Université Paris 8 de Vincennes à Saint-Denis
Département d'Informatique
UFR MITSIC
2, rue de la liberté
93526 Saint Denis CEDEX 02
Tél : +33(0)1 49 40 67 89

Recherche automatique d'heuristiques pour le General Game Playing

ALINE HUFSCHMITT
<alinehuf@gmail.com>

sous la direction de JEAN MÉHAT

Mémoire de Master 1
Informatique et Sciences Humaines

24 Juin 2013

Résumé

Dans ce mémoire, nous présentons une démarche pour générer automatiquement des heuristiques dans le domaine du *General Game Playing* (GGP).

Dans le cadre des compétitions de GGP organisées par Standford, les joueurs programmés reçoivent les règles de jeux originaux sous la forme de description en *Game Definition Langage* (GDL). Ces joueurs utilisent des algorithmes de parcours d'arbre comme le *Upper Confidence Bounds applied to Trees* (UCT) pour déterminer les coups les plus prometteurs pouvant mener à une victoire. Ces algorithmes nécessitent l'usage d'heuristiques pour évaluer les positions et guider la recherche vers les branches les plus prometteuses. Ces heuristiques peuvent être des connaissances propres au domaine (présence d'un plateau de jeu, de marques, de pièces, etc.) établies manuellement et calculées à partir de caractéristiques extraites des règles GDL. Dans notre étude, nous rejetons cette démarche qui fait usage d'un savoir propre au domaine et nous proposons une méthode susceptible de fournir des heuristiques automatiquement à partir de la seule description GDL des positions d'un jeu.

L'utilisation de réseaux de neurones pour la génération automatique d'heuristiques pose des problèmes notamment au niveau du choix de la topologie à utiliser. Nous proposons l'utilisation de l'algorithme de *Neuro-Evolution of Augmenting Topologies* (NEAT) susceptible de découvrir la topologie nécessaire pour la résolution de notre problème. Nous présentons donc le principe des algorithmes génétiques et le fonctionnement de NEAT.

Notre but est de générer automatiquement des heuristiques permettant de guider une exploration UCT à chaque position d'un jeu. Cependant, pour évaluer notre méthode de génération d'heuristiques, nous limitons notre étude à des positions terminales de jeux dont le score est connu.

Dans un premier temps, nous associons les faits GDL de ces positions à des pondérations. La somme des pondérations des faits vrais dans un état final donné permet d'en estimer le score. Nous utilisons un algorithme génétique pour faire évoluer ces pondérations et évaluons la capacité de notre population à prédire le score d'un échantillon de 93 jeux. Nous concluons sur la non-linéarité du problème à traiter et sur l'incapacité d'un simple algorithme génétique à créer des combinaisons de faits nécessaires à l'évaluation de positions de jeux comme le *Tictactoe*.

Dans un second temps, nous utilisons l'algorithme de *Neuro-Evolution of Aug-*

menting Topologies afin de faire évoluer non plus des vecteurs de pondérations, mais des réseaux de neurones. Nous fournissons en entrée de ces réseaux les valeurs booléennes des faits dans chaque position finale. Nous avons obtenu des réseaux de topologies efficaces pour la résolution de problèmes non-linéaires simples s'apparentant au problème du XOR, mais notre programme, fondé sur un exemple d'implémentation en C++ de Buckland [Buckland and Collins, 2002], n'a pas été en mesure de fournir des solutions élégantes et efficaces pour la résolution de problèmes complexes.

Nous concluons que NEAT s'avère une approche valable pour la génération automatique d'heuristiques, mais nous devons apporter des modifications à notre implémentation pour nous rapprocher du modèle original de Stanley [Stanley, 2004] et améliorer les résultats obtenus. Notre programme mis au point nous permettra d'évaluer des positions non-terminales et pourra générer automatiquement des heuristiques pour guider l'exploration UCT d'un joueur GGP.

Remerciements

Je remercie Monsieur Jean Méhat pour ses précieux conseils et sa disponibilité sans réserve.

Je remercie Monsieur Jean-Jacques Mariage pour son aide concernant les réseaux de neurones et ses conseils de méthodologie.

Je remercie Marc Hufschmitt et Maureen Chappuit pour leurs encouragements, leurs conseils et leur aide dans mes recherches.

Je remercie Jean-Philippe Hourcade, mon mari, pour ses encouragements et son soutien quotidien.

Je remercie mes parents pour leur soutien logistique sans lequel je n'aurais pas pu reprendre mes études.

Table des matières

1 Introduction

Nous présentons dans cette introduction les motivations qui nous ont conduits à réaliser un système de génération automatique d'heuristiques pour le *General Game Playing*. Nous commençons par présenter l'objectif des recherches dans le domaine du *General Game Playing*, puis les problèmes posés par la réalisation de joueurs génériques, enfin nous indiquons une piste pour la recherche d'heuristiques, l'utilisation de réseaux de neurones, qui n'a pu être explorée auparavant à cause des différents obstacles qui y sont liés. Nous proposons une solution susceptible de lever ces obstacles et permettant la recherche automatique d'heuristiques.

1.1 Le General Game Playing

Le *General Game Playing* est une des branches de l'intelligence artificielle qui vise à réaliser des programmes polyvalents capables de jouer à n'importe quel jeu. L'objectif qui peut paraître futile à première vue vise en fait à améliorer la compréhension que nous avons de l'intelligence. Il s'agit de créer des programmes capables d'analyser les règles d'un jeu inconnu, d'en comprendre les objectifs, de développer des stratégies gagnantes et d'y jouer avec expertise.

Pour stimuler la recherche dans ce domaine, des compétitions de *General Game Playing* sont organisées [Genesereth et al., 2005] pour confronter les programmes développés par différentes équipes. Pour décrire la large gamme de jeux étudiés, les concurrents utilisent le *Game Definition Language* (GDL). Ce langage de type Prolog, fondé sur la logique du premier ordre, présente les règles du jeu sous forme d'une liste de faits logiques et de théorèmes sur ces faits. Les joueurs programmés analysent ces faits pour établir la situation du jeu, connaître les différents coups légaux et les résultats en fin de partie.

L'objectif étant de jouer avec expertise, le but des chercheurs est de trouver des techniques pour savoir quel coup choisir à chaque étape du jeu. À cette fin, différentes techniques de parcours d'arbre ont été développées pour explorer chaque position du jeu et les différentes branches de possibilités qui dépendent du coup choisi. La racine de l'arbre représente l'état initial du jeu et l'objectif est d'atteindre la feuille correspondant à la fin de partie dont le score est le plus élevé possible pour le joueur.

Récemment, les techniques de *Monte Carlo Tree Search* (MCTS) [Browne et al., 2012] ont remplacé les algorithmes de type *Minimax* avec coupures *Alpha Beta* pour l'exploration d'arbre. Ces nouvelles techniques de MCTS améliorent considérablement l'efficacité des explorations d'arbre en proposant un équilibre entre *exploration*, recherche de nouvelles branches plus prometteuses présentant l'inconvénient d'être coûteuse en temps, et *exploitation*, examen plus approfondi d'une zone de l'arbre qui semble intéressante i.e. susceptible d'amener à un meilleur score.

1.2 L'évaluation des positions

Les techniques de Monte Carlo nécessitent cependant d'établir des heuristiques pour déterminer quelles sont les positions du jeu les plus favorables de manière à guider la recherche et éviter de perdre du temps à examiner des enchaînements de coups sans intérêt. Dans un jeu complexe où il n'est pas possible d'examiner l'enchaînement de tous les coups jusqu'à connaître le score qui leur correspond, il est nécessaire également de pouvoir estimer une position avec des connaissances de plus haut niveau.

Différents joueurs réalisés dans le cadre du *General Game Playing* [Clune, 2007; Schiffel and Thielscher, 2007a] utilisent des heuristiques déterminées plus ou moins à l'avance par le programmeur [1]. Ces heuristiques ne sont pas toujours adaptées et rendent le joueur performant uniquement pour certains types de jeux et très mauvais pour d'autres [2].

Pour la recherche d'heuristiques, il nous paraît nécessaire de mettre au point une technique d'évaluation générique n'utilisant pas de connaissances propres au domaine. Différentes démarches [Kirci et al., 2011; Sharma et al., 2008b] proposent de déduire les heuristiques directement des règles GDL décrivant les jeux. Quand l'exploration d'un enchaînement de coups débouche sur une fin de partie dont le score est connu, les différents faits peuvent être pondérés pour estimer leur contribution à ce score.

Samuel [1959], utilise ainsi un groupe d'heuristiques établies manuellement et pondérées pour guider son programme jouant aux *Dames anglaises*. Une approche similaire est utilisée par Sharma et al. [2008b] dans le cadre du GGP en associant des pondérations aux faits GDL. Cependant, ils s'accordent sur le fait que le problème à résoudre est non linéaire. En effet, les faits interagissent entre eux et une simple pon-

1. Ces heuristiques peuvent être, par exemple, la comparaison du nombre de pièces en possession de chaque joueur ou l'évaluation de la mobilité des pièces.

2. Lors de la compétition de GGP à Boston en 2006, les heuristiques choisies par *ClunePlayer* n'étaient pas adaptées au jeu de *Cylinder checkers*, un jeu de *Dames anglaises* joué sur un plateau cylindrique. Après une phase d'analyse du jeu, *ClunePlayer* a choisi plusieurs heuristiques dont le nombre de pièces possédées par chaque joueur et leur mobilité. Cependant la mobilité, considérée comme plus importante, est devenue l'heuristique dominante de la fonction d'évaluation. Pour limiter les mouvements de l'adversaire *ClunePlayer* a sacrifié ses propres pièces pour contraindre celui-ci à les prendre. Le programme de Clune a donc très mal joué et cela a évidement débouché sur une victoire de son adversaire. [Clune, 2008]

dération est insuffisante ; il est nécessaire de découvrir quelles sont les combinaisons logiques de faits qui sont signifiantes[3]. Samuel [1967] propose une autre démarche qui consiste à réaliser manuellement différentes combinaisons d'heuristiques, mais cette démarche n'est pas applicable dans le cadre du GGP. La combinatoire résultant des combinaisons de faits GDL est bien trop importante.

1.3 Problèmes posés par la recherche d'heuristiques

Plusieurs auteurs de ces recherches [Sharma et al., 2008b; Utgoff, 2001] suggèrent l'utilisation de réseaux de neurones pour réaliser cette recherche automatique d'heuristiques et découvrir la logique sous-jacente à un jeu. Cependant, l'utilisation des réseaux de neurones se heurte à différents obstacles : il est nécessaire de découvrir la topologie adaptée à la résolution du problème et dans le cadre du *General Game Playing* le problème est justement très variable ; le temps d'apprentissage pour qu'un réseau apprenne à résoudre un problème est également très long.

L'utilisation des algorithmes génétiques pour explorer les combinaisons de faits signifiantes est également évoquée, mais leur convergence vers une solution est très lente et leur usage n'est pas conseillé [Utgoff, 2001] pour le *General Game Playing*.

Cependant, la technique de *Neuro-Evolution of Augmenting Topologies* (NEAT) de Stanley [Stanley, 2004] apporte une solution à ces problèmes. En faisant évoluer une population de réseaux de neurones avec un algorithme génétique à partir d'une topologie minimale, son système permet de minimiser le temps de calcul en découvrant progressivement la topologie de réseaux de neurones adaptée à la résolution d'un problème.

En utilisant cette technique nous proposons d'analyser les faits GDL d'une position d'un jeu et d'en réaliser une évaluation. Cette technique, une fois mise au point, pourra permettre la génération automatique d'heuristiques pour l'évaluation de positions dans le domaine du *General Game Playing*.

1.4 Plan de lecture

Dans le chapitre 2 nous présentons le domaine du *General Game Playing* et les problèmes posés par la recherche d'heuristiques pour le parcours des arbres des jeux. Nous montrons que l'utilisation de réseaux de neurones pour cette recherche d'heuristiques se heurte à des problèmes de choix de topologie et de lenteur de calcul.

3. Considérons par exemple le jeu du *Tictactoe* (voir l'annexe A pour un rappel des règles de ce jeu). C'est l'alignement de 3 symboles qui apporte la victoire à l'un des joueurs. Chaque symbole placé dans une case est un fait logique qui peut être décrit en GDL, par exemple `(cell 1 3 x)`. Il est donc nécessaire que le joueur programmé soit en mesure de découvrir les différentes combinaisons de trois faits amenant à une victoire.

Nous consacrons le chapitre 3 à la présentation du principe des algorithmes génétiques et montrons comment ils peuvent être utilisés pour faire évoluer la topologie de réseaux de neurones par une technique nommée *Neuro-Evolution of Augmenting Topologies*. Cette approche peut être utilisée pour la recherche automatique d'heuristiques.

L'objectif de notre recherche est de générer automatiquement des heuristiques pour l'évaluation des positions à chaque étape d'un jeu. Le chapitre 4 présente les données que nous avons générées pour nos tests et nos expérimentations préliminaires effectuées avec les algorithmes génétiques.

Dans le chapitre 5 nous présentons notre implémentation de l'algorithme de *Neuro-Evolution of Augmenting Topologies*. Nous utilisons cette technique pour évaluer des positions terminales et présentons les résultats obtenus lors de nos tests.

Dans la conclusion nous faisons le bilan de nos expérimentations et présentons les améliorations à apporter à notre implémentation pour la rendre réellement efficace. Notre but étant l'évaluation de positions non-terminales, nous indiquons également les futurs développements que nous allons ajouter à notre programme pour atteindre cet objectif.

2 General Game Playing (GGP)

Ce chapitre commence par une brève présentation du *General Game Playing*. Nous parlons des compétitions organisées dans le but de faire progresser ce domaine de recherche et du langage de définition permettant de décrire les règles des jeux étudiés. Nous présentons ensuite des techniques d'exploration d'arbre utilisées pour explorer le déroulement des jeux et enfin nous concluons avec l'utilisation d'heuristiques pour guider et améliorer cette recherche.

2.1 Présentation du GGP

Le *General Game Playing* (GGP) est un champ de recherche faisant partie de l'intelligence artificielle et visant à réaliser des programmes capables de jouer à n'importe quel jeu en partant uniquement de la description de ses règles. Le concept de *General Game Playing* est assez récent [1] et n'a pris sa véritable ampleur qu'avec le projet initié par le *Stanford Logic Group* de l'université de Standford. Leur projet d'une compétition de *General Game Playing* [Genesereth et al., 2005] a débouché également sur la mise en place d'une plateforme de jeu en ligne sur laquelle les différents logiciels développés peuvent s'affronter.

La recherche dans le domaine de la programmation des jeux était motivée au départ par l'idée que la victoire d'un programme sur un joueur humain pourrait être une mesure fiable de son *intelligence* telle que recherchée dans le domaine de l'IA. Pell, dans sa thèse présentant le paradigme de *Metagame* [Pell, 1993], illustre clairement cette idée par une citation de Emanuel Lasker :

1. Dans sa thèse, Quenault [2010] indique que l'étude des jeux représente tout un pan de l'intelligence artificielle. Les premiers programmes étaient des modèles spécialisés capables de battre des joueurs humains dans des jeux particuliers. Le principe de General Game Playing est apparu progressivement par une complexification croissante des modèles mis en place. Un système de jeu générique est défini par la présence de programmes joueurs réunis grâce à un programme capable de les mettre en relation et de les informer régulièrement sur l'état du jeu ; ceci nécessite l'usage d'un langage générique pour décrire le jeu. Quenault [2010] présente les travaux de Pitrat comme la première tentative pour créer un langage générique de description de règles de jeux. Le langage, que Pitrat [1971] décrit, permet de spécifier les règles de jeux à deux joueurs sur plateaux rectangulaires comme le *Tictactoe* ou les *Échecs*. Pitrat [1976] décrit également un programme capable d'apprendre des procédures ou enchaînements de coups favorables aux échecs. Les travaux de Pell [1993] sont similaires à ceux de Pitrat et toujours limités à une catégorie restreinte de jeux de plateau.

> *« In mathematics, if I find a new approach to a problem, another mathematician might claim that he has a better, more elegant solution. In chess, if anybody claims he is better than I, I can checkmate him. »*

Avec la victoire de Deep Blue sur Garry Kasparov en 1997 et la réalisation d'autres programmes spécialisés comme Chinook (*Dames anglaises*) cette idée a montré ses limites : ces succès sont plus le résultat de l'intelligence du programmeur que d'une quelconque avancée en intelligence artificielle. Pell rappelle cela dans l'introduction de sa thèse :

> *« we have been able to produce strong programs for some games through specialized engineering methods, the extreme case being special-purpose hardware, and through analysis of the games by humans instead of by programs themselves. Consequently, it now appears that increased understanding and automation of intelligent processing is neither necessary nor sufficient for strong performance in game-playing. That is, it appears that we can construct strong game-playing programs without doing much of interest from an AI perspective, and conversely, we can make significant advances in AI that do not result in strong game-playing programs. »* [Pell, 1993]

Pell propose alors de changer les données du problème de manière à forcer les chercheurs à concevoir des programmes faisant preuve d'une intelligence plus générale : en ne fournissant que les règles du jeu[2], le programme devrait alors être capable de les analyser pour en déduire une stratégie. Pour Pell, comme pour Clune [Clune, 2008], la généralité semble être une caractéristique fondamentale de l'intelligence et pour illustrer sa pensée il cite Robert A. Heinlein :

> *« A human being should be able to change a diaper, plan an invasion, butcher a hog, conn a ship, design a building, write a sonnet, balance ac- counts, build a wall, set a bone, comfort the dying, take orders, give orders, cooperate, act alone, solve equations, analyze a new problem, pitch ma- nure, program a computer, cook a tasty meal, fight efficiently, die gallantly. Specialization is for insects. »*

Cette citation sera d'ailleurs reprise dans un article de 2005 [Genesereth et al., 2005] présentant la première compétition de General Game Playing.

2.2 Game Definition Language (GDL)

Afin de pouvoir travailler sur une base commune et permettre aux différents joueurs de communiquer entre eux (par l'intermédiaire de l'arbitre dans le cas du GGP), il était nécessaire de concevoir un langage commun pour décrire les jeux. Plusieurs langages ont été proposés : Pell dans le cadre de son *Metagamer* [Pell, 1993]

2. Précisons que pour éviter la tentation d'injecter dans le programme un savoir propre au domaine, son auteur ne doit pas avoir connaissance du jeu en question. Le programme doit être capable de traiter une grande diversité de jeux inconnus uniquement à partir des règles qui les décrivent.

propose une grammaire décrivant les jeux de type échecs. Kaiser [Kaiser, 2005] propose RGL, un langage de définition de jeux fondé sur des expressions régulières décrivant des chemins dans un graphe. Il rappelle également l'existence de différentes notations propres à des jeux spécifiques, comme la *Portable Game Notation* (PGN) pour les mouvements aux échecs ou le *Smart Game Format* (SGF) surtout utilisé pour le jeu de Go, et évoque d'autres tentatives pour créer des langages plus ou moins génériques de description des jeux.

C'est finalement le *Game Definition Language* (GDL), un langage de description des règles des jeux proposé par Michael Genesereth dans le cadre du projet de compétitions de General Game Playing initié par Standford qui semble s'être imposé grâce à sa généralité. Conçu pour les jeux déterministes à information complète, Thielscher [Thielscher, 2011] montre néanmoins qu'il est suffisamment général pour décrire des jeux très différents et peut être facilement étendu aux jeux à information incomplète et soumis à une part de hasard. Il propose d'ailleurs dans ce but une version 2 du GDL [Thielscher, 2010].

Ce *Game Definition Language* (version 1) dont la description est donnée par les créateurs de la compétition [Genesereth et al., 2005] et sur le site de Standford [Group, 2013], est un langage logique similaire à Prolog. Il est fondé sur la logique du premier ordre sans la notion d'arithmétique, bien que les nombres de 0 à 100 soient définis comme constantes pour décrire les scores en fin de partie. Certaines restrictions permettent d'assurer que toutes les règles logiques d'un jeu décrit en GDL sont totalement décidables. Un exemple de jeu décrit en GDL est donné à la figure 2.1. Le style du GDL est purement déclaratif et il possède quelques mots réservés pour décrire le jeu [3] :

- `(role a)` signifie que `a` est un joueur dans le jeu.
- `(init p)` signifie que la proposition `p` est vraie à l'état initial i.e. au début du match.
- `(true p)` signifie que la proposition `p` est vraie à l'état courant.
- `(does r a)` signifie que le joueur `r` fait l'action `a` dans l'état courant.
- `(next p)` signifie que la proposition `p` sera vraie à l'état suivant.
- `(legal r a)` signifie que l'action `a` est un *coup légal* pour le joueur `r` dans l'état courant.
- `(goal r n)` signifie que le joueur `r` obtient la récompense `n`.
- `terminal` signifie que l'état courant est un état final.

D'autre termes constituent le langage logique :

- `(<= x y0 y1 ... yn)` signifie que x implique les faits y0 à yn.
- `?x` signifie que x est une variable.

3. D'autres termes ont été ajoutés par la suite comme `(base a)` ou `(position a)` qui permettent de préciser qu'un fait est une position, et `(input r a)` ou `(action r a)` que `a` peut être un coup légal dans le jeu pour le joueur `r`. `(base a)` et `(input r a)` font partie du standard utilisé par Stanford. Ces termes ne sont pas présents dans le corpus de jeux que nous avons utilisé et qui a été constitué pendant les premières années de la compétition. La version 2 de GDL utilise également les termes `(sees r f)` et `random` pour indiquer respectivement un fait `f` perçu par le joueur `r` dans la position suivante et un élément soumis au hasard.

- `(not x)` permet d'obtenir la négation d'un fait.
- `(distinct x y)` indique que les éléments x et y ne peuvent être identiques.

2.3 Compétitions de GGP

Depuis 2005, une compétition est organisée chaque année à l'occasion de la conférence de l'association pour l'avancement de l'intelligence artificielle *AAAI Conference* pour récompenser les meilleurs programmes réalisés (voir figure 2.2). Les conditions de la compétition ont évolué depuis sa conception, par exemple, une partie de la compétition se déroule désormais en ligne, seule la finale se dispute à l'issue de la conférence AAAI ou IJCAI selon les années. Cependant le principe reste globalement le même.

Une machine-arbitre délivre aux machines des joueurs les règles d'un jeu inconnu des participants ; elle indique à chaque programme quel rôle il doit tenir dans le jeu. La diversité des jeux proposés est grande : ils peuvent être pour un seul joueur, deux ou multi-joueurs, à coups simultanés ou alternés avec une récompense numérique pour chaque joueur définie entre 0 et 100. Ces jeux peuvent également être à somme nulle ou non-nulle. [4]. Chaque programme joueur dispose d'un temps limité pour analyser les règles. À la fin de ce temps de préparation, l'arbitre donne aux programmes un temps limite pour indiquer le coup qu'ils décident de jouer. À la fin de chaque délai, les différents joueurs indiquent l'action choisie. Tous les joueurs jouent à chaque tour mais, pour les jeux à coups alternés, un seul joueur donne un coup *actif*, les autres se contentent d'indiquer un coup sans effet. Pour dissuader les interventions humaines lors des parties en ligne, les règles du jeu sont brouillées de manière à ce qu'aucun mot clef comme *cell* ou *succ* ne puisse servir de guide par son sens. À la place, des mots dénués de sens comme *foo* et *bar* sont utilisés.

Les résultats de cette compétition sont une assez mauvaise évaluation de la réelle valeur des programmes réalisés. Toutes sortes de problèmes peuvent survenir pendant les compétitions : problèmes d'ingénierie ou défaillances des machines qui n'ont pas forcément de rapport direct avec la qualité des modèles développés. Le hasard du tirage des adversaires qui devront s'affronter peut également biaiser les résultats. Il est à noter par exemple que *Ary* n'a jamais affronté *CadiaPlayer* pendant une de ces compétitions. Le nombre de matchs pendant cette compétition est également très faible et il est difficile d'obtenir une idée fiable de la performance d'un programme sur un nombre si restreint d'épreuves. Les programmes réalisés ne sont pas forcément en adéquation avec les jeux présentés une année donnée. La nature des jeux peut favoriser certains concurrents : en 2010, par exemple, aucun jeu pour un seul joueur n'a été proposé.

4. Dans un jeu à somme nulle, ce que gagne un joueur est perdu par l'autre, les gains de l'un correspondent aux pertes de l'autre (exemple : score 100/0, 50/50 ou 0/100), dans un jeu à somme non-nulle, chaque joueur peut accumuler des points indépendamment de l'autre (exemple : score 25/25, 100/30, 75/50)

```
(role player1)
(role player2)

(init (heap a 1))
(init (heap b 5))
(init (heap c 4))
(init (heap d 2))

(init (control player1))

(<= (legal ?p noop) (true (control ?x)) (role ?p) (distinct ?x ?p))
(<= (legal ?p (reduce ?x ?n)) (true (control ?p)) (true (heap ?x ?m))
             (smaller ?n ?m))

(<= (next (heap ?x ?n)) (does ?p (reduce ?x ?n)))
(<= (next (heap ?x ?n)) (true (heap ?x ?n)) (does ?p (reduce ?y ?m))
                        (distinct ?x ?y))
(<= (next (control ?p2)) (true (control ?p1)) (next_player ?p1 ?p2))

(<= terminal (true (heap a 0)) (true (heap b 0)) (true (heap c 0))
             (true (heap d 0)))

(<= (goal ?p 0) (true (control ?p)))
(<= (goal ?p 100) (true (control ?p1)) (next_player ?p ?p1))

(<= (smaller ?x ?y) (succ ?x ?y))
(<= (smaller ?x ?y) (succ ?x ?z) (smaller ?z ?y))

(next_player player1 player2)
(next_player player2 player1)

(succ 0 1)
(succ 1 2)
...
(succ 25 26)
(succ 26 27)
```

FIGURE 2.1 – Description du jeu de nim en GDL.

Année	Programme	Développeur
2005	ClunePlayer	Jim Clune
2006	FluxPlayer	Stephan Schiffel, Michael Thielscher
2007	CadiaPlayer	Yngvi Bjornsson, Hilmar Finnsson
2008	CadiaPlayer	Yngvi Bjornsson, Hilmar Finnsson
2009	Ary	Jean Mehat
2010	Ary	Jean Mehat
2011	TurboTurtle	Sam Schreiber
2012	CadiaPlayer	Yngvi Bjornsson, Hilmar Finnsson

FIGURE 2.2 – Gagnants de la compétition de *General Game Playing* organisée chaque année depuis 2005.

Cependant, malgré tous ces inconvénients, les compétitions de *General Game Playing* de Stanford sont quasiment les seules rencontres internationales organisées dans le monde et les résultats de ces compétitions sont à peu près [5] les seules évaluations disponibles pour se faire une idée de la validité des démarches utilisées.

Depuis sa création en 2005, la compétition de *General Game Playing* a permis de révéler différentes approches intéressantes dont celles de Kuhlmann, Dresner et Stone [Kuhlmann et al., 2006], Clune [Clune, 2007], Schiffel and Thielscher [Schiffel and Thielscher, 2007a], Finnsson and Björnsson [Finnsson and Björnsson, 2008]. On peut remarquer deux type d'approches distinctes, les plus récentes sont fondées sur les algorithmes de Monte Carlo et ses variantes [Finnsson and Björnsson, 2011; Méhat and Cazenave, 2008] qui ont considérablement amélioré la performance des explorations des arbres des jeux. Les plus anciennes mettaient l'accent sur l'extraction de connaissances et la création de fonctions d'évaluation fondées sur des heuristiques, néanmoins elles utilisaient également des algorithmes de parcours d'arbre de la famille *Mini-Max* [Kuhlmann et al., 2006; Schiffel and Thielscher, 2007b; Clune, 2007; Kaiser, 2007; Kirci et al., 2011].

2.4 Exploration d'arbre

Un jeu peut être représenté sous la forme d'un arbre dont chaque nœud représente un état de la partie à un moment donné et chaque lien représente un coup légal i.e. un coup autorisé par les règles du jeu. La racine représente l'état initial du jeu et

5. Divers matchs amicaux sont disputés en ligne notamment sur les serveurs de Standford et de Dresde. On peut noter l'organisation de quelques compétitions nationales ou limitées à une université donnée. La *German General Game Playing Competition* (GGGPC) a été organisée pour la première fois en 2009 ; les participants étaient tous des chercheurs d'universités allemandes. En 2011 une compétition ouverte aux équipes internationales s'est tenue pendant la 34ème Conférence Allemande d'intelligence artificielle (KI-11) ; il est à noter que le GDL-II y a été utilisé pour la première fois afin de proposer des jeux à information incomplète et comportant une part de hasard. Les compétitions de GGP organisées depuis 2006 à l'université de Dresde sont organisées pour leurs étudiants. Une rencontre entre l'université de Dresde et celle de Reykjavik a été organisée en 2009 entre 6 équipes d'étudiants.

chaque feuille représente un état terminal où le score final peut enfin être connu.

À chaque nœud, le joueur doit savoir comment choisir le coup qui lui sera le plus favorable pour obtenir le meilleur score possible. Dans ce but, diverses techniques ont été mises au point, à commencer par celles de la famille *Mini-Max* que nous exposons dans un premier temps. Ces méthodes étaient utilisées par les premiers joueurs programmés pour le *General Game Playing* et deux d'entre eux, *Clune PLayer* [Clune, 2007] et *Flux Player* [Schiffel and Thielscher, 2007a,b] ont gagné les compétitions de 2005 et 2006.

Cependant depuis 2007, les techniques de Monte-Carlo ont été appliquées avec grand succès à l'exploration d'arbre et elle se sont imposées avec la victoire de plusieurs joueurs les utilisant : *Cadia Player* [Finnsson and Björnsson, 2011; Finnsson, 2012], *Ary* [Méhat and Cazenave, 2008, 2010] et *Turbo Turtle*. Nous verrons ces techniques dans un second temps et quelques améliorations utilisées par ces joueurs de GGP.

2.4.1 Famille Mini-Max

Avant l'apparition des techniques de Monte-Carlo (voir paragraphe 2.4.2), ce sont les algorithmes de la famille *Mini-Max* qui étaient utilisés pour le parcours d'arbre. *Mini-Max* consiste à chaque étape à maximiser le score du joueur courant (joueur 1) ou à minimiser celui de l'adversaire (joueur 2). Il part du principe que l'adversaire va choisir judicieusement à chaque étape du jeu le coup qui va minimiser le score de l'adversaire : ce qui implique à la fois que l'adversaire est capable de bien jouer et que minimiser le score du joueur 1 est directement corrélé à l'augmentation de son propre score, donc que le jeu est à somme nulle.

Une branche de l'arbre est d'abord parcourue en profondeur jusqu'à un état terminal et le score du joueur 1, associé à ce nœud, est relevé (voir étape C sur la figure 2.3). On considère que l'adversaire, à l'étape B, essaye d'optimiser son propre score, donc il va choisir les branches qui amènent au score minimal pour le joueur 1. À l'étape A le joueur 1 peut donc déduire qu'il a intérêt à choisir la branche de gauche pour assurer un score de 40 car, si il vise le score 100 (au milieu) mais que le joueur 2 joue bien, il risque de se retrouver avec seulement 25.

