjà tracés. Il est donc plus long de récupérer les informations quand aux corridors et serpents présents dans le jeu, ainsi que de s'assurer que le coup est autorisé.

3.2 Graph

La représentation via graph va permettre de compenser la faiblesse du tableau vis à vis des côtés. Cependant, elle sera moins efficace d'un point de vue graphique. Le graph a représenter doit stocker les informations sur la case. Vis à vis de la grille, une case est un carré, donc un point. Un point est fermé dès que ces 4 côtés sont fermés. Un sommet de notre graph 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 structure 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 moins efficace sur le plan graphique, elle est beaucoup plus puissante pour les algorithmes de résolution et plus rapide.

3.3 Solution choisie

Nous avons donc choisi la structure de graph 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 intéressante du point de vue algorithmique.

Chapitre 4

Information de jeu

Une fois la représentation de l'environnement choisi (en l'occurrence, un graph), il nous faut modéliser les coups et options de jeu. Encore une fois, il y a plusieurs solutions. Cependant, ici, les différentes solutions vont plus s'appliquer au choix d'implémentation d'IA qu'à 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 graph suffit. Les structures sont à définir pour l'IA de niveau 1 et 2.

4.1 IA 1 - Structure

La première IA a besoin d'une connaissance spécifique à chaque configuration du graph : 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 à 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 à l'IA quand elle le fermes).

4.2 IA 2 - Structure

La structure utilisée pour l'IA 2 est assez évidente, elle fait partie de sa définition. Il faut définir un arbre qui va stocker chaque configuration possible du graph en fonction du coup précédent. Nous allons alors avoir un

arbre n -aire ayant pour racine la configuration initiale et où chaque fils est la configuration suivante en fonction du coup jouer.

Quatrième partie

Choix du Langage

Maintenant que nous avons déterminé nos structures de données et nos algorithmes, nous pouvons justifier un choix de langage de programmation. Nos algorithmes et structures de données sont récursifs, par conséquent, s'orienter sur un langage impératif tel que C ou Java est peu adapté (même si les dernières versions gèrent correctement la récursivité, ce ne sont pas des langages conçus dans ce but). De plus, nous avons des algorithmes complexes à mettre en place. Aussi, un langage de programmation fonctionnelle semble plus approprié.

Parmi les différents langages de programmation fonctionnelle que nous connaissons (LISP, élixir et OCaml), nous avons préféré choisir OCaml car un des membres du groupe était plus à l'aise avec OCaml que les autres, l'ayant pratiqué pendant 3 ans.

Nous avons donc choisi le langage sur lequel nous étions le plus expérimentés, et qui nous semble le plus adapté vis-à-vis de nos données.

Cinquième partie

Simulation de fonctionnement

Voyons comment vont fonctionner ces IAs avec un exemple simple.

Pour la simulation, nous utiliserons une grille 2x2 (donc une grille de 4 cases).

Chapitre 5

Initialisation

Voyons comment les différentes IAs et la grille vont être initialisées.

Dans le cas d'une grille 2x2, la grille va être initialisé assez rapidement. Elle sera représentée par un arbre de 4 éléments binaire et équilibré afin de représenter le graph choisi pour la grille (figure : Figure 6.1, page 28).

Du point de vue des IAs, les IAs 0 et 1 n'ont pas besoin de phase d'initialisation autre que leur création. Il n'y a pas d'objets particulier à mettre en oeuvre, seulement des espaces mémoires à réserver.

Pour la dernière IA, il nous faut initialiser l'arbre des possibles. Ce processus est long, et l'arbre obtenue même pour une grille aussi petite, reste assez imposant, aussi nous ne présenterons qu'une dérivation limité de l'arbre des coups afin d'illustrer son fonctionnement sans avoir à présenter un schéma trop lourd.

Pour initialiser l'arbre des coups, nous allons procéder assez simplement :

- Nous prenons une case de la grille.
- Nous créons zéro à quatre (en fonction du nombre de côté disponible dans la grille) « copie » de la grille que nous mettons à jour, chacune en fonction du côté du carré fermé. (figure : Figure 6.2, page : 29)
- Nous rappelons la fonction sur les grilles obtenues afin de générer le rang suivant.

Chapitre 6

Tour de jeu

Projetons nous dans une situation de jeu, en admettant que 3 coups aient été joués et que ce soit au tour de l'IA. La configuration actuelle est représentée dans la figure : Figure 6.3, page : 30.

Au quel cas, nous avons deux IA qui sont dans des états spécifique de mémoire.

L'IA 1, qui voit 2 serpents $([(0, a)], 1)$, $([(1, a), (1, n), (0, b), 3])$ et l'IA 2 qui va avoir évoluée dans l'arbre des possibles jusqu'à l'état actuelle.

Dans cette configuration, faisons jouer chaque IA.

6.1 IA0

Depuis cette configuration, l'IA 0 voit un carré fermable (A0) et le ferme en jouant le coup : $(a, 0)$, 3 pour jouer le côté droit du carré. Une fois ce coups exécuter, elle doit rejouer un coup. Cette fois, aucun carré n'est fermable. Elle va donc tirer un coup au hasard dans la liste des coups possibles. Par exemple : $(a, 1)$, 2 (pour jouer le côté du dessous du carré $a, 1$).

Ce qui nous amène à la configuration représenté dans : Figure 6.4, page : 30

6.2 IA1

L'IA 1 a pour but de marquer plus de points que son adversaire à chaque tour. Ici, elle voit les deux serpents $([(0, a)], 1)$, $([(1, a), (1, n), (0, b), 3])$. Pour marquer plus de points que son adversaire, elle doit donc essayer de fermer le deuxième serpent, valant 3 points. Ce serpents n'est pas fermable pour ce tour. Elle va donc regarder si le suivant est fermable. Le serpent 1 est fermable. Elle va alors regarder si, en fermant ce serpent et en jouant le tour suivant, elle permet à son adversaire de fermer le serpent 2. Ici, l'adversaire ne

pourras pas fermer le serpent 2 après le tour de l'IA. Par conséquent, elle va fermer le carré en jouant le coup $(a,0)$, 3 puis jouer un coup pour déterminer le serpent. Par exemple : $(b,1)$, 3.

Ce qui nous amène à la configuration représenté dans : Figure 6.5, page : 31

6.3 IA2

L'IA 2 va quand à elle regarder les fils directe de la configuration actuelle et chercher celui qui l'amène au plus de condition de victoire. Ici, fermé le carré elle même l'amène a une situation ou, sans erreur du joueur adverse, elle est sûre de perdre. Par conséquent, elle ne va pas fermer ce carré, mais plutôt le laisser à l'adversaire, et joueur un coup quelconque lui permettant de s'assurer une victoire future. Par exemple : $(1,b)$, 2.