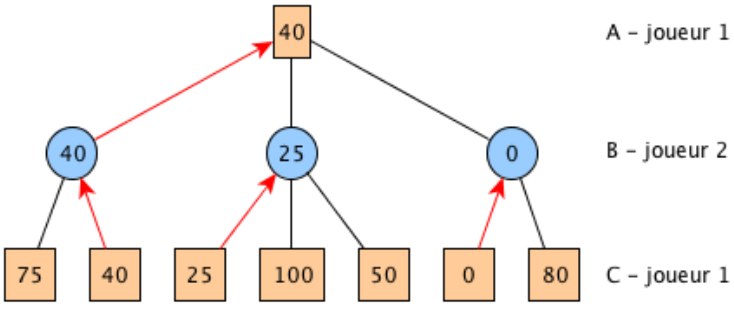


FIGURE 2.3 – Exploration d'un arbre avec *Mini-Max*

Néga-Max est une variante de *Mini-Max* qui apporte essentiellement une simplification du code. Le principe est le même, mais il repose sur le fait que le score du joueur 1 est la valeur du handicap du joueur 2 : $max(score) = -min(-score)$ (voir figure 2.4). En lieu et place de deux fonctions de maximisation et de minimisation, on peut avoir une seule fonction :

```
negamax(position) :
    si la partie est terminée :
        renvoyer le score (score du joueur 1 ou handicap du joueur 2)
    res ← −∞
    pour chaque coup :
        tmp ← - negamax(nouvelle_position)
        if (tmp > res)
            res ← tmp
    renvoyer res
```

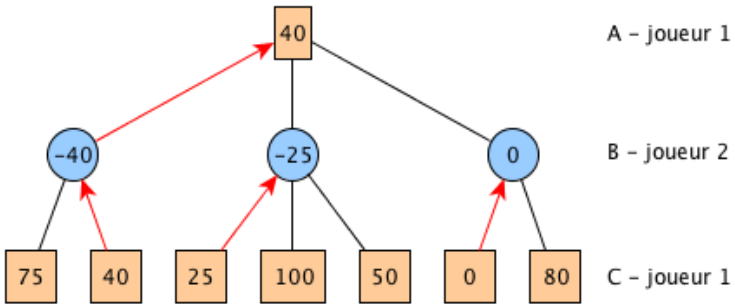


FIGURE 2.4 – Exploration d'un arbre avec *Néga-Max*

Mini-Max implique l'exploration complète de l'arbre du jeu, ce qui est impossible pour des jeux un tant soit peu intéressants et donc complexes [6]. Afin de simplifier l'exploration de l'arbre, *Alpha-Beta* permet de couper certaines branches dont il semble évident qu'elles ne mèneront pas à un meilleur score. À chaque position

6. L'intérêt d'un jeu est une notion relative. Certains jeux très simples peuvent s'avérer intéressants pour des joueurs humains en raison de la récompense (financière ou autre) qui en constitue l'enjeu. Un jeu très simple comme *Monty Hall*, dans lequel trois portes masquent deux récompenses sans intérêt et un lot prestigieux, peut également intéresser le joueur humain grâce à l'espoir d'obtenir le troisième lot, mais pose également un casse-tête probabiliste qui a suscité l'intérêt des mathématiciens (cf : *The Monty Hall Problem : Beyond Closed Doors* de Robert Deaves). Pour un jeu comme le *Tictactoe*, lorsque les deux joueurs ont compris la mécanique du jeu, l'intérêt subsiste uniquement dans la possibilité de déconcentrer l'adversaire pour le piéger. Sinon la partie s'achève invariablement par un match null et l'intérêt du jeu disparaît. D'un autre côté certains jeux complexes peuvent s'avérer dénués d'intérêt. C'est le cas du jeu *Add Game* joué sur un plateau de 2x2 cases : les règles sont ridiculement complexes alors que le jeu est trivial.
La complexité à laquelle nous faisons référence n'est pas celle provenant de l'apprentissage du jeu i.e. la complexité des règles, mais celle qui résulte du grand nombre de coups possibles et de la diversité des stratégies qui en découle. Dans des jeux présentant cette complexité, c'est dans l'exploration de ces possibilités et la mise au point de stratégies que l'intérêt réside.

dans le jeu, le joueur courant tente d'évaluer entre quelles bornes minimales et maximales se situe son score. Ces bornes sont initialisées par défaut à $-\infty$ et $+\infty$, mais elles peuvent être initialisées en fonction des bornes connues du score pour le jeu testé. Une branche est explorée jusqu'à une fin de partie et le score est relevé (voir figure 2.5). Le score est alors remonté dans les nœuds de l'arbre et les bornes minimales et maximales sont ajustées en conséquence. Sur la figure 2.5 on voit par exemple que l'exploration de la branche de gauche assure au joueur 1 un score compris entre 40 et $+\infty$, l'exploration partielle des deux autres branches indique que le score sera de ce côté en dessous de 25 ou en dessous de 0, il est donc inutile d'explorer le reste de ces branches puisqu'il est évident qu'elles ne peuvent apporter un score meilleur.

La simplification vue pour *Néga-Max* peut également être adaptée à *Alpha-Beta*.

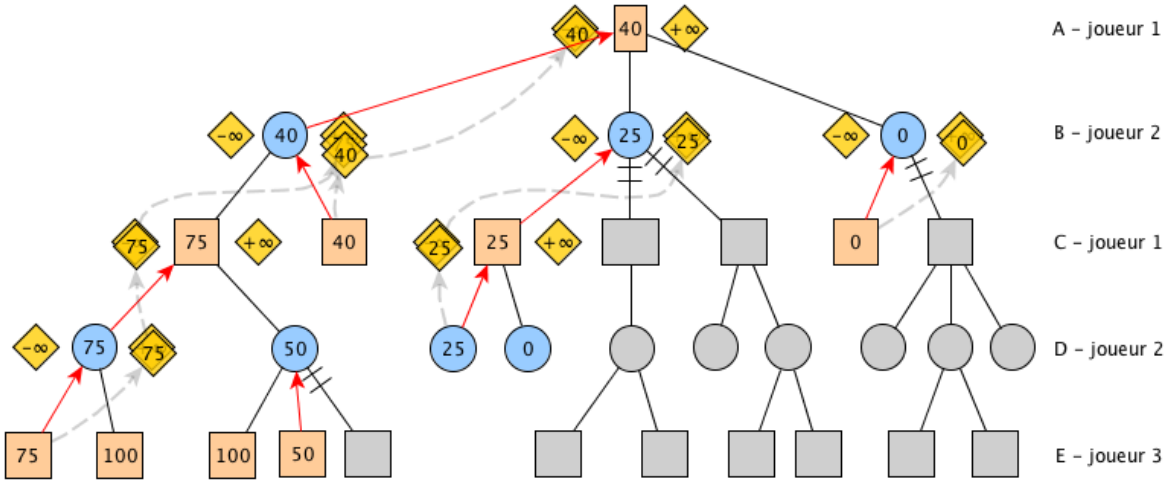


FIGURE 2.5 – Élagage *Alpha-Beta* $(\alpha\beta)$

Ces deux techniques ont été utilisées par les premiers programmes de General Game Playing, par exemple le *Metagamer* de Pell capable de jouer à un groupe de jeux de plateau de type échecs. Pell utilise en plus la technique de l'approfondissement itératif qui consiste à explorer l'arbre en largeur d'abord et jusqu'à une profondeur donnée. Les nœuds non terminaux sont alors évalués avec différentes heuristiques (nombre de pions en possession des joueurs, qui contrôle le centre, etc.). Si le programme dispose de temps pour affiner son évaluation, il explore l'arbre un peu plus profondément et remonte sa nouvelle évaluation dans les nœuds précédents pour mettre à jour leur évaluation. Un classement des coups est effectué pour examiner en premier les coups les plus prometteurs ; ceci permet à *Alpha-Beta* d'être plus efficace en favorisant les coupes.

Kuhlmann, Dresner et Stone [Kuhlmann et al., 2006] et Clune [Clune, 2007] utilisent également *MiniMax* et *Alpha-Beta* pour leur joueurs respectifs *UTexas LARG* et *Clune Player* et y ajoutent le principe des tables de transposition. Ces tables permettent de mémoriser les différentes positions du jeu et leur évaluation de manière à ne pas devoir réévaluer une position qui peut se présenter au bout de différentes

successions de coups. Pour retrouver rapidement une position et son évaluation, les positions sont associées à une clef de hashage à la Zobrist.

Le joueur de Schiffel et Thielscher, nommé *FluxPlayer* [Schiffel and Thielscher, 2007a,b], utilise également ces techniques classiques d'élagage *Alpha-Beta* et des tables de transposition avec une recherche itérative en profondeur d'abord.

2.4.2 Techniques de Monte Carlo

Le principe de base de la technique de *Monte Carlo* brille par sa simplicité. À chaque position du jeu, la liste des coups légaux est établie pour le joueur 1 dont le score doit être maximisé. Un coup est tiré au hasard dans cette liste et un *playout* est réalisé. Le *playout* est une simulation de la partie : tous les coups suivants sont joués au hasard, dans le respect des règles du jeu, jusqu'à atteindre un état terminal i.e. jusqu'à ce la partie soit finie et qu'un score puisse être attribué à chaque joueur. Le score obtenu pour le joueur 1 est alors reporté dans une table en correspondance avec le premier coup choisi. Les simulations se succèdent de manière à ce que chaque coup légal de la liste de départ se voit attribuer un score moyen. Un temps limite est accordé pour la réalisation de ces simulations : à la fin du temps imparti, le coup associé avec le gain moyen le plus fort est joué.

Cependant, cette utilisation basique peut facilement amener à un mauvais choix. Par exemple, si une branche de l'arbre du jeu mène à un ensemble de score moyens (50/100) tandis que l'autre branche amène à un seul score optimal (100/100) et une grande quantité de défaites (0/100), la moyenne obtenue sur la première branche sera plus élevée et le coup choisi annihilera tout espoir d'aboutir au score optimal.

Pour remédier à ce problème, les méthodes de *Monte Carlo Tree Search (MCTS)* proposent une manière plus efficace d'explorer l'arbre du jeu. Un arbre est construit localement de manière incrémentale et asymétrique [Browne et al., 2012]. À chaque itération un nœud est choisi parmi les nœuds déjà explorés et un nouveau nœud est ajouté selon une stratégie donnée (*TreePolicy*). À partir de ce nœud une simulation est effectuée qui peut être guidée et non effectuée au hasard.

Par exemple, la version actuelle de *CadiaPlayer* (2011-2012) [Finnsson and Björnsson, 2011; Finnsson, 2012] utilise UCT pour choisir un nœud de départ pour l'exploration de l'arbre et pendant les playout, les nœuds ne sont pas choisis selon un hasard uniforme, mais selon la distribution de Gibbs (ou Boltzmann) :

$$P(a) = \frac{\mathbf{e}^{Qh(a)/\tau}}{\sum_{b=1}^{n} \mathbf{e}^{Qh(b)/\tau}} \tag{2.1}$$

avec $P(a)$ la probabilité qu'une action a de désirabilité $Qh(a)$ soit choisie. Le paramètre τ permet de doser l'influence de $Qh(a)$: quand τ tant vers 0 la probabilité favorise un nombre plus restreint de coups, et si τ augmente $P(a)$ tend vers une distribution uniforme. La désirabilité d'un coup est évaluée en fonction d'heuristiques

dont nous parlons plus en détail au paragraphe 2.5.

Une fois un état terminal atteint, le résultat est reporté dans les nœuds visités par rétro-propagation (voir figure 2.6).

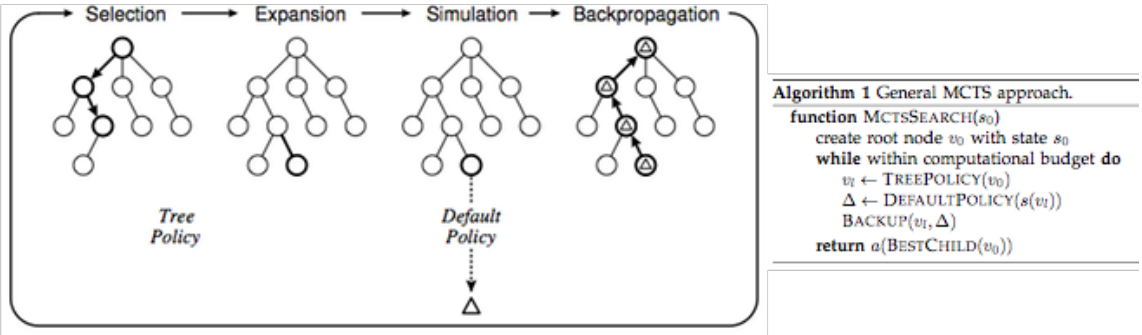


FIGURE 2.6 – Principe général de la méthode Monte Carlo Tree Search [Browne et al., 2012]

Les stratégies de sélection (*TreePolicy*) sont fondées sur un compromis entre exploration et exploitation. Pour avoir une idée fiable de l'ensemble du jeu, il est nécessaire d'explorer la plus grande partie possible de l'arbre, mais cette exploration est coûteuse en temps. D'un autre côté, exploiter les coups les plus prometteurs peut sembler une meilleure idée, mais elle risque de conduire à écarter des coups qui auraient pu se révéler gagnants à long terme e.g. il peut être nécessaire de jouer un coup défavorable (sacrifier une pièce) pour obtenir une victoire. Il faut donc équilibrer ces deux aspects.

L'algorithme *Upper Confidence bound for Tree* (UCT) est le plus populaire parmi les méthodes MCTS et généralement MCTS sous entend l'utilisation de UCT en l'absence de précision. Le principe d'UCT est de choisir comme départ de chaque playout un nœud enfant j qui maximise la valeur de U_j [Browne et al., 2012] :

$$U_j = \overline{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}) \tag{2.2}$$

$\overline{X}_j$ représente le gain moyen obtenu pour ce nœud, C est une constante permettant de pondérer les deux aspects *exploration* et *exploitation*, n est le nombre de fois que le nœud parent a été visité et n_j le nombre de visites pour le nœud j.

On peut noter que si $n_j = 0$ alors la valeur UCT tend vers $+\infty$ ce qui garanti que pour chaque nœud parent, les nœuds enfants non explorés seront visités au moins une fois avant que le choix d'exploiter le meilleur ne survienne. L'autre élément notable, est la nécessité de choisir une valeur pour C, choix qui va déterminer le poids des deux aspects *exploration* et *exploitation* de l'algorithme.

2.4.3 Améliorations de MCTS

Jean Mehat et Tristan Cazenave [Méhat and Cazenave, 2008, 2010] utilisent UCT comme base de leur joueur *Ary*. Afin d'augmenter les performance d'UCT, ils utilisent deux améliorations : la technique des tables de transposition et le Nested Monte-Carlo Search Algorithm pour les jeux à un seul joueur. De leur côté Yngvi Björnsson et Hilmar Finnsson [Finnsson, 2007; Finnsson and Björnsson, 2008; Finnsson and Björnsson, 2009, 2011; Finnsson, 2012] utilisent également UCT pour leur joueur *CadiaPlayer* et ils ont ajouté successivement, entre 2007 à 2009, plusieurs améliorations : Move-Average Sampling Technique (MAST), Rapid Action Value Estimation (RAVE) et Features-to-Action Sampling (FAST).

Pendant une partie, différentes successions de coups peuvent amener à la même situation. Afin de ne pas avoir à explorer deux fois une branche qui est présente à l'identique dans deux parties de l'arbre, les positions et leur évaluation sont stockées dans une *table de transposition*, ce qui permet de diminuer significativement le temps de calcul en réutilisant des informations déjà acquises. Ce temps peut alors être utilisé pour réaliser un nombre plus important de playouts. C'est en utilisant UCT avec une table de transposition que *Ary* a gagné la compétition de GGP de 2009.

Le *Nested Monte-Carlo Search Algorithm* (NMC) [Cazenave, 2009], utilisé pour les jeux à un seul joueur, consiste à orienter les playouts successifs en fonction des playouts précédents en mémorisant le chemin qui a mené au meilleur score lors des simulations précédentes et à jouer le premier coup qui y mène. À chaque coup joué, une nouvelle série de playouts est effectuée et la suite du meilleur chemin est éventuellement remplacée par un enchaînement de coups encore meilleurs. Cette approche apporte un bon équilibre entre exploration et exploitation [Méhat and Cazenave, 2010] et présente l'avantage de ne nécessiter aucun ajustement de paramètre contrairement à UCT. Les performances de UCT et NMC se surpassent l'une l'autre en fonction des jeux, si bien que Mehat et Cazenave ont utilisé pour leur joueur une fonction d'évaluation mêlant les deux approches dans le cas des jeux à un seul joueur. À chaque position, le coup joué est celui recevant la meilleure évaluation combinée des deux algorithmes. Ary a également gagné la compétition de General Game Playing en 2010 mais, cette compétition n'ayant pas proposé de jeux pour un seul joueur, l'efficacité de ces améliorations n'a pu être mise à l'épreuve.

La *Move-Average Sampling Technique* (MAST) consiste, après chaque playout lorsque le score final est remonté à travers les nœuds de l'arbre, à reporter la valeur moyenne de chaque action dans une *lookup table* ; ceci permet d'évaluer la valeur d'une même action quelque soit le moment de la partie où elle est faite. Pour de nombreux jeux cette stratégie présente un avantage clair (occupation du centre au tic-tac-toe ou des angles à Othello). La valeur estimée de chaque action est utilisée pour biaiser le choix des coups pendant les playouts selon la distribution de Boltzmann vue avant.

TO-MAST [Finnsson, 2012] est une petite variation de MAST où les nœuds mis à jour sont limités à ceux de l'arbre déjà exploré plutôt que tous les nœuds du playout ce qui permet de baser les décisions sur des données plus robustes.

La *Rapid Action Value Estimation* (RAVE) hérite d'une méthode appliquée aux programmes de Go connue sous le nom de *All-Moves-As-First* (AMAF) [Browne et al., 2012]. Elle consiste à faire remonter la valeur d'un état final non seulement sur les actions sélectionnées pendant le playout, mais également sur les actions identiques qui peuvent avoir été écartées lors des choix successifs pendant la descente. Contrairement à AMAF qui utilise une évaluation par action, seules les actions de la branche courante sont mises à jour ; ceci permet de distinguer les coups propices dans une situation et néfaste dans d'autres. Pour chaque coup, la valeur estimée par UCT et la valeur RAVE sont toutes les deux calculées séparément. Les valeurs RAVE et UCT sont pondérées de manière à ce que RAVE soit favorisé pour sélectionner l'action de départ d'un playout tant que le nombre de playouts est trop faible pour que la valeur UCT soit fiable. À mesure que la valeur UCT se précise, le poids de RAVE est diminué. Une constante est définie pour indiquer le nombre d'évaluations à partir duquel la valeur UCT est considérée suffisamment fiable pour remplacer totalement la valeur RAVE.

La Predicate-Average Sampling Technique (PAST) fonctionne comme MAST à la différence que c'est la valeur moyenne de chaque couple action-prédicat qui est calculée. Selon le principe du GDL, chaque état du jeu est caractérisé par un certain nombre de prédicats qui sont vrais. Pendant la phase de rétro-propagation, pour chaque nœud ou l'action a a été choisie, la valeur de chaque prédicat vrai est mise à jour. Pendant un playout, pour le calcul de la probabilité dans la formule de Gibbs, c'est la moyenne la plus élevée parmi les prédicats qui est utilisée par *CadiaPlayer*. Cette valeur maximale est utilisée plutôt que la moyenne sur tous les prédicats pour limiter le temps de calcul. PAST permet de choisir les coups qui sont bons en fonction d'un certain contexte.

Comme preuve de l'efficacité de MCTS, *CadiaPlayer* a gagné les compétitions de 2007 et 2008, cependant, les concepteurs de CadiaPlayer indiquent que malgré l'efficacité des algorithmes de Monte Carlo, il reste nécessaire d'utiliser des heuristiques pour introduire une forme de compréhension du jeu de plus haut niveau :

> *« The aforementioned schemes do not use any game-specific domain knowledge. Although this has the benefit of allowing effective deployment over a wide range of disparate games, this approach seems simplistic in contrast to human players, which use high-level features such as piece types and board geometry in their reasoning. The lack of understanding of such high-level game concepts does indeed severely handicap GGP players using simple search-control schemes in certain types of games, for example chess-like games where a good understanding of the different piece type values is essential for competitive play »* [Finnsson and Björnsson, 2011]

Björnsson et Finnsson ont donc ajouté à leur joueur la Features-to-Action Sampling Technique (FAST) qui utilise le Temporal Difference Learning $TD(\lambda)$ [Sutton, 1988] pour évaluer différentes caractéristiques du jeu (position ou déplacement d'une pièce dans une case, etc.). Pendant la phase préparatoire d'analyse du jeu,

dans certains cas[7], FAST détermine des caractéristiques pertinentes pour l'évaluation des positions. Cet ensemble de caractéristiques est utilisé durant le reste du jeu pour estimer la valeur des actions et orienter la recherche.

Finnsonn indique que MAST et FAST peuvent être utilisés conjointement en pondérant les résultats des deux, mais la valeur optimale de cette pondération reste à déterminer. MAST, TO-MAST et PAST sont des variantes inconciliables de la même technique ; elle ne peuvent être utilisées ensemble ; c'est donc une seule de ces variantes qui doit être intégrée à la fonction d'évaluation de *CadiaPlayer* en fonction du type de jeu, mais là aussi le critère de choix est à déterminer et nécessite de futures recherches.

2.5 Recherche d'heuristiques

En 2012, c'est de nouveau *CadiaPlayer*, alliant la puissance de MCTS avec des heuristiques, qui a pris la première place de la compétition. Les programmes de *General Game Playing* semblent de fait avoir beaucoup à gagner en utilisant des heuristiques pour améliorer les algorithmes de recherche utilisés. Ces heuristiques peuvent intervenir à plusieurs niveaux dans l'exploration de l'arbre du jeu : pour choisir l'action à explorer au moment d'effectuer des playouts, pour ordonner les coups légaux à chaque étape du playout ou éventuellement exclure certaines branches qui ne semblent pas devoir aboutir à un score intéressant. Cependant, auparavant, l'extraction de caractéristiques des règles GDL des jeux, l'utilisation de ces caractéristiques pour générer des heuristiques, le choix des bonnes heuristiques et de leur pondérations sont autant de problèmes qui nécessitent d'être résolus.

2.5.1 Players fondés sur l'utilisation d'heuristiques

Lors des différentes compétitions de General Game Playing, plusieurs approches ont été explorées pour l'extraction des caractéristiques du jeu et la création d'heuristiques. Nous allons maintenant voir quelques unes de ces approches.

UTexas LARG - Kuhlmann, Dresner et Stone

Kuhlmann, Dresner et Stone [Kuhlmann et al., 2006] proposent de déduire des caractéristiques de la syntaxe des règles GDL et d'utiliser des heuristiques pour guider l'exploration de l'arbre du jeu et explorer préférentiellement les coups les plus prometteurs. En examinant les règles du jeu, leur programme déduit des éléments de structure comme la relation de successeurs en comparant les arguments possibles d'un même prédicat :

```
(quiet q w) (quiet w e) (quiet e r) (quiet r t)
```

7. Nous sommes sceptiques quant à l'efficacité réelle de FAST

Il peut aussi détecter la présence de compteurs qui utilisent cette relation de successeur pour changer d'état :

```
(<= (next (<counter> ?<var1>))
    (true (<counter> ?<var2>))
    (<successor> ?<var2> ?<var1>) )
```

Kuhlmann et al. proposent également de détecter la présence d'un plateau de jeu en examinant tous les prédicats ternaires e.g. `(cell 1 3) (cell 2 3) (cell ...)`; ceci est contestable car un plateau de jeu peut être en 3D voire plus. Ils distinguent également des *marqueurs* (placés dans différentes cases, comme les pierres au Go) ou des *pièces* (mobiles et présentes dans une seule case à la fois). Ces éléments (plateau, marqueurs, pièces, etc.) sont déduits par une analyse de la variance/invariance des paramètres des règles durant les premières simulations. 10% du temps initial est utilisé pour ces simulations qui déterminent ces éléments de structure. Ces caractéristiques permettent de créer des heuristiques en fonction par exemple de la position, du nombre ou de la distance entre les pièces. Ces heuristiques consistent à voir si la maximisation ou minimisation d'une caractéristique est liée au score obtenu.

Leur joueur *UTexas LARG* a obtenu la $3^{ème}$ place au concours de GGP en 2006 ; il n'a jamais gagné la compétition, mais la démarche de Kuhlmann et al. est néanmoins remarquable par son originalité. Kuhlmann et Stone proposent également une autre idée intéressante : mémoriser les jeux déjà rencontrés et les heuristiques utilisées en codant les règles sous forme de graphe pour pouvoir les comparer ultérieurement à d'autres jeux rencontrés dans le but de réutiliser les connaissances déjà acquises [Kuhlmann and Stone, 2007].

Ogre - Kaiser

Une approche similaire est utilisée par David Kaiser [Kaiser, 2007] pour son joueur *Ogre* mais, tandis que le système de Kuhlmann, Dresner et Stone émet des hypothèses d'après la structure syntaxique des règles GDL d'un jeu puis élimine celles qui apparaissent fausses à mesure des simulations, celui de Kaiser utilise une approche statistique. Les différents prédicats sont comparés pour en extraire les différentes valeurs prises par les arguments et en calculer la variance, afin de déterminer par exemple les arguments représentant des pièces dans le jeu. Cette démarche ne se révèle pas plus efficace ; *Ogre* n'obtiendra que la $4^{ème}$ place lors de la compétition de GGP de 2006.

FluxPlayer - Schiffel et Thielscher

Le joueur de Schiffel et Thielscher, nommé *FluxPlayer* [Schiffel and Thielscher, 2007a,b], opère une détection de structures similaire à celle de Kuhlman et al. [Kuhlmann et al., 2006] mais en analysant la structure sémantique et non syntaxique des prédicats.

FluxPlayer tient son nom du système Prolog Flux développé par Thielscher en 2005, fondé sur le *Fluent Calculus*, qui permet d'analyser les règles GDL. Il utilise une recherche itérative en profondeur d'abord, les techniques classiques d'élagage comme *Alpha-Beta* et des tables de transposition. Les heuristiques sont utilisées pendant les recherches à faible profondeur pour réordonner les coups en fonction de leur valeur estimée. Puis une recherche plus profonde est effectuée pour ces coups estimés plus prometteurs.

Les éléments structurels du jeu sont définis de manière plus large que chez Kuhlman et al. . Schiffel et Thielscher donnent comme exemple qu'une relation binaire qui est antisymétrique, fonctionnelle, injective et dont le graphe est acyclique [Schiffel and Thielscher, 2007b] peut être considérée comme une relation de successeur. Cette façon de procéder permet de détecter des relations de plus haut niveau comme la relation d'ordre et des notions plus larges. Par exemple, la définition d'un plateau de jeu n'est plus limitée aux prédicats ternaires (2 paramètres pour les dimensions et 1 pour le contenu de la case) mais se trouve étendue à N dimensions et M éléments de contenu. Les prédicats sont analysés pour détecter les paramètres d'entrée (position) et de sortie (état) [8].

Les heuristiques utilisées sont fondées sur les règles définissant les états finaux et les buts. Les éléments structurels détectés permettent d'évaluer des caractéristiques comme la position de pièces sur le plateau de jeu ou la valeur de compteurs. La distance entre la valeur courante et la valeur à atteindre définie dans les buts (position à atteindre) ou dans les états finaux (nombre de coups maximum avant la fin du jeu) permet d'évaluer la valeur d'une position : plus ou moins proche du but à atteindre pour maximiser le score.

Le système de Schiffel et Thielscher utilise la logique floue pour estimer la proximité de la fin du jeu et le degré de satisfaction des buts. Ainsi, le joueur évite d'atteindre un état terminal tant que le ou les buts permettant de maximiser le score ne sont pas atteints. La maximisation du score est utilisée pour les jeux en coups alternés tandis qu'un comportement paranoïaque est utilisé dans le cas de coups simultanés (on considère que le ou les adversaires jouent le coup le plus défavorable) ce qui n'aboutit pas forcément à un comportement optimal.

FluxPlayer a gagné la compétition de GGP de 2006 et a su se placer parmi les 4 premiers les années suivantes.

ClunePlayer - Clune

Pour constituer une fonction d'évaluation pour son General Game Player, Clune met également l'accent sur les heuristiques [Clune, 2007] de manière à déterminer la

8. Schiffel et Thielscher indiquent que certains arguments d'un fait peuvent être identifiés comme étant des coordonnées sur le plateau de jeu, ces arguments sont considérés comme des paramètres d'entrée. D'autres arguments ne peuvent avoir qu'une seule valeur pour chaque instance du même fait avec les mêmes coordonnées pour chaque état du jeu. Ces arguments sont identifiés comme une description de l'état de la position et sont considérés comme des paramètres de sortie

désirabilité d'un état non-terminal pour un joueur donné. Comme dans les travaux de Kuhlmann, Dresner et Stone [Kuhlmann et al., 2006] des caractéristiques sont extraites des règles du jeu et comme dans les travaux de Schiffel et Thielscher [Schiffel and Thielscher, 2007a], il évalue le degré de satisfaction des buts et fonde ses heuristiques sur 3 aspects : la récompense, le contrôle et la proximité d'un état terminal.

Pour le degré de satisfaction, il détecte les éléments stables et utilise la variance pour établir une prévision de l'évolution du jeu à partir d'une position donnée : les éléments qui varient trop ne sont pas fiables pour évaluer une position. Les éléments stables (dont la valeur évolue régulièrement sans fluctuation) ayant une influence sur l'évolution de la récompense (positive ou négative) sont utilisés pour les heuristiques, ceux dont la valeur n'est pas liée à celle de la récompense sont jugés insignifiants. Pour le contrôle, il sélectionne les éléments qui ont une influence sur le nombre de coups légaux, pour réduire la mobilité de l'adversaire et augmenter celle du joueur. La proximité d'un état terminal est utilisée pour pondérer les fonctions de récompense et de contrôle, l'augmentation de la récompense étant d'autant plus recherchée que l'issue du jeu approche.

Dans l'introduction de sa thèse, il indique bien que, de son point de vue, cette recherche d'heuristiques est un point fondamental du GGP même si l'exploration d'un jeu nécessite d'utiliser des techniques de parcours d'arbres :

> « *In my opinion, automatic construction of effective heuristic evaluation functions from game descriptions is the central challenge of general game playing. [Clune, 2008]* »

Maligne - Kirci

Le joueur développé par Kirci, nommé *Maligne* [Kirci et al., 2011] utilise le Game Independent Feature Learning algorithm (GIFL) qui consiste à extraire les prédicats vrais d'un état terminal qui ont une influence sur le résultat obtenu en conjonction avec la dernière action effectuée. Le programme de Kirci détermine les faits qui ont une influence en supprimant tour à tour chaque fait et en vérifiant si cette ablation a eu un impact sur le score obtenu. Par exemple, au tic-tac-toe, seules les deux marques `X` déjà présentes ont une influence sur la victoire et c'est l'ajout d'une troisième marque qui déclenche cette victoire. Les autres faits présents dans la description de la position n'ont aucune influence sur le score obtenu (voir figure 2.7).

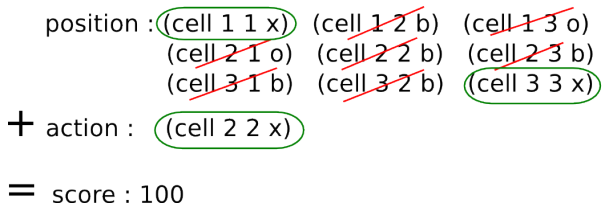


FIGURE 2.7 – Identification des faits ayant une influence sur le score

Si c'est l'adversaire qui obtient une victoire, l'association des prédicats ayant causé cette défaite constitue une caractéristique défensive. L'association de ces prédicats est alors considérée comme une caractéristique à rechercher ou à éviter pour l'évaluation des états non terminaux. Cependant certaines combinaisons plus complexes de prédicats ne sont pas détectées par cette technique ce qui en limite l'efficacité pour certains jeux et provoque même dans certains cas une contre-performance.

MaLigne est arrivé troisième en 2009 et second en 2010 à la compétition de *General Game Playing* mais, en dépit de l'absence de victoire, cette démarche est remarquable par son originalité.

2.5.2 Génération automatique d'heuristiques

Pell décrit dans sa thèse [Pell, 1993] un des premiers programmes de *General Game Playing* nommé *Metagamer* capable de jouer à un groupe de jeux de plateau de type échecs. Son programme est capable de jouer à des jeux déterministes, à deux joueurs et à information complète. En dehors d'une exploration de l'arbre du jeu par la méthode classique de l'algorithme *MiniMax* avec élagage *Alpha-Beta*, son joueur est fondé sur l'évaluation d'heuristiques. L'exploration de l'arbre du jeu se fait par approfondissement itératif et pour augmenter l'efficacité de l'élagage et augmenter les chances d'avoir exploré les coups les plus prometteurs dans le temps imparti, les nœuds sont ordonnés en fonction de leur évaluation par ces heuristiques. Ces dernières, assez simples, sont par exemple la mobilité (comparaison entre les nombres de coups légaux disponibles pour chaque joueur), le matériel (nombre de pièces dont chaque joueur dispose), la centralité (placer les pièces là ou elles ont la mobilité potentielle maximum i.e. occuper le centre), la promotion (pion devenant reine aux dames) et les menaces (valeur de la pièce menacée et effets à craindre).

Les travaux de Pell ont marqué les premières réalisations de General Game Player et bien que limité aux jeux de type échecs, le paradigme de *Metagame* de Pell a inspiré plusieurs concurrents qui ont fondé leur joueur sur des heuristiques.

Samuel en 1959 [Samuel, 1959] proposait déjà d'établir manuellement une liste d'heuristiques pour le jeu de *Dames anglaises* (contrôle du centre, mobilité des pièces, etc.) et suggérait de combiner ces heuristiques avec des opérateurs logiques ($A \bullet B, A \bullet \overline{B}, etc.$). Seulement, il convenait de l'explosion combinatoire qui allait en résulter et seul un nombre restreint de combinaisons d'heuristiques était établi à la main. En faisant jouer deux versions du programme l'une contre l'autre, les pondérations des heuristiques était peu à peu ajustées : 16 heuristiques parmi une réserve de 38 étaient utilisées. À chaque coup, l'écart entre la valeur prévue au préalable pour une position et la valeur actuelle calculée était utilisé pour ajuster les pondérations utilisées par un joueur A. À chaque fin de partie, si la performance du joueur A dépassait celle du joueur B, une copie des heuristiques de A était donnée à B et l'entrainement reprenait. Dans le cas contraire, si le joueur A ne progressait pas, l'heuristique la plus importante était supprimée et remplacée par une autre de la réserve.