Ce qui nous amène à la configuration représenté dans : Figure 6.6, page : 31

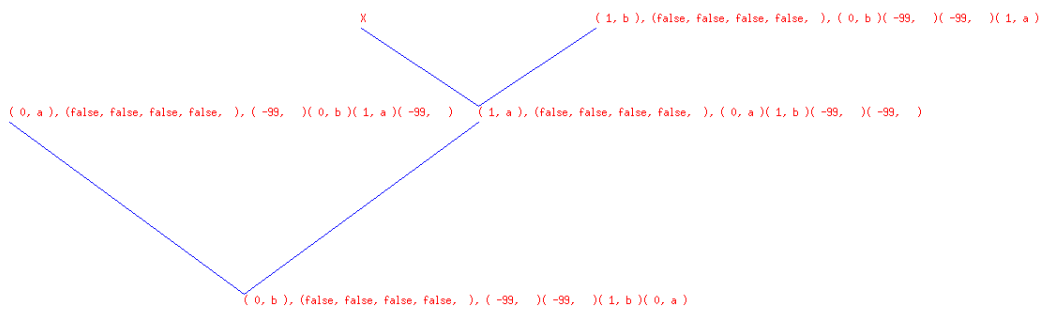


FIGURE 6.1 – Forme de la grille 4x4 après instantiation

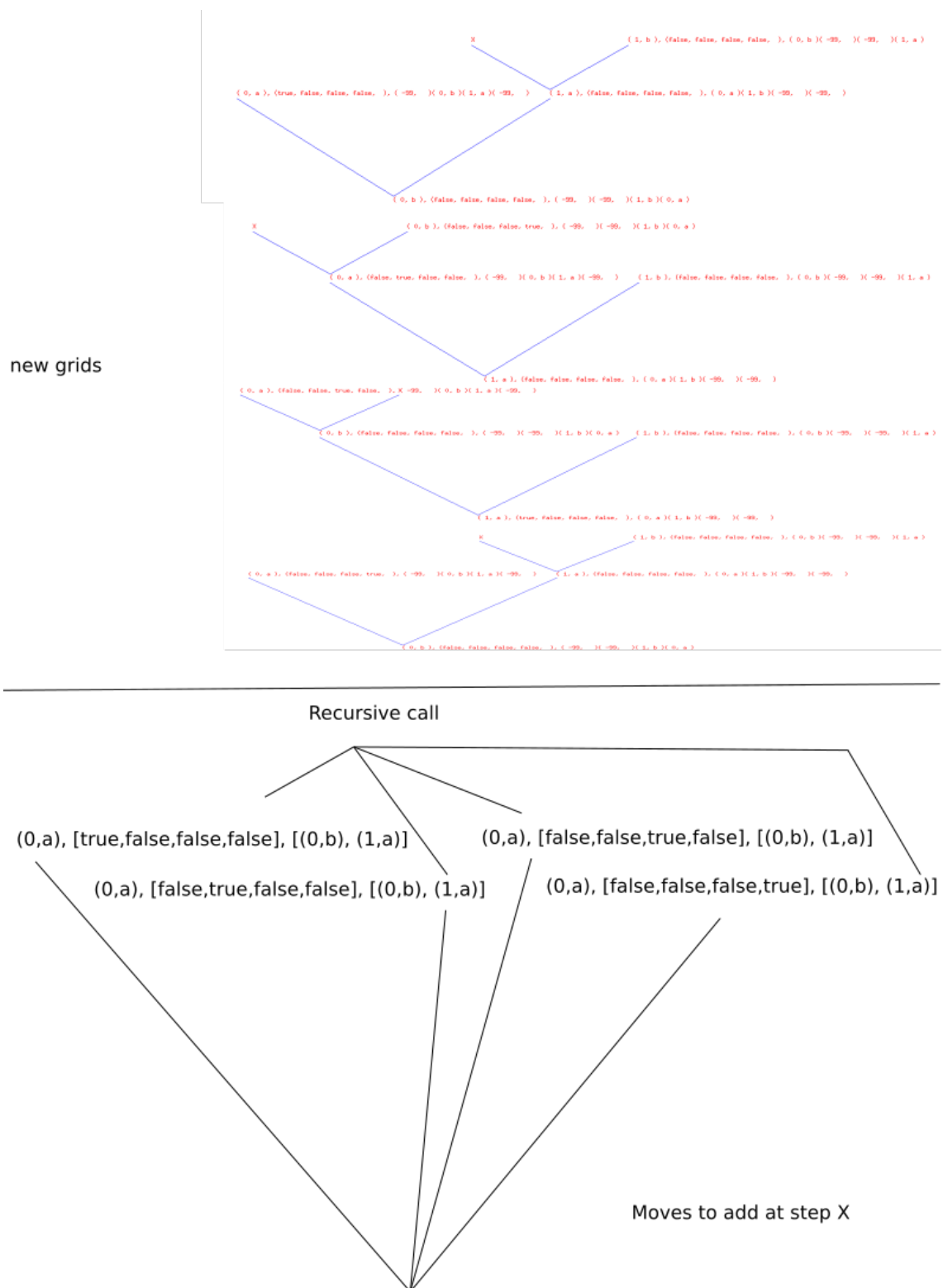


FIGURE 6.2 – Noeud à ajouter dans l'arbre des possible et grille mise à jour pour l'appel récursif

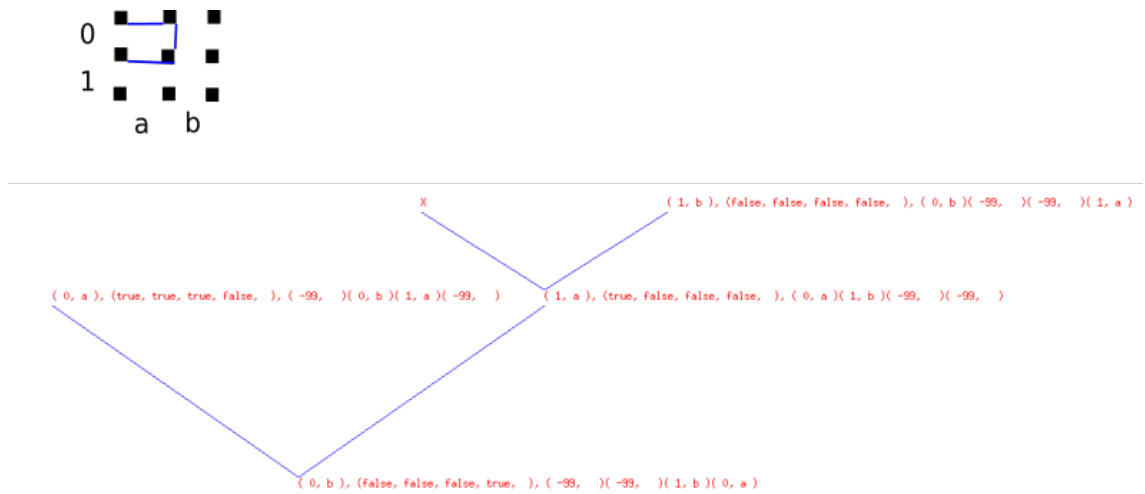


FIGURE 6.3 – Etat de la grille après les 3 coups joués.

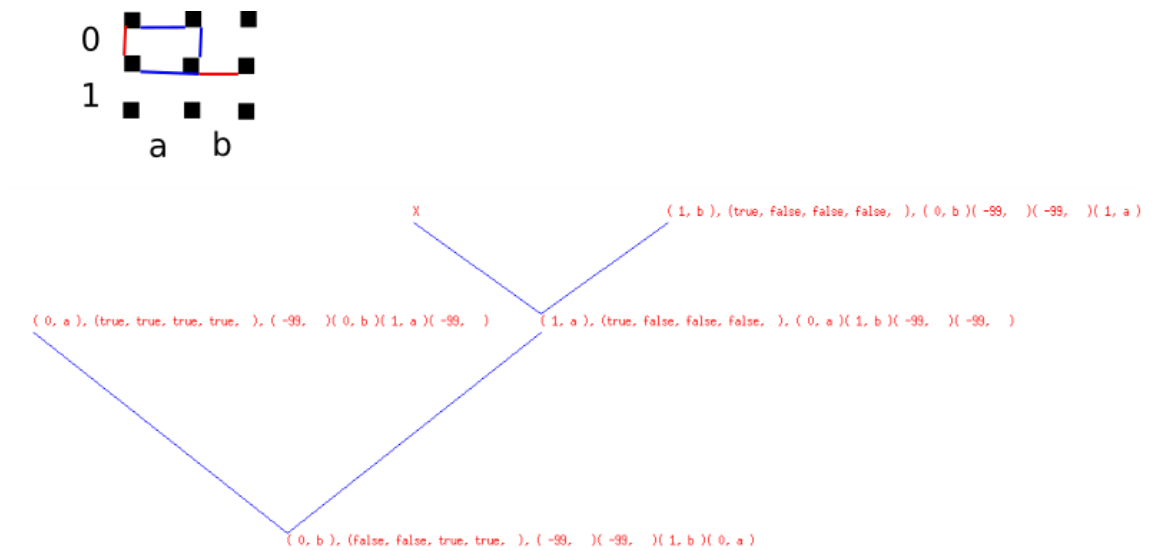


FIGURE 6.4 – Etat de la grille après que l'IA 0 aie joué.