Il est apparu que l'interaction des heuristiques n'est pas linéaire. La modification du coefficient d'une heuristique peut modifier le comportement des autres. Dans sa nouvelle version de 1967 [Samuel, 1967], Samuel propose l'idée d'une table de signatures : les heuristiques sont combinées dans des groupes formant une hiérarchie. Quatre heuristiques sont combinées de 68 manières différentes et leurs signatures sont enregistrées dans une table de niveau 1. Les 3×68 signatures de trois tables de niveau 1 sont combinées pour former 125 signatures différentes dans une table de niveau 2. Enfin le contenu de 2 tables de niveau 2 permet de produire 225 signatures de niveau 3. Ce sont ces combinaisons d'heuristiques qui sont utilisées dans la fonction d'évaluation. L'ajustement des pondérations est effectué en comparant les prévisions du programme avec les coups joués dans une partie modèle.

Cette démarche bien que limitée au jeu de *Dames anglaises* soulève plusieurs questions importantes :

- Comment effectuer le choix des heuristiques au départ ? Rendre la machine capable de trouver elle-même ces heuristiques serait préférable, mais comment faire ?
- D'après Samuel, les heuristiques interagissent de manière non linéaire, comment pourrait-on trouver automatiquement les combinaisons qui ont un sens et en ajuster les pondérations ?

Marcolino [Marcolino and Matsubara, 2011] propose de recombiner des heuristiques appliquées au jeu de Go en utilisant la programmation agent : chaque agent possède une sélection d'heuristiques à appliquer dans un ordre particulier. Un groupe des agents les plus efficaces est sélectionné et un agent du groupe est utilisé à chaque position pour évaluer l'action à faire. Cependant, une fois de plus, les heuristiques sont déterminées à l'avance manuellement.

Nous avons vu (au paragraphe 2.5.1) avec les approches de Kuhlmann, Dresner et Stone [Kuhlmann et al., 2006], Schiffel et Thielscher [Schiffel and Thielscher, 2007b], et Clune [Clune, 2007] que des caractéristiques du jeu peuvent être déduites des règles GDL et ensuite utilisées pour l'évaluation d'heuristiques. Cependant leurs approches sont toutes orientées par le savoir du programmeur : l'analyse des règles est faite de manière à déduire des propriétés connues du programmeur comme les notions de successeurs, de compteurs, de quantité, de position, de pièces.

- Serait-il possible de laisser la machine découvrir elle-même comment les caractéristiques du jeu influencent le score final ?

Dans cette optique, la démarche utilisée par Kirci pour *Maligne* [Kirci et al., 2011] qui consiste à évaluer quelles combinaisons de faits ont une influence sur le score est intéressante. Sa démarche est cependant limitée par l'absence de prise en compte de combinaisons de prédicats plus complexes.

Sharma et al. [Sharma et al., 2008b] proposent d'associer chaque fait vrai définissant une position dans un jeu avec une valeur (une pondération). Pour évaluer une position, il suffit alors de calculer la somme des valeurs des faits et de ramener cette valeur dans l'intervalle [0,1] grâce à une fonction sigmoïde :

$$V(S) = \sigma(\sum_{t \in match(S)} \psi_t) \tag{2.3}$$

avec $V(S)$ la valeur de la position S, $\sigma()$ la fonction sigmoïde, ψ_t la valeur d'un n-uplet (d'un fait) et $match(S)$ l'ensemble des faits vrais pour cette position.

En plus des faits, les actions choisies à chaque étape sont également associées à une pondération calculée en fonction de l'éloignement de l'action de la fin du jeu et de la récompense obtenue. Cette pondération est ajustée à chaque playout en fonction de la longueur de chaque nouvelle simulation et du score obtenu. Les pondérations des faits sont également ajustées en fonction de la récompense obtenue et d'un coefficient d'apprentissage qui décroît avec le temps. Ces deux évaluations (valeur d'une position et valeur d'un coup) sont associées pour orienter la recherche effectuée avec UCT.

L'approche de Sharma et al. est intéressante, mais elle part du principe que les n-uplets représentent des caractéristiques indépendantes les unes des autres, hors nous avons vu que, dès 1959 avec l'approche de Samuel [Samuel, 1959], il apparaissait que les heuristiques interagissaient de manière non linéaire. Sharma et al. concluent d'ailleurs sur ce fait :

> *« In our representation for state knowledge, we have assumed that the tuples are independent of each other. However, in most cases it is the relationship between various state features that matters. An alternative to using linear function approximation is using non-linear function approximation techniques such as neural-networks. »* [Sharma et al., 2008b]

Sharma et al. ont poursuivi leur recherches en s'intéressant finalement à la technique *Ant Colony Optimisation* [Sharma et al., 2008a, 2009] et n'ont pas poursuivi cette idée d'utiliser des réseaux de neurones.

Utgoff [Utgoff, 2001] évoque l'utilisation d'un réseau de neurones pour combiner des caractéristiques décrites par des règles logiques similaires aux règles GDL, mais il indique que cette technique présente plusieurs inconvénients qui sont le risque de bloquer sur un minimum local et la lenteur d'apprentissage d'un tel système :

> *« While this kind of approach is quite useful for many problems, it suffers several drawbacks. First, such an algorithm can become trapped at a local minimum when descending the error gradient, though for some applications a local minimum may be useful nevertheless. Second, such an algorithm can be very slow, sometimes requiring weeks to months, or longer, to reach its best performance level. »* [Utgoff, 2001]

De plus, il indique qu'il est malaisé de savoir quelle topologie donner au réseau :

> *« An associated task is to determine the network architecture. How many hidden units shall there be, and how shall the units be connected ? »* [Utgoff, 2001]

Il indique que différentes combinaisons d'heuristiques peuvent être codées de différentes manières et que chacune implique un réseau contenant un certain nombre de couches cachées. L'augmentation du nombre de couches ne garantit donc pas une solution optimale :

> « *The sentence* $((a \vee b) \wedge (c \vee d)) \vee ((e \vee f) \wedge (g \vee h))$ *requires seven (7) binary operators and three (3) layers to evaluate, whereas the logically equivalent sentence* $((a \wedge c) \vee (a \wedge d) \vee (b \wedge c) \vee (b \wedge d) \vee (e \wedge g) \vee (e \wedge h) \vee (f \wedge g) \vee (f \wedge h))$ *requires fifteen (15) binary operators and two (2) layers to evaluate. One possible motivation for minimizing the number of layers would be to facilitate the parameter adjustment scheme.* » [Utgoff, 2001]

Utgoff évoque aussi l'usage des algorithmes génétiques tels que décris par Goldberg [Goldberg, 1989] mais affirme que les résultats obtenus avec ces algorithmes ne sont pas satisfaisants et qu'une approche plus directe serait préférable selon lui.

2.6 Conclusion

Les meilleurs programmes de *General Game Playing* actuels utilisent pour la plupart la technique Monte Carlo Search Tree (MCTS) avec les diverses améliorations décrites par Cameron Browne [Browne et al., 2012]. Pour améliorer encore leurs performances il apparaît nécessaire de les doter d'une compréhension de concepts de plus haut niveau et ceci semble nécessiter la création d'heuristiques.

Nous avons vu que l'extraction de caractéristiques par l'analyse de la syntaxe ou de la sémantique des règles GDL impliquait l'injection de savoir humain dans le programme et que, pour laisser à la machine la possibilité de construire elle-même ce savoir, une approche similaire à celle de Kirci [Kirci et al., 2011] ou Sharma et al. [Sharma et al., 2008b] était préférable.

Pour que le programme puisse combiner ces caractéristiques de manière non linéaire et découvrir lui-même quelles interactions entre ces caractéristiques sont signifiantes pour évaluer les positions dans un jeu, nous avons vu que l'usage de réseaux de neurones est plusieurs fois suggéré [Utgoff, 2001; Sharma et al., 2008b].

Les critiques que Utgoff formule à l'encontre des réseaux de neurones et des algorithmes génétiques, l'incertitude dans le choix de la topologie et le problème du temps de calcul, semblent pouvoir trouver une solution avec le modèle de *NeuroEvolution of Augmenting Topologies* (NEAT) proposé par Stanley en 2002 [Stanley and Miikkulainen, 2002]. Le modèle proposé par Stanley combine justement algorithmes génétiques et réseaux de neurones. En faisant évoluer la topologie d'un réseau avec un algorithme génétique, il démontre qu'il est possible de découvrir la topologie idéale d'un réseau pour résoudre un problème donné. Il propose de réaliser cette évolution en partant d'un réseau minimal, ce qui permet également de réduire au maximum le temps de calcul.

Dans le prochain chapitre nous détaillons ce modèle, précédé d'une présentation des algorithmes génétiques.

3 Algorithmes génétiques et neuro-évolution

L'ajustement de la topologie d'un réseau de neurones pour la résolution d'un problème particulier est un problème complexe. Le modèle de *Neuro-Evolution of Augmenting Topologies* de Stanley [Stanley and Miikkulainen, 2002] apporte une solution élégante à ce problème en utilisant le principe des algorithmes génétiques pour faire évoluer cette topologie.

Dans ce chapitre, nous commençons par présenter la technique des algorithmes génétiques et divers aspects de leur mise en oeuvre. Puis nous présentons l'algorithme de *Neuro-Evolution of Augmenting Topologies* (NEAT).

3.1 Algorithmes génétiques

La technique des algorithmes génétiques proposé par John H. Holland [Holland, 1992] est une technique d'exploration d'un espace de données inspirée des mécanismes de la sélection naturelle et de la génétique. Les algorithmes génétiques sont très efficaces pour résoudre des problèmes d'optimisation combinatoire, problèmes dont l'espace de recherche est si vaste qu'il n'est pas envisageable d'explorer systématiquement toutes les possibilités pour trouver la solution. Ils s'appliquent spécifiquement bien aux problèmes pour lesquels aucune solution déterministe n'est connue et pour lesquels une bonne solution obtenue dans un temps fini est préférable à une solution parfaite nécessitant un temps indéterminé.

> « *Convergent behavior without guarantee of optimality bothers many poeple who approach genetic algorithms from other, more traditional, optimization backgrounds. (...) the fact of the matter is that genetic algorithms have no convergence guarantees in arbitrary problems. (...) This does not reduce their utility. Quite the contrary, more convergent methods sacrifice globality and flexibility for their convergence.* » [Goldberg, 1989]

3.1.1 Population et fitness

Un algorithme génétique consiste à faire évoluer une population dont chaque individu représente une solution possible dans l'espace des données. Par des processus de croisement, de mutation et de sélection, les meilleurs individus de chaque génération donnent naissance à une nouvelle population représentant des solutions de plus en plus adaptées au problème traité.

La première étape dans la mise en place d'un algorithme génétique consiste donc à coder une solution au problème sous la forme d'un individu ou plutôt de son génome. Goldberg [Goldberg, 1989] décrit comment on peut simplement coder ces génomes sous forme de chaînes de bit binaire. Dans le cas de la recherche d'heuristiques pour le General Game Playing, ce génome peut être la liste des pondérations (flottants en double précision) associées aux faits d'une position terminale d'un jeu exprimée en syntaxe GDL et permettant de déterminer l'influence de chacun de ces faits sur l'évaluation du score.

Un point délicat dans la mise au point d'un algorithme génétique est la manière dont la performance des individus, nommée *fitness* [1], va être évaluée. Il est nécessaire de pouvoir estimer avec précision si l'individu correspond ou non au résultat attendu et à quel point il s'en éloigne ou s'en rapproche. Notre but étant d'amener la population à une prédiction juste du score d'une partie terminée, nous pouvons utiliser l'écart entre la prédiction et le score réel pour estimer la compétence de nos individus.

3.1.2 Sélection

Une fois chaque individu de la population évalué, différentes techniques de sélection sont utilisées pour choisir les reproducteurs qui donneront naissance à la nouvelle population. Les techniques les plus courantes sont le *Stochastic Sampling with Replacement* ou *Roulette Wheel* (= roue de la fortune), et le *tournament selection* (= tournoi de sélection) [Hackett, 1995].

La technique de *Stochastic Sampling with Replacement* est décrite par Goldberg dans son exemple d'implémentation d'un algorithme génétique [Goldberg, 1989]. Elle est nommée également *Roulette Wheel* car elle fonctionne un peu comme si chaque individu se voyait octroyé un secteur d'une roue proportionnel à son *fitness*. Ensuite un tirage au sort permet de choisir des individus pour la reproduction. Les meilleurs individus de la population auront d'autant plus de chance d'être choisis que le secteur qui leur est attribué est important. Avec cette technique une différence peut être observée entre la quantité de descendance attendue pour un individu et le nombre réel sélectionné. Goldberg décrit une variante, le *Remainder Stochastic Selec-*

1. Nous choisissons d'utiliser à partir d'ici le terme anglais de *fitness* pour bien indiquer que nous faisons référence à la valeur associée à la performance d'un individu. Cette valeur indique non seulement l'aptitude d'un individu à résoudre le problème donné mais détermine également la quantité de descendance qu'il engendrera à la prochaine génération.

tion with Replacement, qui apporte d'après lui plus de diversité dans la population. Dans cette technique, la quantité de descendance souhaitée pour chaque individu est représentée par un nombre flottant. La partie entière de ce nombre est utilisée pour sélectionner le nombre voulu de descendance, ensuite c'est la partie fractionnelle du taux qui est utilisée pour la sélection par roue de la fortune.

Une autre variante proposée par De Jong [Goldberg, 1989] est présentée comme approchant plus finement du nombre de descendance espérée pour chaque individu. Cette technique de *Stochastic Sampling without Replacement* est dite *without replacement*, parce que les individus qui ont déjà été choisis voient leur secteur diminuer. De cette manière la proportion de descendance prévue pour chaque individu est bien conforme à leur *fitness*. Voici l'algorithme correspondant à cette technique :

1. Calculer la taille de la population (POP)
2. Déterminer le taux de fécondité de chaque individu (la part de la population qu'il va engendrer) de manière à ce que la somme des taux de tous les individus soit égale à (POP)
3. Générer un nombre aléatoire (SPIN) entre zéro et POP
4. Sélectionner l'individu suivant
5. Soustraire de SPIN le taux de fécondité de l'individu
6. Répéter les étapes 4 et 5 jusqu'à ce que SPIN soit égal ou inférieur à zéro
7. Utiliser l'individu courant pour la reproduction
8. Si c'est une reproduction par croisement décrémenter son taux de fécondité de 0.5, sinon décrémenter de 1
9. Décrémenter POP
10. Répéter les étapes 3 à 6 jusqu'à ce que la nouvelle population soit constituée

La sélection par tournoi (*tournament selection*) consiste à organiser une compétition entre les individus de la population pour obtenir le droit de se reproduire, un peu comme cela se produit dans la nature. Les individus sont sélectionnés au hasard deux par deux pour s'affronter dans un tournoi en plusieurs manches et l'individu victorieux à chaque tour voit ses chances de se reproduire augmenter.

3.1.3 Crossover

Une fois les reproducteurs sélectionnés, l'étape du *crossover* peut survenir. Cette étape nommée aussi *enjambement* en génétique est le phénomène qui à lieu lors de la méiose et qui consiste en un échange de matériel génétique entre les chromosomes d'une même paire. Ce phénomène contribue au brassage génétique lors de la reproduction.

Dans le cadre des algorithmes génétiques, ce phénomène est simulé par l'échange d'une partie des chaînes de bits représentant les individus à partir d'un point de croisement [Goldberg, 1989] :

```
parent1 : 0111 00|10 0101 0100
parent2 : 1010 10|01 1110 1001

enfant1 : 0111 00|01 1110 1001
enfant2 : 1010 10|10 0101 0100
```

D'autres techniques sont décrites proposant plusieurs points de croisement (*multi-point crossover*) mais le principe reste globalement le même : un échange de matériel génétique entre les deux individus.

```
parent1 : 01|11 0010 0|101 0100
parent2 : 10|10 1001 1|110 1001

enfant1 : 01|10 1001 1|101 0100
enfant2 : 10|11 0010 0|110 1001
```

Il est possible de conserver les deux génomes résultant d'un croisement ou de n'en conserver qu'un seul ; dans ce cas chaque individu est considéré comme ayant donné naissance à 0.5 individu (voir étape 8 de l'algorithme donné au paragraphe précédent).

Cette étape de *crossover* constitue la principale singularité des algorithmes génétiques parmi les méthodes stochastiques. Le croisement des individus permet d'exploiter les éléments de solution déjà découverts par différents membres de la population en les recombinant pour trouver de nouvelles solutions encore plus optimales. Chaque élément de solution découvert se traduit par une augmentation du *fitness* des individus le possédant. Ces individus au *fitness* élevé obtiennent une plus large descendance et peuvent donc transmettre les caractéristiques intéressantes aux générations suivantes. En cherchant la solution parmi les individus représentant déjà une solution partielle, les algorithmes génétiques permettent de réduire progressivement l'espace de recherche et de converger vers une solution même si cet espace de recherche est très vaste au départ.

Goldberg [1989] explique ce phénomène par la démonstration d'un *théorème des shemata*. Les *shemata* sont toutes les combinaisons spécifiques de bits qui peuvent apparaître dans les génomes de la population. Les *schemata* les plus courts, correspondant à la présence d'un bit particulier dans la chaîne, sont sélectionnés tôt dans l'évolution quand ils apportent une augmentation du *fitness*. En revanche, les *schemata* plus longs, correspondant à la présence d'une sous-chaîne de bits particulière, peuvent être détruits par la coupure intervenant dans le processus du *crossover* (voir figure 3.1). Goldberg montre qu'un *schema* particulier se développe dans une population à une vitesse égale au rapport de l'adéquation moyenne du *schema* sur l'adéquation moyenne de la population. Le *schema* sera également d'autant plus représenté dans la population suivante que sa longueur est faible et que son adéquation moyenne est élevée. Le codage du génome est donc un élément déterminant pour la bonne évolution de la population et pour garantir le bon fonctionnement du *crossover*.

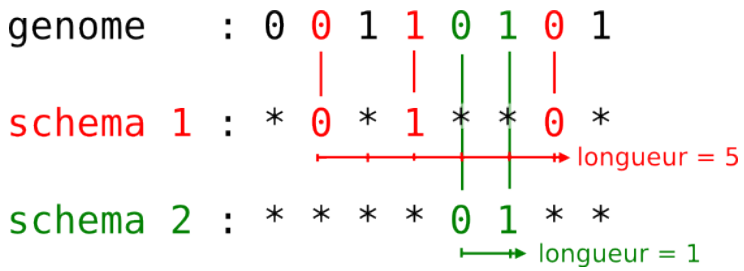


FIGURE 3.1 – Exemple de deux *schemata* différents tous deux présents dans le génome représenté. Les étoiles représentent des bits de valeur quelconque. Le *schema* 1, plus long, a moins de probabilité d'être transmis intact lors de l'étape du *crossover* que le *schema* 2.

Bien que constituant une étape fondamentale dans le fonctionnement des algorithmes génétiques, l'étape du *crossover* n'est pas suffisante pour amener la population vers une bonne solution. Le tirage aléatoire de la population de départ ne garantit pas, si l'espace de recherche est vaste et la population limitée, que tous les éléments de solution seront représentés dans la diversité des génomes. Afin d'explorer l'espace des données et découvrir de nouvelles solutions possibles l'algorithme génétique utilise le principe de mutation.

3.1.4 Mutations

La reproduction des individus peut être réalisée par (*crossover*) ou parfois par simple clonage. Le clonage consiste à répliquer l'individu à l'identique ; ceci est parfois effectué avec le meilleur individu de la population pour provoquer une forme d'élitisme et accélérer la convergence vers une solution. Cependant, la plupart du temps les individus ainsi répliqués sont soumis, de même que les individus issus de croisement, à des mutations.

Comme dans la nature, les mutations permettent de donner ponctuellement naissance à des individus différents et plus performants. Cependant, ces mutations peuvent également donner naissance à des monstres. La quantité de mutations introduites dans les génomes des individus doit donc être limitée pour permettre une bonne exploration de l'espace des données sans pour autant empêcher la population d'évoluer.

L'équilibre entre croisement et mutation détermine la balance entre *exploration* de l'espace des données et *exploitation* des solutions trouvées.

Ces mutations peuvent être représentées, par exemple, par la permutation de deux bits (en rouge) dans un génome défini par une chaîne de bits ou bien par l'inversion de la valeur d'un bit :

```
      genome original :  0110 1001 1101 0100
genome après mutation :  0110 1000 1101 0101

      genome original :  0110 1001 1101 0100
```

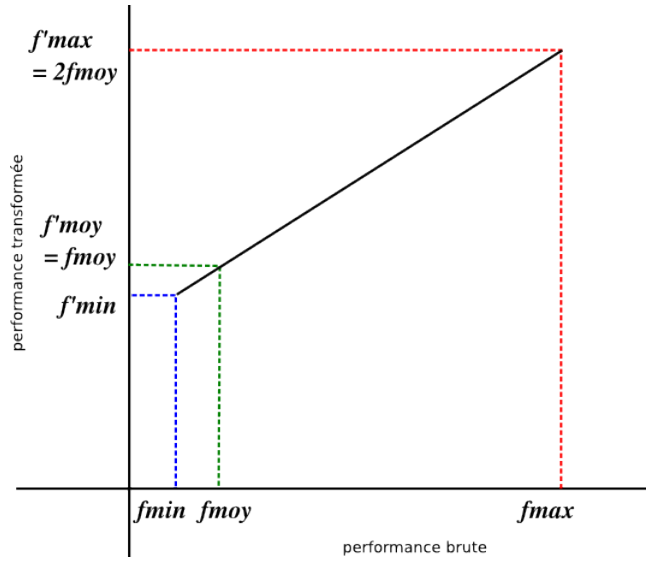


FIGURE 3.2 – Transformation linéaire des *fitness* pour diminuer l'écart entre les individus au *fitness* élevé et les autres. La valeur du *fitness* moyen est conservée, tandis que la valeur du *fitness* maximum est ramenée à N fois le *fitness* moyen (ici N = 2). Toutes les autres valeurs de *fitness* sont transformées linéairement par rapport à ces deux points.

```
genome après mutation :  0111 1001 1101 0100
```

Dans le cas où les génomes sont codé d'une autre manière que par des chaînes de bits, il est nécessaire de trouver des modes de mutations pertinents par rapport à ce codage.

3.1.5 Rescale : ajustement des fitness

Goldberg [Goldberg, 1989] indique que dans les premières générations d'une population, il n'est pas rare d'obtenir quelques individus performants noyés dans une population médiocre. Le risque est de voir ces individus sélectionnés pour donner naissance à une grande partie des générations suivantes, provoquant une convergence prématurée de la population vers une solution non optimale (problème du maximum local). Plus tard dans l'évolution, on peut se retrouver au contraire avec une très faible différence de performance entre les individus les plus faibles et les meilleurs. Pour remédier à cela, Goldberg propose d'opérer un changement d'échelle (*rescale*) des *fitness* de toute la population par transformation linéaire (voir figure 3.2). Le *fitness* moyen est utilisé comme "pivot" et le *fitness* maximum est ramené à N fois le *fitness* moyen, N étant un coefficient multiplicateur désigné par Golberg sous le nom de *Cmult* :

> *« for typical population (n = 50 to 100) a Cmult = 1.2 to 2 has been used successfully »* [Goldberg, 1989]

Cependant un effet indésirable peut survenir quand, dans une population qui a bien évolué, quelques individus peu adaptés subsistent : la transformation linéaire

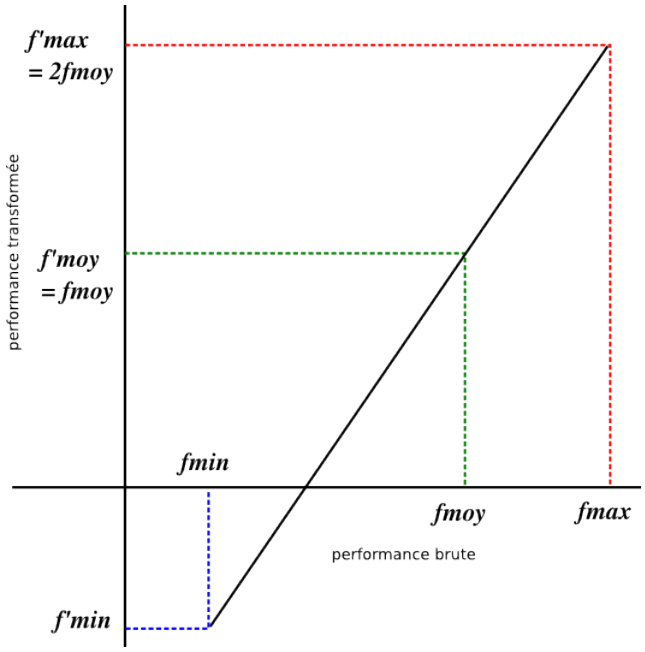


FIGURE 3.3 – Effet indésirable pouvant survenir lors de l'ajustement des *fitness* par transformation linéaire. La valeur du *fitness* moyen est conservée, tandis que la valeur du *fitness* maximum est ramenée à N fois le *fitness* moyen (ici N = 2). Toutes les autres valeurs de *fitness* sont transformées linéairement par rapport à ces deux points ce qui provoque l'apparition de valeurs négatives pour les *fitness* les plus faibles et viole la règle selon laquelle les valeurs de *fitness* doivent être strictement positives.

peut faire apparaître des *fitness* négatifs (voir figure 3.3). La fonction d'ajustement des *fitness* doit donc tenir compte de ce fait et dans ce cas, le *fitness* minimum est fixé à 0 et le reste des valeurs de *fitness* est ajusté en gardant la valeur du *fitness* moyen inchangée.

3.2 Neuro-Evolution of Augmenting Topologies

L'algorithme de *NeuroEvolution of Augmenting Topologies* (NEAT) proposé par Kenneth O. Stanley [Stanley and Miikkulainen, 2002; Stanley, 2004] permet de faire évoluer la topologie d'une population de réseaux de neurones grâce à un algorithme génétique. Pour cela il utilise trois concepts : *Historical Marking*, spéciation et initialisation des réseaux avec une topologie minimale.

Les sources de NEAT implémenté en C++ par Stanley sont disponibles sur une page de son site personnel [Stanley, 2013] consacrée entièrement à NEAT. D'autres implémentations en Delphi, Java, C# ou MatLab sont également disponibles sur cette page. Un tutoriel expliquant pas à pas l'implémentation de NEAT en C++ est proposé par Buckland [Buckland and Collins, 2002] et constitue un point de départ idéal pour comprendre les détails de de la mise en œuvre de cet algorithme.

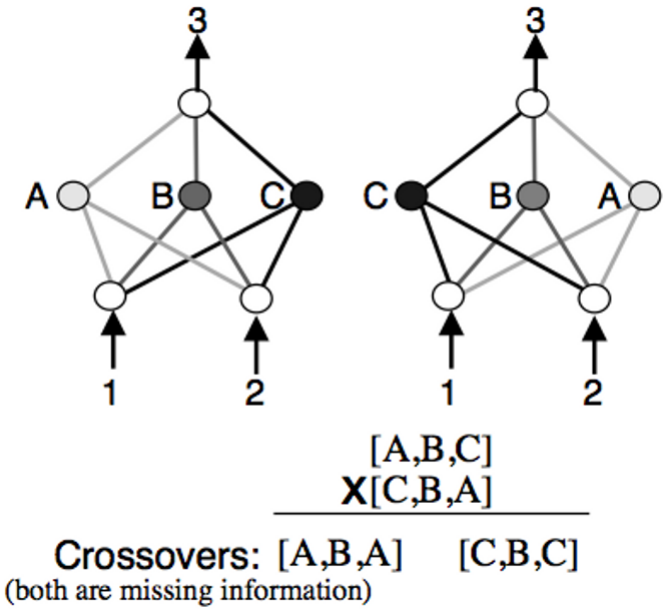


FIGURE 3.4 – Exemple de croisement proposé par Stanley [Stanley and Miikkulainen, 2002] : les deux réseaux parents sont deux des six permutations possibles d'un même réseau. Les deux réseaux sont de fait incompatibles et les réseaux enfants résultant de ce croisement sont incohérents : chacun d'eux a perdu une partie du patrimoine génétique des parents et se retrouve avec une part d'information redondante.

3.2.1 Historical marking

Le principal problème concernant l'application d'un algorithme génétique à une population de réseaux de neurones intervient lors de l'étape du croisement car il est nécessaire d'établir quels réseaux sont compatibles et comment les faire correspondre. Stanley et Miikulainen [Stanley and Miikkulainen, 2002] donnent comme exemple le croisement de deux réseaux identiques mais miroirs l'un de l'autre : un tel croisement est problématique car les réseaux issus de ce croisement perdent inévitablement une partie du patrimoine génétique présent chez les parents (voir figure 3.4). Une analyse de la topologie de chaque réseau pour déterminer sa compatibilité avec un autre au moment du croisement serait extrêmement coûteuse en temps de calcul. Pour remédier à ce problème Stanley propose le système du *historical marking*.

Le *historical marking* consiste à attribuer un numéro à chaque nœud du réseau. Le génome correspondant au réseau est alors représenté par une liste des connexions entre ces nœuds avec une pondération associée à chacune. À côté, la liste des nœuds est conservée. Lors d'un croisement, toutes les connexions des deux réseaux parents sont alignées et comparées ; de cette manière la correspondance entre les deux réseaux peut être établie directement sans nécessiter une analyse coûteuse de leur structure (voir figure 3.5). Les connexions peuvent alors être héritées d'un parent ou de l'autre.

Comme pour un algorithme génétique classique, NEAT possède aussi un prin-

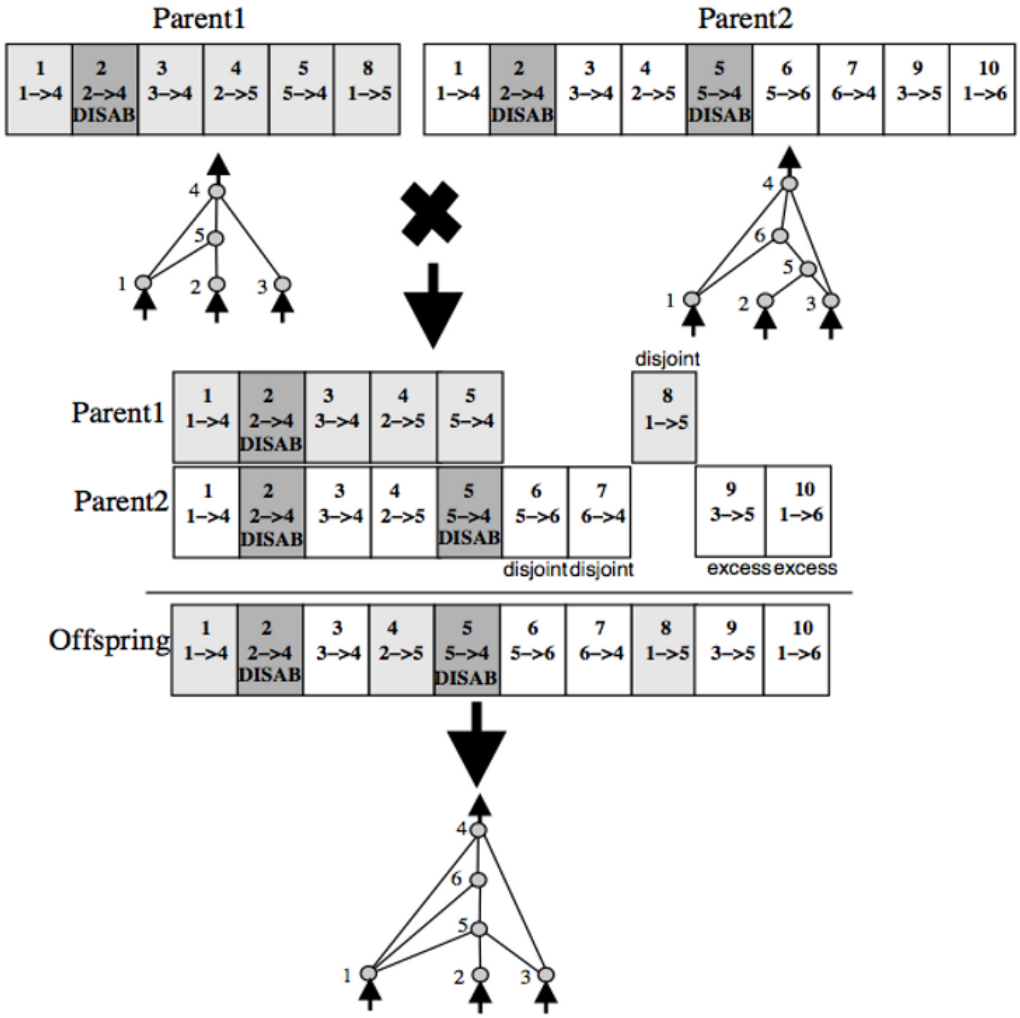


FIGURE 3.5 – « *Matching up genomes for different network topologies using innovation numbers. Although Parent 1 and Parent 2 look different, their innovation numbers (shown at the top of each gene) tell us which genes match up with which. (...) Matching genes are inherited randomly, whereas disjoint genes (those that do not match in the middle) and excess genes (those that do not match in the end) are inherited from the more fit parent.* »[Stanley and Miikkulainen, 2002].

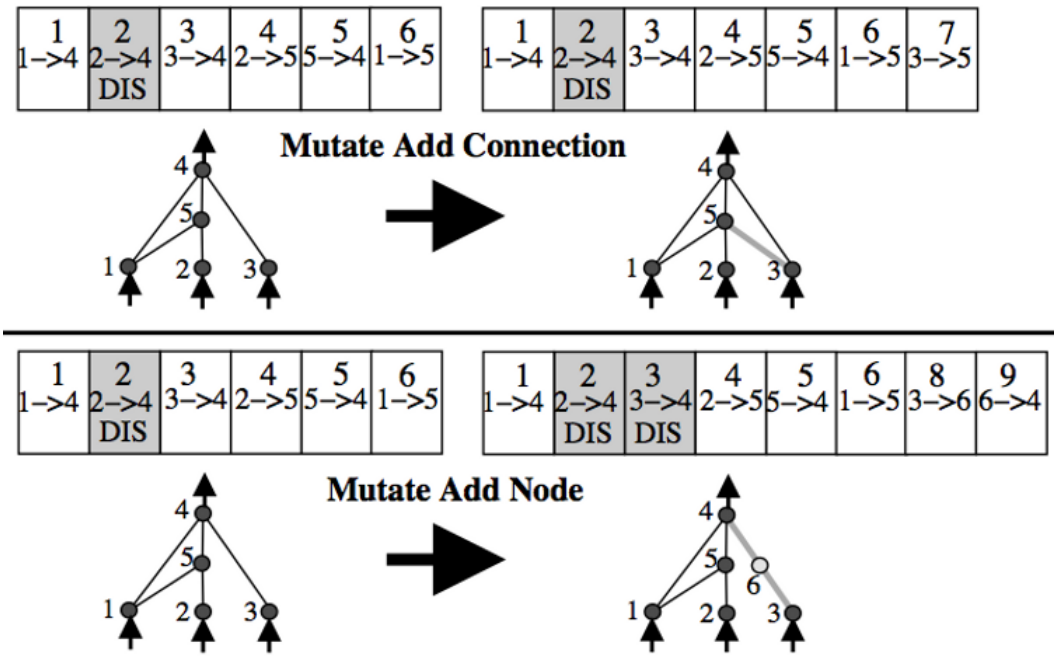


FIGURE 3.6 – « *The two types of structural mutation in NEAT.* »[Stanley and Miikkulainen, 2002].

cipe de mutation. La mutation peut être une modification des pondérations, l'ajout ou la désactivation d'une connexion, ou l'ajout d'un nœud (voir figure 3.6). L'*historical marking* permet d'attribuer automatiquement un numéro aux nouveaux nœuds créés : chaque nœud ajouté constitue une innovation et est enregistré dans une liste des gènes de toute la population avec le numéro des nœuds entre lesquels il s'intercale. Avant d'ajouter un nœud entre deux autres, il suffit donc de vérifier si cette innovation existe déjà et si oui d'utiliser le même numéro. Lors de l'ajout d'un nœud, la connexion entre deux nœuds est déactivée et deux nouvelles connexions sont créées avec le nouveau nœud. Les listes de nœuds et de connexions peuvent être ainsi considérées comme un génome où les mêmes gènes peuvent être utilisés plusieurs fois et s'exprimer avec plus ou moins d'intensité.

3.2.2 Spéciation

Lorsqu'un nouveau nœud ou une nouvelle connexion sont ajoutés, les performances du réseau peuvent baisser temporairement le temps d'ajuster les pondérations. Le principe de la compétition risque alors de provoquer systématiquement le rejet des innovations. Pour éviter cela, un autre principe fondamental de NEAT est la *spéciation*. Les différents réseaux sont regroupés en différentes espèces en fonction de leur ressemblance. La distance entre deux réseaux δ est établie en fonction du nombre de gènes qui les différencient (voir figure 3.5) : cette différence dépend de l'écart moyen des pondérations $\bar{W}$ entre les gènes semblables, du nombre de gènes disjoints D (présents chez l'un et absents chez l'autre entre deux gènes communs), et du nombre de gènes en excès E (gènes supplémentaires présent chez l'un et non chez l'autre) :

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 * \overline{W} \tag{3.1}$$

Le nombre d'espèces maximum est fixé au départ par le programmeur. Quand une mutation ou un croisement intervient, le nouveau réseau est comparé au référent de chaque espèce. S'il n'est semblable à aucune des espèces connues, une nouvelle espèce est créée et le réseau devient le référent de cette nouvelle espèce. D'autres systèmes utilisant la spéciation comparent tous les réseaux deux à deux ce qui est bien plus lourd en traitement que cette comparaison avec l'unique référent de chaque espèce.

Le *fitness* des réseaux est alors calculé à l'intérieur de chaque espèce et la reproduction se fait également entre membres de la même espèce. Afin qu'une espèce performante, mais pas forcément optimale, ne risque pas de prendre le pas sur toute la population, un système de *fitness sharing* est utilisé : le *fitness* d'un individu est d'autant plus affaibli que les membres de la même espèce sont plus nombreux. Ainsi les autres espèces ont leur chance de donner naissance à des individus encore plus évolués. Pour assurer l'évolution de la population, seuls les meilleurs individus de chaque espèce sont conservés en fonction d'un seuil de survie et obtiennent le droit de se reproduire.

3.2.3 Topologie minimale

Faire évoluer un grand nombre de réseaux pendant un nombre important de générations peut être extrêmement coûteux en calcul. La dernière spécificité de NEAT est de partir d'un réseau minimal (voir figure 3.7), avec une couche d'entrée, une couche de sortie et sans couche cachée. Les pondérations sont fixées au hasard et l'évolution va permettre de complexifier peu à peu les structures sans qu'elles ne soient jamais inutilement compliquées puisque la sélection naturelle va favoriser les solutions les plus simples. Ainsi le temps de calcul est minimisé puisque le nombre de nœuds évalués reste bien plus faible que si l'on avait tenté d'ajuster les pondérations d'un réseau complexe avec une topologie fixée au départ.

3.3 Conclusion

Dans ce chapitre, nous avons présenté le modèle de *Neuro-Evolution of Augmenting Topologies* proposé par Stanley [Stanley and Miikkulainen, 2002]. NEAT apporte une solution au problème du choix de la topologie pour un réseau de neurones et au problème du temps de calcul. Grâce à ce modèle, nous pouvons utiliser des réseaux de neurones pour découvrir les caractéristiques signifiantes dans les positions d'un jeu et les combinaisons de ces caractéristiques nécessaires à l'évaluation de positions. Il peuvent donc nous permettre de générer automatiquement des heuristiques pour guider le choix des coups les plus prometteurs.

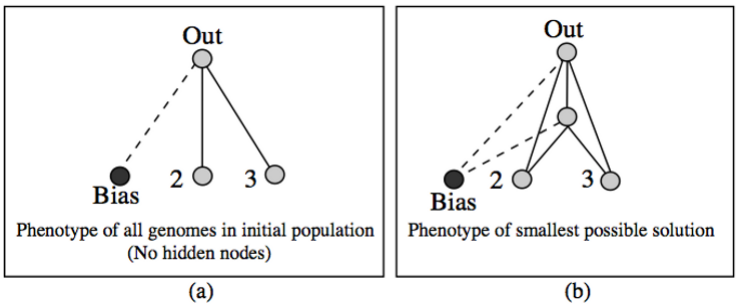


FIGURE 3.7 – « ***Initial phenotype and optimal XOR. Figure (a) shows the phenotype given to the entire initial population. Notice that there are no hidden nodes. In NEAT, a bias is a node that can connect to any node other than inputs. Figure (b) shows an optimal solution with only 1 hidden node. (A network without hidden nodes cannot compute XOR.) The bias connections are not always needed depending on the solution ; All other connections are necessary.*** »**[Stanley and Miikkulainen, 2002].**

Il est à noter que Reisinger [Reisinger et al., 2007; Reisinger and Miikkulainen, 2007] a déjà tenté d'utiliser NEAT dans le but de fournir des heuristiques pour l'exploration d'arbre dans le domaine du *General Game Playing*. Cependant, une fois de plus les heuristiques sont établies à la main et c'est leur résultat (différence dans le nombre de pièces des deux joueurs, mobilité, etc.) qui est donné en entrée des réseaux de neurones évolués avec NEAT.

Il ressort des éléments que nous venons de voir une piste de recherche qui n'a pas encore été exploitée : utiliser NEAT pour faire évoluer une population de réseaux recevant en entrée les faits issus de la description GDL d'une position dans un jeu et pouvant fournir en sortie une évaluation de cette position. C'est cette approche que nous détaillerons dans les prochains chapitres.

4 Expérimentations préliminaires

Inspiré par les travaux de Samuel [Samuel, 1959, 1967] et de Sharma et al. [Sharma et al., 2008b], notre but est de déterminer s'il est possible d'amener la machine à découvrir elle-même les combinaisons de faits qui sont significatives pour l'évaluation d'une position dans un jeu, et les pondérations associées à ces faits.

Dans ce but et pour circonscrire notre champ d'étude à un problème bien précis, nous avons décidé d'utiliser les états finaux de différents jeux pour entrainer notre programme et voir si il pourrait y découvrir le lien entre la description de la position et le score obtenu. Le choix des positions terminales a été fait pour une raison pratique : à ce stade le score est connu et donc il est possible de comparer facilement le score prévu par notre système avec le score réel.

Nous avons également restreint notre étude à des jeux à deux joueurs dans l'idée que ce champ pourrait être étendu par la suite à des jeux solitaires ou multi-joueurs si nos résultats s'avéraient concluants. Dans cette optique notre programme a été implémenté pour être facilement étendu à ce type de jeux.

Afin d'avoir une base de comparaison pour estimer les performances de NEAT, nous avons implémenté un premier programme utilisant un algorithme génétique classique. Cette première étape doit nous permettre également de repérer les jeux pour lesquels une simple pondération des faits permet déjà une estimation juste du score et ceux pour lesquels au contraire, un simple algorithme génétique ne peut opérer une estimation fiable du score parce que ce dernier est déterminé par des combinaisons plus complexes de faits.

La première étape de notre travail a donc consisté à générer les données nécessaires à l'entrainement de notre programme : c'est l'objet du prochain paragraphe dans lequel nous décrivons la manière dont ces données ont été obtenues et leurs caractéristiques, ainsi que les structures utilisées dans notre programme pour stocker ces données.

Dans un second temps nous présentons les détails de l'implémentation de notre algorithme génétique et le résultat des tests effectués sur notre jeu de données.

4.1 Génération des données

Pour générer les états finaux des jeux accompagnés de leur score, nous avons utilisé le player *Ary* de Jean Mehat. Afin de faire tourner le programme il était nécessaire de lui fournir la description GDL des jeux. Les descriptions utilisées sont des fichiers qui ont été récupérés sur le site de General Game Playing de l'université de Dresde [Günther and Schiffel, 2013]. Ces descriptions GDL ont été collectées par Jean Mehat dans le cadre la mise au point de son player *Ary*, et modifiées légèrement pour y corriger les erreurs éventuelles ou en nettoyer les instructions. Nous avons ainsi fourni à *Ary* la description de 93 jeux GDL et obtenu pour chaque jeu et chacun des deux joueurs 3000 états terminaux accompagnés de leur score, donc 6000 états terminaux en tout pour chaque jeu.

Pour que Ary génère ces données, nous lui avons ajouté quelques instructions demandant de placer dans un fichier, lors de chaque playout UCT, la liste des faits GDL vrais à ce moment ainsi que le score. Pour chaque jeu GDL, deux joueurs sont donc lancés sans utilisation de threads puis un arbitre de match créé par Jean Mehat pour ses tests est utilisé pour analyser le fichier GDL et servir d'intermédiaire entre les joueurs pour guider le jeu. Pour automatiser cette tâche, nous avons utilisé un script bash.

Les fichiers obtenus contiennent sur chaque ligne une position terminale représentée par son score, suivie d'une liste de faits logiques vrais. Par exemple, pour le jeu de *Tictactoe* :

```
((100 0) (control oplayer) (cell 1 1 b) (cell 1 2 o) (cell 1 3 b)
         (cell 2 1 b) (cell 2 2 b) (cell 2 3 o) (cell 3 1 x)
         (cell 3 2 x) (cell 3 3 x) )
((50 50) (control oplayer) (cell 1 1 o) (cell 1 2 x) (cell 1 3 x)
         (cell 2 1 x) (cell 2 2 o) (cell 2 3 o) (cell 3 1 o)
         (cell 3 2 x) (cell 3 3 x) )
((0 100) (control xplayer) (cell 1 1 x) (cell 1 2 b) (cell 1 3 o)
         (cell 2 1 x) (cell 2 2 b) (cell 2 3 o) (cell 3 1 b)
         (cell 3 2 x) (cell 3 3 o) )
```

4.1.1 Présentation des données utilisées

Afin de pouvoir comprendre par la suite les résultats obtenus lors de notre expérimentation, il est nécessaire d'examiner plus avant le contenu de ces fichiers (voir Annexe A pour les règles des jeux). En effet la représentation des états finaux par des faits logiques et leur génération lors de playouts UCT amène plusieurs particularités qui peuvent se révéler problématiques.

Premièrement, certains fichiers ne contiennent que quelques états finaux au lieu des 3000 demandés, par exemple pour les jeux *Add Game* et *MiniChess* ; ceci s'explique par le fait que lorsque l'arbre du jeu a été entièrement exploré par UCT, il

est inutile de faire d'avantage de simulations. Il ne génère donc pas de nouveaux playouts, ce qui influence les résultats obtenus ; il convient dont d'examiner les résultats sans perdre de vue cette particularité.

Deuxièmement, les jeux présentés sont à somme nulle (*Tictactoe*, *Nim*, *Connect4*, etc.) ou non (*Battle*, *Point Grab*, etc.). Nous devront vérifier si les résultats obtenus pour ces deux types de jeux sont différents. De plus, dans le cas de *Beat Mania*, le joueur 1 fait office de générateur aléatoire et donc son score est toujours maximal tandis que le joueur 2 reçoit une récompense en fonction de sa performance. Il apparaît donc nécessaire que notre programme analyse le score des deux joueurs (tour à tour ou simultanément) pour pouvoir comparer les résultats obtenus.

Troisièmement, pour certains jeux, la situation à l'état final donne des données triviales. Par exemple, à *Nim* la fin de partie est toujours marquée par l'absence d'allumette dans chaque tas. Seule l'identité du joueur courant indique qui a perdu ou gagné :

```
((0 100) (control player1) (heap a 0) (heap b 0) (heap c 0) (heap d 0))
((100 0) (control player2) (heap a 0) (heap b 0) (heap c 0) (heap d 0))
```

Cependant il est intéressant de voir si notre programme sait découvrir l'inutilité de ces faits dans la détermination du score et tirer parti des faits `(control ...)` qui doivent lui permettre assez facilement d'évaluer le score. Les 4 variantes de *Nim* présentes dans notre base de jeux varient par leurs règles GDL mais pas par la représentation des états finaux. Il est donc à prévoir que les résultats seront identiques pour chacun.

Quatrièmement, pour certains jeux les états finaux gagnants sont très rares (*Wallmaze*, *Racer*, *Mummy Maze*, *Ghost Maze*) ou absents : pour *Blocks* le score maximal atteint est de 25/100, et tous les états finaux de *Hallway* et *Javastrike* sont des matchs nuls (50/50). Au contraire, le fichier de données obtenu pour *EndGame* ne contient que des positions finales dont le score est gagnant pour le joueur 2 (0/100). Il convient de voir si les résultats obtenus pour ces jeux diffèrent spécifiquement.

Pour toutes ces raisons, nous n'avons pas systématiquement testé tous les jeux avec NEAT, ce qui de surcroît se serait avéré très long en temps de calcul. Nous nous sommes concentrés sur certains jeux plus représentatifs :

- *Nim* car ces états finaux simplistes nous permettent de vérifier l'efficacité de notre programme sur un problème simple ;
- *Point Grab* car c'est un exemple de jeu à somme non nulle dont les faits sont directement liés aux résultats obtenus. La description de l'état final comporte également un compteur dont nous verrons si l'absence de corrélation avec le score sera découverte :
  ```
  ((25 10) (step 30) (points first 25) (points second 10))
  ((30 35) (step 30) (points first 30) (points second 35))
  ((20 20) (step 30) (points first 20) (points second 20))
  ```

- *Roshambo* car l'évaluation de son score semble pouvoir être effectuée de manière linéaire ou non ;
- *Tictactoe* parce que ce jeu est simple à analyser, son mécanisme nous étant familier, mais la détermination de son score est un problème non linéaire et complexe.

4.1.2 Stockage des données dans notre programme

Pour pouvoir stocker les données utilisées dans notre programme, nous avons utilisé une structure (langage C) contenant à la fois un vecteur `vAllFacts` stockant les descriptions de tous les faits différents rencontrés dans les états finaux (par exemple `(cell 1 1 b)`), et un vecteur `vAllFinalStates` stockant chacune des positions terminales. Une position terminale est décrite par un vecteur `vFactIdxs` de taille variable contenant les indices des différents faits vrais la décrivant et d'un autre vecteur `vScore` contenant le score des différents joueurs (voir figure 4.1).

```
typedef struct {
  int * vFactIdxs; // indexes of the different True facts
  int * vScore;    // scores of each players
  int iNumFacts;   // number of facts in this final state
} finalState;

typedef struct {
  int iNumPlayers; // number of players (determines the size of the vectors
      of scores)
  char ** vAllFacts; // vector containing all the different facts
      encountered
  int iNumFacts;    // number of facts
  int iMaxFacts;    // maximum size of the vector
  finalState ** vAllFinalStates; // vector containing all the final states
  int iNumFinalStates;        // number of final state
  int iMaxFinalStates;        // maximum size of the vector
} testBase;
```

FIGURE 4.1 – Structures utilisées pour le stockage des données dans nos deux implémentations d'un algorithme génétique et de NEAT

Comme nous l'avons déjà précisé, nous avons limité nos expérimentations aux jeux à deux joueurs, mais nous avons néanmoins prévu ces structures et les fonctions de lecture des fichiers de données pour pouvoir les étendre à différents nombres de joueurs.

Nous avons utilisé ces structures et les fonctions de chargement des données pour nos deux programmes implémentant un algorithme génétique et l'algorithme de *Neuro-Evolution of Augmenting Topologies* (NEAT) de Stanley [Stanley and Miikkulainen, 2002].

4.2 Implémentation d'un algorithme génétique

Les remarques de Samuel [Samuel, 1959] laissent entendre que la recherche d'une corrélation directe entre l'évaluation d'une position et les éléments la décrivant est vaine. D'après lui, ce sont des combinaisons de caractéristiques, comme l'alignement de 3 symboles au *Tictactoe*, qui permettent d'évaluer efficacement une position. Pour vérifier cette assertion et pour obtenir une base pour une future comparaison avec les résultats obtenus pour NEAT, nous avons implémenté un algorithme génétique classique.

Notre implémentation [1] de cet algorithme génétique a été réalisée en C d'après la description fournie par Goldberg [Goldberg, 1989]. Nous avons choisi d'associer chaque fait décrivant les états finaux à des pondérations qui pourraient en déterminer l'influence (positive ou négative) sur le score du joueur courant. Les individus de notre population sont donc représentés par des vecteurs de nombres flottants en double précision dont les valeurs sont bornées entre -1 et $+1$.

4.2.1 Evaluation du fitness

Nous avons choisi de faire évoluer notre population de manière à prédire le score d'un seul joueur pour des raisons pratiques : prédire le score du deuxième joueur aurait nécessité un vecteur de pondérations d'une taille double et les résultats obtenus auraient été plus confus, rendant l'e résultat de l'analyse moins net. Nous avons à la place préféré lancer l'algorithme génétique deux fois sur chaque fichier de données pour prédire le score des deux joueurs l'un après l'autre.

Le *fitness* des individus est évalué de la manière suivante. Une position finale P étant décrite par une liste de N faits vrais représentés par des n-uplets t, la valeur de cette position finale $V(P)$ est estimée en faisant la somme des valeurs associées aux n-uplets ψ_t qui le décrivent :

$$V(P) = \sum_{t \in P} \psi_t \tag{4.1}$$

La somme des valeurs associées aux n-uplets est ramenée dans l'intervalle $[-1, 1]$ simplement en plafonnant les valeurs de manière à ce que l'évaluation ne soit pas faussée par le nombre variable de faits qui peuvent composer un état final. Pour pouvoir comparer cette valeur estimée avec le score réel compris dans l'intervalle $[0, 100]$, la valeur $V(P)$ est également ramenée linéairement dans cet intervalle. Le score estimé pour cette position $S(P)$ est donc :

1. Le code de notre programme est disponible dans un dépôt GIT à cette adresse : https://github.com/alinehuf/gaGGPfinalStates.git

$$S(P) = \begin{cases} 0 & \text{si} \quad V(P) < -1 \\ (V(P)+1) * 50 & \text{si} \quad -1 < F(S) < 1 \\ 100 & \text{si} \quad V(P) > 1 \end{cases} \tag{4.2}$$

La valeur de chaque position terminale est ainsi évaluée et l'erreur commise est calculée pour les N positions finales, entre l'estimation effectuée $V(P_i)$ et la vraie valeur du score $T(P_i)$. Comme les scores sont évalués entre 0 et 100, le *fitness* d'un individu F peut être considérée comme un pourcentage de réussite :

$$F = \frac{\sum_{i=N}^{i=0} 100 - |T(P_i) - V(P_i)|}{N} \tag{4.3}$$

La première évaluation du *fitness* que nous avons réalisée utilisait une fonction sigmoïde pour ramener l'évaluation de la valeur d'un état dans l'intervalle $[0, 100]$ plutôt que d'utiliser une transformation linéaire. Il s'est avéré que ceci faussait l'évaluation du score et diminuait l'erreur d'estimation pour les scores extrêmes (0 ou 100) amenant artificiellement de meilleurs résultats pour les jeux dont le score est binaire.

4.2.2 Ajustement du fitness

Pour éviter les problèmes de convergence prématurée signalés par Goldberg (voir paragraphe 3.1.5), nous avons choisi dans notre implémentation d'ajuster les *fitness* de la population comme il le préconise. Nous avons donc utilisé une transformation linéaire en utilisant un facteur multiplicateur de 2 (Goldberg conseillant une valeur entre 1.2 et 2) : le *fitness* moyen est inchangé, tandis que le *fitness* maximum est ramené à 2 fois la valeur du *fitness* moyen. Le *fitness* minimum et tous les autres sont ajustés en conséquence.

Avant d'effectuer les transformations la valeur du *fitness* minimum est contrôlée pour ne pas être négative. Dans le cas contraire, le *fitness* minimum est ramené à 0, le *fitness* moyen est inchangé et les autres valeurs sont ajustées en conséquence.

Ce *fitness* ajusté est utilisé pour déterminer la quantité de descendance attribuée à chaque individu, Le *fitness* original est lui plus représentatif de la performance réelle des individus et est utilisé dans les fichiers de compte rendu d'expérience.

4.2.3 Sélection des reproducteurs et mutations

Nous avons testé deux techniques pour la sélection des reproducteurs. La première consiste à dupliquer chaque individu un certain nombre de fois en fonction de son taux de reproduction et à placer ses clones dans une réserve de reproducteurs. L'individu est donc d'autant plus représenté que son taux de reproduction est élevé.

Tant que la nouvelle population ne compte pas suffisamment d'individus, deux individus sont piochés au hasard dans la réserve et en fonction du taux de *crossover* établi pendant le paramétrage de l'algorithme, les deux individus sont soit croisés soit placés tels quels dans la nouvelle population.

Le croisement consiste en un échange de gènes et Goldberg propose, dans son modèle binaire, de définir un point de croisement à partir duquel la valeur des bits définissant les individus sont échangées. Dans notre cas, les individus ne sont pas représentés par des chaînes de bits mais par des chaînes de flottants mais nous avons utilisé le même principe et opéré un croisement en échangeant le contenu des vecteurs représentant les individus à partir d'un point de croisement choisi aléatoirement, par exemple :

```
père    +0.543 -0.153 | -0.324 +1.000 +0.923
mère    -0.293 -0.223 | -0.913 -0.432 +0.245

enfant1 +0.543 -0.153 | -0.913 -0.432 +0.245
enfant2 -0.293 -0.223 | -0.324 +1.000 +0.923
```

Cette première technique de sélection que nous avons utilisée présente un inconvénient majeur. Le taux de reproduction de chaque individu n'est pas un entier et il est bien difficile de mettre par exemple 2.234 copies d'un individu dans la réserve. L'erreur qui résulte des arrondis effectués s'est manifestée par des résultats catastrophiques lors des premiers tests. Aucune évolution n'était constatée dans le *fitness* moyen de la population.

Nous avons donc opté pour une autre forme de reproduction, plus précise, le *stochastic sampling without replacement*. Le taux de reproduction attribué à chaque individu l'est de telle manière que la somme des taux de reproduction est égale à la taille de la population ; ceci implique que si l'on respecte scrupuleusement ces taux, on obtient une nouvelle population d'une taille identique à la précédente. On considère donc ces taux de reproduction comme des zones de taille plus ou moins importante attribuées à chaque individu sur une roue de la fortune. Un nombre est tiré au hasard entre 0 et la taille de la population et le pointeur de la roue se positionne à l'endroit désigné : l'individu qui possède cette zone est élu pour se reproduire. Son droit de reproduction est alors diminué de 1 et le tirage recommence en sélectionnant un nombre au hasard dans un intervalle égal au nombre d'individus restant à ajouter.

Pour explorer l'espace des données, nous avons introduits différents types de mutations. Les individus sont représentés par des vecteurs de flottants représentant les pondérations associées à chaque fait. La première mutation consiste à perturber les pondérations qui sont comprises dans l'intervalle $[-1, 1]$ en ajoutant ou soustrayant une valeur aléatoire entre 0.1 et 0.5. Si cette perturbation fait sortir la valeur de l'intervalle [-1, 1], la valeur est tronquée. Un autre type de mutation consiste à inverser le signe d'une pondération. Un troisième type consiste à permuter deux valeurs de pondérations tirées au hasard.

4.3 Évaluation de 93 jeux avec l'algorithme génétique

Dans le but d'avoir une vision d'ensemble de la performance de l'algorithme génétique sur les 93 jeux de notre base, nous avons lancé un banc de test systématique sur la totalité de ses jeux. Puis en fonction des résultats obtenus, nous avons effectué de nouveaux tests sur certains jeux particuliers. Nous présentons donc dans une première partie les hypothèses que nous avons formulées et les résultats attendus. Dans un second temps nous décrivons les conditions de notre expérimentation, puis nous exposons les résultats obtenus. Enfin, nous présentons une expérience supplémentaire réalisée dans le but de confirmer nos résultats.

4.3.1 Hypothèses de départ

Notre première hypothèse est que pour certains jeux, dont le score est directement lié à l'un des faits décrivant l'état final, notre algorithme génétique devrait atteindre un *fitness* maximum élevé. En revanche pour des jeux plus complexes dont le score dépend d'une interaction entre différents faits, l'évaluation du score devrait mettre l'algorithme génétique en difficulté et le *fitness* obtenu devrait être plus faible.

Par exemple pour le jeu de *Nim*, la description de l'état final est triviale :

```
((100 0) (control player2) (heap a 0) (heap b 0) (heap c 0) (heap d 0) )
((0 100) (control player1) (heap a 0) (heap b 0) (heap c 0) (heap d 0) )
```

Nous nous attendons donc à ce que l'algorithme génétique puisse trouver une pondération de $+1$ et -1 pour les faits `(control ...)` et atteigne un taux de réussite de 100% dans ses prévisions. Les faits `(heap ...)` n'ont aucune influence sur le score et devraient être associés à une pondération de 0.

Pour *Point Grab* le score dépend directement du nombre de points accumulés, la relation est donc également directe :

```
((40 65) (step 30) (points first 40) (points second 65) )
((80 75) (step 30) (points first 80) (points second 75) )
((40 65) (step 30) (points first 40) (points second 65) )
((70 100) (step 29) (points first 70) (points second 100) )
((45 65) (step 30) (points first 45) (points second 65) )
((40 70) (step 30) (points first 40) (points second 70) )
((35 65) (step 30) (points first 35) (points second 65) )
```

Un fait comme `(points first 40)` devrait avoir une pondération positive pour le joueur 1 de $+0.40$ tandis que l'autre fait `(points second 65)` ne devrait avoir aucune influence sur son score et une pondération de 0. Réciproquement le fait `(points second 65)` devrait avoir une pondération de $+0.65$ pour le joueur 2 et le

fait `(points first 25)` une influence nulle. Les faits de type `(step x)` avec $x < 10$ surviennent dans notre jeu de données quand le joueur 2 a accumulé le score maximum et que la partie prend fin prématurément. Le hasard des simulations UCT fait que, dans notre jeu de données, la partie n'est jamais prématurément arrêtée en faveur du joueur 1. Ces faits pourraient donc avoir une influence négative sur le score du premier joueur et positive sur celui du deuxième joueur.

Pour le jeu *Roshambo*, l'algorithme génétique ne peut découvrir un concept comme *« le premier joueur a plus de points que le second »* :

```
((0 100) (step s9) (beaten black s5) (beaten white s2) )
((50 50) (step s10) (beaten black s3) (beaten white s3) )
((50 50) (step s10) (beaten black s3) (beaten white s3) )
((100 0) (step s10) (beaten black s3) (beaten white s4) )
((0 100) (step s10) (beaten black s4) (beaten white s2) )
((100 0) (step s10) (beaten black s1) (beaten white s4) )
((100 0) (step s8) (beaten black s0) (beaten white s5) )
((0 100) (step s9) (beaten black s5) (beaten white s1) )
((0 100) (step s10) (beaten black s5) (beaten white s2) )
```

Nous nous attendons, pour le joueur 1, à voir les faits `(beaten white s5)` et `(beaten white s4)` associés à une pondération proche de $+1$, tandis que les faits `(beaten white s2)`, `(beaten white s1)` et `(beaten white s0)` seront proches de -1. Réciproquement, les faits concernant l'adversaire `(beaten black ...)` devraient avoir des pondérations inverses. Les faits `(beaten ... s3)` sont souvent associés à un score de 50/50, leur pondération devrait donc être proche de 0. Cependant le match nul peut aussi intervenir avec `(beaten black s4)` `(beaten white s4)` ou d'autres valeurs, et `(beaten ... s3)` n'intervient pas uniquement dans des match nuls, il est donc possible que l'algorithme génétique rencontre des difficultés dans l'évaluation de ce jeu.

Pour un jeu comme *Tictactoe* où le score est déterminé par la combinaison de 3 symboles alignés, nous nous attendons à ce que l'algorithme génétique échoue à produire une prévision efficace. Cependant, nous espérons que l'algorithme saura identifier que les croix ont une influence positive sur le score du joueur 1 et les cercles une influence négative. S'il utilise les faits de type `(control ...)`, avec une pondération respective de $+1$ et -1 comme pour `Nim`, il est possible qu'il puisse toutefois prédire le score.

4.3.2 Conditions d'expérimentation

Pour pouvoir comparer les résultats obtenus sur un panel de jeu assez large, nous avons lancé notre algorithme génétique sur l'ensemble des fichiers produits par Ary pour les 93 jeux. Pour chaque jeu nous avons deux fichiers de 3000 états finaux produits lors les playouts UCT par les deux players Ary jouant l'un contre l'autre, sauf

pour certains jeux ou l'exploration complète de l'arbre amène une fin prématurée des simulations comme expliqué au paragraphe 4.1.1. Il est apparu très vite que les deux fichiers générés par les players UCT donnaient des résultats équivalents, nous avons donc utilisé uniquement les fichiers générés par l'un des players et non les deux pour établir le graphe présentant nos résultats.

Comme le taux de mutation et de *crossover* est particulièrement délicat à ajuster et que les valeurs idéales pour ces paramètres peuvent varier d'un jeu à l'autre, nous avons lancé l'algorithme plusieurs fois en faisant varier ses deux paramètres. Les mutations implémentées ayant une faible influence sur les génotypes, notre paramètre de mutation doit être assez élevé pour permettre une bonne exploration de l'espace des données. La valeur idéale a été située empiriquement entre 0.1 et 0.4, nous avons donc lancé l'algorithme successivement avec des taux de mutation de $0.1, 0.2, 0.3$ et 0.4. Le taux de *crossover* idéal a été estimé empiriquement entre 0.1 et 0.7 nous avons donc lancé l'algorithme successivement avec des taux de *crossover* de $0.1, 0.3, 0.5$ et 0.7.

Nous avons donc au total réalisé 16 tests pour chaque fichier, ce qui fait 32 lancements du programme pour chaque jeu pour prédire le score des deux joueurs. Le temps de calcul qui en résulte étant assez long, nous avons du limiter à 100 le nombre d'individus et à 1000 le nombre de générations pendant lesquelles la population évolue. Pour pouvoir visualiser les résultats, nous avons conservé pour chaque jeu, parmi les 16 combinaisons de paramètres, celle qui a donné les meilleurs résultats. Le graphe des figures 4.2-4.3 présente les *fitness* (minimal, moyen et maximal) de départ et d'arrivée, ainsi que le meilleur *fitness* jamais obtenu au cours des 1000 générations d'évolution. Les jeux à somme nulle sont signalés par le niveau nul de la courbe verte tandis que les jeux à somme non-nulle sont signalés par une légère croissance de cette courbe.