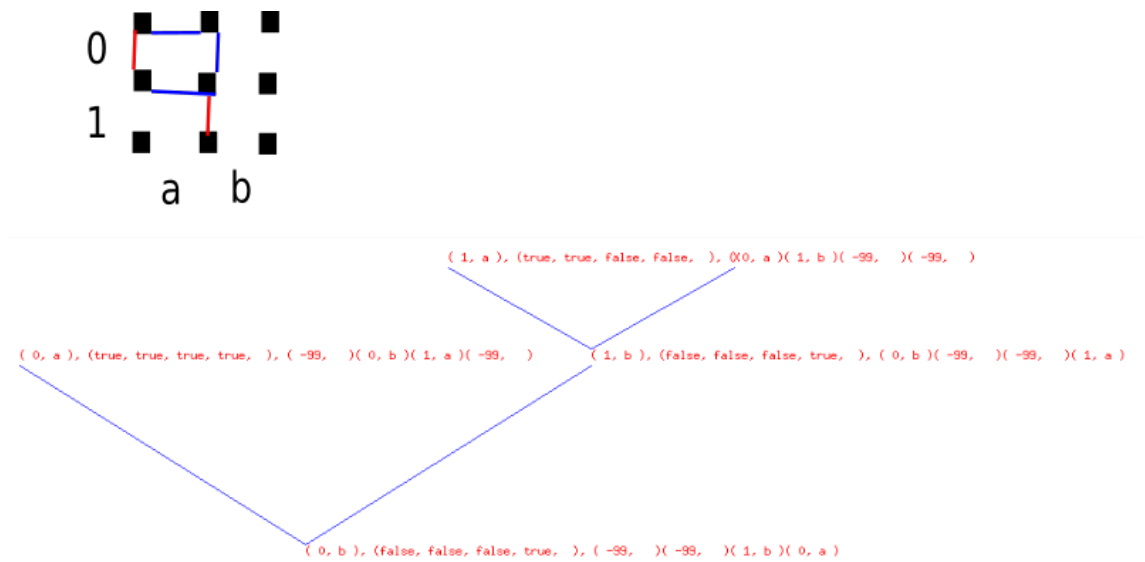


FIGURE 6.5 – Etat de la grille après que l'IA 1 aie joué.

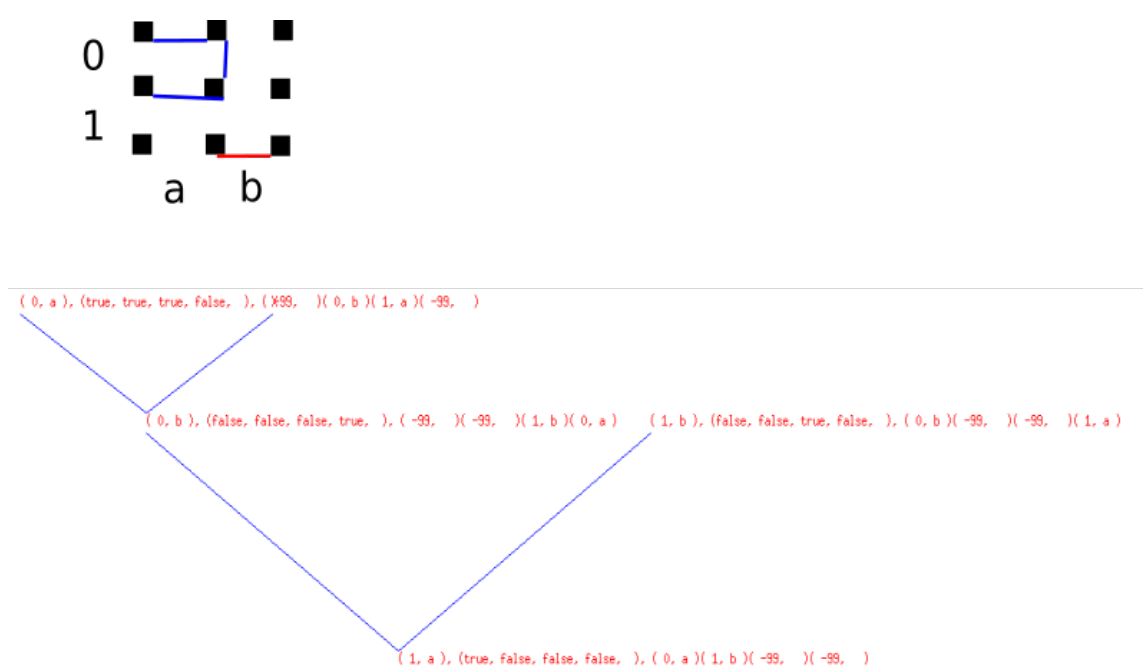


FIGURE 6.6 – Etat de la grille après que l'IA 2 aie joué.