4.3.3 Analyse des résultats

La première constatation que l'on peut tirer de l'observation du graphe (figure 4.2-4.3) est que les jeux à somme nulle ou non nulle ne donnent pas de résultats particuliers, ce qui semble tout à fait normal si l'on considère que nous avons évalué séparément le score de chaque joueur. De même, pour les jeux dont les données ne présentaient que peu ou pas de de scores gagnants, les résultats sont très variables prouvant bien que le résultat obtenu est plus fonction de la complexité du jeu et de la relation (directe ou non) existant entre les faits et le score.

Les quatre jeux que nous avons examinés plus en détail, *Nim*, *Grab Point*, *Roshambo* et *Tictatoe*, se situent parmi ceux pour lesquels l'algorithme génétique a pu effectuer une prévision relativement bonne avec respectivement un *fitness* maximum de 99.5, 95.7, 93.9 et 89.8 (voir figure 4.3).

Nous pouvons constater que les jeux pour lesquels la prévision du score est la moins bonne sont des jeux complexes : parmi eux des jeux de type échecs, dames et des versions complexifiées de tictactoe jouées en parallèle sur plusieurs grilles.

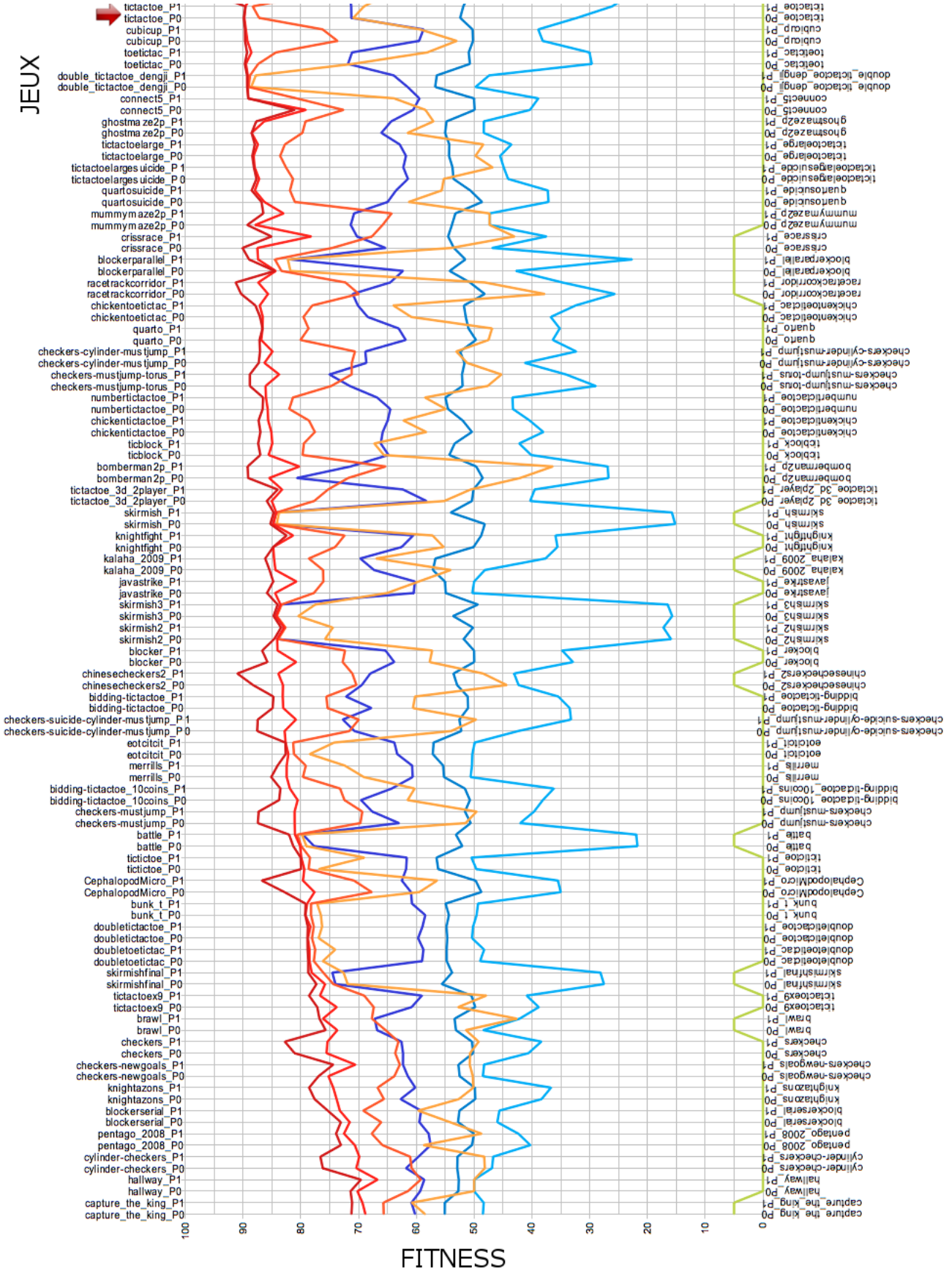


FIGURE 4.2 – *fitness* de départ et d'arrivée pour chaque jeu ...

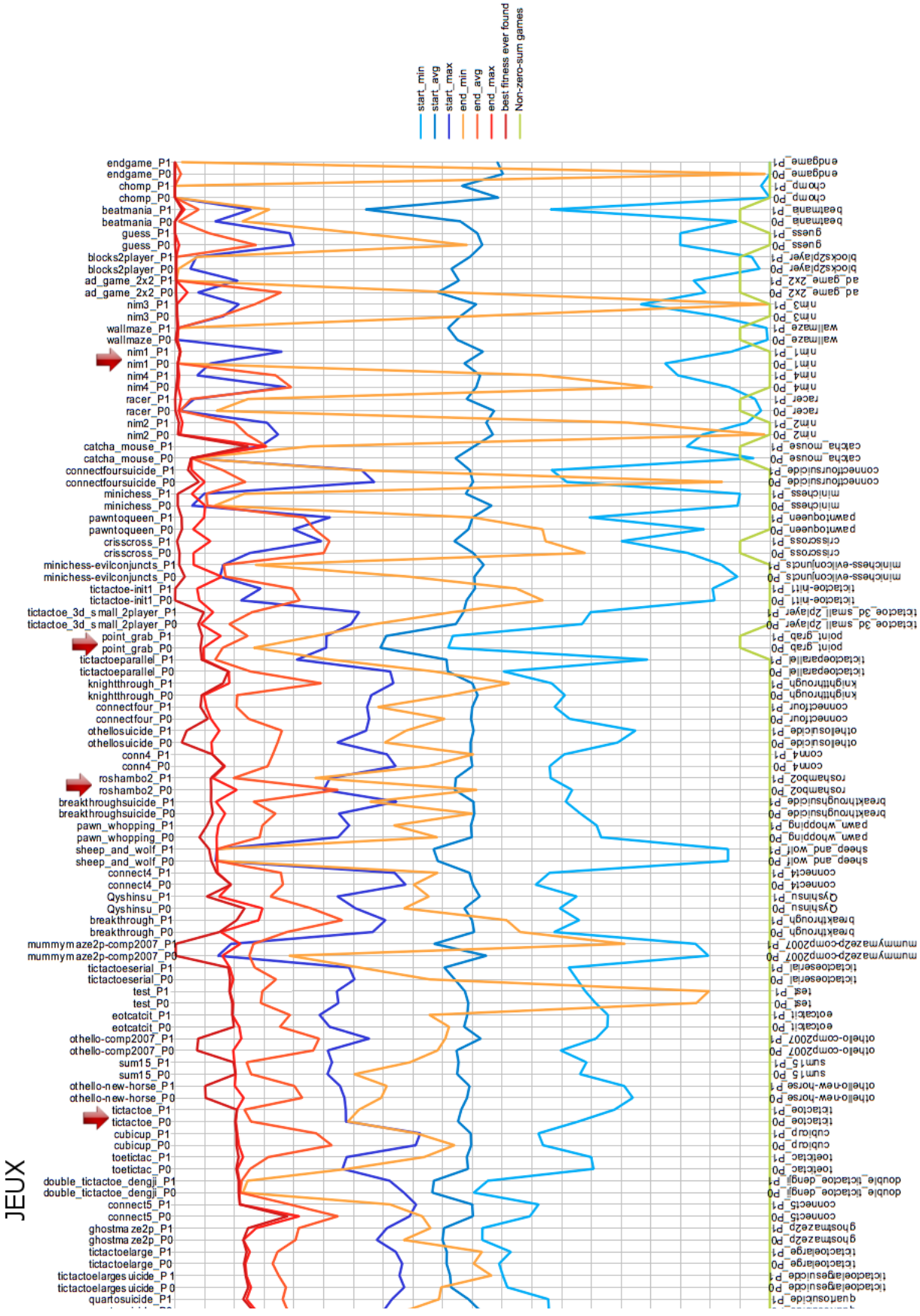


FIGURE 4.3 – ... et pour l'évaluation du score de chaque joueur (P0 et P1)

Le fait que l'algorithme génétique échoue à prédire un score fiable pour ces jeux est une fois de plus un résultat attendu.

Les jeux pour lesquels l'évaluation est particulièrement bonne sont pour la plupart des jeux dont les états finaux sont triviaux. *Nim* en est un exemple type que nous examinerons plus en détail. Il est à noter que les résultats obtenus pour les jeux *Add Game* et *Minichess* semblent faussés par le faible nombre de positions terminales présentes dans les fichiers de test : le résultat est artificiellement bon car les pondérations sont ajustées en fonction de la répartition statistique des faits.

Le tableau ci-dessous présente les résultats obtenus pour le jeu de *Nim*. Les résultats obtenus pour les différentes versions sont comme prévu extrêmement similaires :

Jeu de Nim	**start**	**end**	**start**	**end**
joueur	1	1	2	2
fitness max	94.046919	99.511853	82.061944	99.078385
fitness avg	51.350817	99.511853	48.217363	99.078385
fitness min	17.597419	99.511853	6.429969	99.078385
best ever fitness	94.046919	99.547654	82.061944	99.133515
found at epoch	1	12	1	23
(control player1)	-0.750526	-0.986617	+0.633820	+0.994140
(heap a 0)	+0.589070	+0.205715	-0.038462	-0.122680
(heap b 0)	+0.503067	-0.543373	-0.431609	+0.521943
(heap c 0)	-0.950684	-0.111127	-0.045213	-0.127445
(heap d 0)	-0.139629	+0.452385	+0.110706	-0.300191
(control player2)	+0.994264	+0.994264	-0.648304	-0.969017

Le tableau reprend la valeur des *fitness* minimum, moyen et maximum obtenus en début et en fin d'évolution pour chacun des scores évalués (joueur 1 et 2). L'algorithme génétique a fait évoluer la population pendant 1000 générations mais nous voyons que le meilleur *fitness* a été atteint bien plus tôt dans l'évolution. Les positions finales étant décrites très simplement, des individus au *fitness* élevé sont présents dès la première génération.

Comme prévu, les faits de type `(control ...)` ont été associés à une pondération proche de de $+1$ et -1. En revanche, les faits `(heap ...)` ne sont pas pondérés à 0 comme prévu mais sont associés à des pondérations qui s'annulent réciproquement.

Pour le jeu *Point Grab* nous constatons que les pondérations sont éloignées de celles attendues mais sont pourtant à même de fournir une évaluation relativement bonne du score :

Point Grab	**start**	**end**	**start**	**end**
joueur	1	1	2	2
fitness max	74.581628	95.207031	74.525571	92.304604
fitness avg	65.462245	91.969607	64.484052	87.785093

fitness min	54.041173	86.723343	53.290689	76.347003
best ever fitness	74.581628	95.711399	74.525571	95.439035
found at epoch	1	893	1	709
(points first 95)	-0.158031	-0.080248	+0.591420	-0.324139
(points first 90)	+0.074644	+0.384716	-0.694838	-0.238377
(points first 85)	+0.507135	-0.218068	-0.619337	+0.069631
(points first 80)	-0.198499	+0.384716	-0.903291	+0.217427
(points first 75)	+0.929544	+0.449986	+0.550802	+0.031679
(points first 70)	-0.108891	+0.449986	-0.347231	+0.031679
(points first 65)	-0.624684	+0.384716	+0.102728	+0.067611
(points first 60)	+0.843299	+0.222800	-0.665626	+0.217427
(points first 55)	+0.812348	+0.222800	-0.361287	+0.080018
(points first 50)	+0.188575	+0.101848	-0.733736	+0.009855
(points first 45)	+0.053272	-0.026617	-0.249575	+0.071536
(points first 40)	-0.816450	-0.080248	+0.087840	+0.080018
(points first 35)	-0.781155	-0.218068	-0.559439	+0.080018
(points first 30)	-0.080248	-0.218068	+0.329177	+0.080018
(points first 25)	-0.647636	-0.218068	-0.964950	+0.080018
(points first 20)	-0.975883	-0.218068	+0.822539	+0.080018
(points first 15)	-0.730666	-0.218068	+0.471598	+0.031679
(points first 10)	-0.560227	-0.131104	-0.665312	+0.080018
(points first 5)	+0.056015	+0.449986	+0.270086	-0.352758
(points second 5)	-0.165633	-0.218068	+0.835046	-0.238377
(points second 10)	-0.655914	-0.035841	-0.614490	-0.665626
(points second 15)	-0.862239	-0.080248	-0.151005	-0.665626
(points second 20)	-0.879823	-0.080248	-0.484582	-0.665626
(points second 25)	+0.658062	-0.080248	+0.508464	-0.665626
(points second 30)	+0.329824	-0.080248	+0.412390	-0.665626
(points second 35)	+0.354913	-0.035841	+0.055264	-0.406722
(points second 40)	-0.307333	-0.026617	+0.142828	-0.324139
(points second 45)	+0.146120	-0.035841	-0.052666	-0.238377
(points second 50)	-0.378405	-0.035841	+0.641872	-0.045620
(points second 55)	+0.268318	-0.026617	+0.094166	+0.031679
(points second 60)	+0.210081	-0.035841	+0.622000	+0.080018
(points second 65)	+0.838590	-0.035841	-0.045620	+0.217427
(points second 70)	-0.137574	-0.035841	+0.085744	+0.067611
(points second 75)	-0.677581	-0.035841	+0.817972	+0.416298
(points second 80)	-0.176470	-0.080248	+0.392295	+0.521703
(points second 85)	+0.539091	-0.080248	-0.144024	+0.416298
(points second 90)	-0.585110	-0.026617	+0.803029	+0.918736
(points second 95)	+0.059609	-0.035841	+0.510418	+0.521703
(points second 100)	-0.734875	-0.026617	-0.188130	+0.080018
(step 30)	-0.046694	-0.035841	+0.246174	+0.069631
(step 29)	-0.021106	-0.035841	-0.010959	-0.324139

(step 28)	+0.961276	-0.218068	+0.100964	+0.069631
(step 27)	+0.163922	+0.384716	+0.910260	+0.080018

En les examinant plus attentivement, on constate que les faits de type `(points first X)` avec X >= 50 sont considérés globalement comme positifs pour le joueur 1, et négatifs si X < 50. Les faits de type `(points second ...)` qui concernent donc le joueur 2, sont associés à de faibles pondérations négatives. Une répartition symétrique inverse peut être observée pour l'évaluation du score du joueur 2. Les pondérations atypiques comme celle du fait `(points first 95)` s'expliquent par la répartition statistique de ces faits dans les données analysées : `(points first 95)` n'apparaît qu'une seule fois, ce qui n'est pas suffisant pour ajuster sa pondération conformément au résultat attendu i.e. une pondération fortement positive.

Pour *Roshambo* les pondérations obtenues sont proches de celles prévues. Pour le joueur 1 par exemple, les faits `(beaten white sX)` et `(beaten black sY)` sont positifs pour X > 3 et Y < 3, et inversement négatifs pour X < 3 et Y > 3 ; ceci montre bien que si le joueur 1 a plus de points que la moyenne il peut espérer un bon score mais qu'il est dans une situation également positive si le joueur 2 a, lui, accumulé un nombre de points inférieur à la moyenne. De plus on constate que, pour les faits pour lesquels X et Y sont proches de la moyenne (2 ou 3) et qui sont parfois associés à des matchs nuls (50/50), les pondérations sont moins marquées que pour les faits associés plus franchement à une victoire (0/100). Les faits `(step ...)` ont eux des pondérations variables qui sont avant tout fonction de leur répartition statistique dans les données de départ.

Roshambo	**start**	**end**	**start**	**end**
joueur	1	1	2	2
fitness max	75.002823	89.619063	73.948069	93.648655
fitness avg	52.975430	72.635534	52.764683	90.145067
fitness min	33.232504	49.305827	36.836980	76.423026
best ever fitness	75.002823	93.925558	73.948069	93.973183
found at epoch	1	869	1	303
(beaten white s5)	-0.305834	+0.993640	-0.220509	-0.843115
(beaten white s4)	+0.513596	+0.993640	+0.148909	-0.913245
(beaten white s3)	+0.014420	+0.023482	+0.715243	-0.006051
(beaten white s2)	-0.173623	-0.990729	+0.599512	+0.599512
(beaten white s1)	-0.486961	-0.990729	+0.770393	+0.943164
(beaten white s0)	-0.159562	-0.990729	+0.292833	+0.943164
(beaten black s0)	-0.075216	+0.993640	-0.006051	-0.931056
(beaten black s1)	+0.350889	+0.993640	-0.917482	-0.931056
(beaten black s2)	+0.671488	+0.023482	+0.378758	-0.349192
(beaten black s3)	-0.613088	+0.023482	-0.120033	+0.071729
(beaten black s4)	-0.483339	-0.990729	+0.069266	+0.943164
(beaten black s5)	-0.729368	-0.990729	+0.269933	+0.997320

(step s10)	-0.359403	+0.023482	+0.000956	-0.036325
(step s9)	+0.905853	+0.023482	-0.413258	-0.548732
(step s8)	-0.153551	+0.023482	-0.102679	-0.548732
(step s7)	+0.245244	-0.990729	-0.354312	-0.036325
(step s6)	-0.189935	+0.993640	-0.917503	-0.006051

Le jeu de *Tictactoe*, est celui des quatre jeux examinés pour lequel le *fitness* observé est le plus bas. Ce résultat est sans surprise puisque nous avions prévu que l'algorithme génétique ne serait pas à même d'effectuer une combinaison non linéaire des faits susceptibles de représenter les états gagnants.

Tictactoe	**start**	**end**	**start**	**end**
joueur	1	1	2	2
fitness max	71.258860	89.786112	71.285708	89.591279
fitness avg	52.416925	87.222138	51.875115	88.323960
fitness min	32.191960	70.920141	26.127311	69.074095
best ever fitness	71.258860	89.878657	71.285708	89.591279
found at epoch	1	982	1	870
(control oplayer)	+0.928607	+0.965854	-0.849359	-0.970338
(cell 1 1 o)	+0.668338	+0.010632	+0.306441	-0.002113
(cell 1 2 o)	-0.433775	+0.010632	-0.605471	-0.002113
(cell 1 3 o)	-0.895107	-0.025442	+0.820391	-0.002113
(cell 2 1 o)	+0.696466	-0.025442	-0.141491	-0.002113
(cell 2 2 o)	-0.493502	-0.025442	-0.034904	-0.002113
(cell 2 3 o)	-0.281135	+0.010632	-0.624091	-0.002113
(cell 3 1 o)	+0.474570	+0.010632	+0.821794	-0.002113
(cell 3 2 o)	+0.500130	-0.025442	-0.814759	-0.002113
(cell 3 3 o)	-0.343504	+0.010632	-0.835831	-0.002113
(cell 1 1 b)	-0.321064	+0.010632	+0.349476	-0.002113
(cell 1 2 b)	+0.723621	+0.010632	+0.195624	-0.002113
(cell 1 3 b)	-0.130172	+0.010632	-0.349120	-0.002113
(cell 2 1 b)	-0.066497	-0.025442	+0.306719	-0.002113
(cell 2 2 b)	+0.252164	+0.010632	-0.113125	-0.002113
(cell 2 3 b)	-0.785764	+0.010632	+0.853166	-0.002113
(cell 3 1 b)	+0.199950	+0.010632	+0.339439	-0.002113
(cell 3 2 b)	+0.090889	+0.010632	-0.115702	-0.002113
(cell 3 3 b)	+0.896348	+0.010632	-0.622648	-0.002113
(cell 1 1 x)	-0.108154	+0.010632	-0.153990	-0.002113
(cell 1 2 x)	+0.750077	+0.010632	+0.357086	-0.002113
(cell 1 3 x)	+0.544933	+0.010632	-0.457794	-0.002113
(cell 2 1 x)	+0.617749	-0.025442	+0.295900	-0.002113
(cell 2 2 x)	+0.390862	+0.010632	-0.978351	-0.002113
(cell 2 3 x)	+0.117003	+0.010632	+0.712966	-0.002113
(cell 3 1 x)	+0.968376	+0.010632	+0.898069	-0.002113

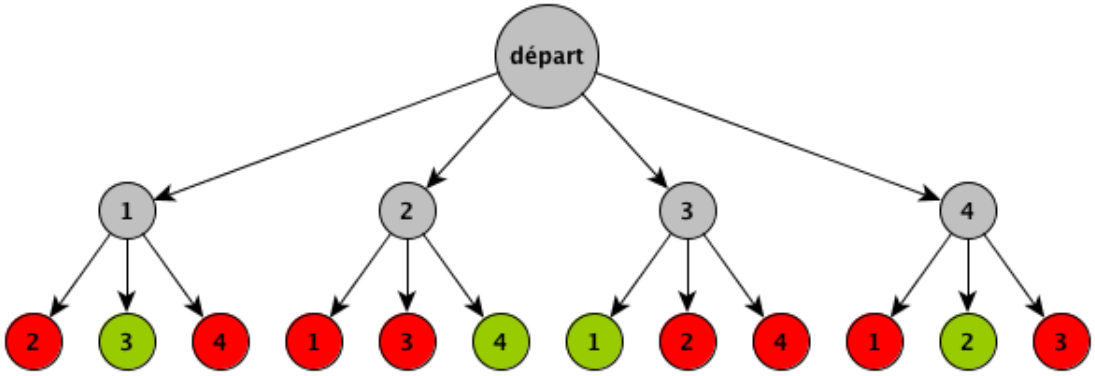


FIGURE 4.4 – Arbre du jeu *Paire* : en vert les positions gagnantes pour le joueur 1, en rouge les positions perdantes

(cell 3 2 x)	-0.505688	+0.010632	-0.160565	-0.002113
(cell 3 3 x)	-0.460010	+0.010632	-0.154085	-0.002113
(control xplayer)	-0.409908	-0.981519	+0.479580	+0.964322

Nous constatons que pour la prévision du score, notre programme a tiré parti des faits indiquant le joueur courant `(control ...)`. Si c'était au joueur `oplayer` de de jouer quand la partie s'est stoppée, c'est que le joueur `xplayer` a posé le dernier symbole et il y a de forte chance que la partie se soit stoppée parce qu'il a créé un alignement. Le jeu peut aussi se solder par un match nul mais dans ce cas, le joueur gagne quand même 50 points ce qui confirme l'aspect positif de ce fait sur son score. Tous les autres faits sont associés à une faible pondération positive ou négative et nous constatons que l'influence positive des `x` pour le joueur `xplayer` et des `o` pour le joueur `oplayer` n'a pas été découverte.

L'examen des résultats de notre banc de test nous montre que l'évaluation du score effectuée par un algorithme génétique n'est, comme nous le prévoyons, fiable que pour les positions finales dont les faits ont une influence directe (positive ou négative) sur le score.

4.3.4 Paire : jeu spécifique pour tests complémentaires

Afin de confirmer l'incapacité de notre algorithme à résoudre un problème non linéaire, nous avons créé un jeu pour un seul joueur, simpliste mais représentatif de ce problème. Voici les règles de ce jeu que nous nommerons *Paire* : une liste de quatre chiffres uniques 1, 2, 3 et 4 est placée devant le joueur. Il doit choisir deux chiffres distincts. Si la somme de ces chiffres est paire il gagne, sinon il perd.

En examinant l'arbre du jeu présenté sur la figure 4.4, nous constatons que le joueur 1 est susceptible de gagner 1/3 du temps. Nous constatons également que le problème ainsi représenté s'apparente à un XOR, un *ou exclusif*. Pour pouvoir tester

Paire	start	end	start	end
calcul du fitness	equation 4.3		equation 4.4	
fitness max	69.610097	74.138856	75.331613	81.365945
fitness avg	51.206012	72.227391	64.289619	81.337872
fitness min	28.515146	58.059827	33.700000	81.120549
best ever fitness	69.610097	74.646824	75.331613	81.365945
found at epoch	1	274	1	18
(nombre 1)	-0.117391	-0.008419	-0.055122	+0.215740
(nombre 2)	-0.988566	-0.985841	-0.437782	-0.968944
(nombre 3)	-0.454362	-0.008419	-0.048535	+0.279698
(nombre 4)	-0.833489	-0.985841	+0.203960	-0.993255

TABLE 4.4 – Résultats obtenus pour le jeu *Paire*

le comportement de notre programme face à ce problème, nous avons généré au hasard (directement sans utiliser Ary) 3000 positions terminales accompagnées de leur score :

```
((0) (nombre 1) (nombre 2))
((0) (nombre 4) (nombre 1))
((100) (nombre 3) (nombre 1))
((0) (nombre 2) (nombre 1))
((0) (nombre 3) (nombre 4))
((100) (nombre 4) (nombre 2))
```

Si le programme se comporte comme prévu, il devrait être incapable de résoudre ce problème : il pourrait prédire efficacement la victoire et atteindre 33% d'estimation juste, mais dans sa quête d'un *fitness* élevé, il tentera plutôt de prédire efficacement la défaite et devrait s'approcher plutôt d'un *fitness* autour de 66%.

Si on examine le tableau des résultats (voir les deux premières colonnes de la table 4.4), on se rend compte que le meilleur *fitness* atteint est en fait plus élevé, autour de 74%. Pourtant, comme prévu, il a pris le parti d'évaluer de manière juste les positions perdantes en les associant à une pondération proche de -1 ($-0.008419 + -0.985841 = -0.994259$) et il évalue les combinaisons gagnantes de manière erronée : le coupe 4-2 donne une pondération plafonnée à -1 et donc un score de 0 ; et le coupe 1-3 obtient une pondération proche de zéro et donne donc un score de 50. C'est ce score de 50 qui explique l'obtention d'un *fitness* supérieur à 66%, l'erreur produite sur l'évaluation des scores gagnants est ainsi minimisée, sans que le problème du ou exclusif ne puisse pour autant être résolu.

Suite à cette expérience, l'idée nous est venue d'évaluer l'erreur plus finement en augmentant le poids des écarts les plus importants entre le score réel et le score prévu. Pour cela, nous avons modifié le calcul de l'erreur (voir équation 4.3) en remplaçant la valeur absolue par le carré de l'erreur :

$$F = \frac{\sum_{i=N}^{i=0} 100 - \frac{(T(P_i) - V(P_i))^2}{100}}{N} \tag{4.4}$$

Il se trouve que cette évaluation de l'erreur permet à l'algorithme génétique de converger vers une solution ou l'erreur produite dans l'évaluation des scores gagnants est encore d'avantage minimisée et donc le *fitness* encore plus élevé.

Nous constatons qu'il est donc possible d'avoir une évaluation fiable à environ 80% sans avoir aucunement réussi à résoudre le problème : trouver la combinaison des faits qui est signifiante pour l'évaluation du score. Pour voir si l'algorithme de *Neuro-Evolution of Augmenting Topologies* de Stanley peut réussir dans cette tâche i.e. peut être capable d'une analyse non linéaire des faits, nous en avons réalisé l'implémentation que nous présentons dans le chapitre suivant.

5 Expérimentations avec NEAT

Après une première phase de tests qui nous a permis de mieux connaître les spécificités de notre jeu de données et de mieux comprendre comment la pondération de chaque fait GDL peut permettre d'approcher l'évaluation d'une position finale d'un jeu grâce à un algorithme génétique, nous avons implémenté l'algorithme de *Neuro-Evolution of Augmenting Topologies* de Stanley [Stanley and Miikkulainen, 2002]. Nous espérons qu'un réseau de neurones saura évaluer plus finement le score correspondant aux positions finales de jeux en créant des combinaisons de faits cohérentes.

Dans une première partie nous présenterons les détails de notre implémentation de NEAT, ensuite nous présenterons les tests que nous avons réalisés et leur résultat.

5.1 Implémentation de NEAT

La lecture du code de NEAT implémenté en C++ par Stanley apparaît complexe à cause de la présence de différentes variantes des mêmes fonctions contrôlant les mutations, le *crossover* et la spéciation. De nombreux paramètres sont directement spécifiés dans ces fonctions et s'ajoutent aux déjà très nombreux paramètres regroupés dans un fichier de configuration. De prime abord, il est difficile de distinguer les fonctionnalités qui ne sont pas utilisées [1] des fonctionnalités qui s'avèrent fondamentales. Afin de mieux cerner la structure de base de NEAT, nous avons donc fondé notre implémentation [2] sur le code de Mat Buckland [Buckland and Collins, 2002]. Dans un ouvrage de vulgarisation, il propose un exemple d'implémentation de NEAT en C++ que Stanley recommande pour une première approche de son système. Nous nous sommes donc largement inspiré de cette implémentation tout en gardant un œil sur la version proposée par Stanley [Stanley, 2013].

1. Nous parlons des fonctionnalités comme les *traits* ou le vol de progéniture (*stolen babies*) qui ont été implémentées dans l'idée de futurs développements.
2. Le code de notre programme est disponible dans un dépôt GIT à cette adresse : https://github.com/alinehuf/neattest.git

5.1.1 Génomes et Phénotypes

Dans leurs implémentations, Stanley comme Buckland séparent les *génomes*, les structures de données représentant les réseaux de neurones utilisés dans les étapes de reproduction et de mutation, des *phénotypes*, les mêmes réseaux représentés par des structures de données différentes dans le but de calculer leur réponse à des données d'entrée. Nous avons choisi de suivre ce choix d'implémentation qui permet de simplifier le reste du code.

Les génomes sont composés de la liste des liens et des connexions qui les composent. Nous avons donc utilisé des structures (voir figure 5.1) pour représenter les neurones `sNeuronGene` et les liens `sLinkGene`. Chacun de ces neurones est muni :

- d'un identifiant unique qui nous permettra de réaliser le *Historical Marking*,
- du type du neurone (input, bias, output ou hidden),
- d'un indicateur booléen pour savoir si le neurone fait l'objet d'une connexion récurrente (utile lors de l'étape de mutation),
- d'un taux d'activation, compris entre 0 et 1, déterminant la pente de la fonction sigmoïde utilisée dans le neurone,
- et de coordonnées utiles lors de la représentation graphique des génomes mais également pour déterminer si un nouveau lien créé sera récurrent.

Les liens `sLinkGene` sont munis :

- d'un identifiant (numéro d'innovation) servant de *Historical Marking*,
- des identifiants des deux neurones liés,
- d'une pondération,
- et de deux indicateurs booléens pour savoir si le lien est bien actif et si il est récurrent,

La structure représentant un génome `sGenome` possède aussi un pointeur `pPhenotype` vers le phénotype correspondant, des variables pour stocker la valeur du *fitness*, le *fitness* ajusté utilisé pour la reproduction, et le taux de reproduction. Les nombres d'entrées et de sorties sont également stockées dans le génome pour des raisons pratiques (ces valeurs sont communes à tous les génomes et auraient pu être stockées dans la structure représentant la population). Le nombre de neurones récurrents présents dans le génome est également stocké pour ajuster la réponse du réseau lors de son évaluation.

Le phénotype créé pour pouvoir évaluer la réponse du réseau reprend une partie des mêmes éléments (voir figure 5.2). Cependant, les neurones possèdent cette fois une liste de connexions entrantes et sortantes permettant de calculer la diffusion du signal dans le réseau.

La variable `iDepth` permet de stocker la profondeur du réseau pour connaître le nombre d'itérations nécessaires pour propager le signal des entrées jusqu'aux sorties. Le nombre d'itérations est augmenté en fonction du nombre de neurones récurrents présents dans le réseau.

Il n'est pas sûr que cette séparation entre génotype et phénotype soit totalement

```
/*************************************************************************
 * neuron gene
 *************************************************************************/

// dSliptX, dSplitY : usefull for graphic representation of the genotype and
// also to determines if a link is reccurent
typedef struct neuronGene {
  int iId;
  neuron_type eNeuronType;
  bool bRecurrent;
  double dSigmoidCurvature; // activation rate AR y = 1 / (1 + e^(-x/AR))
  double dSplitX, dSplitY;
} sNeuronGene;

/*************************************************************************
 * link gene
 *************************************************************************/

typedef struct linkGene {
  int iInnovId;
  int iFromNeuron;
  int iToNeuron;
  double dWeight;
  bool bEnabled;
  bool bRecurrent;
} sLinkGene;

/*************************************************************************
 * genome
 *************************************************************************/

typedef struct genome {
  int iId;
  sNeuronGene ** vNeurons;
  int iNumNeurons;
  int iTotalNeurons;
  sLinkGene ** vLinks;
  int iNumLinks;
  int iTotalLinks;
  sPhenotype * pPhenotype;
  double dFitness;
  double dAjustedFitness;
  double dAmountToSpawn;
  int iNumInputs, iNumOuputs;
  int iNumRecur; // to iterate few more times (recurrent inputs)
} sGenome;
```

FIGURE 5.1 – Structures utilisées pour représenter les *génomes*, les réseaux de neurones évolués avec NEAT

```
/*************************************************************************
 * link of the final ANN (phenotype)
 *************************************************************************/
typedef struct neuron sNeuron;

typedef struct {
  //pointers to the neurons this link connects
  sNeuron * pNeuronIn;
  sNeuron * pNeuronOut;
  //the connection weight
  double dWeight;
  //is this link a recurrent link?
  bool bRecurrent;
} sLink;

/*************************************************************************
 * neuron of the final ANN (phenotype)
 *************************************************************************/

struct neuron {
  int iID;                 // its identification number
  neuron_type eNeuronType; // what type of neuron is this?
  double dActivationResponse; // sets the curvature of the sigmoid function
  sLink ** vLinksIn;       // all the links coming into this neuron
  int iNumLinksIn;
  int iTotalLinksIn;
  sLink ** vLinksOut;      // and out
  int iNumLinksOut;
  int iTotalLinksOut;
  double dSumActivation;   // sum of weights x inputs
  double dOutput;          // the output from this neuron
  //used in visualization of the phenotype
  double dSplitX, dSplitY;
};

/*************************************************************************
 * the final ANN (phenotype)
 *************************************************************************/

typedef struct {
  sNeuron ** vNeurons;
  int iNumNeurons;
  int iDepth; //the depth of the network
} sPhenotype;
```

FIGURE 5.2 – Structures utilisées pour représenter les *phénotypes* permettant l'évaluation de la réponse du réseau

justifiée. La complication du code qui résulterait de la maintenance des listes de liens entrants et sortants pour chaque neurone ne serait pas forcément supérieure à celle résultant du code ajouté pour la génération des phénotypes. Cependant ne disposant pas du temps suffisant pour tester une autre implémentation, nous avons préféré suivre ce modèle proposé.

5.1.2 Innovations

La population de départ est composée de génomes identiques totalement connectés avec des entrées, un biais et des sorties. La topologie de ces réseaux étant identique, les neurones et les liens reçoivent tous les mêmes listes d'identifiants. Ces identifiants sont reportés dans une table d'innovations qui est organisée comme suit (exemple pour des réseaux de 2 entrées et 1 sortie) :

```
innovTable (7 innovations, next neuron ID = 4) :
innov 0  NEW_NEURON in=-1 out=-1 neuronID=0  neuronType=INPUT
innov 1  NEW_NEURON in=-1 out=-1 neuronID=1  neuronType=INPUT
innov 2  NEW_NEURON in=-1 out=-1 neuronID=2  neuronType=BIAS
innov 3  NEW_NEURON in=-1 out=-1 neuronID=3  neuronType=OUTPUT
innov 4  NEW_LINK   in=0  out=3  neuronID=0  neuronType=NONE
innov 5  NEW_LINK   in=1  out=3  neuronID=0  neuronType=NONE
innov 6  NEW_LINK   in=2  out=3  neuronID=0  neuronType=NONE
```

Chaque innovation est donc identifiée par un numéro. Il ne faut pas confondre ce numéro avec l'identifiant des neurones `neuronID`, car les innovations peuvent être également des liens ajoutés au réseau. Chaque innovation possède un type (NEW_NEURON ou NEW_LINK). Les indications `in` et `out` permettent de repérer, pour chaque lien, les neurones d'entrée et de sortie (grâce à leur identifiant unique) mais ils permettent également d'indiquer entre quels neurones un nouveau neurone est inséré lors d'une mutation :

```
innov 7  NEW_NEURON in=1  out=3  neuronID=4  neuronType=HIDDEN
innov 8  NEW_LINK   in=1  out=4  neuronID=0  neuronType=NONE
innov 9  NEW_LINK   in=4  out=3  neuronID=0  neuronType=NONE
```

Dans cet exemple, on voit que le nouveau neurone a été inséré entre les neurones 1 et 3. Le lien 4 a été désactivé dans le génome puis remplacé par deux nouveaux liens reliant le neurone 1 au nouveau neurone 4 et ce nouveau neurone 4 au neurone 3. L'ajout d'un nœud a donc abouti à la création de 3 nouvelles innovations.

Enfin, le type du neurone est également stocké, la valeur `NONE` étant utilisée pour les liens.

Chaque fois qu'une mutation intervient dans un génome, la table d'innovations est consultée pour vérifier qu'une mutation similaire n'a pas déjà été effectuée. Si

oui, le nouveau lien ou le nouveau neurone ajouté est identifié respectivement par le même neuronID ou le même numéro d'innovation. Comme nous l'avons vu au paragraphe 3.2.1, c'est cet *Historical Marking* qui permet ensuite de réaliser l'étape de reproduction des réseaux entre eux.

5.1.3 Spéciation

Pour permettre aux génomes d'évoluer et de complexifier leur topologie sans être systématiquement éliminés par la pression de la sélection naturelle, nous avons vu (voir paragraphe 3.2.2) qu'il est nécessaire d'opérer une spéciation. Un vecteur contenant les espèces est donc créé (voir la structure `sSpecies` sur la figure 5.3). Chaque organisme est comparé au représentant de chaque espèce (`sLeader`) et s'il est suffisamment semblable il est ajouté au vecteur des membres `vMembers`. Sinon, si aucune espèce ne lui convient, une nouvelle espèce est créée et une copie de l'organisme est conservée comme modèle de comparaison.

```
1  typedef struct {
2    sGenome * sLeader; // keep a local copy of the first member of this
         species
3    sGenome ** vMembers; // pointers to all the genomes within this species
4    int iNumMembers;
5    int iTotalMembers;
6    int iSpeciesId;    // the species needs an identification number
7    double dBestFitness;  // best fitness found so far by this species
8    int iGensNoImprovement; // generations since fitness has improved
9    int iAge;
10   double dSpawnsRqd; // how many of this species should be spawned for next
         pop
11 } sSpecies;
```

FIGURE 5.3 – Structure utilisée pour représenter une espèce

L'évolution des espèces est contrôlée. Plus une espèce vieillit (`iAge`), plus elle reçoit un handicap lors de la reproduction. De même, une espèce qui compte de plus en plus de membres reçoit un handicap pour éviter qu'elle ne finisse par englober toute la population ; ceci est effectué par *fitness sharing*, la division du *fitness* par le nombre de membres de l'espèce. Les espèces les plus jeunes sont au contraire avantagées de manière à stimuler l'apparition de topologies originales susceptibles d'apporter de nouvelles aptitudes à la population.

Lors de la création d'une nouvelle génération de la population, chaque espèce est vidée de ses membres. Les espèces qui sont restées durant trop de générations sans évoluer (`iGensNoImprovement`) sont détruites. Les individus de la nouvelle population sont alors comparés au `sLeader` de chaque espèce et répartis dans l'espèce qui leur convient.

Il peut arriver que le seuil de comparaison des génomes soit trop bas et que donc chaque individu se retrouve seul représentant de son espèce ou au contraire qu'il soit trop haut et qu'une seule espèce contienne toute la population. Pour éviter une explosion du nombre d'espèces, un nombre maximum conseillé est indiqué, et s'il est dépassé, le seuil est diminué progressivement à chaque génération ; sinon, si le nombre d'espèces est trop faible, le seuil est augmenté.

Une autre implémentation possible aurait été d'augmenter le seuil, de vider les membres de chaque espèce, et de recommencer la spéciation jusqu'à ce que tous les individus soient ajoutés à une espèce ou à une autre sans dépasser le nombre maximum. Cependant, cette façon de procéder nous a paru moins souple, le nombre d'espèces pouvant ponctuellement être dépassé suite à un nombre exceptionnellement haut de mutations dû aux fluctuations des tirages aléatoires. Ce nombre d'espèces peut alors décroître rapidement si ces nouvelles mutations ne sont pas transmises aux générations suivantes.

Ce contrôle du nombre d'espèces est suggéré par Mat Buckland mais n'est pas implémenté dans son programme. De même, aucun contrôle n'est opéré par Stanley dans son implémentation (Neat 0.1).

5.1.4 Calcul du fitness

Avant de pouvoir ajuster le *fitness* des génomes en fonction de leur espèce, ce *fitness* doit être calculé. Comme précédemment (voir paragraphe 4.2.1) la performance des individus est évaluée en calculant l'erreur moyenne d'évaluation du score correspondant à chaque état final.

La différence fondamentale par rapport à notre expérience précédente est que nous cherchons à voir si les génomes/réseaux sont capables de découvrir dans leur topologie la corrélation entre des combinaisons de faits et le score des deux joueurs. Nos génomes ont donc deux sorties, une pour chaque score à évaluer. Notre fonction d'évaluation prend donc en compte l'erreur commise pour les deux sorties. Pour faciliter l'évaluation le score réel a été ramené à une valeur entre 0 et 1 :

$$E = \frac{\sum_{i=N}^{i=0} \sum_{j=J}^{j=0} |T_j(P_i) - V_j(P_i)|}{N * J} \tag{5.1}$$

$$F = (1 - E)^2 * 100 \tag{5.2}$$

Avec J le nombre de joueurs, N le nombre d'état finaux examinés, P_i l'état final courant, T_j le score réel pour le joueur j et V_j le même score évalué par le réseau. La valeur du *fitness* F d'un individu est finalement exprimée sous forme d'un pourcentage de réussite en fonction de l'erreur moyenne E. Comme le suggèrent Stanley et Miikkulainen [Stanley and Miikkulainen, 2002], nous utilisons le carré du *fitness* pour augmenter l'impact des plus grandes erreurs d'évaluation.

5.1.5 Crossover et mutations

Une fois le *fitness* de chaque individu calculé puis ajusté en fonction de son espèce (handicap ou avantage), il est possible de procéder à la création d'une nouvelle génération.

Pour créer un phénomène d'élitisme, le champion de chaque espèce est cloné dans la nouvelle population. Ensuite si l'espèce ne compte qu'un seul représentant, il est copié et soumis à des mutations. Si l'espèce compte plusieurs représentants, un tirage au sort est effectué, parmi les meilleurs individus (en fonction d'un taux de survie défini dans le paramétrage) pour sélectionner deux parents. Cette sélection est soumise au taux de croisement fixé dans les paramètres et dans le cas où le tirage au sort échoue un trop grand nombre de fois pour trouver deux individus distincts, l'individu est copié à l'identique et soumis à des mutations.

Comme le nombre d'individus créés pour chaque espèce est forcément un nombre entier, il peut arriver que la nouvelle population compte moins de membres que la précédente. Dans ce cas un *tournoi de sélection* (*tournament selection*) est organisé : pendant un nombre fini d'itérations (déterminé dans les paramètres) un individu est sélectionné au hasard dans toute la population, si son *fitness* est plus élevé que l'individu précédemment sélectionné, il le remplace. Le clone de l'individu qui remporte le tournoi est ajouté à la population.

Les mutations ajoutées aux génomes sont de quatre sortes et quatre paramètres déterminent la probabilité que ces mutations interviennent.
La première consiste à perturber les pondérations des connexions du réseau. Un paramètre détermine la probabilité qu'une pondération soit remplacée par une tout autre valeur. Si elle n'est pas remplacée, la pondération est plus ou moins perturbée en fonction d'un taux défini à l'avance.

Une autre mutation peut toucher le taux d'activation des neurones. Un paramètre détermine alors l'ampleur de la perturbation de ce taux qui détermine la pente de la fonction sigmoïde présente dans le neurone.

La troisième mutation peut être l'ajout d'une connexion dans le réseau. Dans ce cas un paramètre détermine si la connexion peut être récurrente ; dans le cas contraire, pendant un nombre d'itérations limité, le programme tente de trouver deux neurones non reliés ensemble, et le cas échéant ajoute un lien entre les deux.

La dernière mutation consiste à ajouter un neurone. Nous avons vu, au paragraphe 5.1.2, que dans ce cas le programme choisit une connexion puis la scinde en deux pour ajouter un neurone. Le choix de la connexion est aléatoire sauf pour les réseaux de petite taille. En dessous de 5 neurones cachés, un réseau est trop petit pour pouvoir choisir le lien aléatoirement. Un effet de *chaînage* risque de survenir : divers neurones sont ajoutés sur ce qui est à l'origine une seule et même connexion amenant une complexification totalement inutile du réseau (voir figure 5.4). Pour éviter cela, le choix de la connexion est biaisé pour les réseaux de petite taille en faveur des connexions les plus anciennes.

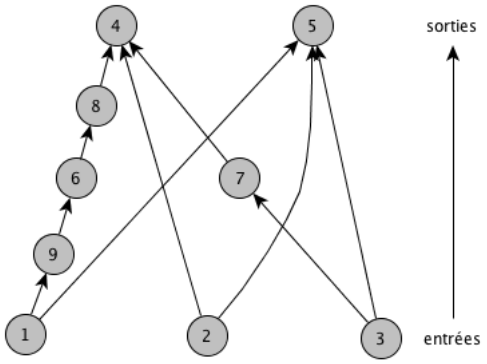


FIGURE 5.4 – Effet de *chaînage* qui peut survenir sur les réseaux de petite taille lorsque des nœuds sont ajoutés aléatoirement. La même connexion se retrouve scindée de manière répétitive provoquant l'apparition d'une chaîne de neurones (1, 9, 6, 8, 4). La probabilité de choisir un des liens formant cette chaîne pour ajouter le prochain nœud augmente avec leur nombre provoquant une réaction en chaîne. Cette complexification du réseau est inutile et n'apporte pas de modification intéressante de la topologie.

5.1.6 Paramétrage

La description de l'implémentation de NEAT que nous venons de faire révèle que notre programme fait usage d'une grande quantité de paramètres pour contrôler le système. Notre implémentation utilise un fichier de paramétrage spécifique passé en paramètre, permettant d'ajuster les réglages en fonction des expériences. Voici un récapitulatif de ces paramètres avec les valeurs que nous avons utilisées dans nos premières expérimentations. Ces valeurs sont celles indiquées par Stanley [Stanley and Miikkulainen, 2002] et permettent d'après lui de résoudre le problème du XOR dans 100% des simulations.

dWeightMutationRate	0.8	Probabilité de mutation du poids d'une connexion (voir remarques ci-dessous)
dMaxWeightPerturbation	0.9	Perturbation maximum infligée aux poids d'une connexion (voir remarques ci-dessous)
dProbabilityWeightReplaced	0.1	Probabilité que le poids d'une connexion soit remplacé par une tout autre valeur (voir remarques ci-dessous)
dActivationMutationRate	0.1	Probabilité de mutation de l'activation d'un neurone (voir remarques ci-dessous)
dMaxActivationPerturbation	0.1	Perturbation maximum infligée au taux d'activation d'un neurone (voir remarques ci-dessous)
dChanceAddLink	0.05	Probabilité de mutation par ajout d'une connexion

dChanceAddRecurrentLink	0	Probabilité d'ajout d'une connexion récurrente (les connexions récurrentes n'étant pas souhaitable pour le problème que nous souhaitons traiter, nous avons mis ce paramètre a 0. La valeur conseillée est de 0.05 quand ces connexions sont souhaitées)
iNumTrysToFindLoop	5	Nombre d'itérations maximum pour trouver un neurone pouvant faire l'objet d'une connexion récurrente
iNumTrysToAddLink	5	Nombre d'itérations maximum pour trouver deux neurones pouvant faire l'objet d'une connexion
dChanceAddNode	0.03	Probabilité d'ajout d'un nouveau neurone
iNumTrysToFindOldLink	5	Nombre d'itérations maximum pour trouver une ancienne connexion pouvant être scindée
dCompatibilityThreshold	0.6	Limite de compatibilité entre deux génomes (utilisé pour la spéciation)
dExcessGenesCoef	1.0	Poids des gènes en excès dans le calcul de la compatibilité (constante $C1$ voir paragraphe 3.2.2)
dDisjointGenesCoef	1.0	Poids des gènes disjoints dans le calcul de la compatibilité (constante $C2$ voir paragraphe 3.2.2)
dWeightDiffCoef	0.4	Poids de l'écart moyen des pondérations dans le calcul de la compatibilité (constante $C3$ voir paragraphe 3.2.2)
iMaxSpecies	20	Nombre d'espèce maximum souhaitable (ce chiffre peut être dépassé, voir paragraphe 5.1.3)
iYoungBonusAgeThreshhold	10	Age en dessous duquel une espèce reçoit un bonus pour aider son développement
dYoungFitnessBonus	1.1	Bonus accordé pour les jeunes espèces (coefficient multiplicateur). Stanley ne donne aucun bonus (paramètre à 1.0), Buckland le met à 1.3. Nous avons empiriquement ramené cette valeur à 1.1 qui donne un résultat globalement satisfaisant sur les différents jeux testés.

iOldAgeThreshold	30	Age au dessus duquel une espèce reçoit un handicap. Buckland [Buckland and Collins, 2002] met ce paramètre à 50 ; cela s'est révélé bien trop élevé au cours de nos tests. Stanley propose 15 : cette valeur très basse risque de pénaliser les espèces les plus adaptées (voir paragraphe 5.2.3)
dOldAgePenalty	0.1	Handicap infligé aux espèces âgées (coefficient multiplicateur). Buckland met ce paramètre à 0.7 ce qui s'est révélé être un handicap trop faible. Stanley propose 0.01 ce qui met trop de pression aux espèces âgées (voir paragraphe 5.2.3)
dSurvivalRate	0.2	Taux de survie appliqué aux individus d'une espèce pour déterminer la part des individus au *fitness* le plus élevé qui prennent part à la reproduction.
dCrossoverRate	0.7	Probabilité de croisement pour la reproduction
iNumTrysToFindMate	5	Nombre d'itérations maximum pour trouver deux génomes parents
iNumGensAllowedNoImprov	15	Nombre de générations maximum avant qu'une espèce qui n'évolue pas soit éliminée
iNumIndividuals	150	Nombre d'individus de la population (= nombre de génomes)
iNumEpoch	1000	Nombre de génération pendant lesquelles la population évolue
iDumpEvery	200	Paramètre ajouté pour demander un rapport toutes les N générations (liste des espèces et génomes de chaque leader)

En examinant le code de NEAT (version 0.1) implémenté par Stanley, la mutation des poids des connections s'est révélée plus complexe que ce qu'il semble décrire dans son article et qui correspond à l'implémentation de Buckland. Un biais est ajouté en faveur des connexions plus récentes en partant du principe que les connexions anciennes sont déjà ajustées :

> « *The power of mutation will rise farther into the genome on the theory that the older genes are more fit since they have stood the test of time* » (commentaire du code de Neat0.1 [Stanley, 2013])

Nous avons donc dans certains tests remplacé les paramètres `dWeightMutationRate`, `dMaxWeightPerturbation` et `dProbabilityWeightReplaced` par d'autres paramètres utilisés par Stanley et modifié notre fonction de mutation des poids. Pour chaque

connexion, la fonction teste d'abord la probabilité d'effectuer une mutation ordinaire et si elle est choisie, elle tente soit de perturber la pondération, soit de la remplacer. Si la mutation ordinaire n'est pas choisie et que la connexion est récente, elle retente une perturbation ou un remplacement avec des probabilités plus importantes, sinon aucune mutation n'est apportée.

dProbOrdinaryMutation	0.5	Probabilité d'effectuer une mutation qui touche n'importe quelle connexion.
dMeanWeightPerturbation	0.4	Probabilité moyenne de perturber le poids d'une connexion : si cette mutation touche n'importe quelle connexion cette probabilité est diminuée de 10%, si elle touche une connexion récente (les derniers 80% du génome) elle est augmentée de 10%
dMeanWeightRemplacement	0.2	Probabilité moyenne de remplacer totalement le poids d'une connexion : si cette mutation touche n'importe quelle connexion cette probabilité est diminuée de 10%, si elle touche une connexion récente (les derniers 80% du génome) elle est augmentée de 10%
dMutationPower	1.8	Intervalle en plus ou moins [-1.8, 1.8] dans lequel sera tirée au hasard la valeur de la perturbation ou de la nouvelle pondération. Stanley monte cette valeur à 2.5 dans certaines de ses expérimentations mais dans notre cas cela introduit une trop grande perturbation.

Stanley indique également [Stanley and Miikkulainen, 2002] qu'il a utilisé une fonction d'activation de pente constante dans ses neurones alors que l'implémentation de Buckland propose d'utiliser une sigmoïde de pente variable qui est soumise également à des mutations contrôlées par les paramètres `dActivationMutationRate` et `dMaxActivationPerturbation`. Nos premiers tests ont révélé que cette variation de la fonction d'activation était contre-productive et perturbait le développement des réseaux. Nous avons donc fini par désactiver cette fonctionnalité et remplacer notre fonction d'activation par une sigmoïde de pente constante conseillée par Stanley :

$$1/(1 + (exp(-4.924273 * S))) \tag{5.3}$$

avec S la somme des activations des neurones

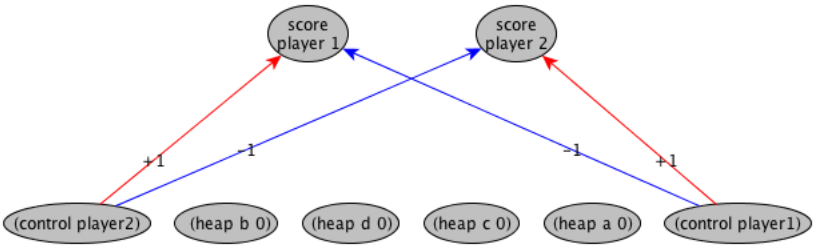


FIGURE 5.5 – Graphe de la topologie espérée pour le jeu de *Nim* : les faits n'ayant pas d'influence sur le score ne devrait plus être connectés à celui-ci. Les faits directement corrélés devrait avoir une pondération de +1 ou -1

5.2 Tests effectués avec NEAT

Nous avons limité nos tests à cinq jeux représentatifs : *Nim* dont les états finaux sont triviaux et le score, qui devrait être évalué sans difficulté par NEAT, est binaire ; *Paire* qui permettra de tester la capacité de notre programme à générer un réseau capable de résoudre le problème du XOR ; *Point Grab* qui nous permettra de vérifier si l'ajustement des pondérations des connexions se fait correctement ; *Roshambo* dont le score peut être déduit par une combinaison de faits ou directement par la pondération des faits ; et *Tictactoe* dont l'estimation juste du score nécessite de découvrir des combinaisons de faits complexes et qui de ce fait constitue un défi pour notre programme.

5.2.1 Hypothèses de départ

Pour chacun des cinq jeux, nous voulons voir si NEAT saura découvrir la topologie du réseau permettant d'estimer le score avec justesse. Les différents faits apparaissant dans les états finaux correspondent chacun à une entrée du réseau. Ainsi, par exemple, pour le jeu de *Tictactoe* nous avons 27 entrées pour les 3 états possibles de chaque case de la grille, plus deux entrées correspondant aux faits indiquant quel joueur a la main. Pour chaque position terminale que nous allons tester, le signal sur chaque entrée est 1 si le fait est vrai (présent dans la liste de faits associée à cette position) et 0 si le fait est faux. Les topologies idéales que nous espérons obtenir sont les suivantes :

Pour *Nim* (voir figure 5.5) nous espérons obtenir un réseau qui relie les faits `(control ...)` aux sorties avec des connections inhibitrices (en bleu) et excitatrices (en rouge) en fonction du score. Les faits `(heap ...)` n'ayant pas d'influence sur le score, nous espérons que NEAT saura supprimer toute liaison à partir de ces entrées.

Pour *Point Grab* (voir figure 5.6) nous espérons que les pondérations s'ajusteront pour chaque fait en fonction de leur contribution au score.

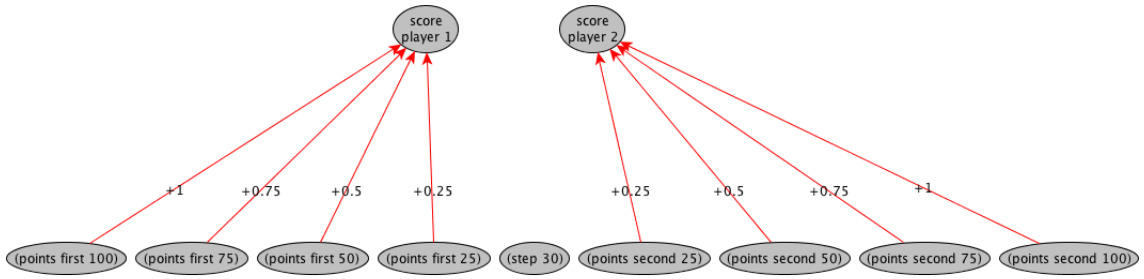


FIGURE 5.6 – Graphe partiel de la topologie espérée pour le jeu de *Point Grab* : les faits n'ayant pas d'influence sur le score ne devrait plus être connectés à celui-ci. Les faits indiquant le nombre de points accumulés ont une influence proportionnelle au nombre de points qu'ils représentent.

Pour *Roshambo*, il est possible que le problème soit résolu par ajustement des pondérations associées aux faits (voir figure 5.7), mais dans l'idéal, nous aimerions voir NEAT découvrir une topologie présentant une couche cachée où chaque neurone relie les faits deux par deux (voir figure 5.8). Cependant, il est peu probable qu'une telle topologie soit créée et la première hypothèse semble plus sensée.

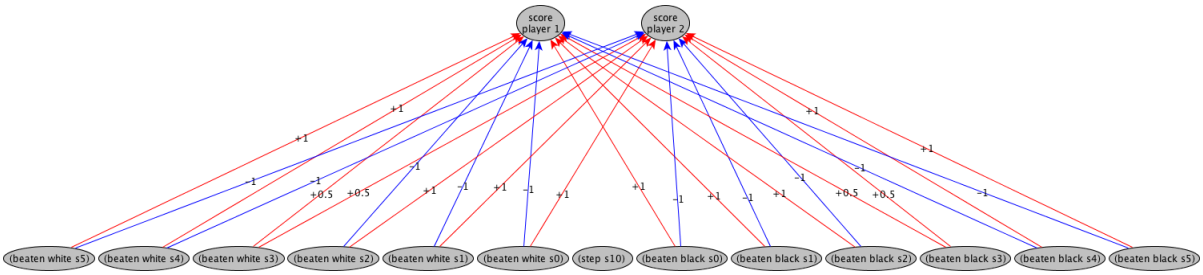


FIGURE 5.7 – Graphe partiel de la topologie attendue pour le jeu de *Roshambo* : il est possible que les faits indiquant que plus de 3 manches ont été gagnées influencent positivement le score tandis que les faits indiquant que moins de manches ont été gagnées aient une influence inverse ; les faits indiquant que 3 manches ont été gagnées pourraient alors avoir une influence de +0.5.

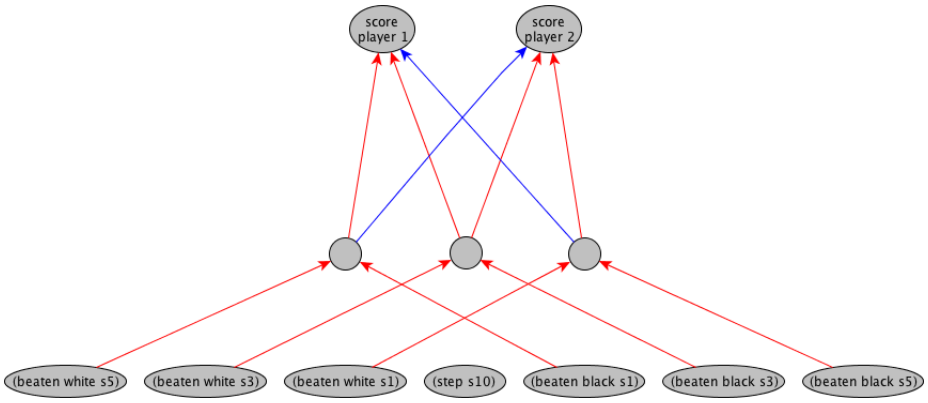


FIGURE 5.8 – Graphe partiel d'une topologie rêvée pour le jeu de *Roshambo* : si NEAT parvient à découvrir la corrélation entre les faits définissant le nombre de manches gagnées, il pourrait combiner les entrées deux à deux grâce à une couche de neurones cachés.

La bonne estimation des scores pour le jeu de *Tictactoe* nécessite de découvrir une topologie complexe reliant les entrées trois par trois (voir figure 5.9). Nous émettons l'hypothèse que NEAT découvrira plus simplement la relation directe entre le score et le joueur qui a la main. Nous vérifierons donc ce fait avant de relancer le test avec un jeu de données débarrassé de cette information pour voir si NEAT peut résoudre ce problème complexe.

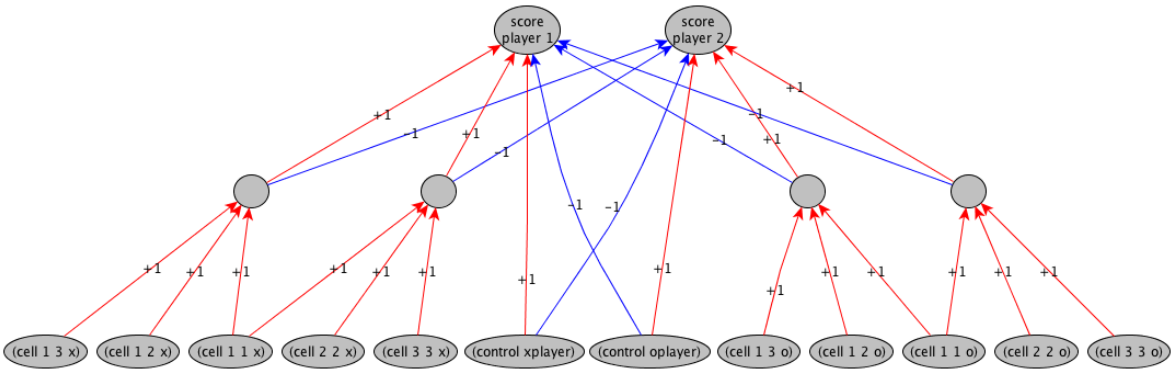


FIGURE 5.9 – Graphe partiel de la topologie espérée pour le jeu de *Tictactoe* : les faits dont l'association a une influence sur le score des joueurs devraient être reliés à des neurones de la couche cachée...

5.2.2 Conditions d'expérimentation

Nous avons lancé NEAT successivement sur les cinq jeux que nous avons choisis avec le même jeu de paramètres décrit au paragraphe 5.1.6 et avec les paramètres modifiés. Notre programme génère à chaque génération une liste des espèces et le *fitness* de leur leader ainsi que la liste des *fitness* de toute la population. Toutes les 400 générations, le programme donne également le détail des génomes des leaders et génère un graphe des réseaux au format DOT dont il est possible ensuite de faire un rendu en image avec le logiciel GraphViz.

Nous avons fait le choix de ne pas générer les graphes à la volée avec la librairie GraphViz pour ne pas ralentir l'exécution de notre programme.

Afin de vérifier le bon fonctionnement de notre programme, nous avons tenté de reproduire l'expérience du XOR décrite par Stanley dans son article [Stanley and Miikkulainen, 2002]. Nous avons obtenu des résultats similaires mais avec une population plus importante que celle qu'il indique avoir utilisée (1000 individus au lieu de 150) et en désactivant le contrôle du nombre d'espèces comme il le fait lui-même. La différence de performance semble due à la manière dont nous avons implémenté l'étape de la reproduction : le modèle proposé par Buckland présente diverses simplifications qui, en rendant le code plus intelligible, ont cependant modifié son comportement. Nos diverses expérimentations ont permis de mesurer l'influence de ces différences d'implémentation.

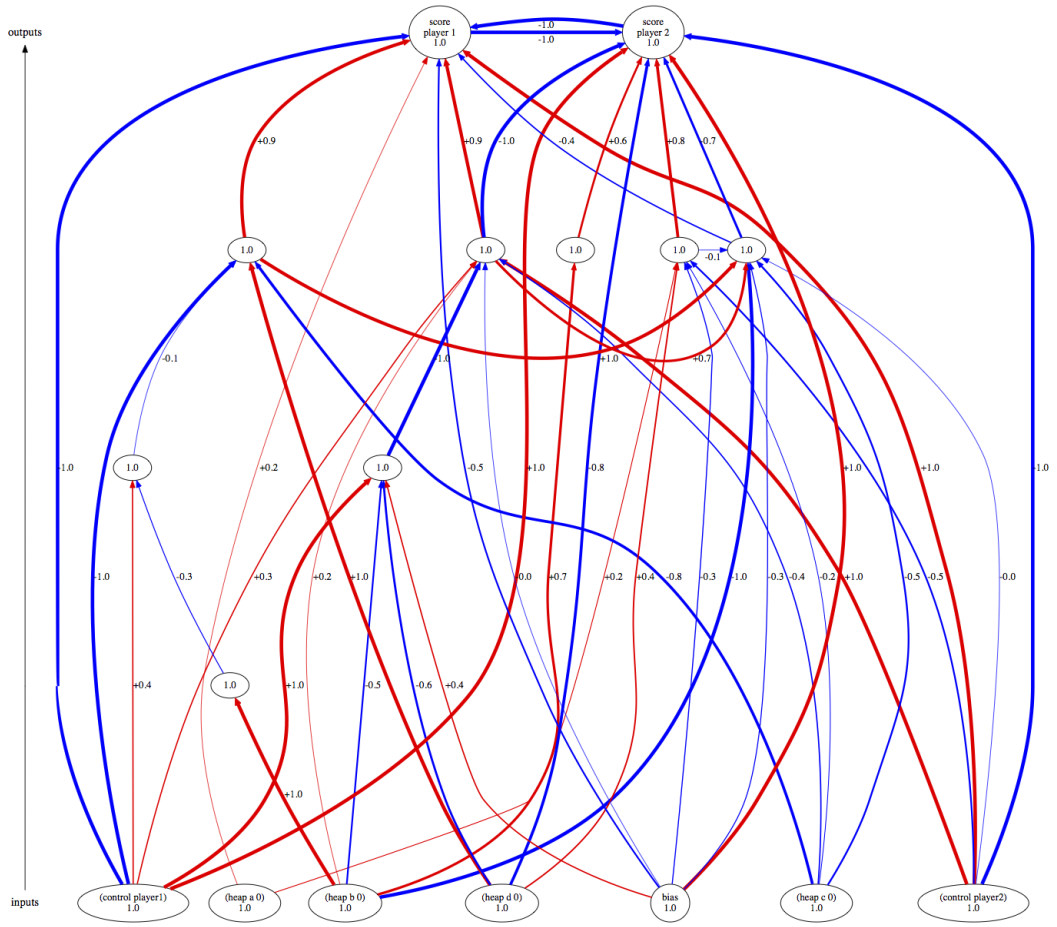


FIGURE 5.10 – Réseau de *fitness* le plus élevé obtenu au bout de 1000 générations pour le jeu de *Nim*. Le réseau est capable de prédire le bon score 99.99% du temps mais sa topologie est loin d'être optimale.

5.2.3 Analyse des résultats

Nos premiers tests effectués en suivant l'implémentation proposée par Buckland ont immédiatement révélé d'importantes difficultés pour obtenir une topologie efficace. Pour le jeu de *Nim* par exemple, un réseau de *fitness* élevé était généré mais sa topologie était inutilement compliquée (voir figure 5.10). L'importance des faits `(control ...)` n'était pas découverte.

La gestion de la spéciation s'est avérée primordiale. Alors que pour résoudre le problème du XOR il semblait nécessaire de supprimer tout contrôle du nombre d'espèces, au contraire, pour ce problème, il semble primordial de maintenir le nombre d'espèces dans une fourchette constante en faisant varier dynamiquement le seuil de similarité (`dCompatibilityThreshold`).

De plus nous avons dû ajuster les limites d'âge et les bonus/malus infligés aux espèces pour favoriser l'exploration de l'espace des données et l'émergence de nouvelles solutions.

Il est apparu également que la limite que nous avions choisie pour la fourchette de valeurs autorisées pour la pondération des connexions était trop limitée. Afin de pouvoir laisser NEAT ajuster librement les connexions, nous avons supprimé toute limite. Les nouvelles connexions sont cependant initialisées avec une pondération entre -1 et $+1$. Ajoutée à la modification que nous avons faite au niveau de la fonction de mutation des pondérations (voir paragraphe 5.1.6), cette modification permet à NEAT de trouver des solutions plus simples dans un temps bien plus réduit : comparez les graphes des figures 5.11 et 5.10.

L'analyse des réseaux générés pour le jeu de *Nim* nous a révélé que l'inutilité des entrées `(heap ...)` n'était pas réellement découverte. Nous avons donc ajouté deux nouveaux types de mutations présentes dans le code de Stanley et absentes de celui de Buckland : les mutations par désactivation de lien ou réactivation. En ajoutant cette possibilité de supprimer des connexions nous pensions obtenir le résultat espéré. Malgré cela, le résultat n'est pas flagrant, sur les graphes de la figure 5.11, on peut voir que les connexions subsistent et que le réseau a été également été inutilement compliqué en ajoutant des nœuds sur ces connexions. Cependant un fait intéressant est la présence de la connexion inhibitrice entre les deux sorties donnant le score : la relation entre les scores a été découverte, le jeu est effectivement à somme nulle. Cette connexion dispense d'ailleurs le deuxième graphe d'avoir une connexion excitatrice reliant le fait `(control player2)` au score du joueur 1.

Pour le jeu *Paire*, notre programme a réussi a générer deux solutions possibles à la 100$^{\text{ème}}$ génération (voir figure 5.12), mais ceci n'a été possible qu'en utilisant les valeurs de bonus et malus proposées par Buckland et il apparaît qu'en diminuant la pression exercée sur les espèces plus âgées, NEAT ne se concentre pas sur des solutions minimales et complexifie exagérément les réseaux générés en ajoutant de nombreux nœuds inutiles.

Nous avons tenté d'augmenter la pression en rapprochant ces valeurs de celles utilisées par Stanley et nous avons pu constater que lorsqu'un nouvel individu possède la mutation nécessaire à la résolution du problème (un neurone de la couche cachée reliant les faits `(nombre 1)` et `(nombre 3)`) il est associé à une nouvelle espèce. Il obtient peu de stimulation, le bonus accordé aux jeunes espèces étant assez faible. En tant que seul représentant de son espèce il se reproduit peu malgré son *fitness* très supérieur au reste de la population. Rapidement l'espèce vieillit avant d'avoir réussi à améliorer encore la solution trouvée, se voit infligée un malus important et n'a plus la possibilité de se reproduire. Ainsi la solution trouvée disparaît.

Stanley propose d'utiliser un âge limite assez bas et une pénalité assez forte, cependant il n'opère pas la reproduction des individus de la même manière que Buckland. À chaque génération 25% des individus sont obtenus par mutation seule sans *crossover* et il propose d'effectuer de temps en temps une reproduction inter-espèce. En cas de stagnation de la population, il propose également de produire la nouvelle génération uniquement à partir des deux espèces dominantes.

> *« In rare cases when the fitness of the entire population does not improve for more than 20 generations, only the top two species are allowed to reproduce,*

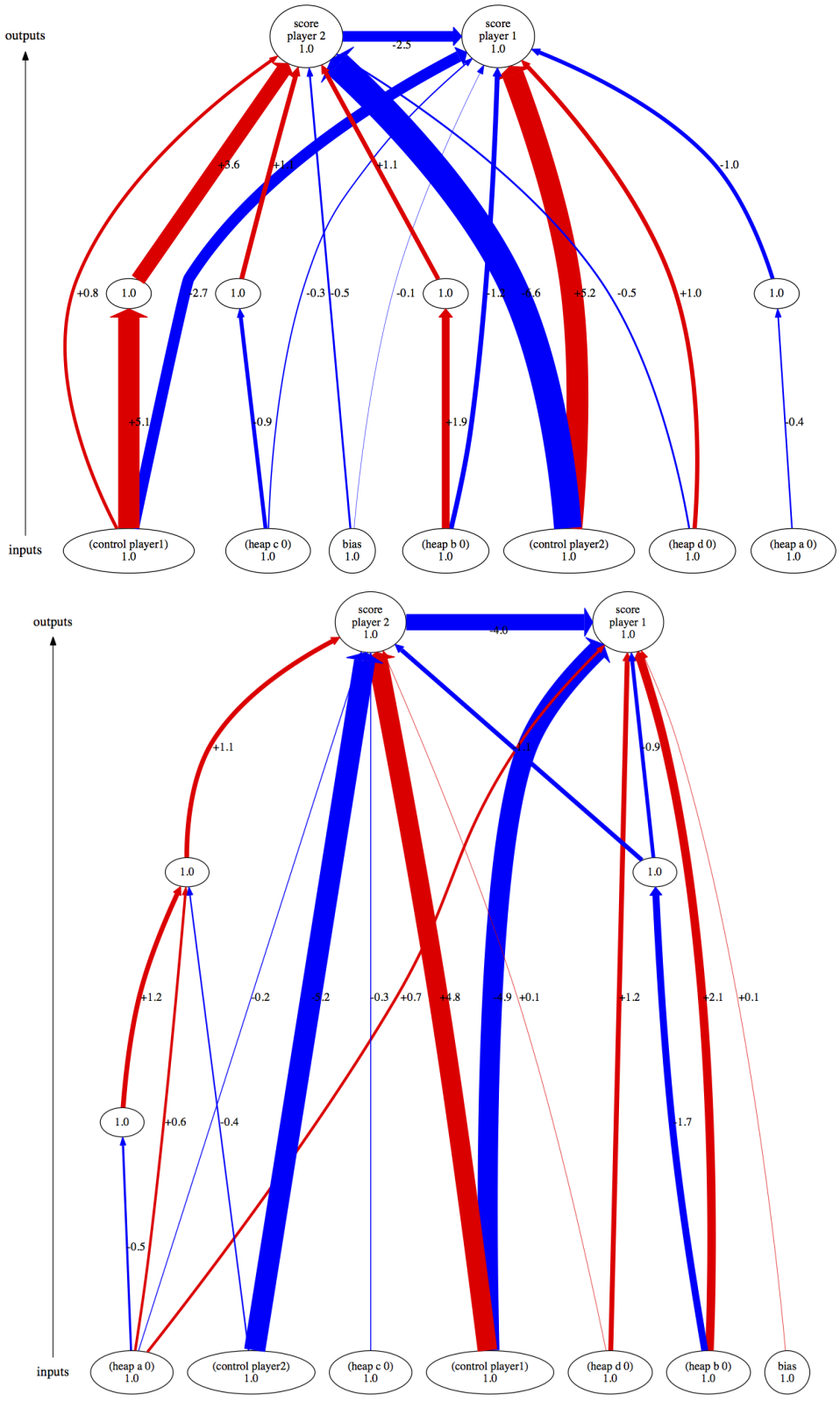


FIGURE 5.11 – Deux exemples de réseaux obtenus au bout de 50 générations pour le jeu de *Nim*. Ces réseaux peuvent prédire le score avec 100% de fiabilité. Le réseau du bas a été découvert en utilisant une liste de positions terminales uniques.

refocusins the search into the most promissing spaces » [Stanley and Miikkulainen, 2002]

Sa technique de sélection des reproducteurs est également différente de celle proposée par Buckland. Les deux techniques semblent devoir aboutir à peu près à la même quantité de descendance en fonction du *fitness* mais il est difficile de mesurer l'impact des possibles légères différences dans des cas précis comme celui-ci sans effectuer des tests approfondis.

Finalement, il apparaît que la technique de reproduction utilisée par Stanley permet d'éviter la disparition des innovations, il peut donc infliger à sa population une forte pression sélective qui lui permet d'obtenir des solutions plus élégantes i.e. des réseaux de topologie minimale, et ceci plus rapidement.

Nos tests sur le jeu *Roshambo* ont révélé les difficultés du système à ajuster les pondérations. Il est apparu que le phénomène que nous avons observé lors de nos tests avec l'algorithme génétique s'accentue avec NEAT : la redondance de certaines positions terminales biaise la recherche en augmentant ou baissant le taux d'erreur artificiellement. Nous avons effectué un tri des données, pour ne conserver que des états uniques dans le but d'améliorer la recherche d'une solution. Une position est considérée identique si elle propose la même liste de faits quelque soit leur ordre. Par exemple les positions terminales suivantes sont considérées comme identiques et une seule est donc conservée :

```
((100 0) (fact A) (fact B))
((100 0) (fact B) (fact A))
```

Dans certains cas, le tri des positions finales n'apporte pas d'amélioration probante du *fitness* mais au pire, le résultat est aussi bon que sans effectuer de tri. Il est à noter néanmoins, que ce tri améliore dans tous les cas la vitesse de calcul en diminuant le nombre d'évaluations nécessaires pour mesurer le *fitness* d'un génome et représente donc un net progrès.

Malgré nos améliorations, la solution trouvée par NEAT n'est pas totalement satisfaisante. Les faits contribuant nettement à la victoire sont bien identifiés (voir figure 5.14), mais les autres pondérations ne sont pas optimales et le réseau a une fois de plus tendance à se complexifier inutilement.

Pour le jeu *Point Grab*, NEAT n'a pas été en mesure de fournir une solution satisfaisante, malgré l'ajout d'un biais en faveur des connexions les plus récentes comme le suggère Stanley (voir paragraphe 5.1.6). Nous avons cependant observé qu'avec l'ajout de ce biais, NEAT a moins tendance à produire de nouveaux nœuds à la recherche d'une solution et se perd moins dans une course à la complexification qui n'amène, de plus, pas de réelle amélioration du *fitness*.

Pour le jeu de *Tictactoe*, nous pouvons constater une fois de plus que, si les diverses améliorations apportées ne permettent pas d'obtenir un *fitness* maximum, la

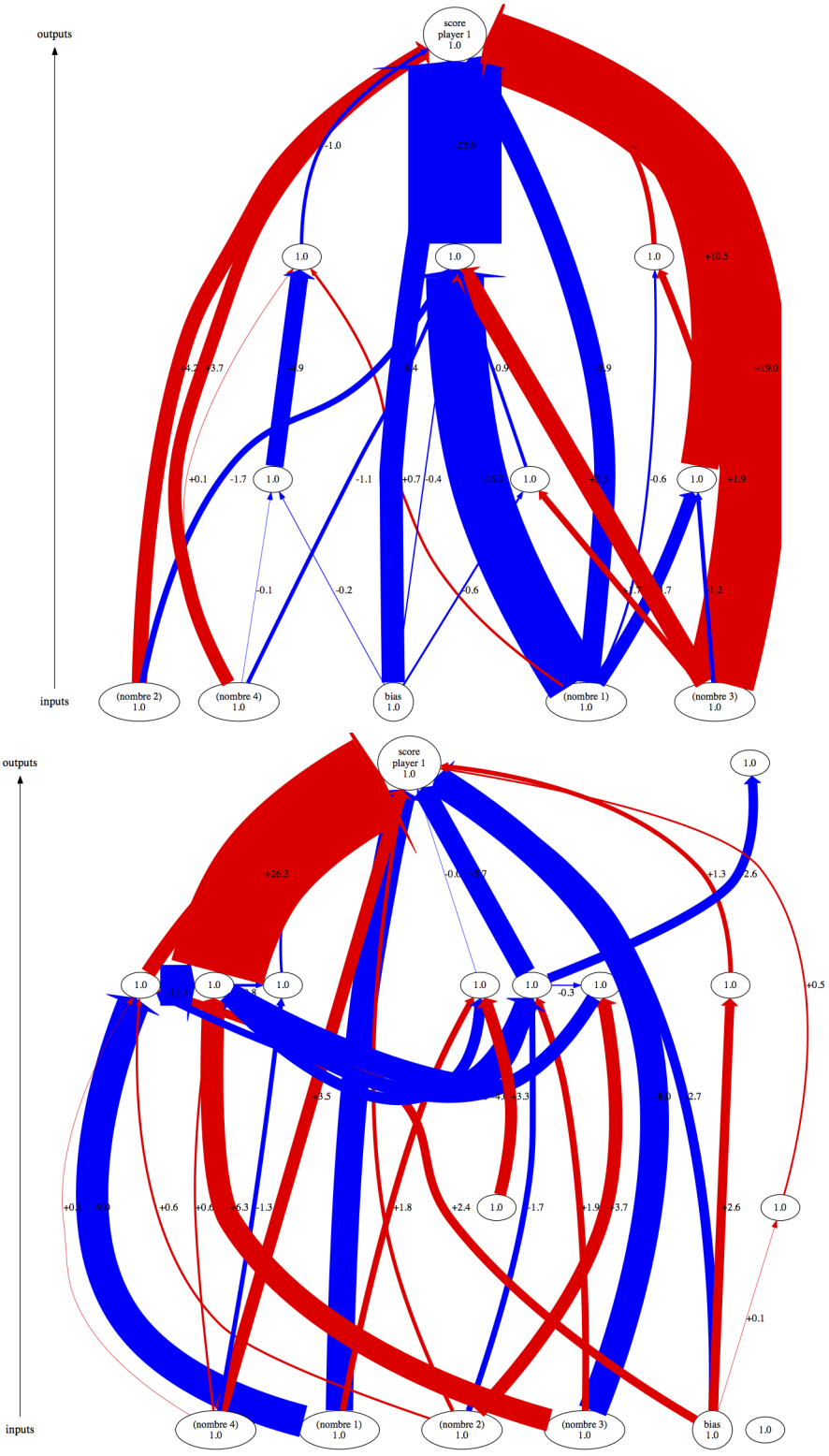


FIGURE 5.12 – Deux réseaux obtenus au bout de 100 générations pour le jeu *Paire*. Ces réseaux peuvent prédire le score avec 100% de fiabilité.

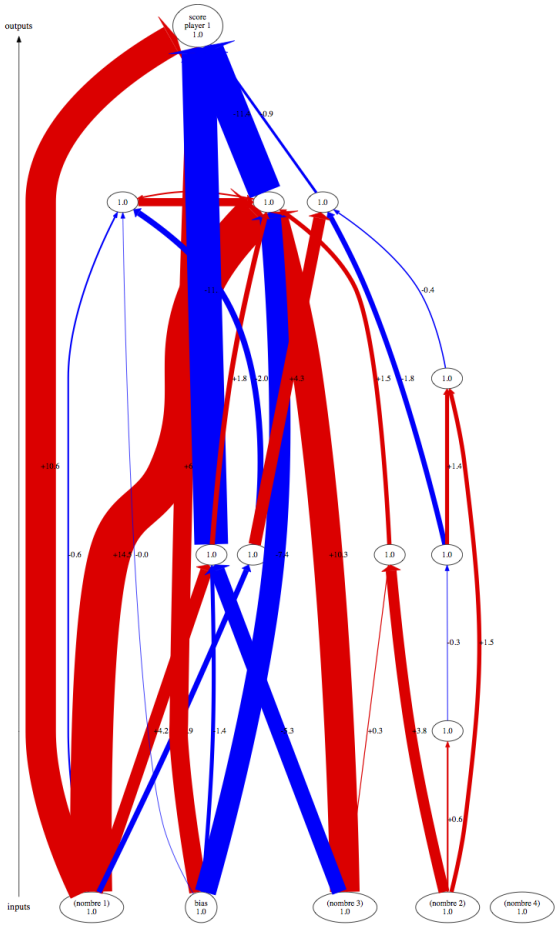


FIGURE 5.13 – Réseau obtenu au bout de 500 générations pour le jeu *Paire*. Ce réseau peut prédire le score avec 100% de fiabilité.

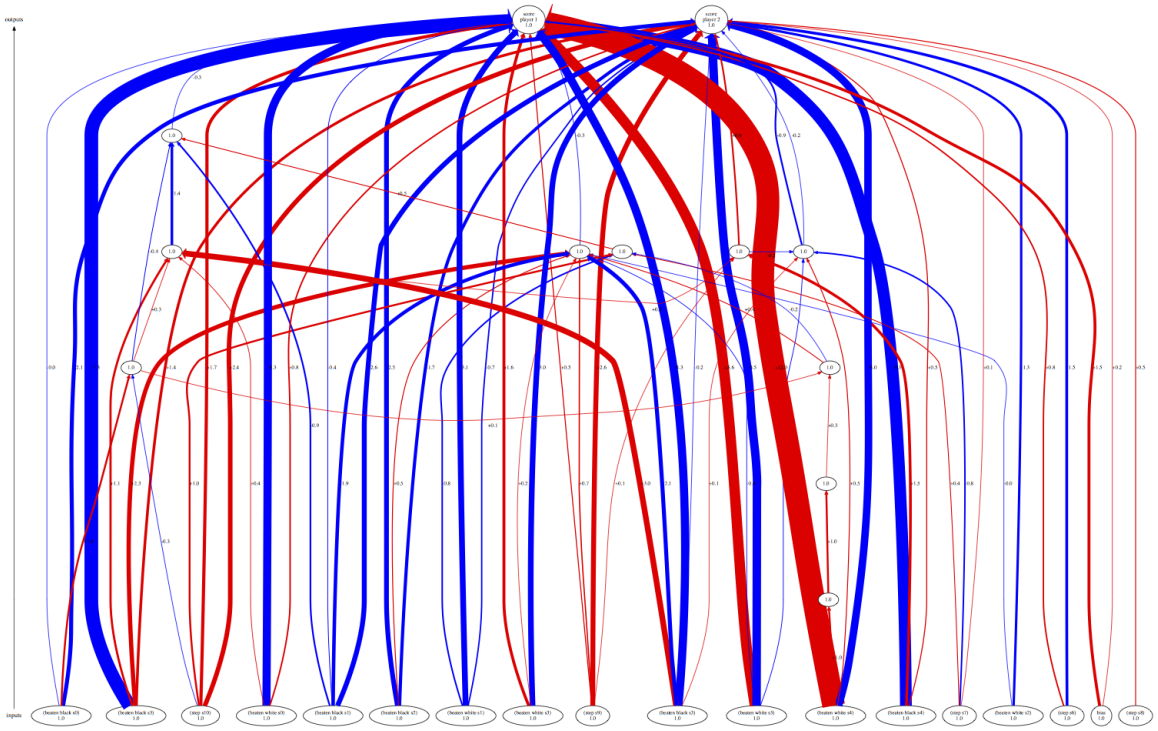


FIGURE 5.14 – Réseau obtenu au bout de 1000 générations pour le jeu *Roshambo*. Ce réseau peut prédire le score avec 94% de fiabilité.

topologie des réseaux est significativement améliorée (comparer les graphes des figures 5.15 et 5.16). Elle est moins complexe (moins de noeuds ajoutés) et les faits `(control ...)` dont l'influence est importante pour l'évaluation du score ont été identifiés.

Comme cette information sur le joueur courant permet à NEAT de s'approcher d'une évaluation juste du score sans découvrir réellement les combinaisons de faits que nous espérions le voir utiliser, nous avons renouvelé le test en supprimant cette information (voir figure 5.17).

Nous constatons que NEAT n'est alors plus en mesure de prédire le score de façon fiable. Il découvre bien l'importance stratégique de la case centrale `(cell 2 2 ...)` et également que les marques de chaque joueur ont une influence positive sur leur score. Cependant, on peut remarquer que des éléments qui ne devraient pas être signifiants, comme le fait `(cell 2 1 b)` se voient apporter une pondération très importante ; ceci s'explique par leur apparition statistique plus importante dans des positions gagnantes pour l'un des joueurs. Le fait `(cell 2 1 b)` est présent 160 fois dans des positions gagnantes pour le joueur 1 et seulement 95 fois dans des positions gagnantes pour le joueur 2, ce qui explique la pondération de cette connexion.

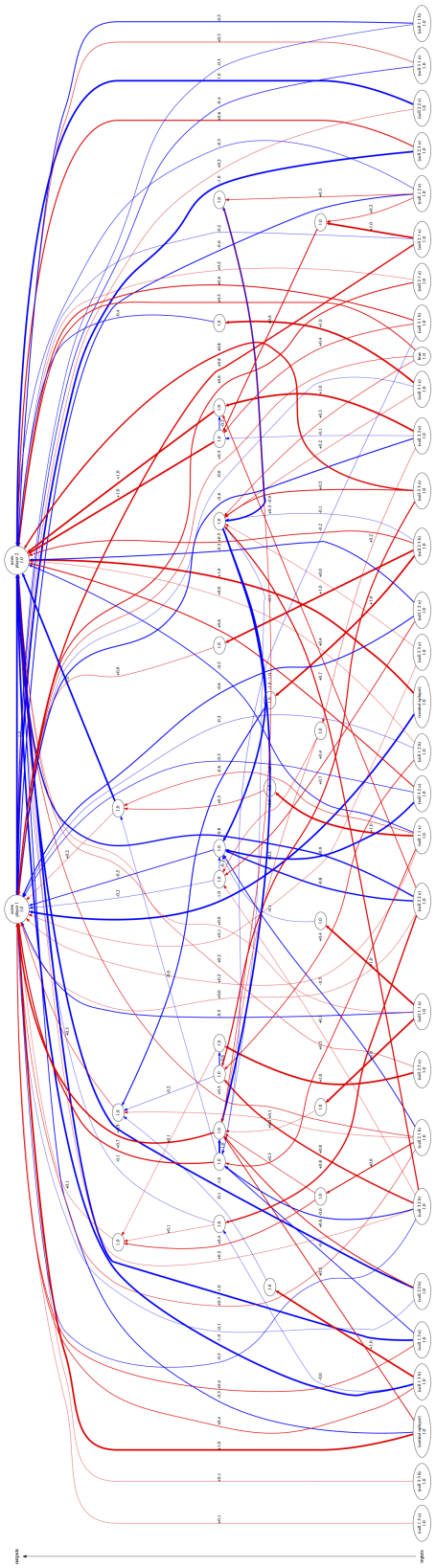


FIGURE 5.15 – Réseau de *fitness* le plus élevé obtenu au bout de 1000 générations pour le jeu de *Tictactoe*. Le réseau ne prédit le bon score qu'à 89.24%**.**

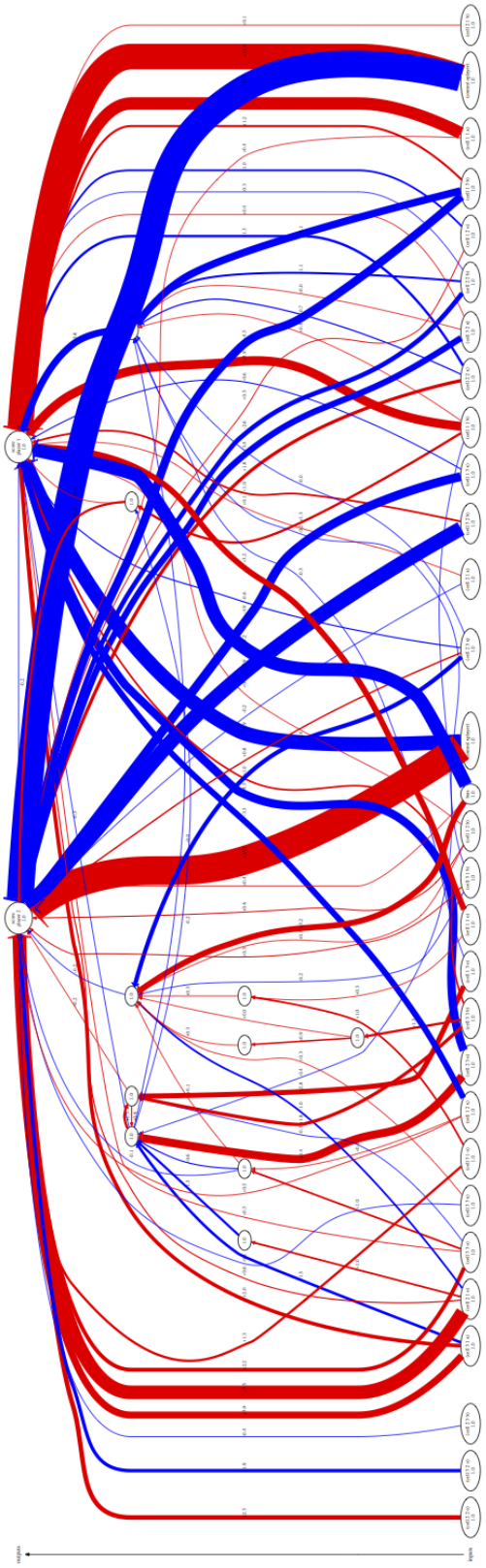


FIGURE 5.16 – **Réseau de *fitness* le plus élevé obtenu au bout de 1000 générations pour le jeu de *Tictactoe*. Le réseau prédit le bon score à 97.74%. Les bonus et handicap de Buckland ont été utilisés et les positions terminales ont été triées pour ne garder que des positions uniques.**

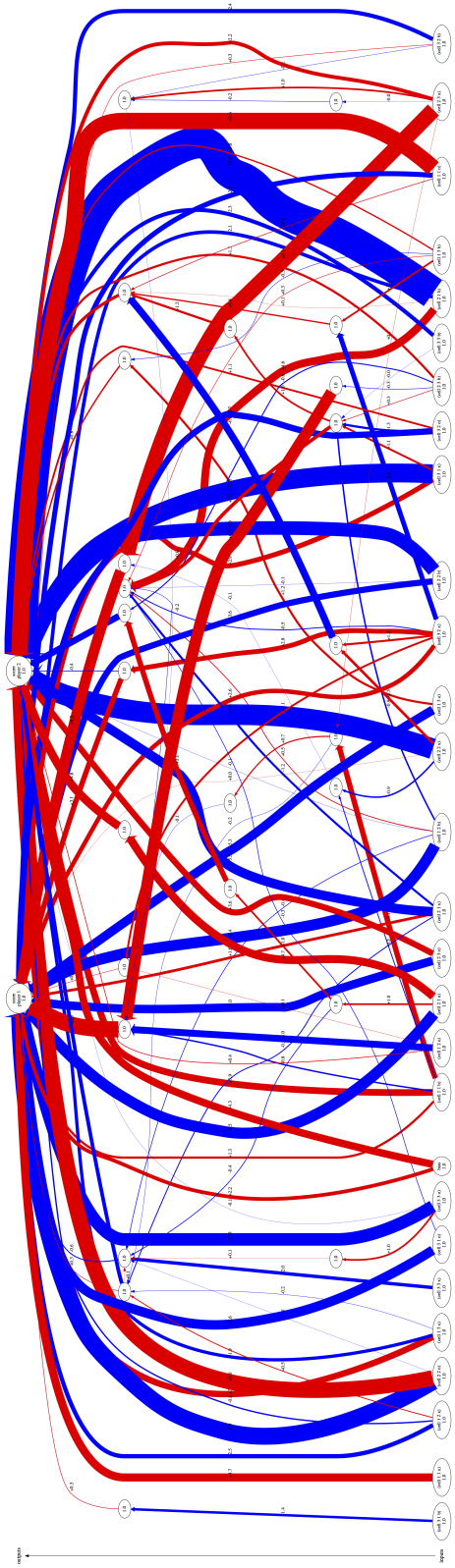


FIGURE 5.17 – **Réseau de *fitness* le plus élevé obtenu au bout de 1000 générations pour le jeu de *Tictactoe*. Le réseau prédit le bon score à $70,33\%$. Les bonus et handicap de Buckland ont été utilisés et les positions terminales ont été triées pour ne garder que des positions uniques.**

5.2.4 Bilan et perspectives

À la fin de ces différents tests le bilan est mitigé. Nous n'avons pas réussi à atteindre notre objectif : notre implémentation de NEAT n'a pas été en mesure de fournir une évaluation efficace du score quand celle-ci nécessite de découvrir une topologie spécifique et complexe. Cependant il apparaît que NEAT est capable dans des cas simples de découvrir la topologie nécessaire à la résolution d'un problème non linéaire.

NEAT nous est donc apparu comme une approche valable pour la résolution de notre problème mais notre implémentation nécessite de fortes modifications pour en améliorer la performance.

L'implémentation proposée par Buckland est réalisée à la base pour contrôler un groupe de petits robots d'exploration devant couvrir un maximum de territoire. Les différentes remarques de Stanley dans la FAQ de son site [Stanley, 2013] laissent penser que le réglage des paramètres et le contrôle des différents aspects de NEAT (*crossover*, mutation et spéciation) sont très différents pour la résolution de ce type de tâche, où le réseau doit évaluer des entrées de manière continue et pour une tâche comme l'évaluation de positions dans un jeu de plateau qui s'apparente à un travail de classification. Il est donc apparent que notre implémentation doit être revue pour s'adapter plus efficacement à notre problème.

Plusieurs points doivent être modifiés. Le mode de sélection des reproducteurs et les fonctions de reproduction doivent être totalement revus pour intégrer les modes de *crossover*, de reproduction inter-espèces et d'élitisme (reproduction des deux meilleures espèces en cas de stagnation) proposés par Stanley. Il nous faut également prévoir de pouvoir définir un taux de reproduction sans croisement (de 25% chez Stanley).

Une fois le mode de reproduction ajusté, les taux de bonus et malus devront permettre d'accentuer la pression de l'évolution pour accélérer la découverte de structures efficaces. Pour contrôler la spéciation et préserver les innovations utiles, il est également nécessaire de vérifier le calcul de la similarité des génomes (équation 3.1). Stanley propose de ne pas effectuer de normalisation pour le calcul des gènes disjoints et en excès (N=1), ou du moins de ne la faire que si le génome des deux parents est de taille suffisante (plus de 20 gènes i.e. plus de 20 connections). Il serait bon de voir dans quelle mesure cela peut influencer l'évolution de la population.

La topologie des réseaux obtenus pour le jeu *Paire* repose la question de savoir si les pondérations des connexions doivent être bornées pour limiter l'apparition de connections tellement fortes qu'elles risquent d'empêcher les nouvelles connexions d'apporter une quelconque modification. Stanley [2013] indique que de telles bornes sont inutiles à moins d'avoir une évolution très longue :

> *« The mutation strength itself basically causes a natural soft bound to arise on its own. By imposing your own bounds, you may have created a range that is intolerant to the strength of your mutation operator. Note that it is the rela-*

> *tive strengths of weights that matters towards creating their functionality, so no bound is necessary. (...) In very long evolutionary runs (say 500 or more generations), a limit may be helpful in preventing some connections from becoming so powerful that new connections cannot hope to compete. It is a simple matter to put in hard bounds if you desire.* » [Stanley, 2013]

Cependant, si on fait le choix d'ajouter de telles limites, la question se pose de la valeur à leur donner ; ceci devra être déterminé expérimentalement.

Stanley propose de démarrer avec un réseau non connecté pour découvrir progressivement les entrées vraiment utiles :

> « *The reason this question is important is that if we can start NEAT with some inputs disconnected, then we can allow NEAT to decide which inputs are important. This process has two good effects : (1) You can start minimally even in problems with many inputs and (2) you don't need to know a priori what the important features of the domain are.* » [Stanley, 2013]

Cette idée a été explorée par Whiteson et al. [Whiteson et al., 2005] et se révèle efficace pour déterminer quelles entrées du réseau sont vraiment utiles dans les cas comme le nôtre où toutes les entrées n'apportent pas une information nécessaire.

Whiteson et al. évoquent aussi l'idée d'ajuster dynamiquement le taux de mutations structurelles au cours du temps :

> « *Intuitively, such mutations are more likely to be helpful early in evolution, when most networks lack the topology and input connections necessary to solve the task. Later in the search, when good structures have already been found and need only fine-tuning, structural mutations are less likely to improve performance.* » [Whiteson et al., 2005]

Un tel système dynamique pourrait également se révéler utile dans notre cas pour s'adapter à la complexité du problème à traiter. En effet, le nombre d'entrées du réseau et la complexité de la topologie à découvrir varient énormément d'un jeu à l'autre, et Stanley indique que le taux de mutation peut varier considérablement en fonction des problèmes à traiter :

> « *Preliminary experiments indicate that high weight mutation rates (i.e.* 50% *or more) are useful for control tasks, but lower rates (i.e. under* 1%*) are more appropriate for high input games like Othello. It may be that the number of inputs is the critical factor* » [Stanley, 2013]

Pour forcer la découverte d'heuristiques utiles pour l'évaluation de positions non terminales, il pourrait être utile de supprimer systématiquement des positions finales les indications concernant le joueur qui a la main, pour éviter que NEAT ne contourne la difficulté en les utilisant. Nous avons vu également que NEAT en découvrant que le jeu est à somme nulle, s'abstient de découvrir certaines connexions

entre les faits et le score. Il serait donc peut être utile d'interdire toute connexion entre les deux sorties dans le cas des jeux à deux joueurs. Dans le cas des jeux multi-joueurs, ce type de connexion peut en revanche se révéler utile pour repérer les coalitions.

Enfin, dans le but d'accélérer la découverte des combinaisons de faits utiles pour l'évaluation du score, certaines mutations orientées pourraient être tentées : en sélectionnant au hasard deux (ou plus) faits vrais dans un état final gagnant, en les reliant à un nœud caché et en reliant ce nœud aux sorties.

Cette idée mérite d'être examinée en détail. Cette sélection de faits présents simultanément dans une position terminale évoque la démarche de Kirci pour son joueur *Maligne* [Kirci et al., 2011]. En s'inspirant de son travail, nous pouvons aussi examiner si, en partant du principe que le dernier fait ajouté correspond au coup qui à provoqué la fin du jeu, inclure systématiquement ce dernier fait parmi les entrées connectées au nouveau nœud peut être utile.

Guider la découverte de structures utiles à la génération d'heuristiques peut sembler aller à l'encontre du principe des algorithmes génétiques qui doivent découvrir par eux-mêmes les solutions réellement performantes. Stanley et Miikkulainen [Stanley and Miikkulainen, 2002] indiquent que démarrer avec une population de topologie minimale est une caractéristique importante déterminant l'efficacité de NEAT :

> « *We examined this question by starting evolution with random topologies as in other TWEANNs. Each network in the initial population received between 1 and 10 hidden neurons with random connectivity (as implemented by Pujol and Poli (1997)). The result is that random-starting NEAT was 7 times slower than full NEAT on average. The random-starting system also failed to find a solution within 1000 generations* 5% *of the time. The result suggests that starting randomly forces NE to search higher-dimensional spaces than necessary, thereby wasting time. If topologies are to grow, they should start out as small as possible.* » [Stanley and Miikkulainen, 2002]

En ajoutant un nœud et plusieurs connections en une seule fois, il est possible que NEAT soit forcé d'explorer un espace de recherche inutilement élargi et que ses performances diminuent de manière similaire. D'un autre côté, dans la FAQ de sa page internet consacrée à NEAT, Stanley indique à propos de FS-NEAT :

> « *One nice mutation which that be helpful if you start with some sensors disconnected is an add_sensor mutation, which immediately connects a randomly chosen sensor which was previously disconnected to all the outputs at once. That way, the sensor can be fully integrated immediately.* » Stanley [2013].

Cette mutation est très similaire à celle que nous proposons et, introduire ces nouvelles connections sous forme d'une mutation aléatoire et non d'une structure imposée à l'initialisation des génomes, doit permettre à l'évolution de déterminer si cette structure est réellement utile.

6 Conclusion

Nous avons présenté notre démarche pour la génération automatique d'heuristiques dans le domaine du *General Game Playing* (GGP).

Nous avons pris comme données d'étude, des jeux proposés dans le cadre des compétitions de GGP de Stanford et dont les règles sont codées grâce au *Game Definition Language* (GDL). Les joueurs programmés pour participer à ces compétitions utilisent des algorithmes de parcours d'arbre pour identifier les positions plus ou moins favorables et savoir quels coups jouer pour maximiser le score. Ces algorithmes de parcours d'arbre nécessitent l'usage d'heuristiques pour l'évaluation des positions. Ces heuristiques peuvent être des connaissances propres au domaine, établies manuellement et calculées à partir des données du jeu (mobilité des pièces, contrôle du centre, etc.). Nous nous opposons à cette démarche qui fait usage d'un savoir propre au domaine et nous avons cherché une technique susceptible de fournir des heuristiques automatiquement à partir de la seule description GDL des positions.

6.1 Etat de notre projet

Dans un premier temps, nous avons associé les faits GDL intervenant dans les différentes positions terminales d'un jeu à des pondérations. La somme des pondérations des faits vrais dans une position terminale indique la valeur de la position et permet d'estimer le score associé. Nous avons utilisé un algorithme génétique pour faire évoluer ces pondérations et nous avons fait plusieurs constatations.

Premièrement, les faits n'ayant pas d'influence sur le score ne sont pas convenablement identifiés par l'algorithme génétique. Au lieu de les associer à une pondération de 0, une pondération négative ou positive leur est attribuée, de manière à ce que la valeur de plusieurs faits inutiles s'annule ou, du moins, qu'elle n'influence pas les pondérations des faits signifiants. Cette particularité de nos données de contenir une proportion d'informations inutiles doit être prise en compte.

Deuxièmement, la présence de positions finales identiques dans nos données biaise l'apprentissage de l'algorithme génétique en faussant l'évaluation du *fitness*. Par exemple, si une position gagnante décrite entre autre par un fait inutile est statistiquement présente plus souvent que les autres, ce fait inutile se verra attribué une

pondération positive alors que son influence réelle est nulle. Nous en avons déduit que pour la génération automatique d'heuristiques il est préférable d'utiliser un jeu de données ne contenant pas de doublon.

Troisièmement, nous avons pu constater, comme prévu, qu'un simple algorithme génétique ne peut réaliser une évaluation correcte d'une position quand le score dépend de l'interaction de plusieurs faits. Par exemple au *Tictactoe*, c'est l'alignement de trois symboles qui détermine un score maximum et non la présence de chaque symbole pris séparément. L'évaluation juste du score nécessite donc l'utilisation d'un système capable de résoudre un problème non linéaire.

Dans un second temps nous avons par conséquent décidé d'utiliser les valeurs booléennes des faits comme signaux d'entrée d'un réseau de neurones. Suite aux remarques d'Utgoff [Utgoff, 2001] concernant les difficultés inhérentes au choix de la topologie d'un réseau de neurones pour la résolution de ce type de problème, nous avons choisi d'utiliser l'algorithme de *Neuro-Evolution of Augmenting Topologies* de Stanley [Stanley, 2004]. Cette technique permet de faire évoluer la topologie d'un réseau pour l'adapter à un problème donné, grâce à un algorithme génétique. Notre but était d'obtenir des topologies à même d'évaluer avec justesse le score correspondant à des positions terminales en découvrant les combinaisons de faits signifiantes.

Nos différents essais pour obtenir une topologie efficace ont abouti à un résultat mitigé.

Nous avons constaté que l'algorithme de *Neuro-Evolution of Augmenting Topologies* est bien capable de découvrir des topologies à même de produire une évaluation juste pour des jeux très simples, mais notre implémentation n'a pu produire des résultats satisfaisants pour des jeux nécessitant des combinaisons plus complexes de faits.

Nous avons pu établir que le mode de reproduction que nous avons implémenté ne permet pas d'exercer une pression sélective suffisante pour permettre une évolution rapide de réseaux de topologies minimales, sans provoquer une perte des innovations. Notre système de spéciation ne remplit pas de ce fait son rôle protecteur des innovations, et sans une pression suffisante pour favoriser les topologies minimales, le système se perd dans l'exploration d'un espace de données trop important et fournit des solutions complexes et peu performantes.

NEAT se révèle donc au final être une approche valable pour la génération automatique d'heuristiques mais nous devons apporter des modifications à notre implémentation pour améliorer les résultats obtenus.

6.2 Perspectives

Nous avons dans le cadre de ce mémoire pris comme modèle l'implémentation de Buckland [Buckland and Collins, 2002]. Cette implémentation diffère de celle de Stanley par la manière de créer une nouvelle génération et par la sélection des re-

producteurs. Buckland conserve systématiquement un clone du meilleur individu d'une espèce puis un autre clone soumis à des mutations. Les autres enfants sont produits en tirant au hasard deux parents parmi les meilleurs de leur espèce pour se reproduire par croisement. Si le nombre d'enfants produit est inférieur à la taille de la population souhaitée, les derniers enfants sont des clones des vainqueurs de tournois de sélection. Cette technique ne semble pas efficace pour préserver les innovations et permettre leur transmission dans la population.

6.2.1 Correction de l'implémentation

Nous allons donc modifier cette partie de notre programme et effectuer comme Stanley une sélection des reproducteurs par *stochastic tournament without replacement* qui, nous l'avons constaté dans nos tests préliminaires avec un algorithme génétique, permet d'obtenir une meilleure approximation de la proportion de descendance pour chaque individu. Le meilleur individu de l'espèce ne sera cloné que si l'espèce compte plus de 5 membres et 25% de la nouvelle génération seront obtenus par mutation seule, sans *crossover* comme le fait Stanley.

En cas de stagnation de la population, Stanley préconise de ne faire reproduire que les deux meilleures espèces (celle dont le taux de descendance prévu était le plus important) et de leur faire générer chacune 50% de la population. Il conviendra de vérifier que cette technique d'élitisme ne risque pas de faire disparaître une espèce moins développée qui pourrait contenir le meilleur individu de la population.

Cette modification du mode de reproduction aura une influence sur l'évolution des espèces et nous espérons pouvoir augmenter la pression de l'évolution en utilisant les paramètres indiqués par Stanley pour les bonus et malus infligés respectivement aux jeunes espèces et aux plus âgées.

Nous effectuerons également d'autres petites modifications et vérifications en fonction des résultats de ce nouveau mode de reproduction. Stanley affirme que les pondérations des connexions ne doivent pas être bornées. Nous avons vu dans notre implémentation que les pondérations peuvent devenir excessives et masquer totalement l'influence de nouvelles connexions. Il conviendra donc de voir si des bornes sont nécessaires et il faudra en ajuster la valeur.

Dans le calcul de la proximité des génomes, Stanley indique qu'il ne normalise pas l'influence des gènes disjoints et en excès, et il conseille de ne le faire que si les génomes sont de taille suffisante i.e. s'ils ont au moins 20 gènes (20 connexions). Il sera nécessaire d'évaluer l'influence de ce mode de calcul et d'adapter la valeur du seuil de compatibilité en fonction.

Grâce à ces modifications nous espérons obtenir une évaluation efficace à 100% pour un jeu comme le *Tictactoe* et une solution élégante i.e. un réseau de topologie raisonnable.

6.2.2 Évaluer des positions non terminales

Notre but ne se résume évidemment pas à évaluer des positions terminales. Une fois notre système correctement ajusté pour réaliser des évaluations fiables des positions terminales sur tous les jeux de notre base de test, notre optique est d'utiliser le réseau pour évaluer des positions non terminales.

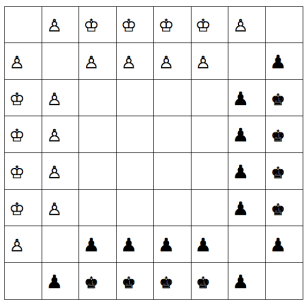
Pour pouvoir ajouter des entrées aux réseaux à mesure que de nouveaux faits sont ajoutés à la description des positions, nous envisageons d'utiliser le modèle FS-NEAT [Whiteson et al., 2005] dont la particularité est de démarrer l'évolution de la population de réseaux, non pas avec des réseaux totalement connectés sans couche cachée comme pour NEAT, mais avec des réseaux sans aucune connexion.

Les différents faits rencontrés lors de l'exécution de playouts UCT pourront être ajoutés aux entrées du réseau à mesure de leur apparition dans la description des positions explorées. L'influence de ces faits sur le score pourra être découverte au cours de l'évolution, de manière à fournir les heuristiques nécessaires à l'orientation de l'exploration de l'arbre du jeu.

À terme, nous espérons pouvoir intégrer ces heuristiques générées automatiquement dans l'algorithme de Monte Carlo d'exploration d'arbre utilisé par un joueur GGP.

A Règles des jeux évoqués

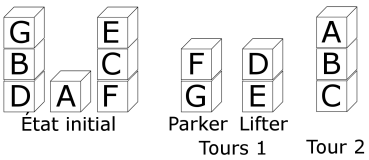
Battle : Sur un damier de 8 cases par 8, chaque joueur possède deux fois 4 rois et 6 pions rangés sur deux bords adjacents du damier. Les rois peuvent se déplacer d'une case à l'horizontale, verticale ou en diagonale, les pions peuvent uniquement bouger sur les horizontales et verticales.



Chaque joueur doit capturer un maximum de pièces de l'adversaire. Le jeu s'achève au bout de 30 tours maximum ou bien lorsque l'un des joueurs a capturé 10 pièces de l'adversaire.

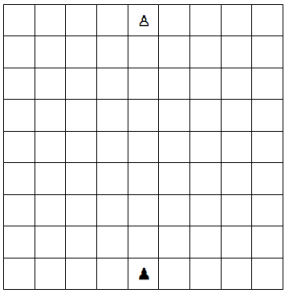
Beat Mania : Un joueur nommé *dropper* dispose de 3 percussions. Pendant 30 tours, il frappe une des percussions au hasard à chaque tour. Au bout de ces 30 tours l'autre joueur doit reproduire exactement le même motif de percussions. Le *dropper* gagne 100 points à chaque fois, l'autre joueur gagne 3 points chaque fois qu'il a tapé sur la même percussion au même tour que le *dropper*. S'il vise juste 25 fois, les 5 derniers coups justes lui rapportent 5 points chacun, si bien que s'il ne se trompe à aucun moment il obtient un score de 100.

Blocks : Ce jeu consiste à déplacer des blocs et à les empiler dans un certain ordre. Les blocs ne peuvent être déplacés que si ils sont en haut d'une pile. Les joueurs *lifter* et *parker* jouent à tour de rôle. Le *lifter* ne peut que prendre un des blocs, tandis que le *parker* ne peut que le déposer sur la table ou sur une des piles existantes. Si le *lifter* réussi à créer une tour avec E et D, il gagne 25 points. Si le *parker* arrive a créer une tour avec G et F, il gagne 25 points. Si l'un des deux arrive à former la tour 2 (C, B, A) sans avoir formé sa tour de type 1, il gagne 75 points. Si l'un des joueurs arrive à former sa tour de type 1 et la tour 2, il gagne 100 points. Le jeux prend fin au bout de 19 tours ou bien lorsque la tour 2 est formée.

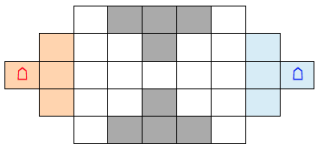


Connect4 : Il s'agit du jeu connu en France sous le nom de *Puissance 4* qui consiste à empiler des pions sur 6 ou 7 colonnes jusqu'à former un alignement de 4 pions. Les joueurs jouent tour à tour et placent un pion de leur couleur (noir ou blanc) sur une des piles. Le premier joueur qui arrive à aligner 4 pions à l'horizontale, verticale ou en diagonale gagne 100 Points et l'autre 0. En cas de match nul chacun reçoit 50 points.

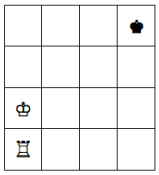
Hallway : Sur un damier de 9x9 cases, chaque joueur possède un pion placé au milieu de deux faces opposées. À chaque tour le joueur peut placer un mur entre deux cases, ou bien se déplacer d'une case à la verticale ou à l'horizontale. Le but du jeu est d'atteindre le côté opposé du damier. Si l'un des joueurs y parvient, il obtient 100 points et l'autre 0. Si aucun des deux n'y parvient au bout de 31 coups, les deux joueurs obtiennent 50 points.



Javastrike : Ce jeu de guerre se déroule sur le plateau dessiné ci-dessous. Chaque joueur possède une base (bleu ou rouge) et peut déployer des unités (pioneer, scout, interceptor, defender, sniper) sur les trois cases qui fond face à sa base. Chaque joueur possède 5 unités de chaque type. Les unités ne sont pas de même force et un ensemble de règles définissent qui l'emporte sur qui. L'issue des combats à force égale est soumise à une part de hasard. Certaines cases du damier sont des zones de brouillard (en gris), sur ces cases l'issue des combats est différente. Les différentes unités peuvent également changer la nature du terrain, sur ces cases l'issue des combats est également différente. Le but de chaque joueur est de prendre possession de la base de l'autre. Si l'un parvient à posséder les deux bases, il obtient 100 points et l'autre 0. Si les deux joueurs arrivent à se vaincre mutuellement, ils obtiennent chacun 50 points. Si à la fin du jeu personne n'a gagné, les deux joueurs obtiennent 0 point. Le jeu prend fin quand l'un des joueurs a pris le contrôle de la base de l'autre, quand un des joueurs a perdu toute ses unités, ou au bout de 200 tours.



MiniChess : Sur un mini-échiquier de 4x4 cases, le joueur blanc possède un roi et une tour tandis que le joueur noir possède juste un roi (voir dessin ci-dessous). Le joueur blanc commence. Si au bout de 10 tours, le joueur blanc n'arrive pas à mettre le joueur noir échec et mat, ce dernier gagne 100 points et le joueur blanc 0, sinon le joueur blanc gagne 100 points et l'autre 0.



Nim : Quatre tas d'allumettes contenant respectivement 1, 3, 5 et 7 allumettes (ou d'autres quantités dans certaines variantes) sont placés devant les joueurs. Chaque joueur doit tour à tour prendre autant d'allumettes qu'il le souhaite mais dans un seul des tas à la fois. Celui qui prend la dernière allumette a gagné (ou perdu dans certaines variantes) et remporte 100 points tandis que l'autre joueur en reçoit 0.

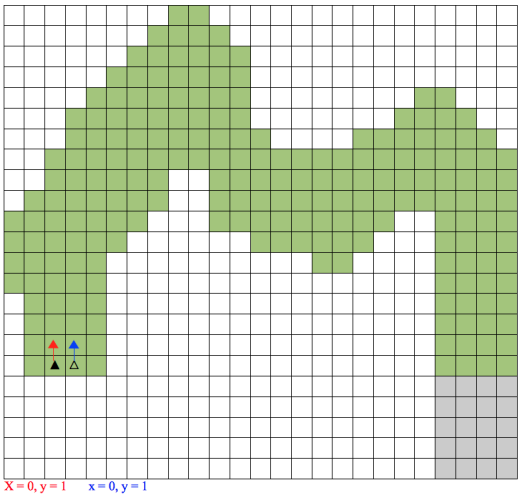
Tictactoe : Il s'agit du jeu également connu en France sous le nom de Morpion [1] qui consiste à aligner trois croix ou trois cercles sur une grille de 3x3 cases. Par convention X est considéré comme étant le joueur 1 et commence. Les joueurs jouent chacun leur tour et déposent leur symbole dans une des cases encore vide. Le premier joueur qui aligne trois symboles dans une ligne, une colonne ou une diagonale remporte 100 points et l'autre 0. En cas de match nul chacun remporte 50 points.

Point grab : Dans ce jeu les deux joueurs jouent simultanément. Chacun a pour but d'accumuler un maximum de points. Différentes actions sont possibles : *scream*, *shout*, *cry*, *whine*, *complain*, *sulk*, *grumble*, *mope*, *a_point* et *b_point*. Seules les actions *a_point* et *b_point* sont susceptibles de rapporter 5 points. Cependant si les deux joueurs choisissent la même action, aucun des deux ne remporte de point. À la fin d'un compte à rebours de 30 tours le jeu prend fin et chacun reçoit le compte de ses points comme score. Si l'un des joueurs atteint 100 points avant la fin, le jeu se termine prématurément.

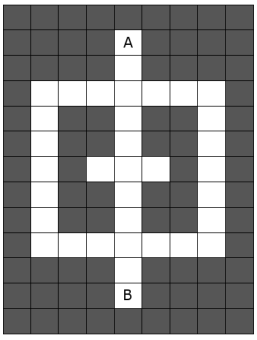
Roshambo : Il s'agit d'un tournoi de "pierre, papier, ciseaux, puits". En 10 manches, les deux joueurs doivent choisir un des quatre éléments. La pierre casse les ciseaux et gagne, le papier enveloppe la pierre et gagne, le papier bouche le puits et gagne, les ciseaux coupent le papier et gagnent, les ciseaux tombent dans le puits et perdent, la pierre tombe dans le puits et perd. Chaque fois que le symbole qu'ils choisissent est gagnant ils obtiennent un point. Deux symboles identiques face à face ne rapportent aucun point à personne. Le jeu prend fin au bout de 10 manches et si l'un des joueurs a plus de points que l'autre, il gagne 100 points, l'autre 0. Si les joueurs sont à égalité (3 points chacun) ils reçoivent chacun 50 points. Si l'un des joueurs atteint 5 points avant la fin des 10 manches, le jeu prend fin prématurément.

1. moins précis puisqu'il désigne plusieurs jeux différents qui n'ont rien à voir entre eux

Racer : Deux véhicules sont placés sur une piste de course. Chacun d'eux peut ajuster sa vitesse de déplacement selon les deux axes de la piste en indiquant les deux valeurs entières de x et y définissant son déplacement. Au départ de la course chacun a un déplacement de $(0, 1)$. À chaque tour du jeu chaque véhicule se déplace de x cases horizontalement et y cases verticalement en fonction des valeurs définies au tour d'avant. Les joueurs ne sont pas autorisés à sortir de la piste (en vert et gris), x et y ne peuvent être égaux à 0 en même temps, et x ne peut être négatif. Si un joueur se retrouve bloqué, il doit déclarer forfait, il obtient alors aucun point et son adversaire 100. Si les joueurs n'ont pas atteint la zone d'arrivée (en gris) au bout de 50 tours, ils gagnent 25 points. Si un joueur est seul à atteindre la zone d'arrivée, il obtient 100 points. Sinon, si les deux sont dans la zone d'arrivée, il gagnent chacun 75 points.



Wallmaze : Deux joueurs A et B sont placés dans un labyrinthe (dessiné ci-dessous). Les deux joueurs peuvent avancer d'une case à chaque tour, leur but est de se rendre là où se trouve l'autre joueur au départ. Les deux joueurs ne peuvent pas se croiser i.e. ils ne peuvent se trouver sur la même case au même moment. S'ils annoncent leur intention de se déplacer sur la même case, ils restent placés là ou ils étaient au coup d'avant. Si les deux joueurs ont atteint leur but en moins de 30 tours, celui qui a parcouru le moins de chemin gagne 75 points, l'autre 25. Si ils ont parcouru la même distance pour parvenir au but, ils gagnent chacun 50 points. Si au bout de 30 tours un des joueurs à atteint son but et pas l'autre, il gagne 100 points et l'autre 0. Si au bout des 30 tours aucun n'a atteint son but, ils ont 0 point tous les deux.



Parmi les 93 jeux utilisés pour nos tests, différentes variantes des jeux que nous avons présentés sont présentes, ainsi que différentes variations des jeux de *Dames anglaises*, *Go* et *Échecs* qui sont suffisamment connus pour qu'il soit superflu de les décrire ici. Nous n'avons pas non plus considéré utile de décrire le jeu *Add Game* car il présente des règles d'une absurde complexité. Appliqué sur un damier de 2x2 cases comme dans la description GDL que nous avons utilisée, ce jeu se révèle totalement dénué d'intérêt. Son arbre se trouve rapidement exploré dans son intégralité par UCT et très peu de positions finales sont générées.

Bibliographie

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. (2012). A survey of monte carlo tree search methods. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1) :1–43.

Buckland, M. and Collins, M. (2002). *AI Techniques for Game Programming*. Premier Press.

Cazenave, T. (2009). Nested Monte-Carlo Search. In Boutilier, C., editor, *IJCAI*, pages 456–461.

Clune, J. (2007). Heuristic evaluation functions for general game playing. In *Proceedings of the 22nd national conference on Artificial intelligence - Volume 2*, AAAI'07, pages 1134–1139. AAAI Press.

Clune, III, J. E. (2008). *Heuristic evaluation functions for general game playing*. PhD thesis, University of California, Los Angeles, Los Angeles, CA, USA. AAI3346923.

Finnsson, H. (2007). Cadia-player : A general game playing agent. Master's thesis, Reykjavík University, School of Computer Science, Reykjavík.

Finnsson, H. (2012). *Simulation-Based General Game Playing*. Doctor of philosophy, School of Computer Science, Reykjavík University.

Finnsson, H. and Björnsson, Y. (2008). Simulation-based approach to general game playing. In *Proceedings of the 23rd national conference on Artificial intelligence - Volume 1*, AAAI'08, pages 259–264. AAAI Press.

Finnsson, H. and Björnsson, Y. (2009). Simulation Control in General Game Playing Agents. In *IJCAI'09 Workshop on General Intelligence in Game Playing Agents*.

Finnsson, H. and Björnsson, Y. (2011). CadiaPlayer : Search Control Techniques. *KI Journal*, 25(1) :9–16.

Genesereth, M. R., Love, N., and Pell, B. (2005). General Game Playing : Overview of the AAAI Competition. *AI Magazine*, 26(2) :62–72.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition.

Group, S. L. (2013). General Game Playing, http ://games.stanford.edu/. Stanford university.

Günther, M. and Schiffel, S. (2013). General Game Playing, http ://130.208.241.192/ggpserver/public/show_games.jsp. Computational Logic group, TU Dresden.

Hackett, P. (1995). *A Comparison of Selection Methods Based on the Performance of a Genetic Program Applied to the Cart-pole Problem.* PhD thesis, Faculty of Engineering and Applied Science, Griffith University, Gold Coast Campus, Queensland.

Holland, J. H. (1992). *Adaptation in Natural and Artificial Systems : An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA.

Kaiser, D. M. (2005). The structure of games. In *Proceedings of the 43rd annual Southeast regional conference - Volume 1*, ACM-SE 43, pages 61–62, New York, NY, USA. ACM.

Kaiser, D. M. (2007). Automatic feature extraction for autonomous general game playing agents. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, AAMAS '07, pages 93 :1–93 :7, New York, NY, USA. ACM.

Kirci, M., Sturtevant, N., and Schaeffer, J. (2011). A GGP Feature Learning Algorithm. *KI - Künstliche Intelligenz*, 25(1) :1–8.

Kuhlmann, G., Dresner, K., and Stone, P. (2006). Automatic Heuristic Construction in a Complete General Game Player. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence*, pages 1457–62.

Kuhlmann, G. and Stone, P. (2007). Graph-Based Domain Mapping for Transfer Learning in General Games. In *Proceedings of the 18th European Conference on Machine Learning*.

Marcolino, L. S. and Matsubara, H. (2011). Multi-agent Monte Carlo Go. In *The 10th International Conference on Autonomous Agents and Multiagent Systems - Volume 1*, AAMAS '11, pages 21–28, Richland, SC. International Foundation for Autonomous Agents and Multiagent Systems.

Méhat, J. and Cazenave, T. (2008). An Account of a Participation to the 2007 General Game Playing Competition.

Méhat, J. and Cazenave, T. (2010). Combining UCT and Nested Monte Carlo Search for Single-Player General Game Playing. *IEEE Trans. Comput. Intellig. and AI in Games*, 2(4) :271–277.

Pell, B. D. (1993). *Strategy generation and evaluation for meta-game playing*. PhD thesis, Trinity College, University of Cambridge.

Pitrat, J. (1971). A general game playing program. In Findler and Meltzer, editors, *Artificial Intelligence and Heuristic Programming*, pages 125–155. Edinburgh University Press.

Pitrat, J. (1976). A program to learn to play chess. In *Pattern Recognition and Artificial Intelligence*, pages 399–419. Academic Press Ltd. London, UK.

Quenault, M. (2010). *Une application de jeux génériques pour les jeux symboliques*. PhD thesis, Université Paris Dauphine.

Reisinger, J., Bahceci, E., Karpov, I., and Miikkulainen, R. (2007). Coevolving Strategies for General Game Playing. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, pages 320–327, Piscataway, NJ. IEEE.

Reisinger, J. and Miikkulainen, R. (2007). Acquiring evolvability through adaptive representations. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, GECCO '07, pages 1045–1052, New York, NY, USA. ACM.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM J. Res. Dev.*, 3(3) :210–229.

Samuel, A. L. (1967). Some studies in machine learning using the game of checkers. II : recent progress. *IBM J. Res. Dev.*, 11(6) :601–617.

Schiffel, S. and Thielscher, M. (2007a). Fluxplayer : A successful general game player. In *In : Proceedings of the AAAI National Conference on Artificial Intelligence*, pages 1191–1196. AAAI Press.

Schiffel, S. and Thielscher, M. (2007b). *Automatic Construction of a Heuristic Search Function for General Game Playing*. Hyderabad, India.

Sharma, S., Kobti, Z., and Goodwin, S. (2008a). Learning and Knowledge Generation in General Games. In *IEEE Symposium On Computational Intelligence and Games*, CIG '08, pages 329–335, Sch. of Comput. Sci., Univ. of Windsor.

Sharma, S., Kobti, Z., and Goodwin, S. (2008b). Knowledge Generation for Improving Simulations in UCT for General Game Playing. In *Proceedings of the 21st Australasian Joint Conference on Artificial Intelligence : Advances in Artificial Intelligence*, AI '08, pages 49–55, Berlin, Heidelberg. Springer-Verlag.

Sharma, S., Kobti, Z., and Goodwin, S. G. (2009). Coevolving intelligent game players in a cultural framework. In *Proceedings of the Eleventh conference on Congress on Evolutionary Computation*, CEC'09, pages 638–645, Piscataway, NJ, USA. IEEE Press.

Stanley, K. O. (2004). *Efficient Evolution of Neural Networks Through Complexification*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin.

Stanley, K. O. (2013). The NeuroEvolution of Augmenting Topologies (NEAT) Users Page, http ://www.cs.ucf.edu/ kstanley/neat.html. Department of Electrical Engineering and Computer Science, College of Engineering and Computer Science, University of Central Florida.

Stanley, K. O. and Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evol. Comput.*, 10(2) :99–127.

Sutton, R. S. (1988). Learning to Predict by the Methods of Temporal Differences. *Mach. Learn.*, 3(1) :9–44.

Thielscher, M. (2010). A General Game Description Language for Incomplete Information Games. In *AAAI*.

Thielscher, M. (2011). The general game playing description language is universal. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence - Volume Volume Two*, IJCAI'11, pages 1107–1112. AAAI Press.

Utgoff, P. E. (2001). *Machines that Learn to Play Games*, chapter 7, Feature Construction for Game Playing, pages 131—-152. Nova Science Publishers, Huntington, NY.

Whiteson, S., Stone, P., Stanley, K. O., Miikkulainen, R., and Kohl, N. (2005). Automatic Feature Selection via Neuroevolution. In *Proceedings of the Genetic and Evolutionary Computation Conference*.